Alma Mater Studiorum
Università di Bologna
Campus di Cesena

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

Ottimizzazione dei percorsi di evacuazione e gestione delle notifiche in un sistema di allerta

Elaborato in:
Basi di Dati Avanzate

Relatore:
Prof.ssa Alessandra Lumini

Correlatore:
Prof.ssa Annalisa Franco

Presentata da:
Lisa Vandi

Sessione II
Anno Accademico 2024-2025

Alla mia famiglia,
linfa della mia esistenza:
è con voi e per voi che ogni mio sogno diventa realtà.
È vostro un capo del filo, con l'altro io correrò nel mondo.
Se dovessi perdermi, voi, tirate.

Introduzione

Il presente elaborato di tesi investiga la problematica della gestione dell'evacuazione in ambienti interni in contesti emergenziali. Lo studio si concentra sull'analisi di un caso pratico applicato al Campus di Cesena dell'Università di Bologna, proponendo una soluzione che supera i limiti dei sistemi di allerta pubblica macro-geografici. Il sistema ideato offre la granularità necessaria per una gestione efficace dell'evacuazione, garantendo una comunicazione personalizzata e un percorso di fuga ottimizzato per ogni occupante.

La gestione efficace delle emergenze richiede di superare il divario tra sistemi di allerta su larga scala e strategie di evacuazione dettagliate e personalizzate. Come illustrato nel Capitolo 1, ciò implica la capacità di localizzare gli individui, acquisire in tempo reale informazioni sui loro movimenti, e generare percorsi personalizzati che eludano aree di rischio e colli di bottiglia. La letteratura e le pratiche industriali indicano che un'architettura modulare ed event-driven - quale quella descritta nel Capitolo 2 - è la soluzione ottimale per soddisfare requisiti di bassa latenza, reattività e scalabilità.

La soluzione progettata e implementata, le cui tecnologie principali sono analizzate nel Capitolo 3, adotta un'architettura event-driven a microservizi in Python. Il sistema integra RabbitMQ per la messaggistica asincrona e PostgreSQL/PostGIS per la persistenza dei dati geospaziali. La topologia dell'edificio, modellata come un grafo multi-piano, consente il calcolo in tempo reale dei percorsi sicuri, escludendo dinamicamente aree a rischio. La comunicazione tra i servizi è gestita attraverso un meccanismo di handshake che garantisce la consistenza dei dati.

I microservizi principali comprendono il Gestore degli alert, il Centro notifiche, il Simulatore delle posizioni, il Gestore delle posizioni, il Visualizzatore della mappa e il Gestore della mappa, ciascuno dedicato a responsabilità funzionali specifiche nell'ambito della gestione delle emergenze. I dettagli implementativi dei microservizi sviluppati sono discussi nei Capitoli 4-6.

I contributi originali di questa tesi includono: l'ideazione di un'architettura a microservizi scalare e modulare; lo sviluppo di una pipeline deterministica per il ricalcolo adattivo dei percorsi; la creazione di un'applicazione web interattiva per la visualizzazione e l'editing del grafo; l'implementazione di un protocollo di sincronizzazione per garantire la comunicazione tra i microservizi.

La validazione sperimentale, presentata nel Capitolo 7, ha attestato l'efficienza, la correttezza topologica e la robustezza del sistema: esso genera e comunica in tempi utili percorsi di evacuazione corretti, che evitano sistematicamente aree pericolose e e deviano il flusso in caso di criticità, avvalorando l'efficacia dell'approccio.

Il lavoro si conclude con le riflessioni finali e le prospettive di sviluppo futuro 7.5.4.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Descrizione del problema affrontato

Il presente capitolo analizza il contesto applicativo ed esamina le criticità tecniche e organizzative connesse alla gestione dell'evacuazione indoor in scenari emergenziali, con specifico riferimento al caso di studio del Campus di Cesena dell'Università di Bologna.
In primo luogo, la Sezione 1.1 formalizza il problema, evidenziando i requisiti funzionali e non funzionali che hanno orientato la progettazione di una soluzione alternativa, basata su un'architettura modulare e tecnologie avanzate di comunicazione e localizzazione. Segue, nella Sezione 1.2, un'analisi critica dello stato dell'arte, con particolare attenzione ai sistemi di allerta pubblica IT-Alert e IPAWS, valutandone caratteristiche, punti di forza e limitazioni rispetto al contesto *indoor*. Infine, la Sezione 1.3 giustifica la necessità di un'architettura innovativa, evidenziando le carenze delle soluzioni esistenti rispetto ai requisiti definiti e motivando lo sviluppo di un sistema scalabile, interoperabile e orientato alla gestione in tempo reale di scenari emergenziali.

1.1 Descrizione del problema

La gestione dell'evacuazione di edifici complessi in scenari emergenziali rappresenta una sfida di natura tecnica e organizzativa che trascende la mera diffusione di un segnale di allarme. In contesti caratterizzati da alta densità di occupanti, variabilità spaziale e vincoli topologici, è necessario affrontare criticità specifiche per garantire un'evacuazione rapida, sicura ed efficiente. Le principali difficoltà riscontrate nello sviluppo progettuale possono essere sintetizzate nei seguenti punti:

- **Tempestività dell'allerta**: le prime informazioni devono raggiungere gli occupanti con latenza nulla o trascurabile, in maniera tale da ridurre la probabilità di vittime in scenari critici quali incendi, rilasci chimici o eventi sismici.

- **Affidabilità del canale di comunicazione**: i sistemi di messaggistica tradizionali (SMS, reti dati) decadono drasticamente in presenza di congestione, blackout parziale o saturazione cellulare. Occorre dunque un'infrastruttura che sia resiliente a fault di rete e integrabile con standard di allerta pubblica (CAP) [1].

- **Consapevolezza situazionale in tempo reale**: il tracciamento fine-grained degli occupanti, le cui densità e distribuzione spaziale variano dinamicamente fra aule, corridoi e spazi comuni, riduce i colli di bottiglia sulle vie di fuga.

- **Ottimizzazione dei percorsi di evacuazione**: occorre generare, in modo automatizzato e con complessità prossima al real-time, percorsi che minimizzino tempo di evacuazione, esposizione al rischio e sovraccarico dei nodi critici, tenendo conto di vincoli topologici (scale, uscite di sicurezza, barriere architettoniche) e di scenari in evoluzione.

- **Scalabilità e robustezza architetturale**: la soluzione deve essere modulare, distribuibile su microservizi e tollerante ai guasti; deve inol-

tre supportare l'estensione a edifici ulteriori e a classi di rischio differenti senza reingegnerizzazione sostanziale.

In questo contesto, il **Campus di Cesena dell'Università di Bologna** rappresenta un caso di studio esemplare. Contraddistinto da una struttura complessa, con aule, corridoi, spazi comuni e una popolazione eterogenea, esso presenta sfide specifiche legate alla variabilità della densità degli occupanti e alla necessità di coordinare un'evacuazione efficace in presenza di vincoli topologici. In termini operativi, il quesito si traduce in un ecosistema software che: (i) integri messaggi CAP generati da fonti autorevoli, (ii) localizzi gli utenti in tempo reale, (iii) generi percorsi di evacuazione ottimali, (iv) diffonda istruzioni personalizzate con bassa latenza e (v) mantenga una rappresentazione topologica aggiornata dell'edificio.
Ne risulta un problema intrinsecamente complesso che coinvolge orchestrazione distribuita, ottimizzazione e comunicazione fault-tolerant.

1.2 Analisi dello stato dell'arte

Nel contesto della gestione delle emergenze pubbliche, la tempestività e l'efficacia della comunicazione si configurano come parametri cruciali per la protezione della popolazione. A tale scopo, diversi paesi hanno sviluppato soluzioni avanzate per la trasmissione di messaggi urgenti. Tra i sistemi di allerta più rilevanti, IT-Alert in Italia [2] e IPAWS [3] negli Stati Uniti costituiscono esempi di piattaforme sofisticate che utilizzano tecnologie moderne per garantire una comunicazione rapida ed efficace in scenari di emergenza.
La presente sezione esplora in dettaglio le caratteristiche distintive di IT-Alert e IPAWS, esaminando le modalità operative di ciascun sistema e le sinergie con altre tecnologie internazionali, con l'obiettivo di evidenziare il loro contributo al miglioramento delle capacità di gestione delle emergenze in tempo reale.

1.2.1 IT-Alert

IT-Alert è il sistema nazionale italiano di allerta pubblica, progettato per la trasmissione immediata e diretta di informazioni relative a situazioni di emergenza alla popolazione. Il sistema consente di notificare i dispositivi mobili situati in aree geografiche specifiche, interessate da pericoli imminenti o in corso, assicurando una comunicazione rapida ed efficace, essenziale per la sicurezza pubblica [2].

Nella sua fase iniziale, IT-Alert è stato concepito per la trasmissione dei messaggi di allerta esclusivamente attraverso il Dipartimento della Protezione Civile, che ha assunto il ruolo centrale nella gestione delle notifiche. Questa fase pilota ha permesso di testare l'efficacia del sistema nel raggiungere la popolazione in situazioni di emergenza. Tuttavia, il sistema è destinato a un ampliamento progressivo, che lo renderà accessibile a tutte le componenti del Servizio Nazionale di Protezione Civile [2]. L'integrazione graduale di enti locali, regionali e nazionali ha come obiettivo quello di rendere la gestione delle emergenze più capillare ed efficiente, ottimizzando l'uso dei canali di comunicazione disponibili in funzione degli specifici scenari di rischio. L'obiettivo è consentire una diffusione istantanea e performante delle informazioni iniziali e delle istruzioni di autoprotezione.

IT-Alert utilizza la tecnologia di *cell broadcast* per l'invio simultaneo di avvisi a tutti i dispositivi mobili accesi e connessi alle celle di rete nella zona di interesse [4]. Questa tecnologia assicura robustezza anche in condizioni di scarsa copertura o congestione della rete telefonica. I dispositivi spenti o fuori copertura non ricevono il messaggio, mentre quelli attivi emettono un tono distintivo che identifica il mittente come "IT-Alert".

Il sistema è unidirezionale, non prevedendo feedback né raccolta di dati personali, tutelando così la privacy. Non è richiesta l'installazione di applicazioni dedicate né procedure di registrazione; i telefoni abilitati ricevono automaticamente i messaggi. IT-Alert è conforme allo standard internazionale *Common Alerting Protocol* (CAP) [5], che definisce un formato globale per la trasmissione degli avvisi di allerta pubblica. L'adozione del CAP garan-

tisce un'interpretazione uniforme e facilita l'integrazione e l'interoperabilità con altre piattaforme di allerta internazionali, come IPAWS, discusso nella Sezione 1.2.2. Questo contribuisce a una gestione delle emergenze coordinata a livello globale [6].
Tuttavia, la piattaforma non effettua localizzazione indoor e non fornisce routing personalizzato.

Dal febbraio 2024, IT-Alert è operativo con messaggi reali per scenari di rischio elevato, quali incidenti nucleari, emergenze industriali, collassi di grandi dighe e attività vulcanica critica. Per altre circostanze di pericolo, come terremoti, maremoti e alluvioni, il sistema è ancora in fase di sperimentazione, con l'intenzione di estendere la copertura alle emergenze naturali. Grazie a test continuativi e miglioramenti del sistema, si prevede un progressivo affinamento della tecnologia e l'ampliamento delle aree geografiche coperte, assicurando una capacità di intervento sempre più robusta e pronta a rispondere a ogni tipo di emergenza.

1.2.2 IPAWS

IPAWS (Integrated Public Alert and Warning System) è il sistema nazionale statunitense, gestito dalla *Federal Emergency Management Agency* (FEMA), finalizzato alla diffusione capillare e simultanea di allerte di emergenza attraverso molteplici canali di comunicazione [3]. Esso consente alle autorità federali, statali, locali, tribali e territoriali di inviare un unico messaggio di allerta che raggiunge la popolazione tramite TV, radio, dispositivi mobili e altri media.

IPAWS utilizza un'ampia gamma di tecnologie per una diffusione ottimale delle allerte. Il sistema impiega il *Wireless Emergency Alerts* (WEA) per dispositivi mobili, l'*Emergency Alert System* (EAS) per trasmissioni via radio e televisione, e i canali del *NOAA Weather Radio* per eventi meteo specifici [3]. Questi canali operano in parallelo, assicurando una copertura completa e multifocale. Il sistema è stato sviluppato dalla FEMA in collaborazione con il *Department of Homeland Security* (DHS) per modernizzare e unificare

l'infrastruttura di comunicazione d'emergenza nazionale, creando un sistema altamente integrato e affidabile in situazioni di criticità [7].

Il processo tecnico prevede che le autorità autorizzate redigano un messaggio di emergenza nel formato *Common Alerting Protocol* (CAP) e lo inviino alla piattaforma centrale IPAWS-OPEN. Da lì, il messaggio viene autenticato e diffuso simultaneamente tramite tutti i canali disponibili [8]. Questo approccio centralizzato consente a IPAWS di raggiungere il massimo numero di cittadini con un'unica emissione, riducendo i tempi di risposta e migliorando la gestione delle emergenze a livello nazionale.

I principali canali utilizzati da IPAWS includono [3]:

- EAS (Emergency Alert System): trasmissione di avvisi su radio AM/-FM e TV, essenziale per una rapida diffusione delle allerte in aree ad alta densità di popolazione.
- WEA (Wireless Emergency Alerts): notifiche geolocalizzate inviate direttamente ai dispositivi mobili, funzionanti anche in caso di rete congestionata.
- NOAA Weather Radio: trasmissione via radio dedicata a eventi meteorologici specifici [9].
- altri canali: sirene, pannelli elettronici stradali, applicazioni e piattaforme online, che completano la rete di comunicazione e permettono una copertura completa [10].

Il sistema consente la segmentazione geografica (*geofencing*), indirizzando l'avviso solo ai dispositivi presenti nella zona di interesse, senza la necessità di iscrizioni da parte dell'utente [3]. Questo approccio assicura che gli avvisi siano inviati solo a chi si trova effettivamente nella zona di pericolo, evitando allarmi inconsistenti.

Dunque, IPAWS è concepito per coperture macro-geografiche: manca di precisione spaziale in ambienti confinati e di percorsi di evacuazione personalizzati. Restano inoltre assenti meccanismi di feedback utente e di adattamento topologico in tempo reale.

Pertanto, IPAWS si distingue per la sua scalabilità e interoperabilità, essendo progettato per una copertura massima in situazioni di emergenza, e per l'efficienza con cui gestisce la trasmissione dei messaggi su più canali, garantendo una risposta rapida e coordinata a livello nazionale e internazionale [11].

1.3 Motivazioni dello sviluppo

Sebbene IT-Alert e IPAWS rappresentino soluzioni valide e avanzate per l'allerta pubblica, demandano alle autorità locali la gestione indoor, dove granularità spaziale e dinamiche di folla richiedono approcci dedicati.

Le loro limitazioni principali riguardano:

- **Mancanza di localizzazione indoor**: entrambi i sistemi operano a livello territoriale, senza capacità di localizzazione precisa all'interno di edifici, essenziale per gestire la densità variabile e i vincoli topologici di un campus universitario.

- **Assenza di ottimizzazione dei percorsi**: IT-Alert e IPAWS si limitano a trasmettere notifiche generiche, senza generare percorsi di evacuazione personalizzati o adattarsi a scenari dinamici;

- **Limitata interattività**: l'approccio unidirezionale non consente feedback dagli occupanti né la gestione dinamica delle informazioni situazionali.

Queste lacune giustificano lo sviluppo di un sistema innovativo, modulare e interoperabile, che integri:

- **Localizzazione in tempo reale** basata su tecnologie indoor.

- **Algoritmi di ottimizzazione per percorsi di evacuazione** personalizzati, basati su modelli topologici e dati dinamici.

- **Diffusione di istruzioni personalizzate** a bassa latenza, utilizzando protocolli fault-tolerant.
- **Architettura a microservizi** per garantire scalabilità e robustezza.

La piattaforma proposta colma così il divario tra i sistemi d'allerta macro-geografici e le necessità di evacuazione indoor, soddisfacendo i requisiti critici del Campus di Cesena e costituendo un modello replicabile in contesti analoghi.

Capitolo 2

Architettura del sistema

Nel presente capitolo si analizza in dettaglio l'architettura a microservizi adottata per il sistema di gestione delle emergenze, scelta e sviluppata per rispondere alle problematiche descritte nel Capitolo 1. L'obiettivo è fornire una visione chiara e strutturata delle scelte progettuali adottate e delle soluzioni implementate, con particolare attenzione ai parametri di scalabilità, manutenibilità e interoperabilità del sistema.

La trattazione si apre con un'analisi comparativa tra l'architettura a microservizi e quella monolitica (Sezione 2.1), allo scopo di giustificare la scelta effettuata in funzione dei requisiti specifici del progetto. Si prosegue con una panoramica dell'architettura proposta (Sezione 2.2), in cui vengono delineati i principali componenti del sistema e il flusso informativo tra essi. Infine, la Sezione 2.3 approfondisce le caratteristiche dei singoli microservizi, analizzandone le responsabilità, le interfacce esposte e il ruolo che ciascuno riveste all'interno dell'intero ecosistema software.

2.1 Scelta dell'architettura a microservizi

2.1.1 Architettura a microservizi

Il paradigma a microservizi definisce uno stile architetturale che struttura un'applicazione come un insieme di servizi autonomi e indipendentemen-

te deployabili[1], orientati a singole funzionalità di dominio, che comunicano mediante protocolli leggeri (quali HTTP/REST) e mantengono la proprietà esclusiva dei propri dati[12, 13].
Le caratteristiche cardine comprendono:

- **Loose coupling** (accoppiamento debole): le dipendenze tra servizi sono ridotte al minimo indispensabile, in maniera tale che variazioni in un servizio non impongano modifiche negli altri.

- **High cohesion** (alta coesione): ogni servizio racchiude funzionalità strettamente correlate allo stesso dominio o bounded context[2].

- **Decentralizzazione** della governance e del data management.

- **Possibilità di scalare** e rilasciare ogni servizio in modo isolato.

Dal punto di vista formale, un microservizio è un processo autonomo che implementa un bounded context ben definito[14]. Esso espone un'interfaccia contrattuale stabile (tipicamente REST), possiede un modello di dati privato e non condiviso, e può essere sviluppato, testato, distribuito e scalato indipendentemente dagli altri servizi. Pertanto, l'unità di deployment coincide con l'unità di responsabilità funzionale: secondo il principio *"one team, one service, one database"*[13], ogni team controlla in autonomia l'intero ciclo di vita del proprio microservizio e del relativo database, riducendo al minimo le dipendenze organizzative e tecniche.

[1] Un servizio è indipendentemente deployabile quando può essere rilasciato in produzione senza richiedere la ricostruzione o la riconfigurazione degli altri componenti del sistema.

[2] Nel Domain-Driven Design, un bounded context è un confine semantico entro il quale un modello di dominio ha validità e coerenza; al suo interno termini e regole hanno significato univoco, mentre all'esterno possono divergere.

2.1.2 Analisi delle alternative: l'architettura monolitica

Un'architettura monolitica è un modello tradizionale di sviluppo software in cui l'intero applicativo è costruito e distribuito come un unico artefatto eseguibile. In tale approccio, tutti i componenti (interfaccia utente, logica di dominio, persistenza dati) risultano strettamente integrati e condividenti le medesime risorse e dipendenze [15, 16]. Da un punto di vista organizzativo, il monolite adotta un paradigma di *single code-base*: qualsiasi evoluzione funzionale, seppur circoscritta, richiede la ricompilazione e il redeploy dell'intera applicazione, comportando cicli di rilascio più lunghi e un workflow di sviluppo centralizzato [17].

L'adozione di un unico artefatto comporta indubbi benefici in termini di semplicità gestionale e coerenza operativa, ma introduce vincoli significativi sulla scalabilità, la resilienza e l'evoluzione tecnologica dell'applicazione.

Di seguito, in forma schematica, si riportano i principali punti di forza e le relative criticità del modello architetturale monolitico.

Punti di forza:

- **Semplicità di avvio progetto**: una base di codice unica semplifica l'avvio del progetto e consente al team di configurare rapidamente pipeline di build, test e deploy [18].

- **Debug e testing centralizzati**: l'intera logica di esecuzione risiede in un solo processo, agevolando il tracciamento end-to-end delle transazioni e l'analisi dei malfunzionamenti [15].

- **Deployment semplificato**: la distribuzione avviene con un singolo comando di rilascio ("one-shot deploy"[3]) senza necessità di orchestrazione tra componenti [19].

[3]Un unico artefatto contenente l'intero stack applicativo.

- **Prestazioni intraprocesso**: l'assenza di chiamate di rete interne elimina overhead di serializzazione e latenza, aumentando il throughput in scenari I/O bound [15].

Criticità:

- **Scalabilità orizzontale limitata**: per gestire un aumento di carico è necessario replicare l'intero artefatto, con conseguente inefficienza nell'utilizzo delle risorse [20].

- **Forte accoppiamento**: una modifica locale impone la ricompilazione e il redeploy dell'intera applicazione, rallentando i cicli di rilascio [16].

- ***Single point of failure*** (SPoF): un guasto in qualunque componente può compromettere l'intero sistema, riducendone la resilienza [20].

- **Technology lock-in**: l'adozione di nuove tecnologie o linguaggi è vincolata allo stack esistente, aumentando la rigidità e ostacolando l'evoluzione architetturale [21].

La Tabella 2.1 evidenzia come l'architettura monolitica privilegi la semplicità iniziale a scapito di flessibilità, scalabilità e resilienza. Al contrario, i microservizi offrono un paradigma di decomposizione verticale, con servizi autonomi e indipendenti, sebbene introducano complessità operativa aggiuntiva (orchestrazione, osservabilità, gestione del traffico) e latenza di rete nelle comunicazioni inter-servizio[18]. La scelta architetturale deve pertanto bilanciare i requisiti di tempi di deploy, scalabilità attesa e tolleranza ai guasti nell'ambiente di esecuzione.

Dimensione	Monolite	Microservizi
Unità di deployment	Singolo artefatto eseguibile	Servizi autonomi, indipendentemente deployabili
Scalabilità	Replica dell'intero stack	Scale-out mirato per ogni servizio
Accoppiamento / Coesione	Forte accoppiamento, coesione eterogenea	Loose coupling, high cohesion per bounded context
Resilienza	SPoF intrinseco	Fault isolation e degrado parziale
Governance dati	Database condiviso	Database per servizio, responsabilità locale

Tabella 2.1: Confronto tra architettura monolitica e microservizi.

2.1.3 Motivazioni della scelta

L'adozione di un'architettura a microservizi per il sistema di gestione delle emergenze è stata guidata da una valutazione approfondita dei requisiti individuati nel Capitolo 1 e dalle limitazioni riscontrate nei modelli monolitici. In particolare, quattro fattori critici ne hanno determinato la preferenza:

1. **Modularità e indipendenza**: grazie al bounded context e al principio di *single responsibility*, ogni microservizio incapsula un sottoinsieme coerente di funzionalità, riducendo la complessità cognitiva e permettendo cicli di sviluppo, test e rilascio autonomi per ciascun modulo. Ciò agevola l'applicazione del paradigma *"one team, one service"* e facilita la localizzazione e la correzione di eventuali malfunzionamenti senza impattare sull'intero sistema[13, 14].

2. **Flessibilità tecnologica**: l'architettura a microservizi supporta un approccio *polyglot*, consentendo di selezionare il linguaggio di program-

mazione, il framework e il sistema di persistenza dati più idonei per le specifiche esigenze di ogni servizio. Questa libertà tecnologica riduce il technology lock-in e favorisce l'adozione di innovazioni qualitativamente rilevanti[13, 22].

3. **Scalabilità orizzontale**: poiché ciascun microservizio implementa un dominio funzionale autonomo, è possibile scalare in modo fine-grained soltanto i componenti sottoposti a maggior carico, ottimizzando l'utilizzo delle risorse e riducendo i costi infrastrutturali. Tale modello si adatta particolarmente ai picchi di accesso tipici dei sistemi di allerta in situazioni emergenziali[15, 22].

4. **Resilienza e fault tolerance**: l'isolamento processuale dei microservizi favorisce il contenimento dei guasti, evitando che un malfunzionamento locale si propaghi all'intero sistema.

In sintesi, l'architettura a microservizi si allinea in modo ottimale con i requisiti di modularità, flessibilità, scalabilità e tolleranza ai guasti definiti per il sistema, superando le limitazioni intrinseche del modello monolitico e ponendo le basi per una soluzione evolutiva e manutenibile nel lungo periodo.

2.2 Panoramica dell'architettura proposta

Lo scopo di questa sezione è fornire una visione d'insieme delle principali entità software e dei flussi informativi che le collegano, evidenziando come l'orchestrazione distribuita e la separazione dei contesti di responsabilità permettano di soddisfare i requisiti di scalabilità, resilienza e manutenibilità illustrati nel Capitolo 1.

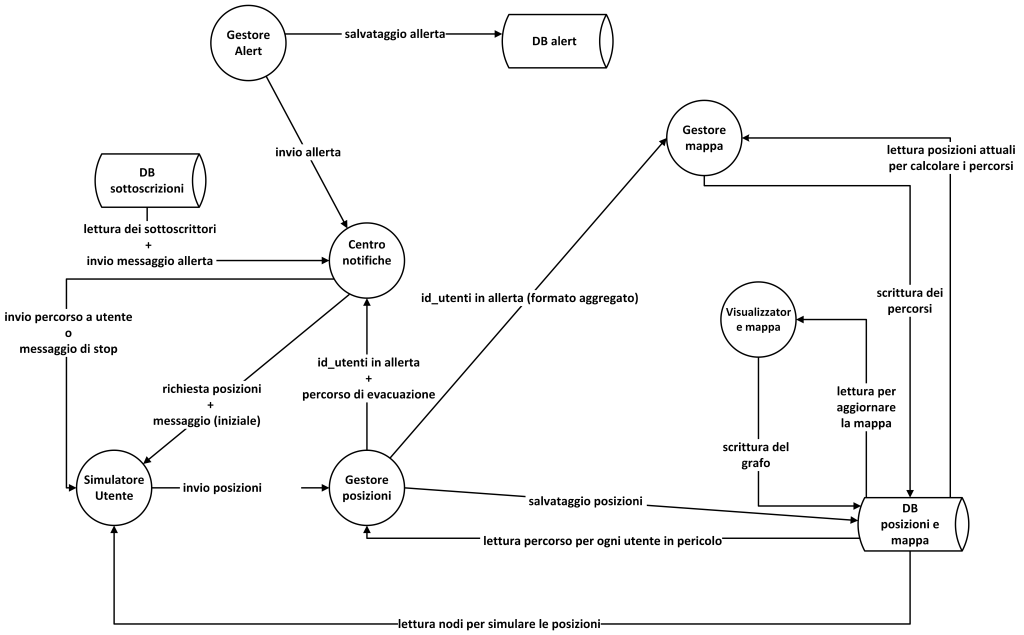

Figura 2.1: Architettura del sistema

Il grafo in Figura 2.1 rappresenta ciascun microservizio come un nodo e utilizza archi orientati per indicare i percorsi di comunicazione e i messaggi scambiati. Tale formalismo semplifica l'individuazione dei bounded context in cui ogni servizio opera, rendendo evidente il confine tra le responsabilità funzionali. La distinzione tra flussi sincroni e asincroni è immediata, agevolando la scelta del paradigma di comunicazione più adatto tra chiamate REST e sistemi di messaging in base ai requisiti di latenza e throughput. Inoltre, la topologia di rete così delineata facilita l'identificazione di eventuali cluster critici e possibili punti unici di guasto, supportando strategie di fault isolation e load balancing. Dal punto di vista operativo, il team di sviluppo può pianificare in modo accurato il dimensionamento delle risorse e l'auto-scaling dei singoli servizi. Inoltre, durante le attività di debug e audit, il grafo assume il ruolo di *"single source of truth"*, consentendo di localizzare rapidamente i servizi coinvolti in eventuali malfunzionamenti e di verificare la compliance dei flussi informativi. Infine, l'estensione futura dell'ecosistema si traduce in un semplice aggiornamento del grafo, che evidenzia

immediatamente l'impatto sulle connessioni esistenti e previene l'insorgere di dipendenze indesiderate.

Nei paragrafi successivi verranno esaminati nel dettaglio ciascuno di questi componenti e il ruolo che essi svolgono all'interno dell'ecosistema complessivo.

2.3 Design dei microservizi

In questa sezione vengono presentati, in ordine di flusso operativo, i microservizi che costituiscono l'architettura del sistema. Per ciascun servizio verranno descritti il contesto di responsabilità, le funzionalità principali, l'interfaccia di comunicazione e i requisiti di performance e resilienza soddisfatti. L'analisi mira a mettere in evidenza come la scomposizione in servizi autonomi risponda alle esigenze di modulabilità, scalabilità e manutenibilità delineate nei capitoli precedenti.

Per una panoramica dei flussi di messaggistica e delle modalità di interazione tra microservizi, si rimanda alla Sezione 2.4. Le specifiche tecnologiche e i dettagli di implementazione saranno invece illustrati nei Capitoli 4, 5 e 6.

2.3.1 Gestore degli alert

Il microservizio Gestore degli alert (*Alert Manager*) riveste un ruolo centrale nella pipeline di gestione delle emergenze all'interno del sistema. Infatti, si occupa della raccolta, elaborazione, filtraggio e archiviazione degli allarmi CAP in ingresso, garantendone un trattamento accurato e conforme ai criteri definiti. La sua funzione principale consiste nell'assicurare una elaborazione efficace e precisa delle allerte, supportando il sistema nel monitoraggio e nella supervisione degli eventi critici.
Il microservizio interagisce con altri componenti del sistema per l'invio delle informazioni relative agli alert, contribuendo alla distribuzione tempestiva delle segnalazioni di allerta agli utenti.

2.3.2 Centro notifiche

Il microservizio Centro notifiche (*Notification Center*) funge da hub centrale per la gestione e la distribuzione di notifiche all'interno dell'architettura a microservizi del sistema. La sua funzione primaria consiste nell'assicurare la disseminazione tempestiva e accurata di informazioni critiche relative a eventi di emergenza. In particolare, il servizio è responsabile di:

- **Diffusione degli alert**: gestione della trasmissione di allerte e aggiornamenti in tempo reale, indirizzati ai destinatari pertinenti in base a criteri di rilevanza e urgenza.
- **Comunicazione tra microservizi**: interfacciamento con gli altri microservizi del sistema, al fine di aggregare e distribuire informazioni contestuali relative all'evento di emergenza.
- **Gestione del ciclo di vita delle notifiche**: supervisione dell'intero ciclo di vita delle notifiche di allerta, dalla generazione alla terminazione, garantendo una comunicazione continua e puntuale fino alla risoluzione dell'evento.

L'implementazione delle funzionalità sopra descritte richiede l'adozione di protocolli di comunicazione efficienti e strategie di notifica ottimali, al fine di minimizzare la latenza e massimizzare l'affidabilità della trasmissione. Tali aspetti tecnici, inclusi i dettagli relativi ai protocolli di invio e alle strategie di notificazione degli utenti, saranno oggetto di analisi approfondita nel Capitolo 4.

2.3.3 Simulatore delle posizioni

Il microservizio Simulatore delle posizioni (*User Simulator*) implementa la generazione di dati di localizzazione sintetici con granularità configurabile, con l'obiettivo di emulare la presenza, il movimento e la distribuzione spaziale di utenti all'interno di un ambiente definito. Tale funzionalità si rivela

imprescindibile nelle fasi di test e validazione, in cui l'accesso a dati di tracciamento reali risulta spesso impraticabile per motivi logistici, economici o etici.

La disponibilità di stream di posizioni realistici è determinante per:

- Verificare la capacità del *Position Manager* di individuare e classificare, in tempo quasi reale, gli utenti esposti al rischio.
- Misurare la reattività del *Map Manager* nel ricalcolo dinamico dei percorsi di esodo.
- Collaudare la robustezza del *Notification Center* nella distribuzione massiva di istruzioni geolocalizzate.

Lo *User Simulator* consente di modellare scenari eterogenei di affollamento, modulare i tassi di mobilità e i pattern di fuga, generando così carichi controllati sull'intera pipeline. Pur configurandosi come componente placeholder, destinato a essere sostituito da un sistema di *real-time indoor positioning*, il servizio riveste un ruolo strategico nell'emulazione di emergenze: consente di orchestrare esercitazioni virtuali, analizzare le metriche di latenza end-to-end e calibrare le soglie di attivazione delle politiche di auto-scaling, mitigando il rischio di regressioni prima del dispiegamento in ambiente operativo.

2.3.4 Gestore delle posizioni

Il microservizio Gestore delle posizioni (*Position Manager*) si configura come il componente fondamentale per la gestione e l'elaborazione dei dati geospaziali nell'infrastruttura complessiva del sistema. Esso ha la responsabilità di raccogliere, analizzare e monitorare le posizioni degli utenti all'interno dell'edificio, al fine di supportare in maniera tempestiva le decisioni in contesti di emergenza.

Le principali funzionalità del *Position Manager* si esplicano nei seguenti ambiti:

- **Analisi spaziale**: il microservizio esegue un'analisi dettagliata delle posizioni e dei movimenti degli utenti, identificando la loro distribuzione spaziale all'interno dell'edificio. Questa capacità di monitoraggio in tempo reale è cruciale per l'adozione di misure preventive, come la gestione delle aree di congestione o la segnalazione di zone a rischio elevato.

- **Valutazione dinamica del rischio**: attraverso l'applicazione di modelli predittivi e regole predefinite, il *Position Manager* è in grado di classificare le situazioni di pericolo in relazione alla posizione degli utenti. Tale valutazione consente di assegnare un livello di criticità ad ogni area, garantendo una risposta rapida ed efficace in scenari di emergenza.

- **Supporto alle operazioni di intervento**: una volta identificati i rischi, il *Position Manager* attiva risposte operative mirate, fornendo gli strumenti necessari per la gestione delle emergenze. Dopo aver verificato che non vi siano più utenti in situazioni di pericolo, il *Position Manager* è responsabile della comunicazione di cessazione dell'allerta, garantendo la conclusione del processo di evacuazione o gestione dell'emergenza.

In stretta collaborazione con il *Map Manager* e il *Notification Center*, il *Position Manager* gioca un ruolo centrale nell'assicurare che i dati relativi alle posizioni siano costantemente aggiornati e coerenti, migliorando l'efficacia complessiva delle operazioni di evacuazione e intervento.

2.3.5 Gestore della mappa

Il microservizio Gestore della mappa (*Map Manager*) è il componente specializzato nella computazione distribuita di percorsi di evacuazione ottimizzati in tempo reale per rispondere a situazioni di emergenza. Esso integra in modo fluido il sistema di gestione delle posizioni con la rete infrastrutturale dell'edificio, elaborando informazioni geospaziali e aggiornamenti provenienti

dalle fonti di localizzazione degli utenti per garantire un flusso continuo di dati durante le fasi dell'evacuazione. Tale approccio permette al *Map Manager* di determinare dinamicamente i percorsi di evacuazione più sicuri e rapidi, adattandosi continuamente alle condizioni mutevoli dell'edificio e alle necessità operative.

A fronte di una segnalazione di emergenza, il modulo determina i percorsi personalizzati mediante l'integrazione con un sistema di gestione delle posizioni, processando lo stato attuale della rete infrastrutturale e la localizzazione degli utenti coinvolti. Il servizio adotta un approccio data-driven: integra feed di posizionamento in tempo reale con un grafo navigabile che modella la rete di evacuazione, applicando algoritmi di pathfinding con vincoli dinamici. Tali vincoli includono metriche di capacità residua delle vie di fuga e segnalazioni crowdsourced, che inducono una riconfigurazione adattiva della topologia di evacuazione.

Le funzioni principali del *Map Manager* comprendono:

- **Calcolo dinamico dei percorsi**: il microservizio analizza costantemente lo stato della rete di evacuazione, calcolando i percorsi migliori per l'esodo degli utenti, in base alla posizione attuale e alle condizioni di rischio. Tali percorsi sono adattati in tempo reale per riflettere le modifiche nei vincoli, come ostruzioni, alterazioni nei flussi di persone o variazioni nella capacità di evacuazione.

- **Integrazione con i dati di posizionamento**: il *Map Manager* riceve in tempo reale i dati di localizzazione degli utenti e li utilizza per aggiornare il modello della mappa, ottimizzando i percorsi di evacuazione in funzione della distribuzione spaziale degli stessi. Le informazioni relative alla capacità delle vie di fuga e agli aggiornamenti degli utenti sono determinanti per ricalcolare rapidamente le rotte più sicure.

- **Adattamento in base a vincoli dinamici**: i percorsi di evacuazione sono costantemente aggiornati tenendo conto di molteplici fattori, quali l'affollamento delle zone, i segnali di emergenza e gli aggiornamenti

provenienti da altre fonti, come i dati crowdsourced. Questo approccio adattivo consente di ottenere un modello di evacuazione flessibile e reattivo alle mutate condizioni di emergenza.

Pertanto, il *Map Manager* rappresenta un elemento cruciale nella gestione delle risorse e nella pianificazione degli interventi di emergenza, poiché integra i dati provenienti dal *Position Manager* con le informazioni sulla rete infrastrutturale dell'edificio. Contribuisce in modo determinante alla tempestività e all'efficacia delle operazioni di evacuazione, ottimizzando l'allocazione delle risorse e garantendo la sicurezza degli utenti in tutte le fasi dell'emergenza.

I dettagli implementativi, incluse le strategie di ottimizzazione adottate e le modalità di interazione con i servizi dipendenti, verranno approfonditi nel Capitolo 6.

2.3.6 Visualizzatore della mappa

Il microservizio Visualizzatore della mappa (*Map Viewer*) è un componente software avanzato, progettato per la visualizzazione interattiva e dinamica delle posizioni degli utenti e dei percorsi di evacuazione all'interno di un edificio. Il suo scopo principale risiede nel fornire una rappresentazione visiva delle informazioni critiche, supportando gli operatori nel monitoraggio continuo e nella gestione delle operazioni di evacuazione in tempo reale.

Le principali funzionalità del microservizio comprendono:

- **Recupero e visualizzazione dei dati di localizzazione degli utenti**: il *Map Viewer* acquisisce e visualizza in tempo reale le posizioni degli utenti, mostrandole su una mappa digitale dell'edificio. Il microservizio ottiene i dati di localizzazione degli utenti da una fonte dati esterna, li elabora e li utilizza per la generazione di una rappresentazione visiva delle posizioni degli utenti sulla mappa. Tale funzionalità consente di monitorare la distribuzione spaziale degli utenti, facilitando l'indi-

viduazione tempestiva di aree congestionate o a rischio, cruciali per la gestione efficace dell'emergenza.

- **Aggiornamento dinamico**: il *Map Viewer* è progettato per aggiornare costantemente la visualizzazione della mappa dell'edificio, sulla base delle variazioni nei dati di posizionamento degli utenti e nelle condizioni dei percorsi di evacuazione. Tale funzionalità garantisce una sincronizzazione accurata e immediata della mappa con l'aggiornamento delle posizioni degli utenti; ciò assicura che gli operatori abbiano sempre una visione chiara, precisa e aggiornata della situazione in tempo reale.

- **Interazione con la mappa**: gli operatori possono interagire in modo intuitivo con la mappa, aggiungendo e modificando i nodi e gli archi attraverso un'interfaccia grafica user-friendly. Tale funzionalità consente di tracciare la planimetria dell'edificio in fase di configurazione iniziale.

In sintesi, il *Map Viewer* fornisce un supporto decisivo per gli operatori di emergenza e per gli amministratori di sistema, offrendo loro una comprensione immediata e intuitiva della distribuzione spaziale degli utenti, dei percorsi di evacuazione calcolati e delle zone interessate dagli eventi critici. Tale funzionalità è determinante per garantire una gestione tempestiva ed efficiente dell'emergenza, favorendo la sicurezza degli utenti e l'efficacia delle operazioni di evacuazione.

I dettagli implementativi relativi alle tecnologie impiegate per la visualizzazione della mappa e alle strategie di gestione dei dati di localizzazione saranno trattati in maniera esaustiva nel Capitolo 5, dove verranno esplorati i principi progettuali e le soluzioni tecniche adottate per realizzare il microservizio.

2.4 Comunicazione tra microservizi: flusso dell'emergenza

Il sistema, strutturato secondo un'architettura distribuita a microservizi, adotta un meccanismo sofisticato per orchestrare l'interazione tra i microservizi nella gestione delle situazioni di emergenza. Fondato su un approccio che integra la propagazione di eventi e la condivisione dello stato, il flusso operativo si sviluppa attraverso una serie di scambi sincronizzati e coordinati tra i componenti. Ogni microservizio opera autonomamente, con responsabilità ben definite, assicurando così un'architettura modulare ed efficiente, che favorisce una gestione scalabile e flessibile del processo emergenziale.

2.4.1 Fase 0: configurazione del sistema

La fase di configurazione iniziale del sistema di gestione delle emergenze riveste un ruolo fondamentale, pur non facendo parte del ciclo operativo continuo. Questa fase preliminare è cruciale per predisporre l'ambiente e i dati necessari al corretto funzionamento di tutti i componenti del sistema. In particolare, si concentra sulla definizione delle informazioni relative alla struttura dell'edificio, che sono imprescindibili per la simulazione delle posizioni degli utenti e per il calcolo dei percorsi di evacuazione. La configurazione del sistema si basa sulla creazione di una rappresentazione astratta dell'edificio sotto forma di un grafo, composto da nodi e archi. I nodi corrispondono alle stanze, classificate in base alla loro tipologia, mentre gli archi definiscono i collegamenti tra di esse. Per effettuare questa operazione, viene utilizzata un'interfaccia dedicata, che consente l'inserimento e la gestione di tali elementi. La rappresentazione grafica dell'edificio è il fulcro su cui si fondano diverse funzionalità del sistema. Nello specifico, il *Map Manager* si avvale di tale mappa per determinare i percorsi di evacuazione ottimali, lo *User Simulator* per simulare la presenza e il movimento degli utenti all'interno della struttura, e il *Position Manager* per registrare le posizioni degli utenti nel database e valutare eventuali situazioni di pericolo in base alla natura

dell'emergenza in corso.
Il diagramma di sequenza mostrato in Figura 2.2 illustra in dettaglio le interazioni principali di questa fase preparatoria, con particolare attenzione al ruolo del *Map Viewer* e al sistema di archiviazione dei dati relativi alla mappa. Infatti, il microservizio *Map Viewer* si occupa della generazione della mappa e della rete infrastrutturale dell'edificio, caricando e strutturando i dati in un formato utilizzabile durante le fasi di gestione dell'emergenza.

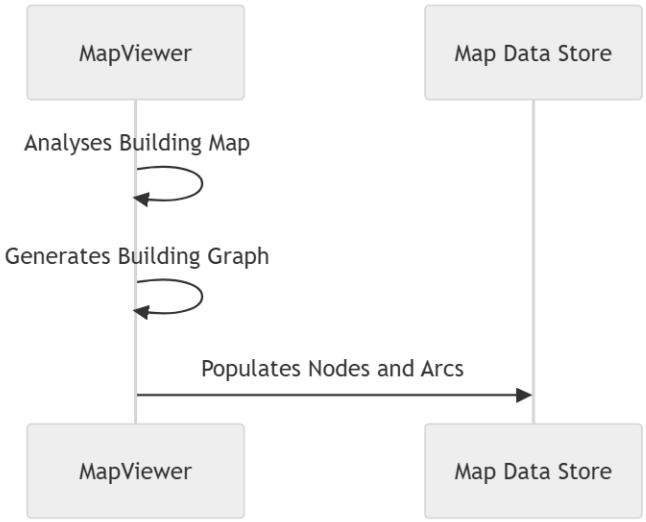

Figura 2.2: Diagramma di sequenza: configurazione iniziale del sistema

2.4.2 Fase 1: gestione dell'allerta

Il processo di gestione dell'emergenza ha inizio con l'*Alert Manager*, che genera e valida un potenziale evento critico, producendo un messaggio di allerta conforme al formato CAP. Il messaggio viene sottoposto a filtri di rilevanza basati su regole specifiche del dominio per valutarne la rilevanza. Se ritenuto significativo, l'evento viene archiviato nel database dedicato e l'*Alert Manager* trasmette al *Notification Center* l'identificativo univoco dell'alert e i relativi metadati, avviando ufficialmente il protocollo di emergenza. Ricevuta la segnalazione, il *Notification Center* accede al database delle sottoscrizioni per recuperare l'elenco aggiornato degli utenti coinvolti e invia le notifiche

iniziali tramite i canali preconfigurati. Contemporaneamente, richiede allo *User Simulator* i dati aggiornati sulle posizioni degli utenti, attivando così l'analisi geospaziale dell'evento. La sequenza delle interazioni tra i principali microservizi durante questa fase iniziale è illustrata nel *sequence diagram* riportato nella Figura 2.3.

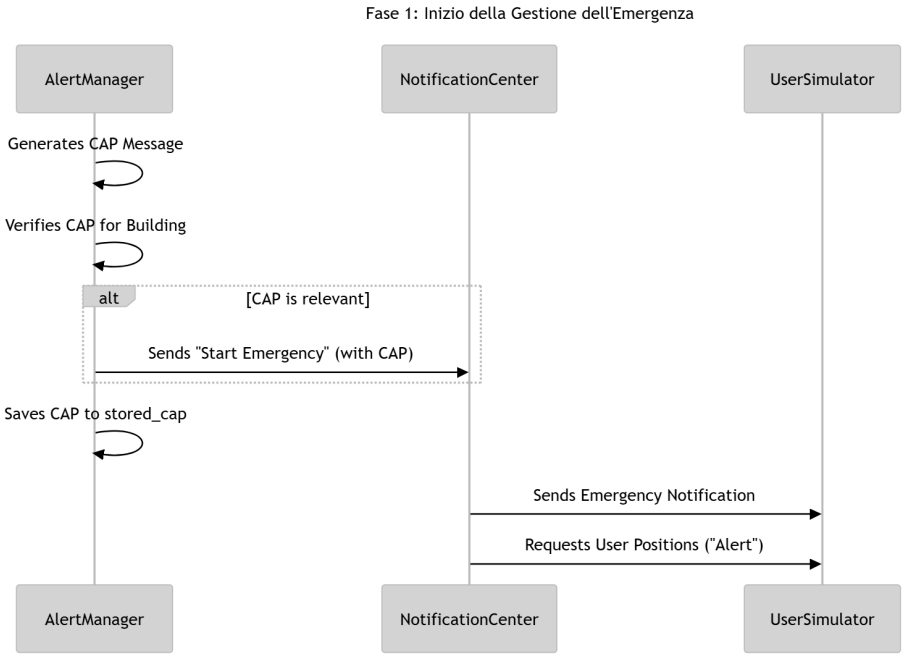

Figura 2.3: Diagramma di sequenza: inizio dell'emergenza

2.4.3 Fase 2: gestione delle posizioni e rilevamento del pericolo

Durante questa fase, lo *User Simulator* trasmette al *Position Manager* i dettagli relativi alla collocazione di ciascun utente e alla tipologia di evento in corso. Il *Position Manager*, una volta acquisite le informazioni di localizzazione, effettua un controllo topologico delle coordinate per determinare quali utenti siano potenzialmente a rischio. I risultati di tale analisi vengono condivisi simultaneamente con due moduli: il *Notification Center* riceve l'identificativo univoco degli utenti in pericolo insieme al percorso di evacua-

zione associato, mentre il *Map Manager* ottiene una panoramica aggregata basata sui nodi topologici. Qualora non vengano rilevati utenti a rischio, il *Position Manager* invia un segnale di "Stop" al *Notification Center*, indicando la conclusione del processo di gestione dell'emergenza. Le interazioni descritte sono rappresentate in dettaglio nel diagramma di sequenza riportato nella Figura 2.4.

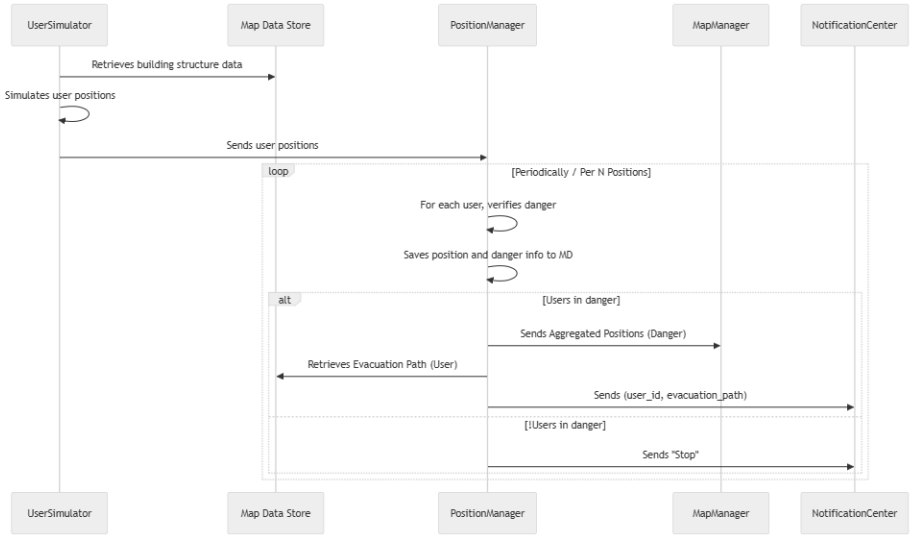

Figura 2.4: Diagramma di sequenza: gestione delle posizioni e rilevamento del pericolo

2.4.4 Fase 3: risposta alle notifiche di evacuazione e aggiornamento delle posizioni simulate

A questo punto della gestione del flusso emergenziale, il *Notification Center* trasmette allo *User Simulator* i messaggi ottenuti dal *Position Manager*. In base al contenuto del messaggio, che può indicare un percorso di evacuazione o un segnale di "Stop", lo *User Simulator* gestisce diversamente la simulazione: segue il percorso indicato per simulare il movimento degli utenti o termina l'elaborazione. Se la simulazione dell'emergenza continua, lo *User Simulator* comunica al *Position Manager* le posizioni aggiornate degli

utenti. Questo processo di interazione è dettagliatamente rappresentato nel diagramma di sequenza della Figura 2.5, che evidenzia chiaramente le diverse reazioni dello *User Simulator* in funzione del tipo di messaggio ricevuto.

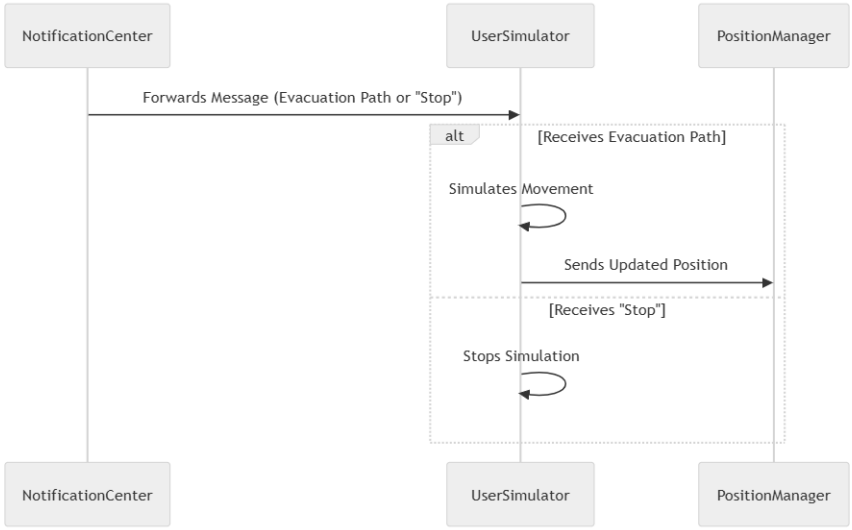

Figura 2.5: Diagramma di sequenza: risposta alle notifiche di evacuazione e aggiornamento delle posizioni

2.4.5 Fase 4: aggiornamento della mappa e ricalcolo dei percorsi

L'aggiornamento della visualizzazione del grafo astratto dell'edificio avviene tramite l'uso di trigger temporali, i quali vengono attivati dal *Position Manager* in base alla frequenza degli aggiornamenti delle posizioni degli utenti. Quando tali trigger si attivano, il *Map Viewer* acquisisce le informazioni più recenti riguardanti le posizioni degli utenti e aggiorna dinamicamente la visualizzazione del sistema. Questo garantisce che la mappa rifletta sempre lo stato attuale delle localizzazioni all'interno dell'edificio. Contemporaneamente, il *Map Manager*, una volta ricevuti i dati dal *Position Manager*, avvia il calcolo dei percorsi di evacuazione ottimizzati. Questo processo considera diversi fattori, come la capacità residua dei percorsi, eventuali ostacoli o

interruzioni, e i vincoli strutturali presenti nell'ambiente. I percorsi generati vengono successivamente memorizzati nel database condiviso delle posizioni e della mappa.

Il sistema di aggiornamento è ciclico: ogni intervallo temporale determinato dal trigger del *Position Manager* avvia un processo che aggiorna sia la visualizzazione della mappa che i percorsi di evacuazione, garantendo una visione in tempo reale della situazione e facilitando una gestione continua ed efficace dell'emergenza.

Il diagramma di sequenza mostrato nella figura 2.6 illustra il flusso di comunicazione tra i vari moduli in questa fase, evidenziando come l'aggiornamento della mappa e il ricalcolo dei percorsi siano ripetuti durante tutta la gestione dell'emergenza.

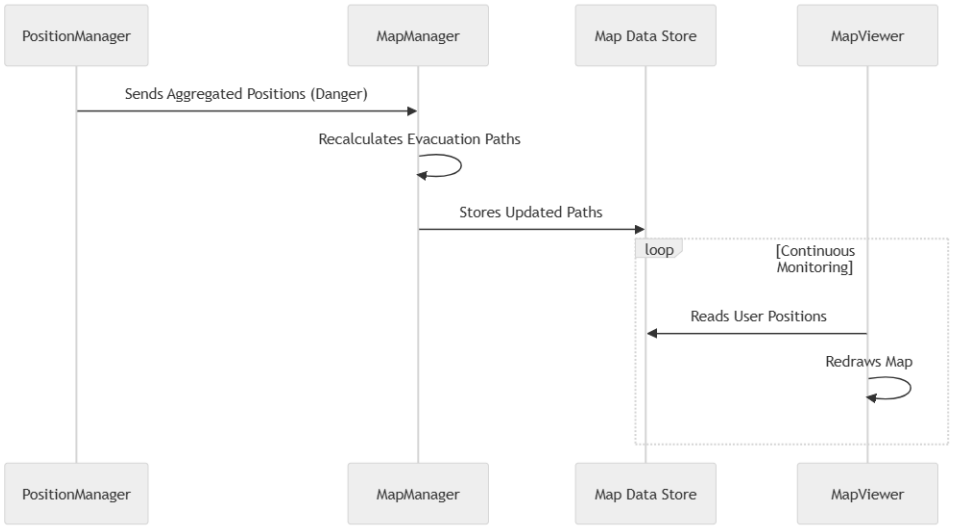

Figura 2.6: Diagramma di sequenza: aggiornamento della mappa e ricalcolo dei percorsi

2.4.6 Fase 5: riassegnamento dei percorsi

Quando il *Position Manager* riceve la posizione di un utente, recupera il percorso di evacuazione associato a quella specifica localizzazione. Tuttavia, i percorsi di evacuazione non sono definitivi, ma vengono costantemente

aggiornati dal *Map Manager*, che si occupa di ricalcolare e ottimizzare le rotte in tempo reale, tenendo conto delle norme di evacuazione e delle aree eventualmente inaccessibili. Pertanto, il *Position Manager* attinge ai percorsi ricalcolati dal *Map Manager* e riassegna agli utenti i nuovi percorsi, garantendo che ogni individuo sia indirizzato verso la via di fuga più sicura e ottimale. Una volta completata questa operazione, le istruzioni per l'evacuazione vengono inviate al *Notification Center*, che provvede alla distribuzione finale delle informazioni agli utenti interessati.

La Figura 2.7 illustra chiaramente come avviene il processo di aggiornamento dinamico dei percorsi e il successivo riassegnamento degli stessi.

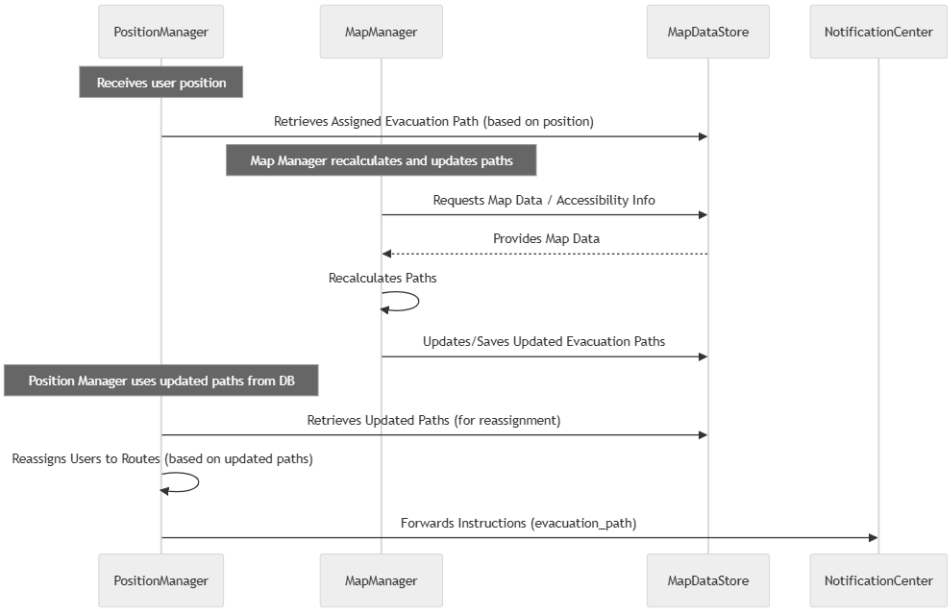

Figura 2.7: Diagramma di sequenza: aggiornamento e riassegnamento dei percorsi di evacuazione

Capitolo 3

Tecnologie fondamentali del sistema

Il presente capitolo illustra le scelte tecnologiche imprescindibili a fondamento dell'architettura del sistema di gestione delle emergenze, selezionate in coerenza con i requisiti funzionali e non funzionali delineati nel Capitolo 1. Verranno analizzati i principali elementi del framework tecnologico, dal linguaggio di programmazione all'infrastruttura di comunicazione asincrona, fino alla piattaforma di persistenza dati e ai formati di configurazione, evidenziando, per ciascuno, il contributo in termini di affidabilità, scalabilità e adattabilità ai vincoli operativi degli scenari emergenziali.

3.1 Linguaggio di programmazione: Python

Il linguaggio di programmazione costituisce il nucleo dello sviluppo dei microservizi del sistema. La scelta di Python è stata guidata dai requisiti del sistema (Capitolo 1) e dalla necessità di integrazione con l'ecosistema tecnologico adottato.

Dopo un'analisi comparativa con Node.js, Python è risultato la scelta ottimale per la sua versatilità e robustezza.

3.1.1 Requisiti del linguaggio di programmazione

I requisiti, derivati dai vincoli del Capitolo 1 e dalle esigenze dei microservizi (Sezione 2.3), includono:

- **Analisi e calcolo avanzati**: supporto per modellazione di grafi e calcoli spaziali per ottimizzare rotte di evacuazione.
- **Comunicazione asincrona**: gestione di interazioni concorrenti con RabbitMQ per notifiche in tempo reale.
- **Produttività e manutenibilità**: sintassi chiara e documentazione estesa per sviluppo rapido e manutenzione semplice.
- **Interoperabilità**: integrazione con Docker, API (REST, gRPC) e PostgreSQL/PostGIS.

3.1.2 Python

La selezione di Python come linguaggio di riferimento per i microservizi si fonda sui requisiti specifici del sistema e sulla ricchezza del suo ecosistema di librerie. In particolare, esso soddisfa:

- **Analisi e calcolo avanzati**: librerie mature come `NetworkX` consentono di modellare la mappa dell'edificio come un grafo e di calcolare percorsi minimi (*shortest path*) in modo efficiente. Queste funzionalità sono centrali per l'ottimizzazione delle rotte di evacuazione senza dover implementare *ex novo* complessi algoritmi di pathfinding [23]. Analogamente, l'ecosistema Python include librerie GIS (es. `GeoPandas`, `Shapely`) e il driver `psycopg2` per PostgreSQL, che insieme a PostGIS permettono di eseguire query spaziali e manipolare geodati direttamente da Python [24, 25, 26]. Tale ricchezza di strumenti scientifici e di calcolo non ha equivalenti altrettanto consolidati in altri ecosistemi, rendendo Python più adatto per microservizi orientati all'analisi dei dati e delle reti complesse [27].

- **Comunicazione asincrona ed event-driven**: grazie al modulo asincrono `asyncio` e al client AMQP `pika`, Python gestisce interazioni concorrenti e I/O non bloccante con RabbitMQ [28, 29], broker di messaggistica selezionato per la comunicazione inter-microservizi (cfr. Sezione 3.2), garantendo integrazione semplice con il sistema di messaggistica [30]. Ciò consente al sistema di ricevere e propagare notifiche d'allerta con bassa latenza. In questo modo, Python unisce la potenza del suo stack scientifico con la reattività tipica di sistemi real-time.

- **Produttività e manutenibilità**: la sintassi chiara e concisa di Python, unita a una comunità estesa e a una vasta collezione di librerie open-source, accelera lo sviluppo e facilita la scrittura di codice manutenibile [27]. La presenza di framework maturi per il testing e di documentazione completa riduce il *time-to-market* e semplifica l'evoluzione del sistema.

- **Interoperabilità e approccio polyglot**: l'architettura a microservizi consente di scegliere il linguaggio più adatto per ogni componente senza vincoli imposti dallo stack tecnologico complessivo. In questo contesto, Python si integra nativamente con container Docker e con API esterne (REST, gRPC), facilitando la comunicazione con componenti in altri linguaggi e riducendo il lock-in tecnologico [23].

3.1.3 Analisi delle alternative: Node.js

Parallelamente a Python, si è valutata l'adozione di Node.js, un runtime JavaScript, progettato per eseguire codice lato server con un event-loop non bloccante. Tale architettura lo rende estremamente efficace in scenari *I/O-bound* e real-time, dove occorre gestire un elevato numero di connessioni concorrenti con bassa latenza[27, 31].

Nel contesto in analisi, tuttavia, il carico principale è di tipo computazionale (calcolo di percorsi ottimali, analisi di vicinanza spaziale) e *data-intensive* più che puramente I/O. In tali circostanze, il modello single-threaded

di Node.js può diventare un collo di bottiglia: operazioni CPU-intensive possono bloccare l'event loop, degradando le prestazioni globali del servizio [27].

Inoltre, benché esistano librerie Node.js per la grafica e l'analisi dei grafi, esse risultano meno consolidate e ottimizzate rispetto alle controparti Python, che beneficiano di anni di sviluppo in ambito scientifico. Ciò implicherebbe maggiore lavoro di integrazione o performance inferiori.

Per quanto concerne la disponibilità di librerie per il messaging e il data access, entrambi i linguaggi dispongono di client AMQP per RabbitMQ [32] e driver per PostgreSQL. Ciononostante, Python vanta anche ORM e toolkit specifici per la geolocalizzazione che possono velocizzare lo sviluppo di funzionalità geospaziali con poche linee di codice, mentre su Node.js molte di queste integrazioni richiedono maggior lavoro manuale o l'uso di pacchetti meno diffusi.

Dunque, la scelta di Python è confermata dal suo equilibrio tra capacità di calcolo scientifico, supporto per messaggistica e facilità di sviluppo, perfettamente allineato ai requisiti funzionali e non funzionali del sistema (Capitolo 1).

3.2 Broker di messaggistica: RabbitMQ

In un'architettura a microservizi, il broker di messaggistica rappresenta il fulcro della *control plane*: assicura la propagazione affidabile degli eventi, il disaccoppiamento temporale fra processi e la resilienza ai guasti di rete. La presente sezione motiva la scelta di RabbitMQ come broker di messaggistica, analizzando i requisiti specifici del sistema e il contributo di RabbitMQ in termini di funzionalità, prestazioni e compatibilità con l'ecosistema tecnologico adottato.

3.2.1 Requisiti del sistema di messaggistica

Il sistema in analisi richiede un'infrastruttura di messaggistica che soddisfi i seguenti requisiti, derivati dai vincoli funzionali e non funzionali delineati nel Capitolo 1:

- **Bassa latenza nella propagazione degli eventi**: i messaggi relativi agli alert di emergenza e ai percorsi di evacuazione devono essere trasmessi in tempo reale o con latenza trascurabile per garantire una risposta tempestiva in scenari critici.
- **Flussi di dati non massivi**: il sistema gestisce un volume di messaggi relativamente contenuto. La priorità è quindi la reattività rispetto alla capacità di gestire carichi estremi.
- **Supporto per comunicazioni asincrone**: l'architettura a microservizi richiede un disaccoppiamento temporale tra produttori e consumatori, con pattern di messaggistica come *publish/subscribe* e *point-to-point*, per consentire l'orchestrazione di flussi complessi come la gestione degli alert e l'aggiornamento delle posizioni.
- **Compatibilità con Python**: il broker deve integrarsi nativamente con Python, linguaggio scelto per i microservizi (cfr. Sezione 3.1), attraverso client robusti e ben documentati, come `pika` per AMQP.
- **Facilità di configurazione e manutenzione**: il broker deve essere configurabile senza complessità eccessive e supportare meccanismi di monitoraggio e fault tolerance senza richiedere infrastrutture pesanti.

Sulla base di tali requisiti, sono stati valutati RabbitMQ e Apache Kafka, due dei principali broker di messaggistica disponibili.

3.2.2 RabbitMQ

RabbitMQ è un broker di messaggistica open-source e general-purpose, progettato per integrare e disaccoppiare microservizi, offrendo comunicazioni

asincrone stabili e affidabili [33]. La sua adozione è particolarmente adatta a sistemi critici come quello proposto, dove la tempestività nella diffusione degli allarmi e la garanzia di consegna sono requisiti fondamentali (cfr. Sezione 1.1).
RabbitMQ implementa il protocollo AMQP (Advanced Message Queuing Protocol) e utilizza un modello *push-based*: i produttori inviano messaggi a un `exchange`, che li instrada a una o più code secondo regole di binding definite, e i consumatori ricevono i messaggi direttamente dalle code [34]. Questo approccio supporta pattern di messaggistica complessi, come `publish/subscribe`, essenziali per orchestrare i flussi operativi descritti nella Sezione 2.4.
Le principali caratteristiche di RabbitMQ che ne giustificano l'adozione includono:

- **Flessibilità nei pattern di messaggistica**: RabbitMQ supporta code di messaggi configurabili e meccanismi avanzati di routing tramite *exchanges*, che consentono di modellare flussi di comunicazione complessi. Questo si adatta perfettamente alla necessità del sistema di propagare alert CAP e di orchestrare aggiornamenti di posizione.

- **Bassa latenza**: RabbitMQ è ottimizzato per scenari in cui la reattività è prioritaria, con tempi di consegna dei messaggi nell'ordine dei millisecondi, ideali per la trasmissione di notifiche di emergenza in tempo reale [35].

- **Supporto nativo per Python**: il client `pika` offre un'integrazione robusta e ben documentata con Python, consentendo la gestione asincrona dei messaggi tramite `asyncio` e garantendo compatibilità con il framework tecnologico del sistema [36].

- **Supporto per JSON**: RabbitMQ consente di inviare messaggi in formato JSON, che è il formato standard per i dati scambiati tra i microservizi del sistema. Questo garantisce una serializzazione e deserializzazione efficiente, fondamentale per l'interoperabilità tra i componenti

e per rappresentare informazioni complesse come coordinate spaziali o istruzioni di emergenza.

- **Fault tolerance e affidabilità**: RabbitMQ supporta meccanismi di persistenza dei messaggi per garantire la consegna dei messaggi anche in caso di guasti parziali, soddisfacendo il requisito di resilienza del sistema [37].

- **Facilità di configurazione**: RabbitMQ è leggero da configurare e gestire, con un'interfaccia di amministrazione intuitiva e supporto per containerizzazione, rendendolo ideale per ambienti con risorse limitate [38].

Le suddette peculiarità rendono RabbitMQ particolarmente adatto a un sistema che richiede comunicazioni rapide, affidabili e flessibili in un contesto con flussi di dati non massivi.

3.2.3 Analisi delle alternative: Apache Kafka

Come alternativa a RabbitMQ si analizza Apache Kafka.
Si tratta di una piattaforma di streaming di eventi distribuita, progettata per gestire volumi di dati massivi ad alta velocità, tipicamente utilizzata in scenari di big data[39].
Differentemente da RabbitMQ, utilizza un modello *pull-based*: i produttori scrivono messaggi in `topic` partizionati e i consumatori ne effettuano il *polling*.
Sebbene potente, grazie al suo throughput elevatissimo e alla capacità di scalare orizzontalmente su più nodi [34], Kafka presenta alcune limitazioni nel contesto del progetto:

- **Orientamento a flussi massivi**: Kafka eccelle nella gestione di throughput elevati (nell'ordine di milioni di eventi al secondo), ma il sistema di implementato non richiede tale capacità. Pertanto, la complessità di configurazione e gestione di Kafka risulta sovradimensionata[40].

- **Latenza più elevata**: Kafka è ottimizzato per il throughput e la persistenza di grandi volumi di dati, ma introduce una latenza leggermente superiore rispetto a RabbitMQ in scenari a bassa intensità, a causa del suo modello basato su log strutturati e partizionamento [41].

- **Complessità operativa**: la configurazione di Kafka richiede un'infrastruttura distribuita più complessa, con nodi multipli e dipendenze esterne, aumentando il carico operativo rispetto a RabbitMQ, che può essere deployato come un singolo nodo o in un cluster leggero [39].

- **Supporto Python**: sebbene Kafka disponga di client Python, l'ecosistema AMQP di RabbitMQ è più maturo e integrato con Python tramite `pika`, offrendo una gestione più semplice delle connessioni asincrone [36, 42].

La Tabella 3.1 sintetizza le principali differenze tra RabbitMQ e Kafka in relazione ai requisiti del sistema.

Caratteristica	RabbitMQ	Apache Kafka
Pattern di messaggistica	*Pub/sub* tramite AMQP; *exchanges* per routing flessibile; modello *push-based* [43]	*Pub/sub* su *topic* partizionati; modello *pull-based* meno flessibile[39]
Latenza	Bassa, ottimizzata per real-time [35]	Più alta in scenari a bassa intensità, dovuta a log strutturati e partizionamento [41]
Throughput	Adatto a flussi moderati [35]	Ideale per flussi massivi (milioni di eventi/s) [39]
Integrazione Python	`pika` con supporto `asyncio`, maturo e ben documentato [36]	Client meno maturi e complessi per connessioni asincrone [42]
Formato dati	Supporto nativo per JSON, ideale per interoperabilità [43]	Supporto per JSON, ma ottimizzato per log strutturati [39]
Complessità operativa	Configurazione leggera, deploy singolo o cluster; interfaccia di amministrazione intuitiva [38]	Richiede infrastruttura distribuita complessa con dipendenze[39]

Tabella 3.1: Confronto tra RabbitMQ e Apache Kafka per il sistema di gestione delle emergenze.

3.3 Database di persistenza: PostgreSQL

Il database di persistenza deve garantire la gestione efficiente dei dati spaziali, il supporto a notifiche in tempo reale e l'integrazione con l'ecosistema tecnologico basato su Python (Sezione 3.1).
Dopo un'analisi comparativa tra PostgreSQL con l'estensione PostGIS e Re-

thinkDB, PostgreSQL è stato scelto come soluzione ottimale. La presente sezione motiva tale decisione, descrivendo i requisiti del sistema, le caratteristiche di entrambe le soluzioni e i motivi che rendono PostgreSQL più adatto al contesto applicativo.

3.3.1 Requisiti del sistema di persistenza

I requisiti per il database di persistenza derivano dai vincoli funzionali e non funzionali delineati nel Capitolo 1 e dalle esigenze operative dei microservizi descritti nella Sezione 2.3:

- **Gestione di dati spaziali**: il database deve supportare l'analisi spaziale avanzata, come il calcolo di percorsi minimi (*shortest path*) su un grafo rappresentante l'edificio, e la manipolazione di geodati per il tracciamento in tempo reale degli utenti.
- **Notifiche in tempo reale**: il sistema deve evitare meccanismi di polling inefficienti, favorendo notifiche immediate in caso di variazioni nei dati (es. nuovi alert o aggiornamenti delle posizioni).
- **Flessibilità nella gestione dei dati**: il database deve supportare sia dati strutturati (per analisi statistiche e query per chiavi primarie) sia dati destrutturati (es. JSON per rappresentare informazioni complesse come coordinate o metadati degli alert).
- **Rappresentazione geometrica dell'edificio**: occorre memorizzare coordinate geografiche/cartesiane e topologie con precisione metrica, così da interrogarle con operatori geometrici e algoritmi di path-finding.
- **Efficienza e scalabilità**: il database deve gestire un volume moderato di dati con prestazioni elevate, supportando query complesse e garantendo scalabilità per futuri ampliamenti.

Oltre ai requisiti elencati, era essenziale che il DBMS disponesse di notifiche push native, così da evitare un ipotetico polling continuo da parte

dell'*Alert Manager* in fase di gestione iniziale dell'emergenza (cfr. Sezione 2.4.2). Infatti, durante l'analisi preliminare, era stata ipotizzata una strategia di polling periodico per intercettare nuovi eventi critici; questa soluzione, però, avrebbe generato query superflue e latenza aggiuntiva, compromettendo la tempestività del sistema. Si è dunque limitata la valutazione a motori che offrono meccanismi di pubblicazione immediata delle variazioni. Alla luce di tali criteri, il confronto si è concentrato su PostgreSQL + PostGIS e RethinkDB.

3.3.2 PostgreSQL con PostGIS

PostgreSQL è un database relazionale open-source noto per la sua robustezza, flessibilità e supporto per estensioni avanzate come PostGIS, che lo rende ideale per la gestione di dati spaziali [44].

Le principali caratteristiche che giustificano l'adozione di PostgreSQL con PostGIS nel sistema includono:

- **Supporto per analisi spaziale avanzata**: l'estensione PostGIS consente di modellare la mappa dell'edificio come un grafo navigabile e di eseguire query spaziali complesse, come il calcolo di percorsi minimi (*shortest path*) utilizzando algoritmi ottimizzati[45]. Ciò è fondamentale per il *Map Manager*, che calcola percorsi di evacuazione in tempo reale.

- **Meccanismi di notifica in tempo reale**: PostgreSQL offre il sistema `LISTEN/NOTIFY`, che permette ai microservizi di iscriversi a canali di notifica e ricevere aggiornamenti immediati in caso di variazioni nei dati (es. nuovi alert CAP o cambiamenti nelle posizioni degli utenti) [46]. Questo elimina la necessità di polling periodico, riducendo il carico sul database e garantendo una reattività adeguata ai requisiti di bassa latenza.

- **Gestione ibrida dei dati**: PostgreSQL supporta sia dati strutturati, organizzati in tabelle relazionali per analisi statistiche e query per

chiavi primarie, sia dati destrutturati tramite il tipo `JSONB`, che consente di memorizzare e interrogare documenti JSON in modo efficiente [47]. Questo è ideale per rappresentare informazioni complesse come coordinate spaziali o metadati degli alert.

- **Primitive geometriche native**: PostGIS estende PostgreSQL con tipi `POINT`, `LINESTRING`, `POLYGON` e funzioni di topologia; ciò permette di rappresentare accuratamente la pianta dell'edificio e di eseguire query metriche fondamentali per il tracciamento delle posizioni degli utenti.

- **Integrazione con Python**: il driver `psycopg2` offre un'integrazione robusta e ben documentata con Python, supportando operazioni asincrone tramite `asyncio` e garantendo compatibilità con l'ecosistema tecnologico del sistema [26].

- **Scalabilità e affidabilità**: PostgreSQL è progettato per gestire carichi moderati con alte prestazioni, supportando indici spaziali (es. GiST per PostGIS) e meccanismi di replica per garantire disponibilità e fault tolerance [48]. La sua maturità e il supporto per transazioni ACID lo rendono adatto a scenari critici come la gestione delle emergenze.

Queste caratteristiche rendono PostgreSQL con PostGIS una scelta ottimale per un sistema che richiede analisi spaziale avanzata, notifiche in tempo reale e flessibilità nella gestione dei dati.

3.3.3 Analisi delle alternative: RethinkDB

RethinkDB è un database NoSQL documentale progettato per applicazioni in tempo reale, con un focus su semplicità e reattività [49]. Sebbene offra alcune caratteristiche interessanti, presenta limitazioni nel contesto del progetto:

- **Supporto per notifiche in tempo reale**: RethinkDB utilizza i *Changefeeds*, un meccanismo che consente ai client di ricevere aggior-

namenti automatici in caso di modifiche ai dati[50]. Questo elimina la necessità di polling, rendendo RethinkDB competitivo in scenari real-time.

- **Flessibilità dei dati**: come database *schema-less*, RethinkDB utilizza JSON come formato nativo, semplificando la gestione di dati destrutturati e consentendo query flessibili tramite ReQL, un linguaggio dichiarativo simile a Python e JavaScript [51]. Tuttavia, le query complesse possono essere meno efficienti rispetto a un database relazionale come PostgreSQL [52].

- **Limitazioni per analisi spaziale**: RethinkDB offre funzionalità di geolocalizzazione di base, ma non supporta analisi spaziali avanzate come il calcolo di percorsi minimi su grafi complessi, a differenza di PostGIS [53].

- **Integrazione con Python**: RethinkDB dispone di un driver Python ufficiale, ma la sua comunità è meno attiva rispetto a quella di PostgreSQL, e il supporto per operazioni spaziali è limitato [54].

La Tabella 3.2 sintetizza il confronto tra PostgreSQL con PostGIS e RethinkDB in relazione ai requisiti del sistema.

Caratteristica	**PostgreSQL con PostGIS**	**RethinkDB**
Analisi spaziale	Supporto avanzato tramite PostGIS (*shortest path*, query spaziali) [45]	Funzionalità geospaziali di base[53]
Notifiche real-time	`LISTEN/NOTIFY` per aggiornamenti immediati [46]	*Changefeeds* per notifiche automatiche [50]
Gestione dati	Ibrida: tabelle relazionali e `JSONB` [47]	*Schema-less* con JSON nativo [51]
Prestazioni query complesse	Ottimizzate per join, aggregazioni e query spaziali [48]	Meno efficienti per query complesse [52]

Tabella 3.2: Confronto tra PostgreSQL con PostGIS e RethinkDB per il sistema di evacuazione indoor.

In conclusione, PostgreSQL con PostGIS è stato scelto per il supporto avanzato all'analisi spaziale tramite PostGIS, notifiche in tempo reale con `LISTEN/NOTIFY`, gestione ibrida di dati strutturati e destrutturati tramite `JSONB`, e integrazione robusta con Python tramite `psycopg2`, garantendo prestazioni, affidabilità e allineamento con i requisiti del sistema [26, 45, 46, 47].

3.4 File di configurazione

I file di configurazione sono un componente fondamentale per la gestione parametrica del sistema, poiché consentono di definire in modo modulare e flessibile le impostazioni operative dei microservizi e dei componenti infrastrutturali senza alterare il codice sorgente. La scelta dei formati di configurazione è stata guidata dai requisiti del sistema (cfr. Capitolo 1) e dalle esigenze di integrazione con l'ecosistema tecnologico basato su Python, Rab-

bitMQ e PostgreSQL (cfr. Sezioni 3.1, 3.2, 3.3).
Dopo un'attenta analisi comparativa, riassunta nella Tabella 3.3, si è optato per un approccio ibrido che combina **YAML**, **JSON** e **file Python**.

3.4.1 Requisiti dei file di configurazione

I requisiti per i file di configurazione, derivanti dai vincoli del Capitolo 1 e dalle esigenze dei microservizi (Sezione 2.3), includono:

- Elevata leggibilità per semplificare modifiche da parte di sviluppatori e amministratori.
- Supporto a strutture gerarchiche per dati complessi (quali parametri di RabbitMQ o PostgreSQL).
- Integrazione fluida con Python per il parsing.
- Interoperabilità con strumenti come Docker e RabbitMQ.

3.4.2 Caratteristiche e casi d'uso

- **YAML**: con una sintassi chiara e indentata, è ideale per configurazioni infrastrutturali, come Docker Compose o connessioni a RabbitMQ e PostgreSQL. È facilmente parsabile con `pyyaml` [55, 56].
- **JSON**: leggero e standard, si usa per messaggi in RabbitMQ e configurazioni interne ai microservizi. La libreria `json` di Python ne garantisce un parsing rapido [57].
- **Python (.py)**: offre flessibilità per configurazioni dinamiche grazie alla logica programmabile. È nativo in Python, ma il suo uso è limitato per ridurre rischi di sicurezza [58].

Caratteristica	**YAML**	**JSON**	**Python**
Leggibilità	Elevata [59]	Buona [57]	Limitata
Parsing in Python	`pyyaml` [56]	`json` [60]	Nativo
Uso tipico	Infrastruttura	Messaggi	Logica dinamica

Tabella 3.3: Confronto tra formati di configurazione.

L'approccio ibrido adottato gode dei punti di forza dei tre formati:

1. **YAML** assicura leggibilità e compatibilità con strumenti di orchestrazione come Docker [61].
2. **JSON** garantisce interoperabilità per i messaggi scambiati tramite RabbitMQ [60].
3. **Python** offre flessibilità per configurazioni complesse, con un uso controllato per minimizzare vulnerabilità [62].

Questa combinazione ottimizza leggibilità, interoperabilità e funzionalità, adattandosi alle diverse esigenze del sistema e supportando una gestione efficiente delle configurazioni.

Capitolo 4

Implementazione: microservizio Centro Notifiche

Il **Centro Notifiche** (o *Notification Center*) è il microservizio deputato alla gestione e distribuzione delle notifiche di emergenza all'interno del sistema (cfr. Sezione 2.3.2). Esso funge da hub centrale, orchestrando la comunicazione tra i componenti del software: riceve allerte dall'*Alert Manager*, elabora dati sugli utenti a rischio dal *Position Manager* e inoltra istruzioni personalizzate allo *User Simulator*, che in fase di test emula i client finali. In un contesto reale, il *Notification Center* dovrà gestire **notifiche push** dirette agli utenti finali, utilizzando i dati di sottoscrizione memorizzati in un apposito database. Nel prototipo attuale, invece, tale funzionalità è sostituita da un meccanismo di inoltro verso lo *User Simulator* via RabbitMQ: gli utenti e i dispositivi reali sono simulati, ma il sistema è già **predisposto** per l'integrazione futura del servizio di push.

Il *Notification Center* opera in modalità fortemente disaccoppiata e asincrona, avvalendosi di code di messaggistica per ricevere gli avvisi di emergenza e propagare le notifiche. Ciò consente di ottenere un'elevata resilienza e scalabilità: i componenti produttori e consumatori di messaggi non sono vincolati temporalmente tra loro, potendo operare a velocità indipendenti senza perdere dati. Come descritto nel Capitolo 3.2, adottando un'architet-

tura *event-driven* basata su RabbitMQ, il *Notification Center* implementa il pattern *publish/subscribe*, nel quale i mittenti (*publisher*) pubblicano eventi su un canale logico e tutti i servizi sottoscritti (*subscriber*) ricevono una copia del messaggio. Per mezzo di questo approccio, l'emissione di un'allerta è immediatamente notificata agli utenti interessati senza che il servizio produttore debba conoscerne i dettagli o gestire direttamente le connessioni ai client. Di conseguenza, il microservizio può concentrarsi esclusivamente sulla logica di smistamento e traduzione dei messaggi.

Le funzionalità principali del *Notification Center* includono:

- **Ricezione ed elaborazione delle allerte**: gestione dei messaggi di allerta in base alla tipologia (`Alert`, `Update` o `Cancel`).

- **Distribuzione delle notifiche**: inoltro di allerte e percorsi di evacuazione personalizzati per la successiva trasmissione ai client finali.

- **Coordinamento tra microservizi**: interazione dinamica con i moduli del sistema per garantire un flusso di informazioni in tempo reale.

In sintesi, il *Notification Center* agisce come punto centrale di smistamento delle notifiche nell'architettura a microservizi dell'applicazione, gestendo il ciclo di vita delle notifiche attraverso tre fasi principali:

1. **Ascolto** delle code di input per nuovi eventi di allerta.

2. **Processamento** di ciascun messaggio in base alla tipologia.

3. **Inoltro** dei messaggi gestiti verso le code di output appropriate.

Le sezioni successive approfondiscono lo stato dell'arte, le scelte tecnologiche e i dettagli implementativi, con particolare attenzione all'integrazione con l'ecosistema tecnologico descritto nel Capitolo 3.

4.1 Analisi dello stato dell'arte

Il problema della notifica tempestiva e affidabile delle allerte verso la popolazione è stato affrontato in numerosi sistemi reali su larga scala, e diversi pattern architetturali sono emersi come soluzioni consolidate. In questa sezione si confronterà la soluzione implementata con sistemi analoghi e si richiameranno i principi tecnologici di riferimento.

Dal punto di vista dei sistemi di allerta pubblica, un primo riferimento è **IPAWS**, il sistema nazionale statunitense.
IPAWS, analizzato e descritto nel Capitolo 1.2.2, costituisce un'architettura unificata che aggrega diversi canali di allarme sotto un'unica piattaforma federata. Un messaggio di allerta originato da un'autorità autorizzata viene inserito nel sistema, il quale provvede a smistarne la distribuzione simultanea su tutti i canali appropriati, garantendo un'ampia diffusione alla popolazione. Il Centro Notifiche sviluppato presenta analogie concettuali con IPAWS, in quanto funge da nodo di aggregazione e ridistribuzione: in miniatura, esso riceve l'allerta da una sorgente centralizzata e la ridirige verso un canale di notifica agli utenti. Tuttavia, a differenza di IPAWS, che opera a livello nazionale e multi-canale, il microservizio implementato si limita a un singolo canale simulato e a scala locale.
Inoltre, IPAWS utilizza in gran parte infrastrutture broadcast e sistemi eterogenei integrati via server CAP, mentre il *Notification Center* adotta un broker AMQP per inoltrare messaggi in una rete chiusa di microservizi. Si noti che entrambi gli approcci perseguono obiettivi comuni di affidabilità e tempestività: IPAWS garantisce la distribuzione contemporanea su multipli canali per massimizzare la copertura, mentre il *Notification Center* garantisce la consegna attraverso meccanismi di acknowledgment propri di RabbitMQ per assicurare che ogni messaggio raggiunga lo *User Simulator*.

Un secondo sistema rilevante è **IT-Alert**, la piattaforma di allerta pubblica italiana. Anch'esso, come esplicitato nel Capitolo 1.2.1, impiega la tecnologia cell broadcast per inviare messaggi di allarme a tutti i telefoni cellulari presenti in una determinata area geografica in caso di emergenze gravi

o disastri imminenti. Il Centro Notifiche implementato, pur non utilizzando cell broadcast, realizza una forma di notifica push analoga in ambito simulativo: infatti, il messaggio viene inoltrato verso lo *User Simulator* non appena disponibile, senza che i client debbano interrogare attivamente il server. Nel sistema in analisi, il ruolo di broadcast è svolto da RabbitMQ: il messaggio di allerta viene inserito in una coda a cui il consumer (simulatore) è collegato, e quindi consegnato immediatamente.

Un'ulteriore soluzione di riferimento nel panorama delle notifiche di emergenza è la piattaforma commerciale **Alertus**. Essa offre un sistema di *mass notification* unificato, pensato per contesti indoor, che permette di attivare con un solo gesto una molteplicità di dispositivi di allarme fisici [63]. L'architettura Alertus prevede un server centrale che, ricevuta un'allerta da una console o fonte esterna, provvede a smistare l'allarme su tutti i canali configurati. In termini architetturali, Alertus adotta un modello simile al publish/subscribe: l'evento di allarme attiva una serie di notifiche verso N endpoint differenti, il tutto orchestrato dal server centrale. Il *Notification Center* può essere paragonato, in miniatura, al server Alertus: anch'esso riceve un input (l'alert) e lo ridistribuisce verso i suoi endpoint. La differenza principale sta nell'ambito di applicazione: Alertus è pensato per reti locali e usa protocolli eterogenei, mentre il Centro Notifiche lavora interamente su messaggistica AMQP interna al sistema software. Entrambi, comunque, beneficiano del decoupling: l'attivazione è *single-point* e la distribuzione è affidata a meccanismi asincroni ai vari subscriber. Questo elimina la necessità di chiamate dirette a ciascun dispositivo e consente di aggiungere/rimuovere canali di notifica senza modificare il produttore dell'alert.

4.1.1 Innovazioni introdotte dal Centro Notifiche

Rispetto allo stato dell'arte, il microservizio *Notification Center* si distingue per:

1. **Architettura *publish/subscribe* AMQP personalizzata**: si è implementato un sistema di code RabbitMQ dedicato alle diverse tipologie di messaggi di emergenza (`Alert`, `Evacuation`, `Update` e `Stop`), realizzando un disaccoppiamento completo tra i produttori di eventi e i consumatori. Ciò consente una scalabilità fine-grained: più istanze del servizio di simulazione utente possono sottoscriversi alle medesime code per gestire un numero elevato di dispositivi in parallelo, senza interferire con la generazione delle allerte.

2. **Notifiche geolocalizzate indoor**: il Centro Notifiche inoltra agli utenti non solo l'allarme generico, ma anche istruzioni personalizzate sotto forma di percorso di evacuazione calcolato in tempo reale dal *Map Manager*. La capacità di tradurre un alert globale in indicazioni specifiche per ciascun individuo all'interno di un edificio, tenendo conto della posizione corrente e delle vie di fuga disponibili, è un elemento innovativo rispetto ai sistemi di allerta esistenti.

4.2 Analisi delle alternative: motivazioni delle scelte

La progettazione del microservizio *Notification Center* ha richiesto la valutazione di diverse alternative tecnologiche per il meccanismo di distribuzione delle notifiche verso gli utenti e per la gestione della comunicazione inter-microservizi. Di seguito si analizzano le principali opzioni considerate e le motivazioni che hanno guidato le scelte implementative finali.

Un primo aspetto valutato è stato **come far pervenire le allerte ai dispositivi utente**. In un sistema reale, esistono essenzialmente due modelli:

1. **Modello push**: utilizzato nelle notifiche push di smartphone attraverso servizi come Firebase Cloud Messaging.

2. **Modello pull**: i client effettuano periodicamente richieste a un server per verificare la presenza di nuove allerte, come avviene in semplici sistemi basati su polling HTTP.

Per un sistema di allerta tempestiva, il modello push risulta nettamente preferibile, poiché consente di ridurre al minimo la latenza di consegna e garantisce che l'utente riceva l'avviso anche fuori da un contesto di applicazione attiva. Nel nostro progetto, la scelta di utilizzare RabbitMQ come intermediario ha permesso di implementare un sistema push efficiente all'interno dell'ecosistema dei microservizi: il Centro Notifiche pubblica sulle code a cui lo *User Simulator* è iscritto, "spingendo" così le notifiche verso i consumer. Un'alternativa inizialmente considerata era l'uso di semplici richieste HTTP dal Centro Notifiche verso il servizio utente (ad esempio tramite una chiamata REST POST contenente l'allerta). Tale approccio, sebbene più semplice da implementare in assenza di un broker, avrebbe comportato una forte accoppiamento tra i servizi e una minore robustezza: in caso di indisponibilità temporanea del Simulatore Utenti, la richiesta HTTP sarebbe fallita e l'allerta poteva andare persa, a meno di implementare manualmente meccanismi di riprova e queueing nel codice. L'uso di RabbitMQ fornisce questi meccanismi *out-of-the-box*: il messaggio rimane in coda finché il consumer non lo riceve con successo, grazie al sistema di ACK, e il Centro Notifiche non deve conoscere l'indirizzo o lo stato del servizio utente, riducendo l'accoppiamento. Pertanto, l'alternativa RESTful-polling è stata scartata in favore della soluzione attuale basata su *message broker*.
Nel prototipo, lo *User Simulator* funge da consumer temporaneo. In un ambiente di produzione, può essere sostituito da un **Notification Gateway**, un servizio specializzato che consuma i messaggi da RabbitMQ e li recapita ai dispositivi reali degli utenti. Esso può gestire:

- **Un registro delle sottoscrizioni**: un database che associa gli utenti ai loro token di notifica, permettendo al gateway di sapere a chi inviare ogni allerta.

- **L'idempotenza lato client**: un meccanismo per prevenire che l'utente riceva notifiche duplicate, un aspetto fondamentale per garantire una buona esperienza utente, specialmente in caso di tentativi di consegna falliti e successivi riavvii.

Il message broker (RabbitMQ) mantiene il suo ruolo cruciale, garantendo il disaccoppiamento e l'affidabilità del sistema: agisce come un punto di smistamento robusto, permettendo al Centro Notifiche di inviare messaggi senza conoscere lo stato o l'indirizzo del destinatario finale.

Inoltre, all'interno dell'ecosistema di microservizi, era necessario scegliere come **orchestrare lo scambio di dati** tra i vari moduli. L'analisi delle alternative considerate è già stata affrontata e approfondita nel Capitolo 3.2.

Dal punto di vista della **struttura interna del microservizio**, un'alternativa discussa riguardava l'uso di un meccanismo di notifiche asincrone da database, invece del broker per alcuni tipi di messaggi di feedback. In particolare, avendo a disposizione un database PostgreSQL come componente condiviso del sistema, si sarebbe potuto sfruttare la funzionalità `LISTEN/NOTIFY` di Postgres per ricevere eventi e far sì che il *Notification Center* vi reagisse. Questa strada, tuttavia, è stata abbandonata per privilegiare un design più coerente centrato su RabbitMQ: introdurre anche il database come bus di messaggi avrebbe aumentato la complessità e creato duplicazione di logica (due canali differenti di notifica). Si è preferito utilizzare il database solo per l'archiviazione di dati persistenti, mentre tutte le notifiche in tempo reale viaggiano sul broker AMQP.

4.3 Sviluppo operativo del microservizio

Lo sviluppo del microservizio *Notification Center* è stato realizzato in Python, sfruttando le tecnologie selezionate nel Capitolo 3. Il microservizio è strutturato per gestire due flussi principali:

1. Ricezione e processamento delle allerte provenienti dall'*Alert Manager*, con successiva notifica immediata agli utenti.

2. Ricezione dei dati sugli utenti in pericolo dal *Position Manager* e dei percorsi di evacuazione calcolati dal *Map Manager*, con inoltro di tali informazioni agli utenti interessati

Di seguito si descrivono i dettagli implementativi e la logica di funzionamento, supportati da diagrammi e frammenti di codice significativi.

4.3.1 Struttura dei componenti e classi principali

Il microservizio è organizzato in moduli distinti, ciascuno con responsabilità specifiche, seguendo il principio di *single responsibility* dell'architettura a microservizi, con classi che gestiscono rispettivamente la connessione al broker, il consumo dei messaggi e l'inoltro ai destinatari.

Le componenti principali del microservizio includono:

- **Main** (`main.py`): punto di ingresso del microservizio, responsabile dell'inizializzazione del *RabbitMQHandler* e della configurazione delle code necessarie. Gestisce anche l'avvio del consumatore dei messaggi di allerta e la chiusura pulita del sistema tramite segnali di sistema.

- **Handler RabbitMQ** (`rabbitmq_handler.py`:) implementa la classe `RabbitMQHandler`, che incapsula la logica di connessione al broker RabbitMQ e fornisce metodi per dichiarare code, inviare messaggi e avviare la consumazione di messaggi in arrivo in formato JSON. Questa classe gestisce automaticamente riconnessioni in caso di drop della connessione, garantendo la resilienza del collegamento al broker.

- **Consumer di allerte** (`alert_consumer.py`): definisce la classe `AlertConsumer`, che si occupa di consumare i messaggi dalla coda `alert_queue`. Quando un nuovo messaggio di allerta viene ricevuto, l'*AlertConsumer* lo processa in base alla tipologia (`msgType`) e intraprende le azioni appropriate:

  - Se `msgType` = `Alert` o `Update`, la notifica viene inoltrata immediatamente agli utenti.
  - Se `msgType` = `Cancel`, viene creata una notifica di stop.

- **Consumer dei percorsi di evacuazione** (`alerted_users_consumer.py`): implementa la classe `AlertedUsersConsumer`, incaricata di consumare i messaggi dalla coda `alerted_users_queue`. Processa i percorsi di evacuazione o i segnali di stop ricevuti dal *Position Manager* e li inoltra allo *User Simulator*.

- **Modulo di inoltro** (`alert_smister_to_user_simulator.py`): fornisce funzioni di utility per l'inoltro delle allerte e dei percorsi di evacuazione sul broker verso le code di output `user_simulator_queue` e `evacuation_paths_queue`.

- **Handler del database** (`database_handler.py`): incapsula la gestione della connessione a PostgreSQL, destinato a ospitare la tabella delle sottoscrizioni degli utenti alle notifiche push, tramite il driver `psycopg2`. L'handler configura il livello di isolamento `AUTOCOMMIT` e abilita il meccanismo `LISTEN/NOTIFY`, permettendo al microservizio di ricevere eventi push direttamente dal DBMS senza ricorrere a polling. Al momento della stesura, la base dati è stata predisposta ma non ancora popolata: la struttura è pronta per registrare, in una fase successiva, i token/device–id degli utenti che desiderano ricevere le notifiche push in tempo reale.

- **Configurazione e logging** (`settings.py` e `logging.py`): il file `settings.py` contiene tutte le configurazioni centralizzate, quali i nomi delle code RabbitMQ, le credenziali di connessione e i parametri di connessione al database. Questo approccio consente di modificare facilmente tali parametri senza modificare il codice logico. Il modulo `logging.py` definisce la configurazione del logging applicativo: si utilizza il modulo standard `logging` di Python con un formato uniforme dei messaggi

e gestori che scrivono su file separati per ogni componente. Questo facilita il debugging, poiché i file di log possono essere consultati per monitorare il flusso delle operazioni di ogni singolo componente in esecuzione.

La struttura di classi e interazioni del *Notification Center* è riassunta nel diagramma UML in Figura 4.1. Si noti che `AlertConsumer` e `AlertedUsersConsumer` utilizzano la medesima istanza di `RabbitMQHandler` per accedere al broker. Il modulo principale inizializza questi oggetti e avvia il ciclo di consumo dei messaggi. Il DatabaseHandler è separato in quanto potenzialmente utilizzabile per sincronizzare informazioni di sottoscrizione in tempo reale (anche se, nel prototipo corrente, il suo utilizzo è limitato).

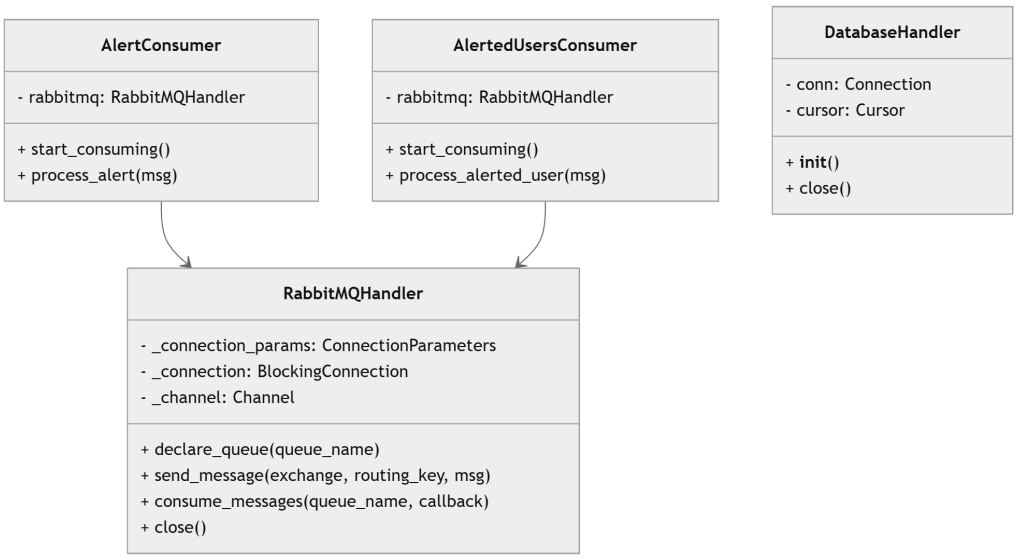

Figura 4.1: Diagramma delle classi: *Notification Center*

4.3.2 Flusso operativo

Il flusso operativo del *Notification Center* può essere suddiviso in tre fasi principali (cfr. Sezione 2.4):

1. **Ascolto**: il microservizio rimane in attesa di nuovi eventi sulle code di input: in particolare, `AlertConsumer` è in ascolto su `ALERT_QUEUE` per ricevere allerte dall'*Alert Manager*, mentre `AlertedUsersConsumer` è in ascolto su `ALERTED_USERS_QUEUE` per ricevere dati dal *Position Manager*.

2. **Elaborazione**: all'arrivo di un messaggio, il rispettivo consumer esegue la logica di business: interpreta il messaggio, logga l'evento, e decide come reagire (inoltro, trasformazione o generazione di nuovi messaggi).

3. **Inoltro**: se previsto, il Centro Notifiche produce a sua volta messaggi in uscita verso le code di output appropriate: tipicamente, le code destinate allo *User Simulator* (`USER_SIMULATOR_QUEUE` per le notifiche generiche e `EVACUATION_PATHS_QUEUE` per i percorsi di evacuazione).

Di seguito, si descrive un tipico scenario end-to-end di gestione di un'emergenza, con focus specifico sul ruolo del microservizio in analisi:

1. **Generazione dell'allerta**: l'*Alert Manager* rileva o riceve una condizione di emergenza e genera un messaggio di allerta in formato CAP, che pubblica, in formato JSON, sulla coda `ALERT_QUEUE` di RabbitMQ. Il messaggio include un identificativo univoco, il tipo di messaggio (`Alert` se è una nuova emergenza, `Update` se è un aggiornamento relativo a un'allerta esistente, oppure `Cancel` per segnalare la fine dell'emergenza), una descrizione testuale ed eventualmente informazioni geografiche sull'area coinvolta.

2. **Notifica immediata agli utenti**: il `Notification Center` (istanza `AlertConsumer`) riceve dalla coda il messaggio di allerta, riconosce il tipo di allerta e inoltra immediatamente il messaggio allo *User Simulator* tramite la coda `USER_SIMULATOR_QUEUE`. In questo modo, ogni utente (simulato) riceve una notifica con il testo dell'allarme e le informazioni base. In un'implementazione futura, questa stessa logica di smistamento potrà essere estesa per inviare direttamente notifiche push agli utenti reali, recuperando i dati di sottoscrizione dal database predisposto.

Contestualmente, il *Notification Center* inoltra lo stesso messaggio di allerta al *Position Manager*, responsabile dell'intercettazione delle aree o utenti coinvolti nell'allerta.

3. **Calcolo dei percorsi di evacuazione**: il microservizio *Map Manager*, una volta informato dell'emergenza, calcola in tempo reale i percorsi di evacuazione ottimali per gli utenti presenti nelle aree a rischio, i cui dati sono inviatogli dal *Position Manager*. Quest'ultimo, leggendo i percorsi di evacuazione aggiornati dal *Map Manager* dal database dedicato, genera un messaggio contenente tali dati e lo pubblica sulla `ALERTED_USERS_QUEUE`. Nel design del sistema, `ALERTED_USERS_QUEUE` può veicolare sia informazioni di evacuazione sia un segnale di stop quando tutti gli utenti sono in salvo o l'emergenza è rientrata.

4. **Inoltro dei percorsi di evacuazione**: il *Notification Center* (istanza `AlertedUsersConsumer`) riceve il messaggio dalla coda `ALERTED_USERS_QUEUE`. Se il messaggio contiene dei percorsi di evacuazione, modifica il campo `msgType` in `"Evacuation"` per differenziarlo da una normale notifica di allerta, e lo inoltra sulla coda `EVACUATION_PATHS_QUEUE` diretta allo *User Simulator*. Se invece il messaggio è di tipo `Stop`, indicante che l'emergenza è terminata, allora `AlertedUsersConsumer` instrada direttamente questo segnale agli utenti.

5. **Chiusura del ciclo di emergenza**: una volta inviati tutti i percorsi di evacuazione e/o il segnale di fine emergenza, il ruolo del *Notification Center* si conclude. Gli utenti avranno ricevuto prima l'allarme, poi eventualmente le istruzioni di evacuazione, ed infine la comunicazione di fine allarme. Il sistema rimane comunque in ascolto di eventuali ulteriori messaggi.

4.3.3 Tecnologie e implementazione

Il microservizio è implementato in Python, sfruttando le librerie descritte nella Sezione 3.1. In particolare:

- **RabbitMQ e pika**: la classe `RabbitMQHandler` utilizza `pika` per gestire connessioni, code e messaggi JSON. La configurazione (`settings.py`) include parametri come `RABBITMQ_HOST`, `RABBITMQ_PORT`, `RABBITMQ_USERNAME` e `RABBITMQ_PASSWORD`, con supporto per riconnessione automatica (`_reconnect`) e gestione degli errori.

- **PostgreSQL e psycopg2**: il modulo `database_handler.py` utilizza `psycopg2` per connettersi al database `subscription_db`, che memorizza le sottoscrizioni degli utenti. Il livello di isolamento `AUTOCOMMIT` supporta il sistema `LISTEN/NOTIFY`, essenziale per ricevere aggiornamenti in tempo reale senza polling, come descritto nella Sezione 3.3.2.

- **Logging**: il modulo `logging.py` configura un sistema di logging centralizzato, con file di log separati per ciascun componente (es. `alertConsumer.log`, `alertedUsersConsumer.log`). Il formato include timestamp, livello di log e messaggio, garantendo tracciabilità e facilitando il debug.

4.3.4 Gestione degli errori e resilienza

Il microservizio implementa meccanismi robusti per la gestione degli errori:

- **Riconnessione a RabbitMQ**: la classe `RabbitMQHandler` include logica di riconnessione con *exponential backoff* (`_reconnect`), che tenta fino a tre connessioni in caso di fallimento, con ritardi crescenti.

- **Acknowledgement dei messaggi**: i consumatori (`AlertConsumer` e `AlertedUsersConsumer`) utilizzano `basic_ack` per confermare la ricezione dei messaggi e `basic_nack` per gestire errori, come messaggi JSON non validi, con possibilità di reinserimento in coda.

- **Chiusura pulita**: il modulo `main.py` gestisce segnali di sistema per garantire una chiusura ordinata, svuotando le code e chiudendo le connessioni a RabbitMQ e PostgreSQL.

In conclusione, il microservizio Centro Notifiche è stato implementato con successo, rispettando i requisiti di modularità, scalabilità e resilienza definiti nel Capitolo 1. L'uso di Python, RabbitMQ e PostgreSQL garantisce un'integrazione fluida con l'ecosistema complessivo, mentre i meccanismi di gestione degli errori e logging assicurano affidabilità e tracciabilità.

Capitolo 5

Implementazione: microservizio Visualizzatore della Mappa

Il *MapViewer* è il microservizio incaricato della visualizzazione dinamica e interattiva della planimetria di un edificio multi-piano. Integra il monitoraggio in tempo reale della posizione degli utenti e l'interazione con la struttura topologica del grafo di navigazione interno. Espone API REST lato **backend** per interrogare e aggiornare lo stato del grafo e offre un **frontend** web per il rendering su immagini raster. Il grafo è mantenuto coerente sia in memoria, per reattività, sia su database relazionale, per persistenza e auditing.
Le sue funzionalità principali includono: (i) visualizzazione della pianta per piano con overlay dei nodi e degli archi; (ii) aggiornamento dello stato del grafo e riflessione immediata in UI; (iii) tracciamento periodico delle posizioni utente; (iv) gestione di un sistema di coordinate locale pixel-based coerente con l'immagine di planimetria.

Il presente capitolo illustra lo sviluppo del microservizio: la Sezione 5.1 analizza lo stato dell'arte; la Sezione 5.2 confronta le alternative e motiva le scelte progettuali; infine, la Sezione 5.3 documenta nel dettaglio lo sviluppo operativo.

5.1 Analisi dello stato dell'arte

La visualizzazione di mappe indoor e la gestione di dati spaziali per la navigazione in ambienti interni sono ambiti in rapida evoluzione, con applicazioni che spaziano dalla gestione delle emergenze alla logistica in edifici complessi. Le tecnologie adottate nel microservizio *Map Viewer* si collocano in un panorama tecnologico che include strumenti di visualizzazione cartografica, database spaziali, framework per microservizi e librerie per la gestione di grafi.
Di seguito si inquadrano le soluzioni principali, con attenzione al rapporto con le scelte progettuali del sistema.

5.1.1 Visualizzazione indoor e cartografia dedicata

La cartografia indoor presenta esigenze differenti rispetto alla mappatura tradizionale outdoor, richiedendo strategie di visualizzazione e modelli concettuali specifici[64]. In letteratura, infatti, si distinguono vari tipologie di mappe per interni: dalle planimetrie architettoniche dettagliate, ideate per progettazione e manutenzione edilizia, a rappresentazioni semplificate orientate alla navigazione degli utenti, fino ad approcci in realtà aumentata. Nello specifico, le *floor plan maps*, piante bidimensionali semplificate per ciascun piano, risultano efficaci nei compiti di wayfinding, poiché omettono dettagli superflui privilegiando informazioni pertinenti al percorso e all'orientamento umano. Questo approccio, adottato anche dal *Map Viewer*, rende l'interfaccia chiara e meno appesantita da elementi accessori. In scenari che richiedono maggiore immersività, la letteratura propone visualizzazioni 3D o in realtà aumentata, evidenziando benefici nella comprensione spaziale e nei tempi di navigazione[65]. Tuttavia, tali soluzioni impongono requisiti hardware/software più elevati. Di conseguenza, la rappresentazione 2D per piano rimane uno standard de facto per molte applicazioni indoor: nel contesto del *Map Viewer*, si adotta deliberatamente una resa 2D per piano, privilegiando semplicità e facilità di interpretazione.

5.1.2 Modelli spaziali indoor e grafi di navigazione

Per la modellazione formale degli spazi interni e delle loro relazioni topologiche, lo standard **IndoorGML** dell'OGC rappresenta lo stato dell'arte: descrive gli ambienti indoor come un grafo di spazi (nodi) e connessioni (archi) con informazioni semantiche dettagliate[65]. IndoorGML adotta il concetto di *Cellular Space*, in cui ogni spazio navigabile è una *CellSpace* e ogni collegamento è modellato come *State* e *Transition* nella rete di navigazione. Questo standard fornisce uno schema concettuale unificato per scambiare informazioni in modo interoperabile[66]. Accanto a IndoorGML, formati industriali mirano a una integrazione applicativa diretta: è il caso di **Indoor Mapping Data Format** (IMDF) di Apple, il quale fornisce un modello per descrivere feature indoor georeferenziate sul piano globale, consentendo calcolo di percorsi all'interno di mappe di edifici. A differenza di IndoorGML, focalizzato su una rappresentazione generalizzata del grafo navigabile indipendente dall'applicazione, IMDF presenta un network pathfinding incorporato, ma meno flessibile ad utilizzi generici[66].
In sintesi, lo stato dell'arte mostra, da un lato, modelli dati indoor formalizzati, dall'altro la prassi diffusa di utilizzare schemi semplificati a seconda dell'ecosistema. Nel sistema di evacuazione implementato, si è optato per un modello leggero orientato al grafo di navigazione, che mantiene compatibilità concettuale con tali standard, senza implementarne l'intero schema formale: i nodi del grafo corrispondono alle unità spaziali navigabili (simili alle CellSpace di IndoorGML) e gli archi rappresentano connessioni fisiche (analoghe alle Transition).

5.1.3 Estrazione automatica e modellazione manuale del grafo indoor

Nell'ambito della rappresentazione astratta della planimetria di un edificio, la principale sfida consiste nella modalità di ottenimento del grafo di navigazione. La letteratura propone approcci automatici che sfruttano i dati

disponibili: piante architettoniche 2D, modelli BIM 3D o scansioni laser degli interni. Alcune ricerche mirano a estrarre il grafo direttamente da planimetrie esistenti tramite algoritmi di visione e di riconoscimento delle strutture. Progetti basati su floor plan digitali applicano tecniche di segmentazione delle immagini per riconoscere stanze e corridoi; alcuni studi hanno sperimentato reti neurali profonde per estrarre elementi strutturali da mappe raster, con risultati promettenti in casi controllati[67]. Tuttavia, tali approcci automatici soffrono di limitazioni significative nel caso di edifici complessi o dati eterogenei: le planimetrie "as-built" spesso differiscono dai disegni originali e richiederebbero continui aggiornamenti manuali, mentre i metodi puramente automatici faticano in presenza di rumore, occlusioni o simboli non standard [68]. In pratica, l'estrazione affidabile richiede un compromesso tra automazione e intervento manuale: per questo motivo, diverse soluzioni industriali adottano workflow semi-automatici, in cui la maggior parte del grafo è generato algoritmicamente, ma l'operatore valida e corregge i risultati.
Coerentemente, dopo tentativi di estrazione completamente automatica via OpenCV sulla pianta raster, il sistema ha adottato una modellazione manuale assistita, che garantisce piena corrispondenza con l'ambiente reale e una semantica corretta (cfr. Sezione 5.2.1). Tale approccio garantisce elevata accuratezza, a scapito di maggiore sforzo iniziale di configurazione, mitigato però dalla persistenza su database e riuso del grafo.

5.1.4 Basi di dati spaziali e gestione del grafo

Nel panorama delle soluzioni per il mapping indoor emergono principalmente due strategie per le modalità di memorizzazione e interrogazione dei dati relativi al grafo di navigazione:

1. **Database relazionali spaziali (PostgreSQL/PostGIS)**: la combinazione di PostgreSQL con l'estensione spaziale PostGIS fornisce tipi di dati geometrici e query efficienti per la gestione di coordinate, distanze e intersezioni, e viene frequentemente associata a librerie per il calcolo di percorsi sul grafo archiviato in tabelle relazionali[69]. Ad esempio,

vari modelli di database indoor ibridi (geometrici e simbolici) sono implementati su PostGIS, in cui le connessioni verticali e orizzontali sono memorizzate in tabelle relazionali e algoritmi di pathfinding basati su varianti estese di Dijkstra gestiscono il routing multi-piano[70]. Tale approccio sfrutta la maturità dell'ecosistema SQL e la disponibilità di operatori spaziali standard.

2. **Database a grafo (Neo4j)**: Neo4j permette di rappresentare naturalmente stanze e corridoi come nodi collegati da archi e di eseguire query di percorso con algoritmi di *graph traversal*. Ad esempio, il sistema *NaviSecure* implementa il modulo di pathfinding su Neo4j, sfruttando il linguaggio Cypher per ottenere il cammino più breve all'interno del grafo indoor e aggiornando dinamicamente i pesi degli archi in base a condizioni di sicurezza[71]. L'adozione di un grafo nativo elimina la necessità di tradurre le relazioni spaziali in chiavi esterne e consente interrogazioni dichiarative, come l'identificazione di tutti i percorsi alternativi in caso di inaccessibilità di un nodo.

Tuttavia, la mancanza di funzioni GIS avanzate nei database a grafo e la necessità di integrare i risultati con coordinate metriche può complicarne l'adozione diretta per applicazioni indoor. Inoltre, in scenari di piccola scala, mantenere il grafo in memoria nell'applicativo backend può risultare più efficiente rispetto alla delega di ogni richiesta a un database esterno. Questa è la scelta operata in *Map Viewer*, dove NetworkX in Python gestisce il grafo di ogni piano in RAM, garantendo una rapida risposta alle API, mentre il database relazionale viene utilizzato principalmente per assicurare persistenza e consistenza globale (cfr. Sezione 5.2.2). Tale architettura ibrida coniuga i vantaggi dei grafi in-memory con la robustezza di un datastore spaziale affidabile per backup, analisi ad-hoc e integrazione futura con dati georeferenziati esterni.

5.1.5 Frontend e librerie di mappatura web

Nell'ambito della mappatura interattiva 2D, le librerie JavaScript più diffuse includono **Leaflet.js** e **OpenLayers**, entrambe in grado di supportare overlay di dati vettoriali e raster, funzionalità di zoom e pan fluidi, nonché componenti di interfaccia utente per l'interazione con la mappa.

OpenLayers si distingue per la sua completezza e flessibilità, caratteristiche che lo rendono ideale per applicazioni GIS complesse e ad alte prestazioni, sebbene a scapito di una maggiore complessità di configurazione e di un pacchetto più corposo.

Leaflet, d'altro canto, privilegia la semplicità e la leggerezza, consentendo di ottenere risultati rapidi con un codice conciso, particolarmente adatto a contesti applicativi più mirati [72].
Confronti indipendenti evidenziano che OpenLayers è preferibile per progetti GIS articolati, mentre Leaflet eccelle in soluzioni intuitive e di facile implementazione. Nel dominio della mappatura indoor, l'utilizzo di un overlay statico di planimetria può essere gestito efficacemente con Leaflet, grazie alla proiezione cartesiana semplice (CRS.Simple), che tratta i pixel dell'immagine come coordinate piane senza distorsioni, eliminando la necessità di proiezioni geografiche [73]. Ciò elimina la dipendenza da servizi esterni, permettendo piena funzionalità offline, vantaggio cruciale in ambienti con connettività limitata o assente.

In alternativa, librerie orientate alla gestione di mappe vettoriali e 3D, come **Mapbox GL JS**, offrono funzionalità avanzate, tra cui rendering sofisticato, stilizzazione dinamica e supporto per visualizzazioni tridimensionali. Tuttavia, tali soluzioni presentano vincoli significativi: l'utilizzo di Mapbox richiede un account attivo, l'inclusione di un token API e l'accettazione di restrizioni d'uso e costi legati al traffico dati [74, 75]. Questi requisiti rendono Mapbox meno idoneo per contesti self-hosted e chiusi, come il sistema in

esame.

Alla luce di tali considerazioni, la scelta di Leaflet per il *Map Viewer* si rivela pienamente conforme alle best practice evidenziate. Tale libreria, completamente open-source, leggera ed estensibile tramite plugin, si distingue per la sua immediatezza nell'utilizzo con immagini statiche e coordinate personalizzate. Consente di caricare planimetrie come Image Overlay a piena risoluzione, sovrapponendo marker e polilinee tramite layer vettoriali, senza richiedere proiezioni geografiche grazie all'opzione CRS.Simple, specificamente indicata per mappe indoor non georeferenziate [73]. Tale configurazione garantisce robustezza, semplicità e adattabilità, rendendo Leaflet una soluzione ottimale per il caso d'uso considerato.

In conclusione, il microservizio **Visualizzatore della Mappa** si inserisce con coerenza nel panorama delle soluzioni avanzate per la rappresentazione e la gestione delle mappe indoor, integrando in modo strategico tecnologie consolidate con adattamenti mirati alle specificità del caso d'uso. L'adozione di immagini raster di pianta arricchite con overlay interattivi segue un paradigma ampiamente riconosciuto in letteratura per favorire una navigazione indoor intuitiva e user-friendly [64]. Sul versante backend, l'architettura basata su microservizi, unitamente all'impiego di PostGIS e NetworkX, assicura conformità alle migliori pratiche industriali, garantendo elevati standard di scalabilità, affidabilità e prestazioni [70].
Pur senza implementare uno standard formale come IndoorGML, la soluzione conserva una congruenza concettuale con esso, in particolare per quanto concerne la rappresentazione del grafo di spazi connessi, aprendo la possibilità di un'evoluzione futura in tale direzione. In definitiva, lo stato dell'arte conferma la validità delle scelte progettuali: un approccio pragmatico, modulare e open-source, in cui ogni componente è selezionato per massimizzare robustezza e usabilità, risulta perfettamente in linea con quanto raccomandato da letteratura e standard per le applicazioni di indoor mapping e navigazione

negli edifici.

5.2 Analisi delle alternative: motivazioni delle scelte

In fase di progettazione e prototipazione del microservizio *Map Viewer*, sono state valutate più tecnologie per la visualizzazione della mappa e la gestione del grafo di navigazione.
La presente sezione, in continuità con l'analisi dello stato dell'arte (Sezione 5.1), confronta la soluzione adottata con la versione iniziale del sistema e con alternative presenti in letteratura, illustrando le motivazioni che hanno condotto all'implementazione finale. Nello specifico: la Sezione 5.2.1 analizza l'impiego di OpenCV per l'estrazione automatica del grafo dalla planimetria; la Sezione 5.2.2 discute la scelta del database e del sistema di coordinate (PostgreSQL/PostGIS) rispetto a un database a grafo (Neo4j); la Sezione 5.2.3 motiva l'adozione di Leaflet come libreria frontend rispetto a OpenLayers e Mapbox GL JS.

5.2.1 Estrazione automatica del grafo con OpenCV

In linea con i tentativi di estrazione automatica discussi nella Sezione 5.1.3, nella prima versione del sistema si è tentato di derivare automaticamente il grafo di navigazione dalla rappresentazione raster della planimetria tramite tecniche di *computer vision*, al fine di ridurre o eliminare la configurazione manuale di nodi e archi. In particolare, l'estrazione del grafo era effettuata utilizzando la **libreria OpenCV** per analizzare un'immagine statica della planimetria dell'edificio [76]. Infatti, OpenCV è una libreria open-source per la visione computazionale, che offre funzionalità avanzate per l'elaborazione di immagini e la creazione di visualizzazioni statiche, tra cui il pre-filtraggio Gaussiano e l'algoritmo di Canny impiegati nel pipeline sperimentale [77].

Il processo comprendenva le seguenti fasi:

- **Elaborazione dell'immagine**: l'immagine della planimetria veniva caricata e convertita in scala di grigi, applicando un filtro di sfocatura Gaussiana e l'algoritmo Canny per il rilevamento dei contorni. I contorni identificati rappresentavano potenziali nodi del grafo [76, 77].

- **Identificazione dei nodi**: i contorni rilevati venivano filtrati in base a un'area minima (`MIN_AREA_PX`) e convertiti in nodi, calcolando coordinate e proprietà, quali capacità e tipologia. Le coordinate venivano trasformate da pixel a unità del modello utilizzando un fattore di scala.

- **Creazione degli archi**: gli archi venivano generati collegando i centroidi dei nodi vicini, con una distanza massima definita (`THRESHOLD_DIST_CM`). Anch'essi erano arricchiti con attributi, quali capacità e tempo di attraversamento.

- **Inserimento nel database**: i nodi e gli archi estratti venivano salvati nel database PostgreSQL, utilizzando query SQL per popolare le tabelle `nodes` e `arcs`.

Sebbene tale approccio offrisse un potenziale di automazione, presentava limitazioni significative:

- **Imprecisione topologica su planimetrie reali complesse**: l'algoritmo di rilevamento dei contorni di OpenCV non era in grado di interpretare correttamente la semantica della planimetria. Ad esempio, spessori non uniformi delle pareti, corridoi stretti o aree complesse (come scale o ascensori) venivano spesso ignorati o rappresentati erroneamente, portando a un grafo di navigazione incoerente con la struttura reale dell'edificio. Inoltre, la dipendenza da parametri come `MIN_AREA_PX` e `THRESHOLD_DIST_CM` richiedeva una calibrazione manuale per ogni planimetria, riducendo la generalizzabilità del sistema[76, 77].

- **Mancanza di interpretazione semantica**: i soli contorni rilevati non distinguono porte, muri e passaggi; sono necessarie euristiche o modelli addestrati, con costi di manutenzione elevati [76].

- **Limitata interattività**: OpenCV, orientato principalmente all'elaborazione di immagini statiche, non supporta nativamente la visualizzazione interattiva web-based. La rappresentazione del grafo richiedeva un'integrazione aggiuntiva con un frontend dedicato, aumentando la complessità architetturale.

- **Prestazioni computazionali onerose**: l'elaborazione delle immagini con OpenCV, specialmente per planimetrie ad alta risoluzione, risultava computazionalmente costosa, con tempi di processamento significativi per l'estrazione dei contorni e la generazione del grafo [77].

I test condotti sul caso reale hanno evidenziato, dunque, i limiti pratici già noti in letteratura (cfr. Sezione 5.1.3): la necessità di *fine tuning* e di correzioni manuali ha annullato il vantaggio di automazione. Pertanto, per garantire accuratezza topologica e semantica, si è adottata una **modellazione manuale assistita** del grafo.

5.2.2 Database e sistema di coordinate

La combinazione di un database relazionale spaziale e di un grafo in memoria, già evidenziata come soluzione efficace nella Sezione 5.1.4, è stata adottata per la persistenza di nodi, archi, posizioni correnti e storico. In particolare, si è scelto **PostgreSQL** con estensione **PostGIS**, mentre i grafi operativi, distinti per piano, sono mantenuti in memoria tramite **NetworkX** con gestione thread–safe [23, 44, 45, 78]. Questa soluzione ibrida combina la solidità e le capacità GIS del database relazionale con la velocità di accesso della cache in memoria.
Benché PostGIS abiliti georeferenziazione e funzioni GIS [45], il sistema di coordinate adotta deliberatamente un *Coordinate Reference System (CRS)*

locale pixel–based ancorato all'immagine di planimetria, giustificato dalle motivazioni cartografiche e frontend presentate rispettivamente nelle Sezioni 5.1.1 e 5.1.5):

- L'origine $(0, 0)$ ricade sull'angolo superiore sinistro dell'immagine del piano; le coordinate di nodi e archi sono espresse in pixel rispetto a tale punto.
- Nel frontend, le coordinate immagine sono mappate invertendo l'asse y nella costruzione dei `LatLng` per l'image overlay[79, 80, 81, 82].
- La conversione pixel $\leftrightarrow$ unità di modello $\leftrightarrow$ metri è centralizzata nella componente `HeightMapper`, parametrizzata tramite un file di configurazione (`settings.py`), per stimare lunghezze, tempi di attraversamento e capacità in modo coerente.

Sebbene al momento si utilizzino coordinate locali, mantenere PostGIS disponibile con un sistema di riferimento spaziale reale (SRID) è strategico per:

- Possibile **georeferenziazione futura** e integrazione con servizi GIS esterni.
- Accesso a **operatori spaziali affidabili** senza necessità di reimplementazioni.
- Utilizzo di un **ecosistema SQL maturo e consolidato**, che facilita tooling, backup e migrazioni [44, 45].

In alternativa, analogamente a quanto analizzato nella Sezione 5.1.4, è stato considerato **Neo4j** per una persistenza nativa a grafo. Si tratta di un DBMS a grafo con linguaggio *Cypher* e algoritmi integrati per l'analisi di rete, ideali per grafi di grandi dimensioni[83].
Tuttavia, nel contesto in analisi: (i) il grafo indoor è compatto e le traversate sono servite efficacemente in memoria da `NetworkX`, evitando round-trip al

database; (ii) la componente spaziale è più matura nell'ecosistema PostGIS, cruciale per le operazioni di overlay pixel-based e conversione di coordinate; (iii) l'adozione di Neo4j introdurrebbe un doppio store o una migrazione completa, con maggiore complessità operativa e competenze specifiche non necessarie; (iv) l'attuale logica di tracciamento dello stato degli archi si basa su trigger e log di PostgreSQL, che dovrebbero essere replicati in Neo4j con nuove procedure [84, 85].

Pertanto, alla scala e ai requisiti del *Map Viewer*, la combinazione **PostgreSQL con PostGIS** per persistenza e auditing, abbinata a **NetworkX** per la gestione operativa del grafo in memoria, bilancia semplicità operativa, integrazione GIS e flessibilità evolutiva, in linea con le evidenze e le best practice presentate dallo stato dell'arte [44, 45, 78].

5.2.3 Libreria di visualizzazione: Leaflet, OpenLayers e Mapbox

Per la visualizzazione operativa del sistema è stata adottata la libreria **Leaflet.js** quale strumento principale per la gestione della planimetria di base e dei layer dinamici contenenti nodi e archi, forniti dal backend in formato JSON. Oltre a tale libreria, sono state confrontate altre soluzioni nella Sezione 5.1.5); ciononostante, la scelta di Leaflet è stata confermata da molteplici considerazioni di natura tecnica e architetturale:

1. **Semplicità e leggerezza**: Leaflet è una libreria open-source leggera, progettata per mappe interattive mobile-friendly, la cui API, intuitiva e priva di dipendenze, risulta ideale per overlay raster indoor e coordinate locali [79, 80, 81, 86].

2. **Interattività**: Leaflet fornisce un set esaustivo di funzionalità interattive essenziali, abilitando una fruizione fluida e reattiva della mappa indoor. Il supporto diretto per layer vettoriali dinamici e overlay di

immagini si rivela pienamente adeguato per sovrapporre i dati di navigazione del grafo in tempo reale, garantendo un'interfaccia utente intuitiva e responsiva [80, 86].

3. **Neutralità di vendor**: Leaflet è completamente open-source e priva di dipendenze da servizi proprietari. Non richiede l'impiego di chiavi API, token di accesso o infrastrutture esterne, offrendo così piena autonomia operativa e portabilità del microservizio anche in scenari offline o ambienti chiusi, requisito chiave già evidenziato nella Sezione 5.1.5, senza vincoli contrattuali o restrizioni d'uso imposte da provider terzi[79].

In fase di valutazione, sono state considerate anche le librerie **OpenLayers** e **Mapbox GL JS**, anch'esse consolidate soluzioni per cartografia web.
Come descritto nella Sezione 5.1.5, **OpenLayers** è stato scartato poiché la potenza e la ricchezza di funzionalità avanzate si traducono in una complessità maggiore, non giustificata per mappe indoor semplici.
Analogamente, **Mapbox GL JS**, seppur efficiente nel rendering vettoriale, richiede un account e token Mapbox e introduce un modello *tile-first* non perfettamente allineato a una planimetria locale non georeferenziata, risultando meno adatto al contesto in analisi [74, 75, 87]).

5.3 Sviluppo operativo del microservizio

La presente sezione funge da documentazione operativa del microservizio *Map Viewer*. Ne descrive avvio e configurazione, API esposte, funzionamento dell'interfaccia web, responsabilità dei moduli e gestione dei dati tra memoria e database, in maniera tale da costituire una guida all'uso e alla manutenzione del software.

Il microservizio offre:

- Una **web-app** per visualizzare le planimetrie e il grafo di navigazione di ogni piano, con funzioni di editing leggero e visualizzazione delle posizioni utente.
- Una **API REST** per elencare le planimetrie, esportare e ricaricare il grafo, creare nodi e archi, disabilitare archi e mostrare le posizioni degli utenti, aggiornate in *near real-time*.
- Una **cache in memoria** del grafo per piano, allineata alla persistenza su PostgreSQL e gestita da un componente dedicato.

In sintesi, *Map Viewer* consente di: (i) elencare le planimetrie disponibili e caricarne il grafo; (ii) aggiungere nodi e archi direttamente dalla UI con persistenza immediata; (iii) disabilitare archi non praticabili con audit automatico; (iv) visualizzare le posizioni utente aggiornate a intervalli regolari.
La soluzione è costruita su FastAPI (backend), Leaflet (frontend), NetworkX (grafi in memoria) e PostgreSQL (persistence), con un modulo di *height mapping* per la conversione coerente tra pixel, unità di modello e metri reali.

5.3.1 Progettazione dell'architettura

Il microservizio è stato progettato come componente indipendente che dialoga con altri servizi tramite API RESTful; tale architettura separa responsabilità e cicli di vita. Il **backend**, implementato in Python con il framework FastAPI, si occupa di elaborare i dati del grafo di navigazione generato con NetworkX, di fornire endpoint per la trasmissione dei dati al frontend e di coordinare letture/scritture su database. Il **frontend** è una single-page basata su JavaScript e Leaflet, che gestisce la visualizzazione interattiva della mappa e l'aggiornamento in tempo reale delle posizioni degli utenti. La persistenza è affidata a PostgreSQL con schema e trigger per occupazione nodi e audit. Le costanti di configurazione (DB, tipi di nodo, fattori di scala) sono centralizzate.
Questa impostazione facilita l'evoluzione indipendente dei componenti e consente di bilanciare reattività (cache) e consistenza (DB).

5.3.2 Implementazione del backend

Il backend è implementato in Python con FastAPI e utilizza **NetworkX** per rappresentare, in memoria, il grafo di navigazione per ciascun piano. I nodi corrispondono a aree funzionali abitabili, mentre gli archi modellano i collegamenti fisici; per ciascun arco vengono stimati capacità e tempo di percorrenza a partire dalle distanze su mappa, convertite in metri tramite una componente di **height/scale mapping**. La serializzazione verso il frontend avviene in un JSON *ad hoc* pensato per la UI, che include immagine di riferimento, dimensioni e liste di nodi/archi filtrate per piano. Inoltre, il backend integra un sistema di localizzazione, che aggiorna dinamicamente le posizioni degli utenti sul grafo.

A livello concettuale, il backend realizza tre responsabilità principali:

1. **Esposizione di risorse applicative**: l'API pubblica risorse per:
   - scoprire le planimetrie disponibili;
   - ottenere lo stato del grafo per un piano (nodi e archi attivi) in un JSON pronto all'uso per il frontend;
   - apportare modifiche mirate: creare nodi, collegare nodi con archi, disabilitare un arco non praticabile;
   - recuperare le posizioni utente da un servizio esterno e arricchirle con una stima del piano, in modo che la UI possa filtrare e rappresentare correttamente i marker.
2. **Gestione del grafo in memoria**: per ogni piano è mantenuta una rappresentazione in memoria del grafo di navigazione. Ciò consente:
   - tempi di risposta prevedibili nella consultazione del grafo;
   - applicazione immediata delle modifiche (aggiunta di nodi/archi, cambio di stato) senza dover ricaricare l'intero dataset;
   - ricarico controllato su richiesta, per riallineare la cache dopo operazioni massicce lato database.

In particolare, è il **Graph Manager** il modulo responsabile della gestione della rappresentazione astratta dell'edificio. Infatti, costruisce e mantiene in memoria, per ciascun piano, il grafo operativo caricando dal database i nodi e gli archi correlati; a ciascun elemento associa gli attributi utili alla logica e al rendering (tipologia, capacità e occupazione per i nodi; stato di attività e tempo di attraversamento per gli archi), cosicchè le interrogazioni della UI trovino già tutto pronto e coerente. Quando si aggiunge un nodo dalla UI, il servizio verifica innanzitutto che non esistano duplicati troppo vicini, quindi determina l'intervallo verticale [*z1, z2*] del piano in centimetri di modello, persiste il nuovo record in tabella e aggiorna contestualmente la cache in memoria. L'aggiunta di un arco segue la stessa filosofia: il sistema misura la distanza in pixel tra i centri dei due nodi, la converte in metri e, sulla base di semplici ipotesi di attraversabilità, stima una capacità indicativa e il relativo tempo di percorrenza; dopodiché registra l'arco nel database e lo inserisce nel grafo in memoria. A garantire coerenza tra rappresentazione grafica e grandezze fisiche interviene l'utility **HeightMapper**, che centralizza tutte le conversioni: dai pixel ai centimetri di modello e quindi ai metri reali, nonché il *mapping* tra numero di piano e intervallo di quota. In questo modo, ogni calcolo si appoggia a un'unica sorgente di verità metrica, evitando discrepanze tra componenti diverse del sistema.

3. **Ciclo di vita e coerenza dati**: all'avvio, il servizio prepara lo schema dati e popola la cache con i grafi presenti, garantendo che la UI, al primo caricamento, possa mostrare un quadro completo. Operazioni potenzialmente ad alta frequenza e ad elevato volume, come la gestione dello storico posizioni, sono normalizzate in fase di *bootstrap* per partire da uno stato pulito; in esercizio, gli aggiornamenti di occupazione e l'audit degli archi sono demandati al database per affidabilità e tracciabilità.

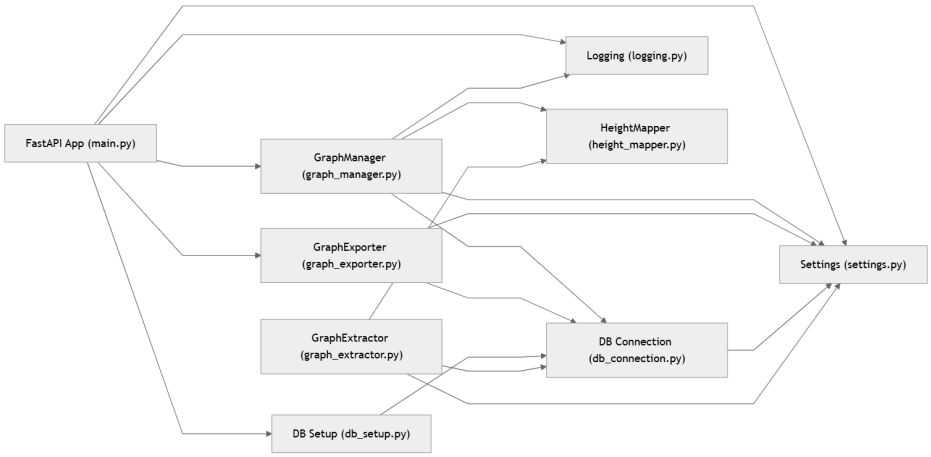

Figura 5.1: Diagramma UML dei componenti del backend di *Map Viewer*.

Il diagramma in Figura 5.1 illustra le componenti backend del *Map Viewer*, le loro interfacce fornite/richieste e le dipendenze applicative.

FastAPI App (main.py): è il punto d'ingresso del servizio. Espone l'interfaccia *REST API* per la UI e l'endpoint di *serving* dei contenuti statici. Orchestra le operazioni ad alto livello delegando la logica di grafo al *GraphManager*, l'esportazione dei dati al *GraphExporter*, l'inizializzazione dello schema a *DB Setup* e attingendo a *Settings* e *Logging* per configurazione e osservabilità.

GraphManager (graph_manager.py): mantiene in memoria (per piano) il grafo di navigazione basato su NetworkX. Carica nodi e archi dal database, applica mutazioni incrementali (creazione nodi/archi, disattivazioni) e tiene allineata la cache con la persistenza. Per stime coerenti di distanza, capacità e tempi di percorrenza dipende da *HeightMapper* e dai parametri in *Settings*. Registra eventi e anomalie tramite *Logging*.

GraphExporter (graph_exporter.py): fornisce un servizio di lettura

ed esportazione del grafo di un piano in un JSON semplice e direttamente consumabile dalla UI. Interroga il database attraverso *DB Connection* e utilizza *Settings* per i metadati necessari (immagini, dimensioni, configurazioni).

HeightMapper (height_mapper.py): è il punto unico per le conversioni fisiche: pixel → centimetri di modello → metri, e mappatura piano → intervallo di quota. Garantisce che tempi di attraversamento, capacità e stima del piano per le posizioni utente siano calcolati con criteri uniformi in tutto il sistema.

DB Setup (db_setup.py): inizializza e mantiene lo schema del database (tabelle per nodi, archi, posizioni, audit) e i trigger/funzioni per audit e aggiornamenti incrementali dell'occupazione. Si appoggia a *DB Connection* per l'accesso transazionale.

DB Connection (db_connection.py): incapsula la creazione e la gestione delle connessioni a PostgreSQL, fornendo un punto di accesso stabile e centralizzato ai dati per le componenti che leggono e scrivono su DB.

Settings (settings.py): raccoglie la configurazione centrale, cioè parametri di connessione al DB, tipi di nodo e capacità di default, fattori di scala e mappatura verticale (altezza per piano). È la fonte di verità per i moduli che necessitano di valori operativi condivisi.

Logging (logging.py): fornisce la strumentazione di log per modulo, con gestione di handler e formattazione coerente, utile alla diagnostica e al monitoraggio in esercizio.

GraphExtractor (graph_extractor.py): strumento di bootstrap/import. Consente l'inserimento di nodi e archi nel DB, includendo la gestione di nodi multi–piano. Dipende da *DB Connection*, *HeightMapper* e *Settings*.

La progettazione aderisce al *Single Responsibility Principle*: ogni componente ha una responsabilità ben delimitata, così da favorire riuso ed evoluzione indipendente. L'obiettivo è chiarire i confini di responsabilità e definire i contratti d'interfaccia espliciti tra i moduli, mantenendo una vista architetturale stabile e indipendente dai dettagli di implementazione.

5.3.3 Sviluppo del frontend

Il frontend è stato implementato utilizzando Leaflet.js per caricare la planimetria dell'edificio come immagine statica, sulla quale vengono sovrapposti i nodi e gli archi del grafo. Un layer interattivo consente agli utenti di interagire con la mappa tramite zoom e pan. La comunicazione con il backend avviene tramite chiamate AJAX per aggiornare i dati in tempo reale, garantendo una visualizzazione fluida anche in presenza di un elevato numero di utenti.

Il front-end adotta un modello semplice e robusto:

- **Rendering per piani**: per ciascuna immagine di planimetria viene istanziata una mappa 2D in cui i pixel dell'immagine fungono da sistema di riferimento: la sovrapposizione di nodi e archi risulta così intuitiva e fedele alla pianta.

- **Strati funzionali**: nodi, archi e utenti sono resi su layer separati per aggiornamenti indipendenti, quale il refresh frequente dei soli utenti.

- **Editing guidato**: l'utente può:

  - Collegare due nodi per creare un arco.
  - Selezionare un arco per disabilitarlo, simulando un passaggio inagibile.

- **Aggiornamenti frequenti ma controllati**: le posizioni degli utenti vengono richieste periodicamente. Il ritmo di aggiornamento è configurato per bilanciare reattività e carico, mantenendo fluida la rappresentazione.

- **Feedback visivo**: i nodi cambiano dimensione e colore in funzione del rapporto occupazione/capacità, fornendo una percezione immediata dello stato, mentre gli archi disabilitati scompaiono dal tracciato praticabile.

In conclusione, il frontend realizza una vista fedele e reattiva dell'edificio: immagini di piano come base, un grafo reso e modificabile in modo controllato e posizioni utente aggiornate periodicamente. La progettazione privilegia semplicità, separazione delle responsabilità tra layer e un contratto chiaro con le API del backend, facilitando manutenzione ed estensioni future.

5.3.4 Flusso dei dati

Il flusso operativo del *Map Viewer* descrive come i dati scorrono tra UI, API, cache in memoria e database lungo l'intero ciclo di vita del servizio.

1. **Bootstrap e preload**: all'avvio, il backend crea o aggiorna lo schema dati, inizializza tabelle e trigger, quindi procede a precaricare in memoria i grafi per piano a partire dal database. Per garantire uno stato iniziale consistente, il microservizio scarica lo storico delle posizioni su file, azzera le tabelle relative alle posizioni correnti e storiche e ripristina le occupazioni dei nodi. Il processo di inizializzazione e precaricamento descritto è rappresentato nel diagramma di sequenza in Figura 5.2, che illustra le azioni orchestrate tra backend e database.

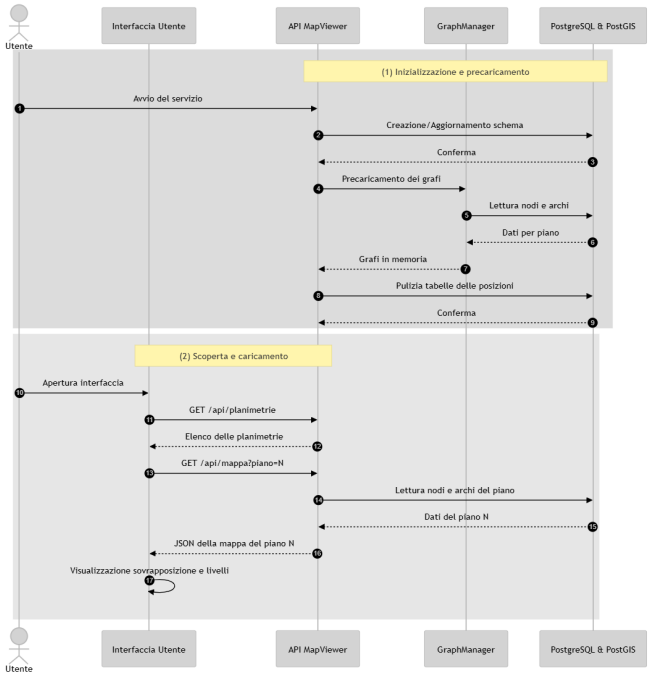

Figura 5.2: Diagramma di sequenza — (1) Inizializzazione e precaricamento; (2) Scoperta e caricamento.

2. **Ricerca e caricamento di mappe e grafi**: l'interfaccia utente interroga le API per ottenere l'elenco delle planimetrie disponibili. Per ciascun piano, viene richiesto un file JSON contenente i metadati dell'immagine, i nodi e gli archi attivi. Sul client, la planimetria è resa come *image overlay*, con i layer vettoriali (nodi/archi) sovrapposti nel sistema di coordinate locale. Le sequenze di richieste e risposte che consentono il caricamento e la composizione della vista sono dettagliate in Figura 5.2.

3. **Aggiornamento delle posizioni (near real-time)**: a intervalli regolari, la UI richiede gli aggiornamenti delle posizioni. Il backend integra i dati esterni, stima il piano di appartenenza attraverso conversioni pixel→metri→piano e restituisce i dati aggiornati. L'interfaccia utente aggiorna esclusivamente il layer dei marker, preservando la stabilità del-

la scena grafica. Il ciclo di aggiornamento e refresh dei dati di posizione è rappresentato nel diagramma di sequenza in Figura 5.3.

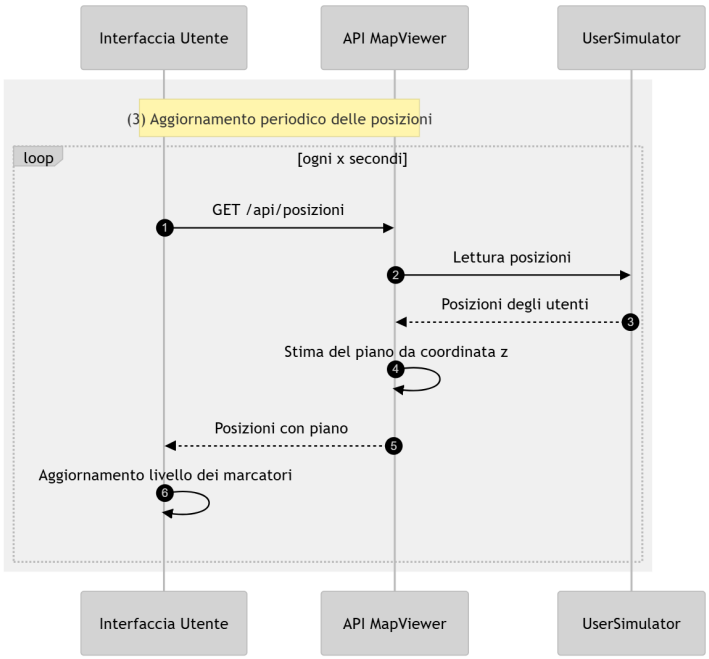

Figura 5.3: Diagramma di sequenza — (3) Aggiornamento periodico delle posizioni.

4. **Modifica del grafo con persistenza immediata**:

   - **Aggiunta di un nodo**: l'interfaccia utente trasmette posizione e tipo del nodo; il backend valida i dati per prevenire duplicati, calcola le grandezze metriche, aggiorna il database e la cache, quindi restituisce il nodo all'interfaccia utente per il rendering.
   - **Aggiunta di un arco**: l'interfaccia utente seleziona due nodi; il backend calcola distanza e tempi di percorrenza, persiste l'arco nel database, aggiorna la cache e notifica l'esito all'interfaccia utente, che rende visibile la polilinea.

- **Disabilitazione di un arco**: l'interfaccia utente richiede la disattivazione; il backend aggiorna lo stato nel database, e l'interfaccia utente rilegge il grafo del piano per riflettere la modifica.

L'intero processo di modifica interattiva del grafo, con la sincronizzazione immediata tra UI, backend, cache e database, è descritto nel dettaglio dal diagramma di sequenza in Figura 5.4.

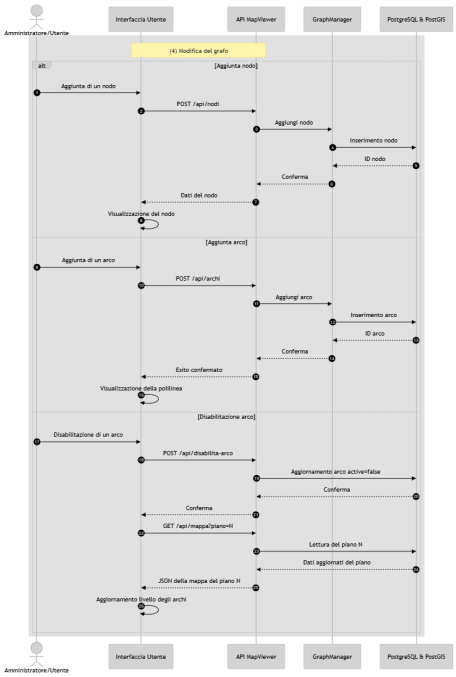

Figura 5.4: Diagramma di sequenza — (4) Modifica del grafo.

5. **Riallineamento cache–database**: su richiesta, le API ricostruiscono la cache in memoria a partire dai dati persistenti nel database, garantendo la coerenza dello stato. Le interazioni e i passaggi di questa procedura sono rappresentati nella Figura 5.5.

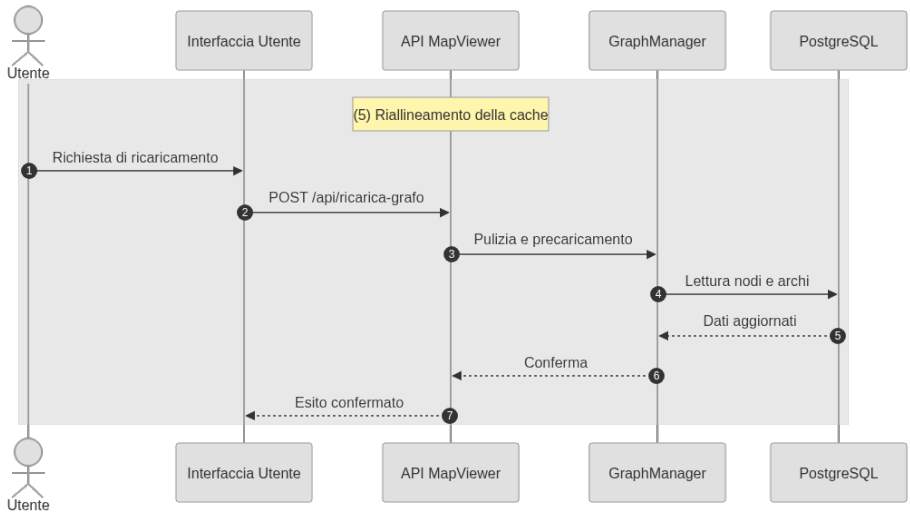

Figura 5.5: Diagramma di sequenza — (5) Riallineamento della cache.

In sintesi, il *Map Viewer* realizza un'architettura integrata e coerente tra interfaccia utente, API e database per la gestione di planimetrie e grafi indoor. L'utente beneficia di un'esperienza di consultazione e modifica fluida e immediata, mentre la coerenza dello stato è assicurata dalla combinazione di cache per piano e persistenza transazionale, supportata da meccanismi reattivi lato database che garantiscono affidabilità e tracciabilità.

Capitolo 6

Implementazione: microservizio Gestore della Mappa

La gestione ottimale delle vie di evacuazione in un edificio richiede un sistema altamente adattivo, capace di reagire in tempo reale a condizioni in continua evoluzione. In questo scenario, il microservizio *Map Manager* si posiziona come un componente strategico all'interno della pipeline di evacuazione, responsabile del calcolo dinamico e reattivo dei percorsi di fuga. Nello specifico, il **Gestore della Mappa** elabora itinerari di evacuazione ottimali in risposta a eventi di emergenza, integrando variabili critiche quali la capacità dei nodi e la disponibilità degli archi. Pertanto, anziché affidarsi a percorsi predefiniti, il sistema genera dinamicamente vie di fuga che riflettono la topologia attuale e le condizioni contingenti dell'edificio. Ciò garantisce soluzioni di evacuazione pertinenti e resilienti, in grado di adattarsi con efficacia all'evoluzione delle circostanze durante un'emergenza.

Le sezioni successive approfondiscono lo stato dell'arte, le scelte tecnologiche e i dettagli implementativi, con un'attenzione specifica all'integrazione con l'ecosistema tecnologico discusso nel Capitolo 3.

6.1 Analisi dello stato dell'arte

La gestione delle evacuazioni in scenari di emergenza si configura come un problema complesso che si colloca all'intersezione di diverse discipline: la teoria dei grafi per la modellazione spaziale, i sistemi real-time per l'acquisizione di dati e l'ottimizzazione combinatoria per la pianificazione dei flussi. L'approccio adottato dal *Map Manager* è il risultato di un'analisi critica delle principali metodologie esistenti, posizionandosi come una soluzione pragmatica che bilancia l'ottimalità teorica con i requisiti di reattività e affidabilità propri di un sistema critico.

6.1.1 Approcci basati su ricalcolo reattivo su grafo

Il paradigma più diffuso per il pathfinding dinamico in ambienti indoor si fonda sulla modellazione dell'edificio come un grafo, dove i nodi rappresentano luoghi significativi e gli archi le connessioni percorribili. In questo contesto, un'emergenza viene tradotta in una modifica dinamica della topologia o dei pesi del grafo: percorsi precedentemente validi possono diventare inagibili (rimozione di nodi o archi) o più lenti (aumento del peso degli archi). Questa è la filosofia operativa adottata dal Gestore della Mappa. Il sistema risponde agli eventi in modo reattivo:

1. **Ricezione dell'evento**: il sistema riceve una notifica di allerta.
2. **Modifica dello stato del grafo**: sulla base di regole predefinite, il sistema aggiorna lo stato dei componenti nel database, contrassegnando nodi come sicuri o non sicuri e disattivando archi.
3. **Ricalcolo del percorso**: quando viene richiesta un'evacuazione, l'algoritmo di pathfinding opera sul grafo filtrato, escludendo gli elementi non sicuri o inattivi e ricalcolando il cammino minimo.

Questo approccio, sebbene computazionalmente intensivo se ripetuto con alta frequenza, garantisce prevedibilità e affidabilità, poiché ogni calcolo si basa

sullo stato più aggiornato del sistema. La sua efficacia è massima in contesti con grafi di dimensioni contenute, come un singolo edificio, dove un ricalcolo completo può essere eseguito in frazioni di secondo. Un esempio di riferimento in questo ambito è **MazeMap** [88], una piattaforma commerciale di mappatura e navigazione indoor che fornisce mappe interattive e servizi di wayfinding per grandi complessi. La tecnologia adottata si basa sulla rappresentazione a grafo degli edifici per calcolare percorsi dinamici, che possono essere aggiornati in tempo reale per riflettere chiusure o eventi temporanei. Applicativi come MazeMap utilizzano un principio simile al *Map Manager*, ma con un focus differente: MazeMap è una soluzione user-facing che guida l'individuo tramite un'applicazione. Il *Map Manager*, al contrario, agisce come un servizio di backend orchestratore: non interagisce direttamente con l'utente finale, ma fornisce i percorsi ottimizzati come dato grezzo ad altri microservizi, che si occuperanno della loro diffusione collettiva.

6.1.2 Approcci basati su apprendimento e ottimizzazione globale

L'evacuazione può essere modellata come un problema di flusso dinamico su rete con capacità, un classico campo della ricerca operativa. In questo approccio, l'obiettivo non è semplicemente trovare il percorso più breve per un singolo individuo, ma minimizzare il tempo totale di evacuazione per l'intera popolazione, tenendo conto della capacità di archi e nodi per evitare congestioni. Questi modelli, spesso risolti con algoritmi complessi, offrono una soluzione teoricamente ottimale per la gestione della folla. Il *Map Manager* non implementa un risolutore di flusso globale, una scelta motivata dalla necessità di privilegiare la semplicità implementativa e la rapidità di calcolo in tempo reale. Tuttavia, ne recepisce i principi fondamentali in forma semplificata:

- **Capacità dei nodi**: il concetto di capacità non viene usato per ottimizzare un flusso, ma come vincolo rigido. I nodi che superano la ca-

pacità massima prestabilita vengono considerati sovraffollati e temporaneamente rimossi dal grafo prima del calcolo del percorso, simulando un blocco dovuto a congestione.

- **Capacità degli archi**: analogamente, la capacità di un arco è ridotta a uno stato binario: *attivo* o *inattivo*. Un arco non viene rallentato a causa del traffico, ma viene direttamente disattivato se le condizioni lo rendono impercorribile.

In questo modo, il *Map Manager* adotta un'euristica efficace che approssima il comportamento di un sistema a capacità limitata senza incorrere nell'onere computazionale dei modelli di flusso completi.

Inoltre, un filone di ricerca emergente utilizza tecniche di *Deep Reinforcement Learning* (DRL). Un esempio è **EvacuAI**[89], un sistema basato su apprendimento per rinforzo in cui un agente software impara la politica di evacuazione ottimale interagendo con un ambiente simulato. Attraverso un processo di "trial and error", l'agente viene premiato o penalizzato in base all'efficacia delle sue scelte, permettendogli di sviluppare strategie complesse che tengono conto di molteplici fattori senza essere esplicitamente programmate. Soluzioni come EvacuAI imparano attraverso migliaia di simulazioni, sviluppando policy complesse che possono superare le euristiche umane, specialmente in scenari con molteplici variabili interdipendenti. Sebbene promettente, questo approccio è stato scartato per il Map Manager per diverse ragioni strategiche:

1. **Complessità e addestramento**: i modelli DRL richiedono un ambiente di simulazione accurato e una fase di addestramento lunga e computazionalmente costosa.

2. **Interpretabilità**: le decisioni prese da un agente DRL possono essere difficili da interpretare, aspetto critico in un sistema di sicurezza dove la prevedibilità e la giustificazione di un percorso sono fondamentali.

3. **Adattabilità al contesto**: il sistema è progettato per operare su una topologia di edificio caricata dinamicamente all'avvio. Un modello

pre-addestrato potrebbe non generalizzare correttamente a una nuova planimetria senza un riaddestramento specifico.

In sintesi, il *Map Manager* si posiziona deliberatamente come un sistema deterministico e reattivo. Utilizza un algoritmo di pathfinding classico e ben compreso (Dijkstra) su un grafo che viene dinamicamente alterato per riflettere lo stato dell'emergenza. Questa scelta sacrifica l'ottimalità globale dei modelli di flusso e l'adattabilità avanzata dei sistemi di apprendimento in favore di velocità, affidabilità e trasparenza, qualità ritenute prioritarie per un microservizio il cui scopo è fornire risposte rapide e sicure in condizioni critiche.

6.2 Analisi delle alternative: motivazioni delle scelte

La fase di progettazione del Gestore della Mappa ha richiesto un'analisi critica di molteplici soluzioni, le cui alternative sono state ponderate e discusse prima di definire l'implementazione definitiva. Le sezioni successive illustrano nel dettaglio le ragioni che hanno guidato tali decisioni.

6.2.1 Modellazione dei nodi *stairs*

Per una corretta rappresentazione della connettività verticale all'interno del grafo, è stata scartata l'ipotesi preliminare di modellare i vani scala multipiano tramite un singolo nodo aggregatore. Sebbene concettualmente compatta, tale soluzione introduceva artefatti topologici, generando archi spuri che collegavano direttamente anche piani non contigui e compromettendo così l'accuratezza del calcolo dei percorsi.

Si è quindi implementato un modello più rigoroso, in cui ogni vano scala viene discretizzato in un nodo specifico per piano. La continuità verticale è garantita da archi che collegano esclusivamente coppie di nodi corrispondenti a piani fisicamente adiacenti. Tale rappresentazione assicura che qualsia-

si percorso calcolato dal pathfinding rifletta fedelmente la sequenzialità dei collegamenti reali, escludendo a priori la possibilità di scorciatoie verticali anomale.

6.2.2 Funzionalità GIS integrate nel database

In fase di prototipazione, è stata analizzata la possibilità di sfruttare le funzionalità geospaziali offerte da PostgreSQL/PostGIS per il calcolo dei percorsi di evacuazione, delegando tale logica direttamente al database. Questa soluzione, sebbene tecnicamente interessante, è stata scartata a favore di un'implementazione in Python, guidata dalle seguenti considerazioni pratiche e architetturali.

1. **Sistema di coordinate non georeferenziato**: l'ambiente di sviluppo utilizza un sistema di coordinate locale e astratto per la mappa interna dell'edificio, non riferito a coordinate geografiche reali, per le quali le funzionalità GIS sono ottimizzate.

2. **Complessità architetturale**: delegare il calcolo dei percorsi al livello dati avrebbe introdotto una complessità non necessaria all'architettura complessiva del sistema. Inoltre, avrebbe potuto comportare potenziali problematiche di performance sotto carico elevato.

3. **Maggiore controllo e flessibilità**: l'implementazione di un algoritmo di pathfinding nell'applicativo Python ha garantito un controllo superiore sulle ottimizzazioni specifiche del dominio, come l'esclusione di nodi sovraffollati o la gestione di archi temporaneamente disattivati. L'integrazione di queste logiche personalizzate in query SQL/GIS sarebbe risultata macchinosa, mentre un algoritmo *ad hoc* in Python ha consentito l'applicazione di filtri e criteri aggiuntivi in modo più agevole prima e durante il calcolo del percorso. Infatti, caricando il grafo una sola volta dal database nella memoria, si eliminano i frequenti accessi al disco che sono molto lenti. Ogni volta che l'algoritmo deve attraversare un nodo o un arco per calcolare il percorso, l'informazione è già

pronta e disponibile per l'uso immediato, il che accelera drasticamente il processo di ricerca.

6.2.3 Percorsi di evacuazione predefiniti

A differenza dei sistemi tradizionali che si appoggiano su itinerari fissi, il Gestore della Mappa abbandona l'approccio convenzionale di percorsi di evacuazione pre-calcolati e archiviati in file di configurazione statici. Questa decisione progettuale non è arbitraria, ma costituisce una diretta conseguenza della filosofia di design del sistema stesso. Infatti, il grafo su cui il microservizio opera non è un insieme di dati predeterminato, bensì viene costruito dinamicamente al primo avvio del sistema. Questa metodologia si discosta marcatamente dalle soluzioni tradizionali, che si appoggiano su itinerari fissi definiti a priori in planimetrie statiche. Poiché la struttura topologica dell'edificio, modellata tramite nodi e archi, non è nota in fase di sviluppo, ma viene definita solo durante l'inizializzazione, risulta impraticabile preimpostare e memorizzare percorsi standard. Di conseguenza, il sistema è progettato per generare i percorsi di evacuazione on-demand: al primo avvio, il microservizio calcola un insieme di percorsi di evacuazione standard basati sulla topologia iniziale del grafo. In caso di emergenza, l'algoritmo di pathfinding non si limita a richiamare tale percorsi precalcolati, ma computa l'itinerario in tempo reale, tenendo conto sia della topologia costruita dinamicamente che delle condizioni attuali dell'edificio. Questa notevole flessibilità assicura che i percorsi individuati siano sempre ottimizzati e adattati alla situazione contingente, conferendo al sistema una superiore resilienza e un maggior grado di sicurezza in situazioni di emergenza.

6.2.4 Coordinamento tra microservizi: gestione della race condition

Un'architettura a microservizi basata su messaggistica asincrona richiede particolare attenzione al coordinamento temporale delle operazioni per evi-

tare condizioni di race. Nel contesto del *Map Manager*, si è individuato un potenziale problema di concorrenza: il rischio che il *Position Manager* legga dal database percorsi di evacuazione non ancora aggiornati, utilizzando itinerari obsoleti mentre il *Map Manager* sta ancora calcolando quelli nuovi. Questo scenario può verificarsi a causa della natura asincrona del flusso di eventi tra microservizi nella pipeline di emergenza.

Per garantire che il *Position Manager* utilizzi percorsi aggiornati, è stato implementato un meccanismo di handshake esplicito tramite RabbitMQ. Nello specifico, al termine del calcolo, il Map Manager pubblica un messaggio di conferma su una coda dedicata. Tale messaggio funge da segnale di completamento: il *Position Manager* si pone in ascolto su tale coda e, solo dopo aver ricevuto la notifica, procede a leggere i percorsi dal database per inoltrarli. Questo approccio garantisce un allineamento temporale tra i microservizi, in cui il *Position Manager* non può precedere il *Map Manager*, ma al contrario reagisce al suo segnale, assicurandosi di operare su dati consistenti e aggiornati. La scelta di implementare questo coordinamento tramite un messaggio di acknowledgment su coda RabbitMQ è motivata da esigenze di coerenza e affidabilità, in quanto le soluzioni alternative sarebbero risultate fragili e inefficienti. Un ritardo arbitrario, ad esempio, non garantirebbe il corretto funzionamento sotto carichi variabili, mentre un intenso polling graverebbe inutilmente sul database, con il rischio di leggere comunque dati non ancora consistenti. Inoltre, un approccio sincrono con chiamate dirette tra microservizi sarebbe in contrasto con il principio di disaccoppiamento dell'architettura, introducendo potenziali colli di bottiglia. L'handshake basato su eventi, invece, mantiene i servizi loosely coupled e reattivi, orchestrandoli in modo da prevenire le condizioni di race e rafforzare l'affidabilità complessiva del sistema di evacuazione.
Il corretto flusso operativo del meccanismo implementato è illustrato in dettaglio dal diagramma di sequenza in Figura 6.1:

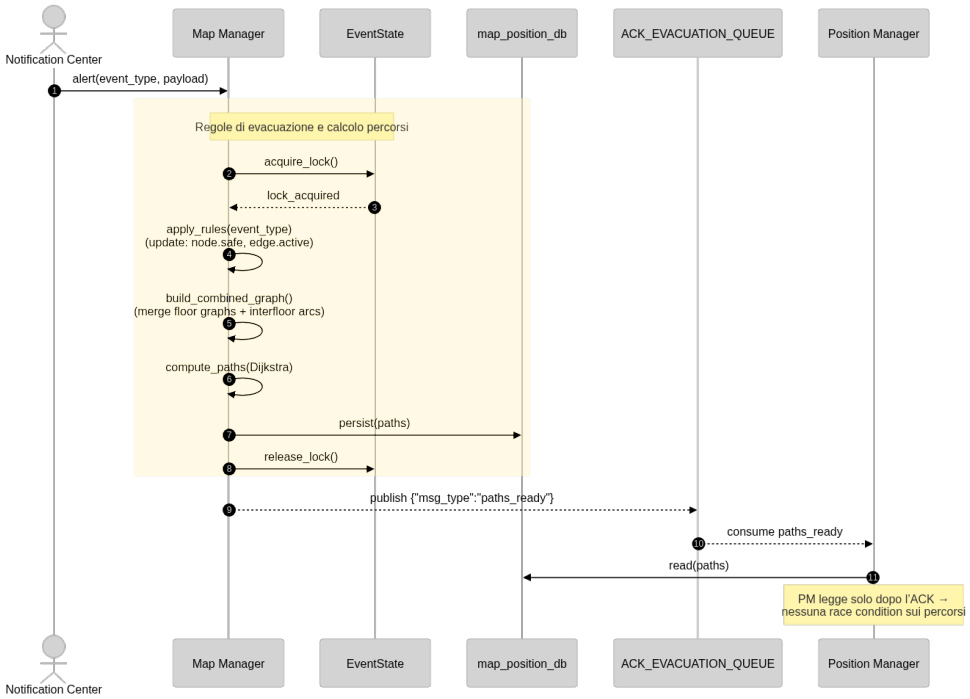

Figura 6.1: Diagramma di sequenza: meccanismo di *handshake* tra *Map Manager* e *Position Manager*

6.3 Sviluppo operativo del microservizio

Il microservizio *Map Manager* è stato sviluppato in Python seguendo i principi architetturali a microservizi, garantendo modularità, indipendenza e scalabilità. Ogni componente interno svolge una funzione specifica, facilitando la manutenzione e l'evoluzione del sistema.

6.3.1 Flusso operativo

Il flusso operativo del microservizio può essere descritto come segue:

1. **Ricezione dati**: il sistema riceve informazioni in tempo reale sulla posizione degli occupanti, sullo stato delle vie di fuga e sulla capacità dei nodi.

2. **Analisi del grafo**: si determinano quali nodi e archi sono disponibili e sicuri.

3. **Calcolo dei percorsi**: utilizzando algoritmi di pathfinding, il microservizio calcola i percorsi di evacuazione ottimali che conducono gli occupanti verso nodi sicuri, tenendo conto delle condizioni correnti.

4. **Aggiornamento dinamico**: il sistema continua a monitorare l'evoluzione dell'emergenza e aggiorna dinamicamente i percorsi calcolati qualora le condizioni mutino.

5. **Comunicazione**: i percorsi calcolati vengono persistiti sul database.

Ogni modulo del Gestore della Mappa è progettato per avere una singola responsabilità ben definita, in linea con i principi SOLID:

- **Modulo di acquisizione dati**: raccoglie informazioni in tempo reale sulla posizione degli occupanti e sullo stato delle vie di fuga.

- **Modulo di gestione del grafo**: si occupa di impostare gli attributi, quali flag *safe/non safe* per i nodi, stato *attivo/inattivo* per gli archi e tempi di attraversamento di una via di fuga.

- **Modulo di calcolo dei percorsi**: implementa l'algoritmo di pathfinding Dijkstra per determinare il percorso di evacuazione ottimale da ciascun nodo occupato verso un'uscita sicura.

- **Modulo di aggiornamento dinamico**: gestisce le modifiche in tempo reale: disattiva archi qualora diventino impraticabili, aggiorna i pesi degli archi se mutano le condizioni e innesca un ricalcolo immediato dei percorsi se necessario.

6.3.2 Consumatori di messaggi e architettura

Il *Map Manager* utilizza due principali consumer di messaggi RabbitMQ, ciascuno dedicato a un flusso di input:

1. **Alert di emergenza**: riceve notifiche di allarme. Le allerte e le relative regole di evacuazione sono configurate in un file YAML dedicato, che definisce per ogni tipologia di evento le azioni da compiere sulla mappa:

   - **Flood**: in caso di allagamento, l'evacuazione è **verticale**, privilegiando lo spostamento verso piani superiori qualora i livelli bassi risultino allagati o impraticabili. Il sistema contrassegna come non sicuri tutti i nodi situati nelle zone a quota inferiore potenzialmente esposte all'acqua, e designa come sicuri i nodi che consentono di transitare ai piani più alti.
   - **Earthquake**: in caso di terremoto, la strategia di evacuazione **orizzontale** verso l'esterno è prioritaria. Il sistema marca come non sicuri tutti i nodi interni all'edificio, data la potenziale instabilità strutturale, e considera sicuri i nodi di tipo outdoor (uscite verso spazi aperti).
   - **Fire**: in caso di incendio, l'evacuazione segue un modello *orizzontale* verso aree non coinvolte dal fuoco. Il sistema identifica una *danger zone* sulla base dei dati dell'allarme antincendio e imposta tutti i nodi in tale area come non sicuri. Vengono contestualmente marcati come sicuri tutti i nodi di tipo outdoor, assicurando che le uscite di emergenza all'esterno siano le destinazioni finali. Inoltre, tutti gli archi che attraversano la zona di pericolo vengono temporaneamente disattivati, cosicché l'algoritmo di pathfinding possa dirottare gli occupanti lungo vie alternative prive di fumo o fiamme.

   Quando il consumer di allerte riceve un messaggio di allarme con la tipologia di emergenza applica immediatamente le conseguenti restrizioni sulla mappa.

2. **Segnalazione di nodi pericolosi**: riceve messaggi contenenti elenchi aggregati di nodi pericolosi, prodotti dal microservizio *Position Mana-*

ger: quest'ultimo monitora la posizione degli occupanti e identifica, in seguito al rilevamento dell'allerta in corso, gli utenti in pericolo. Alla ricezione di tale messaggio, il consumer specifico raggruppa i nodi pericolosi per piano e avvia il calcolo dei percorsi di evacuazione per ciascun gruppo. Ciò innesca la logica di pathfinding per trovare il miglior itinerario di fuga da ogni nodo segnalato verso un'uscita sicura.

In un'architettura in cui due consumer elaborano flussi di dati distinti ma logicamente correlati, la preservazione di uno stato globale coerente dell'evento di emergenza assume un'importanza cruciale. La classe `EventState` assolve a tale funzione, agendo quale repository centralizzato e sincronizzato per lo stato dell'evento. La concorrenza degli accessi è regolata da un meccanismo di locking interno, concepito per prevenire fenomeni di race condition. Tale meccanismo garantisce l'atomicità delle operazioni di lettura e scrittura, evitando che un processo possa operare su dati inconsistenti durante la transizione di stato indotta da un altro processo. Pertanto, tale classe funge da unica fonte di verità (*Single Source of Truth*) per lo stato dell'emergenza all'interno del microservizio. Ciò garantisce la coerenza decisionale tra le varie componenti, scongiurando il rischio che un modulo operi sulla base di informazioni obsolete o parziali.

6.3.3 Calcolo dei percorsi di evacuazione

Il cuore del Gestore della Mappa risiede nell'algoritmo di calcolo dei percorsi ottimali verso l'uscita in condizioni di emergenza. Nell'implementazione corrente, si è scelto di utilizzare l'**algoritmo di Dijkstra** per trovare il percorso più breve da ciascun nodo occupato a un nodo di uscita sicuro. La procedura è la seguente:

1. Viene costruito dinamicamente un **grafo combinato multi-piano**, partendo dal grafo di ogni piano caricato in memoria ed aggiungendo anche gli archi inter-floor, cioè quelli che collegano tra loro i nodi di tipo stairs di piani adiacenti. Questo grafo combinato rappresenta

tutte le possibili connessioni percorribili nell'intero edificio al momento corrente.

2. Il grafo viene ottimizzato con l'esclusione temporanea dei nodi non sicuri o sovraffollati e degli archi inattivi, per assicurare che l'algoritmo operi solo su percorsi validi e sicuri.

3. Per ciascun nodo di partenza, l'algoritmo cerca il cammino minimo verso uno qualsiasi dei nodi marcati come sicuri. Se esistono più possibili uscite, l'algoritmo seleziona automaticamente quella che produce il percorso complessivamente più veloce o più breve in termini di peso, dove il peso di un percorso dipende dal tempo di percorrenza di un arco.

4. Il risultato del calcolo per ogni nodo in pericolo è un elenco ordinato di archi che costituiscono il percorso di evacuazione dal nodo fino a una uscita. Questi percorsi vengono quindi scritti nel database, associandoli ai rispettivi nodi di partenza.

L'utilizzo dell'algoritmo di Dijkstra è motivato dalla natura relativamente ridotta del grafo in confronto a grafi stradali su scala cittadina e dalla necessità di ricalcolare frequentemente i percorsi in tempo reale. Infatti, Dijkstra è semplice da implementare e, operando su grafi di poche migliaia di nodi, fornisce risultati in tempi molto brevi. Va sottolineato che, durante l'emergenza, l'aggiornamento dei percorsi può avvenire ripetutamente; il sistema è stato progettato per eseguire il ricalcolo in background in maniera concorrente senza bloccare la ricezione di nuovi eventi.

La pipeline di evacuazione è rappresentata dal diagramma in Figura 6.2:

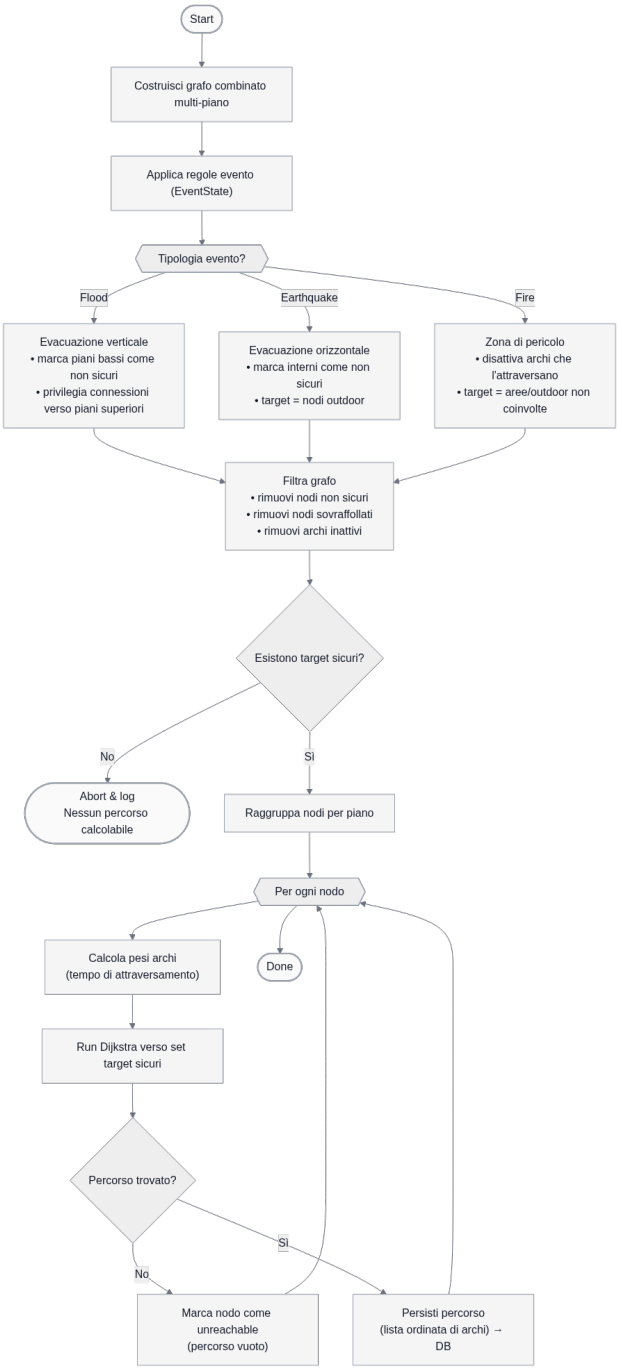

Figura 6.2: Flowchart: Pipeline di pathfinding

6.3.4 Inizializzazione e notifiche

All'avvio del microservizio, il *Map Manager* predispone una condizione di percorso di evacuazione per ogni nodo dell'edificio, anche in assenza di un'emergenza attiva, al fine di avere uno stato consistente. A causa dell'impossibilità di utilizzare percorsi di evacuazione standard basati sull'immagine della planimetria dell'edificio, dal momento che se ne utilizza una rappresentazione astratta semplificata tramite grafo, i percorsi di evacuazione di default vengono calcolati dal microservizio: questi vengono calcolati utilizzando come nodo di tipo safe un nodo all'esterno (tipo = *outdoor*). Per i nodi esterni, il Gestore della Mappa imposta un percorso di evacuazione vuoto a indicare che si è già in un luogo sicuro. Questo calcolo iniziale dei percorsi di default multi-piano viene effettuato sfruttando la stessa logica di path-finding descritta precedentemente ed è utile perché fornisce immediatamente un riferimento di evacuazione qualora un'emergenza partisse senza preavviso.

In conclusione, l'implementazione del Gestore della Mappa rappresenta una soluzione ingegneristica robusta e reattiva per il calcolo dinamico dei percorsi di evacuazione. Il suo design, basato su un approccio deterministico e una modellazione a grafo dell'edificio, offre un equilibrio ottimale tra prestazioni, affidabilità e trasparenza, qualità imprescindibili per un sistema di sicurezza. La scelta di non affidarsi a percorsi predefiniti, ma di generarli in tempo reale in base alle condizioni contingenti, conferisce al sistema una notevole flessibilità e resilienza. L'integrazione con l'architettura a microservizi, mediata da RabbitMQ, e l'adozione di meccanismi di coordinamento come l'handshake garantiscono che il *Map Manager* operi in modo sinergico con l'intero ecosistema tecnologico, fornendo percorsi di evacuazione tempestivi e sicuri. Questo microservizio si configura, quindi, come un pilastro fondamentale per la sicurezza degli occupanti durante situazioni di emergenza.

Capitolo 7

Risultati sperimentali

Il presente capitolo è dedicato all'esposizione e all'analisi critica dei risultati della validazione sperimentale, intrapresa al fine di verificare l'efficacia e la robustezza delle soluzioni architetturali e implementative descritte nei Capitoli 2–6. La campagna di test è stata concepita con l'obiettivo primario di quantificare le prestazioni della piattaforma rispetto a un insieme definito di metriche e obiettivi qualitativi cardine. I parametri di valutazione prioritari sono stati i seguenti:

- **Tempestività end-to-end**: quantificazione della latenza complessiva della pipeline di elaborazione, definita come l'intervallo temporale che intercorre tra l'acquisizione di un evento di allerta e la conseguente consegna della notifica di evacuazione all'utente finale.

- **Correttezza e sicurezza**: validazione della coerenza topologica e della sicurezza intrinseca dei percorsi di evacuazione generati, garantendo che questi escludano sistematicamente le zone identificate come inagibili o pericolose.

- **Robustezza**: analisi della resilienza della soluzione implementata e della sua capacità di adattamento a diverse tipologie di eventi emergenziali e a condizioni ambientali dinamiche, assicurando la generazione di strategie di evacuazione pertinenti e ottimali.

- **Coerenza dei dati**: verifica della sincronizzazione e dell'integrità dei dati tra lo stato persistente, mantenuto sul database, e la sua rappresentazione in-memory, cruciale per l'affidabilità delle decisioni in tempo reale.

- **Osservabilità**: valutazione dell'efficacia degli strumenti di monitoraggio nel fornire una chiara interpretabilità del comportamento dell'architettura in scenari operativi simulati, al fine di diagnosticarne lo stato e le prestazioni.

L'impianto sperimentale si fonda sulle seguenti assunzioni chiave: (i) l'adozione di un modello a grafo pre-validato, rappresentativo della topologia reale dell'edificio in esame; (ii) l'impiego di parametri di capacità dei percorsi e tempi di percorrenza coerenti con la scala metrica e le caratteristiche dell'ambiente; (iii) la generazione di stream di posizionamento con densità e dinamiche rappresentative di scenari di occupazione realistici.

7.1 Analisi del caso di studio

Il dominio applicativo selezionato per la validazione sperimentale è il **Campus di Cesena dell'Università di Bologna**. La complessa topologia dell'edificio è stata formalizzata mediante un grafo orientato e multi-piano, una struttura dati che permette una rappresentazione computazionalmente efficiente degli spazi interni e delle relative interconnessioni. Le planimetrie dei livelli che costituiscono l'edificio, su cui si basa la modellazione, sono illustrate nelle Figure 7.1, 7.2 e 7.3. Il grafo risultante si compone di un totale di 148 nodi e 343 archi.

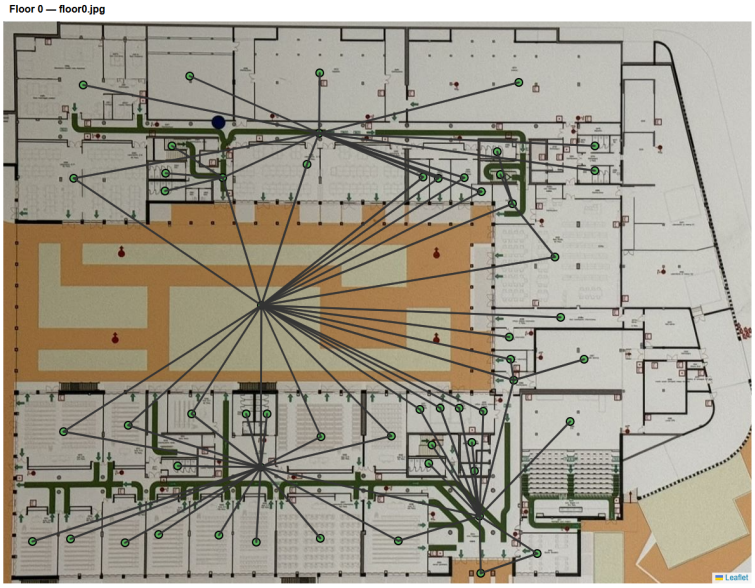

Figura 7.1: Piano 0 del Campus di Cesena

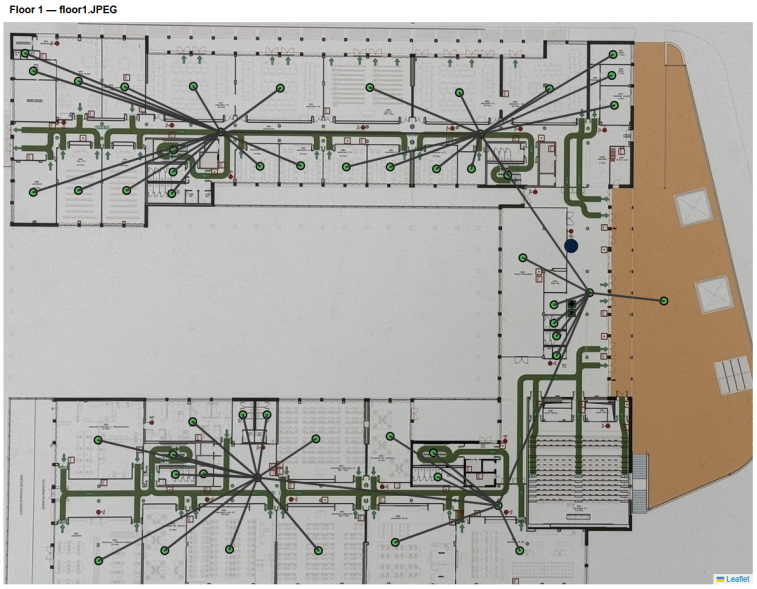

Figura 7.2: Piano 1 del Campus di Cesena

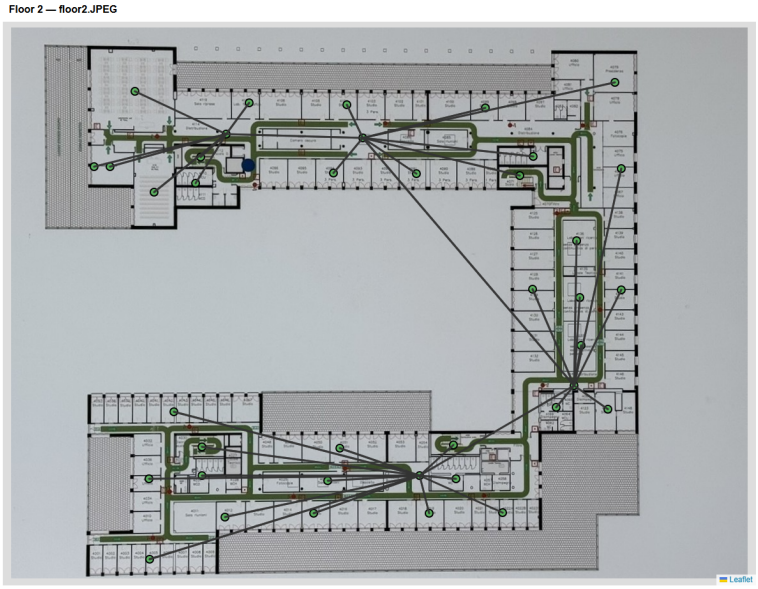

Figura 7.3: Piano 2 del Campus di Cesena

Il nucleo logico della soluzione implementata risiede nella sua capacità di applicare dinamicamente politiche di evacuazione specifiche per la tipologia di emergenza in corso. Tali politiche sono state classificate in base alla loro estensione e alla natura della minaccia:

Allerte a evacuazione totale: impongono l'evacuazione completa dell'edificio da parte di tutti gli occupanti. La strategia associata persegue l'obiettivo di minimizzare il tempo complessivo di sfollamento, guidando gli individui verso punti di raccolta esterni predefiniti. Il calcolo dei percorsi privilegia gli itinerari a costo minimo diretti verso le uscite di sicurezza.

Allerte a evacuazione parziale: interessano unicamente sottoinsiemi specifici della struttura. In questo caso, il piano di evacuazione mira a delocalizzare gli occupanti dalle aree a rischio verso zone sicure interne all'edificio stesso.

Le sezioni successive forniscono una disamina dettagliata dei dataset impiegati, degli scenari di simulazione implementati e dei protocolli di mi-

surazione adottati per la raccolta dei dati quantitativi, culminando nella discussione e interpretazione dei risultati ottenuti.

7.2 Descrizione dei dati

La validazione dell'efficacia del progetto è stata condotta mediante una rigorosa campagna sperimentale, articolata su due direttrici di analisi: una **qualitativa** e una **quantitativa**.

L'analisi *qualitativa* si è focalizzata sull'esame approfondito di scenari operativi specifici. Per un campione rappresentativo di 25 utenti, sono state acquisite e analizzate metriche fondamentali per la convalida funzionale del sistema, tra cui la latenza di notifica dell'allerta, la posizione iniziale degli utenti, i percorsi di evacuazione generati e l'esito del raggiungimento della destinazione sicura. Tale approccio ha permesso di verificare la coerenza topologica e la sicurezza intrinseca delle soluzioni di routing proposte.

L'analisi *quantitativa*, invece, ha avuto come obiettivo la misurazione delle prestazioni dell'applicazione al variare del carico, inteso come numero di utenti concorrenti. Sono state eseguiti test con diverse densità di occupazione simulata, raccogliendo dati sulle performance temporali e sull'efficienza complessiva del processo di evacuazione.

7.3 Validazione qualitativa: terremoto

Per valutare la capacità del sistema di gestire scenari di emergenza su larga scala, è stata condotta una simulazione dettagliata di un **terremoto**: esso impone un'**evacuazione totale**, che richiede a tutti gli occupanti dell'edificio di lasciare la struttura e dirigersi verso punti di raccolta esterni. L'obiettivo era verificare l'efficacia nella generazione di percorsi di fuga sicuri in un contesto di rischio strutturale diffuso.
Per una valutazione puntuale e rigorosa, la presente trattazione si concentrerà sull'analisi di un sottoinsieme rappresentativo di 5 utenti.

7.3.1 Posizioni iniziali e rilevamento del pericolo

La simulazione di un evento sismico ha attivato una politica di evacuazione globale, estesa a tutti gli utenti presenti, indipendentemente dalla loro posizione iniziale. Il sistema ha risposto all'allerta classificando come in pericolo tutti gli occupanti dell'edificio. Questa risposta su vasta scala ne dimostra la capacità di applicare politiche di emergenza non circoscritte a una specifica area, ma che coinvolgono l'intera topologia del grafo.

Come illustrato nelle Figure 7.4, 7.5 e 7.6, le posizioni iniziali degli utenti sono distribuite su tutti i livelli del campus.

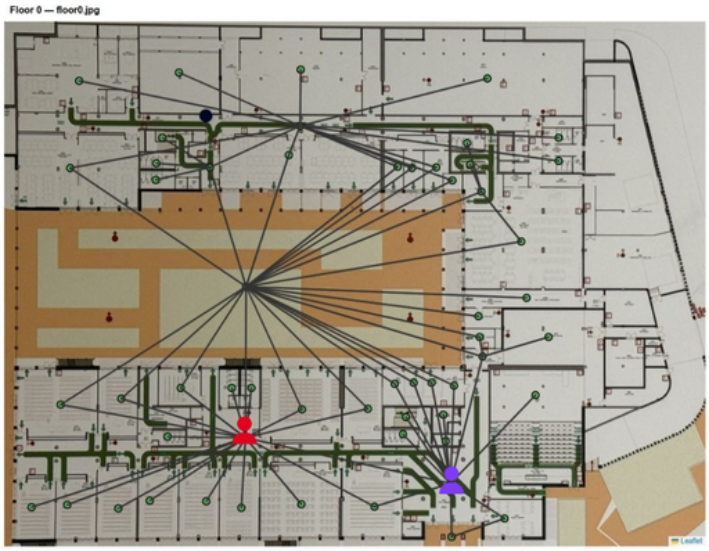

Figura 7.4: Posizioni iniziali prima dell'allerta: piano 0

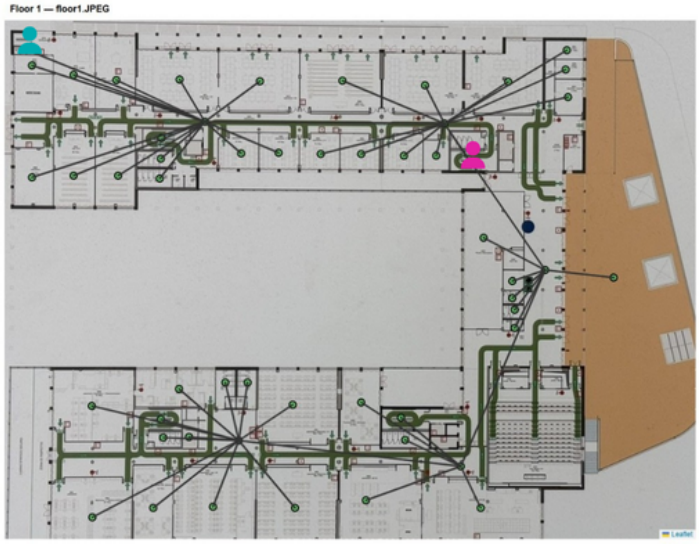

Figura 7.5: Posizioni iniziali prima dell'allerta: piano 1

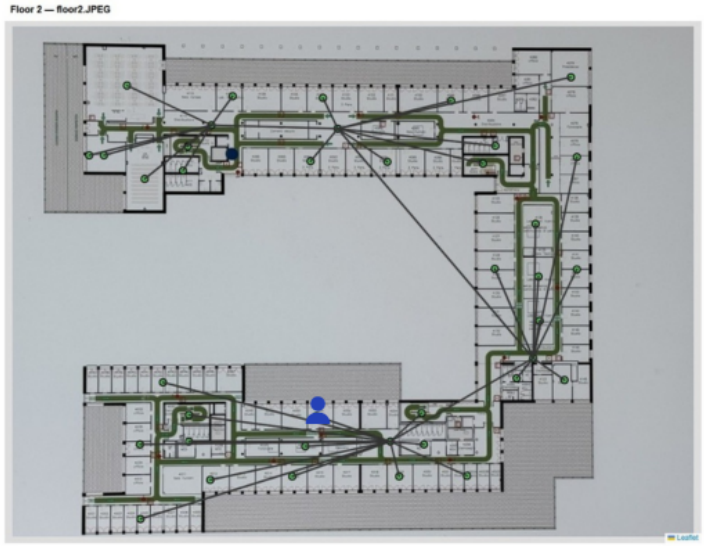

Figura 7.6: Posizioni iniziali prima dell'allerta: piano 2

A seguito dell'allerta, l'infrastruttura software ha notificato simultaneamente tutti gli utenti, confermando la sua capacità di discernere e gestire una minaccia su scala totale. In Figura 7.7 è riportato l'elenco dei nodi che presentano almeno un utente e i relativi percorsi di evacuazione. Tutti i nodi

interni, trattandosi di un'allerta terremoto, sono stati contrassegnati come non sicuri, indipendentemente dalla loro posizione all'interno dell'edificio.

	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 in pericolo durante l'allerta di tipo terremoto

7.3.2 Generazione dei percorsi di evacuazione

A seguito dell'identificazione dei nodi in pericolo, il motore di routing ha computato per ciascun nodo un percorso di evacuazione ottimale verso l'esterno: la strategia implementata ha privilegiato itinerari che minimizzano il tempo di permanenza all'interno della struttura, dimostrando la robustezza dell'algoritmo nel rispondere a una minaccia complessa e diffusa.

L'efficacia di tale strategia è stata confermata analizzando il flusso di spostamento degli utenti, come visualizzato per ogni piano nelle Figure 7.8, 7.9 e 7.10. A livello grafico, queste immagini offrono una rappresentazione chiara del processo, mostrando un movimento coordinato e convergente verso le uscite. Ciò comprova la validità dell'implementazione nel gestire un'evacuazione di massa e la corretta aderenza degli utenti alle direttive fornite.

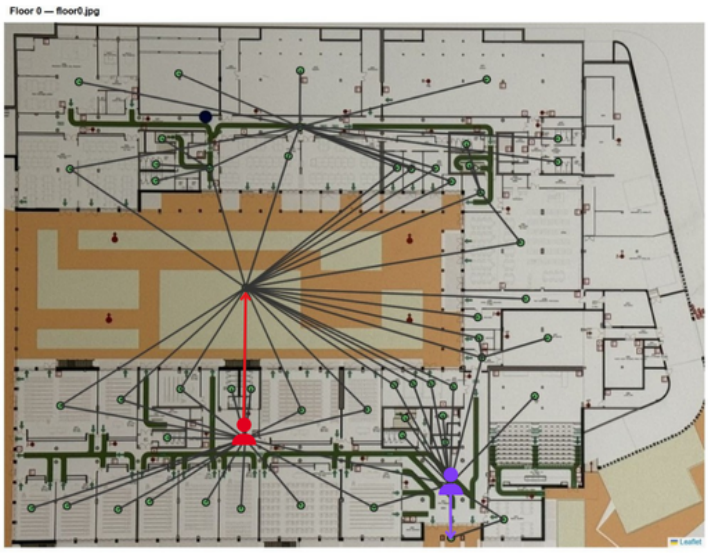

Figura 7.8: Flusso degli utenti in evacuazione: piano 0

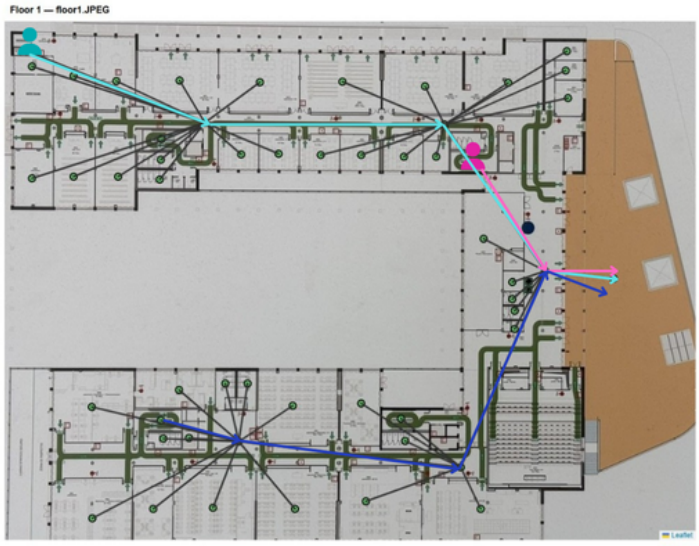

Figura 7.9: Flusso degli utenti in evacuazione: piano 1

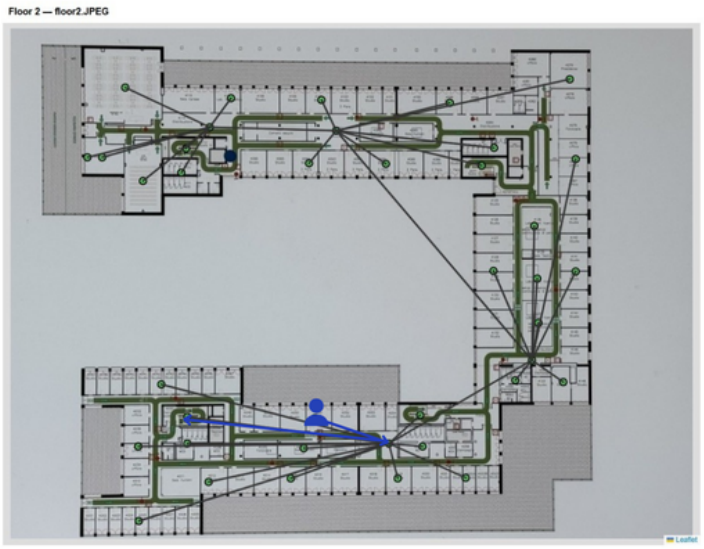

Figura 7.10: Flusso degli utenti in evacuazione: piano 2

7.3.3 Validazione dell'arrivo e visualizzazione del flusso

L'analisi del flusso degli utenti, presentata nelle sezioni precedenti, costituisce la prova conclusiva del successo dell'evacuazione. Il movimento coordinato e diretto verso le uscite di sicurezza, come documentato nelle mappe di flusso (Figure 7.8, 7.9 e 7.10), conferma che tutti gli occupanti in pericolo hanno seguito con successo i percorsi designati, raggiungendo le aree sicure all'esterno della struttura. Questo risultato convalida l'efficacia dell'intero sistema di evacuazione, dalla generazione dei percorsi alla guida in tempo reale degli utenti.

7.4 Validazione qualitativa: alluvione

Al fine di dimostrare l'efficacia del sistema in uno scenario di evacuazione parziale, è stata effettuata un'analisi di caso su una simulazione di **alluvione**, coinvolgendo un campione di 25 utenti. Per una valutazione puntuale e rigorosa, la presente trattazione si concentrerà sull'analisi di un sottoinsieme

rappresentativo di 5 utenti. L'obiettivo primario consisteva nel verificare la capacità dell'implementazione di identificare correttamente l'area di pericolo e di generare percorsi di evacuazione coerenti ed efficaci, guidando gli utenti dalle zone a rischio verso i punti di raccolta sicuri designati.

7.4.1 Posizioni iniziali e rilevamento del pericolo

La simulazione di un alluvione ha permesso di testare una funzionalità cruciale dell'implementazione: l'**identificazione selettiva degli utenti in pericolo**. Data la natura della minaccia, circoscritta al livello inferiore dell'edificio, unicamente gli utenti localizzati al piano 0 sono stati correttamente classificati come in pericolo. Questo approccio ha correttamente inibito la propagazione dell'allerta di evacuazione agli utenti situati ai piani superiori, dimostrando l'efficacia del progetto nel contestualizzare la minaccia in base alla sua origine e diffusione. Come illustrato nelle Figure 7.11, 7.12, 7.13, le posizioni iniziali dei cinque utenti di riferimento sono distribuite su tutti i livelli del campus: la soluzione implementata ha risposto con la precisione attesa, notificando esclusivamente gli utenti sul piano interessato e confermando la sua capacità di discernere con accuratezza le aree di rischio.

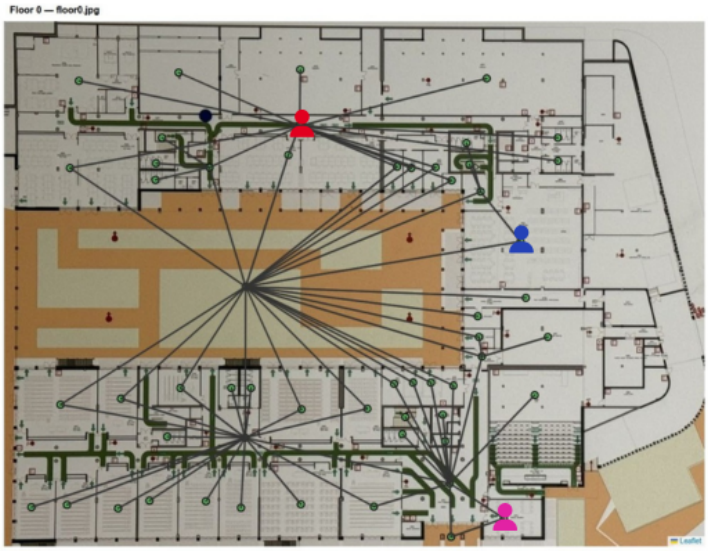

Figura 7.11: Posizioni iniziali: piano 0

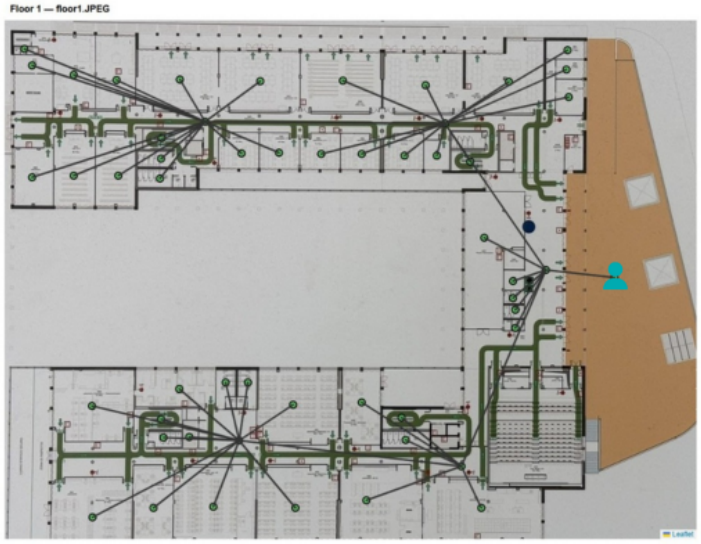

Figura 7.12: Posizioni iniziali: piano 1

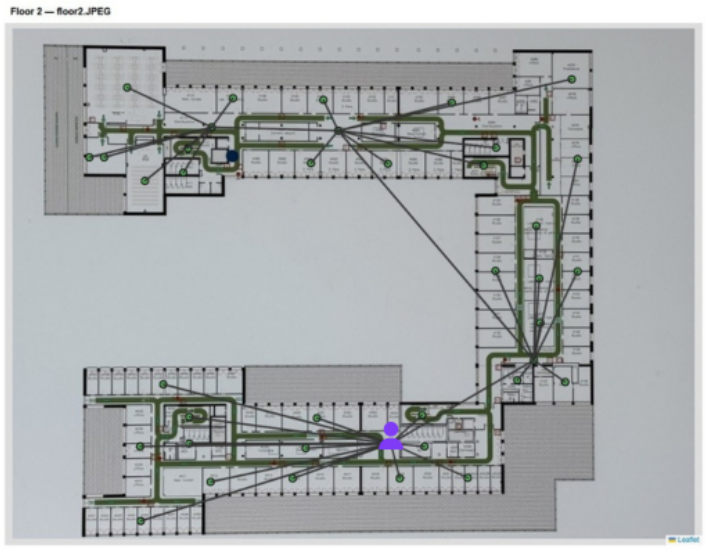

Figura 7.13: Posizioni iniziali: piano 2

Successivamente, l'analisi ha confermato che i nodi sul piano terra sono stati correttamente classificati come pericolosi, attivando la fase successiva di calcolo dei percorsi di evacuazione.

7.4.2 Generazione dei percorsi di evacuazione

A seguito dell'identificazione dei nodi in pericolo, l'infrastruttura ha evidenziato efficacia nell'elaborazione di percorsi di evacuazione sicuri e topologicamente validi. Il componente *Map Manager* ha generato dinamicamente tali itinerari, definiti come una sequenza ordinata di archi che collegano la posizione corrente dell'utente a un punto di raccolta sicuro, escludendo proattivamente i nodi e gli archi che simulavano le aree allagate.

La correttezza di questa fase è stata validata tramite l'analisi del percorso generato. In Figura 7.14 viene mostrato come, a partire da un nodo classificato come pericoloso, il microservizio dedicato associ una lista di archi che costituiscono l'itinerario di evacuazione. Ciascun percorso è composto da una lista ordinata di archi che l'utente deve attraversare in sequenza.

	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 al piano 0 in pericolo: la colonna `evacuation_path` contiene la sequenza di archi computata per la fuga.

La validazione dell'aderenza a tali percorsi è confermata dal flusso di spostamento, illustrato graficamente in Figura 7.15. L'immagine offre una rappresentazione visuale immediata del processo, mostrando la traiettoria completa degli utenti in pericolo, dalle loro posizioni iniziali fino ai punti sicuri. L'analisi del flusso dimostra in modo inequivocabile come gli utenti abbiano seguito con precisione i percorsi calcolati, spostandosi dalle aree allagate del piano 0 verso i vani scala per raggiungere un piano superiore e, di conseguenza, sicuro.

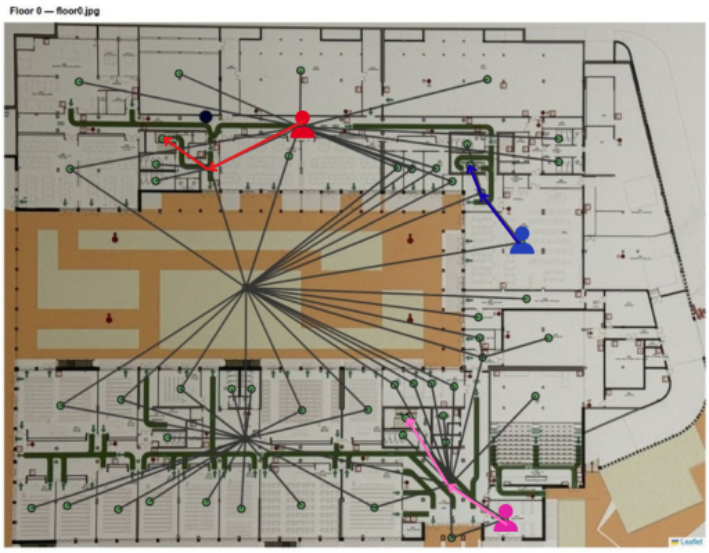

Figura 7.15: Flusso degli utenti in evacuazione

La tracciabilità dei movimenti ha mostrato in modo inequivocabile come gli utenti in pericolo abbiano seguito con precisione i percorsi calcolati. Il movimento si è sviluppato dalle aree allagate del piano 0 verso i vani scala, che hanno consentito il raggiungimento di un piano superiore e, di conseguenza, sicuro. Ciò ha confermato non solo la correttezza algoritmica del calcolo dei percorsi, ma anche l'integrità del processo di tracciamento e della visualizzazione delle strategie di evacuazione.

7.4.3 Validazione dell'arrivo e visualizzazione del flusso

L'analisi del flusso degli utenti costituisce la prova conclusiva del successo dell'evacuazione parziale. Il movimento, tracciato dalle aree di pericolo fino ai punti di raccolta sicuri, conferma che tutti gli occupanti a rischio hanno completato con successo i loro percorsi. Questo risultato attesta l'efficacia della piattaforma nel gestire una minaccia localizzata, guidando selettivamente gli individui al di fuori della zona di pericolo.

7.5 Validazione quantitativa

La presente sezione è dedicata alla validazione quantitativa della piattaforma, con l'obiettivo di valutarne sistematicamente le prestazioni, la scalabilità e la robustezza al variare delle condizioni operative. A tal fine, è stato implementato un framework sperimentale basato su un'**analisi di sensitività univariata**. Tale approccio metodologico prevede la definizione di uno scenario di baseline, rispetto al quale viene variato un singolo fattore sperimentale alla volta, mantenendo costanti gli altri. Questa tecnica permette di isolare e quantificare con precisione l'impatto di ciascun fattore sugli indicatori chiave di prestazione (*Key Performance Indicator*, KPI) della piattaforma. Sono stati definiti due KPI primari: efficacia ed efficienza. L'**efficienza** del sistema è 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.
L'**efficacia**, invece, è quantificata dal numero di utenti salvati, ovvero il totale degli occupanti che completano con successo il percorso di evacuazione designato. Questo indicatore misura la capacità del sistema di garantire il raggiungimento degli obiettivi di sicurezza per la totalità della popolazione coinvolta.

L'analisi è stata strutturata modulando quattro fattori sperimentali primari, ciascuno rappresentativo di una diversa dimensione del problema:

1. **Carico del sistema**: misura la scalabilità dell'architettura al variare della densità di occupazione. Sono stati definiti cinque livelli di carico, corrispondenti alla simulazione di 100, 300, 500, 750 e 1000 utenti concorrenti, per analizzare il comportamento del sistema sotto stress crescente.

2. **Contesto temporale e spaziale**: determina la distribuzione spaziale iniziale della popolazione simulata. Sono state modellate tre fasce ora-

rie rappresentative, volte a emulare profili di occupazione eterogenei e valutare la resilienza del sistema a diverse configurazioni di partenza.

3. **Tipologia di allerta**: definisce la natura dello scenario emergenziale. Sono state investigate due tipologie di emergenza - Terremoto e Alluvione - per investigare la capacità del sistema di adattare la strategia di evacuazione (rispettivamente totale o parziale) alla natura della minaccia.

4. **Capacità del grafo topologico**: modella i vincoli fisici dell'infrastruttura. Sono state confrontate due configurazioni – a capacità infinita e a capacità reale – al fine di comparare le prestazioni algoritmiche ideali con scenari realistici in cui emergono colli di bottiglia e fenomeni di congestione.

L'adozione di un approccio univariato offre il vantaggio di una chiara attribuibilità causale. Ciò permette di derivare conclusioni robuste e interpretabili sull'impatto di ciascuna variabile sull'efficienza e la reattività del processo di evacuazione.

Per ogni scenario, l'evento di allerta definisce $t_0 = 0$; tutti i tempi sono riportati come Δt rispetto a t_0. Per ciascun livello del fattore variato sono state eseguite repliche indipendenti.
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.

Da tali misure derivano le metriche operative:

- **Throughput** (*utenti/s*): è definito come il numero di utenti evacuati per unità di tempo, calcolato come $N/\Delta t_{\text{Stop}}$, dove N è la popolazione simulata. Misura l'efficienza del flusso di evacuazione: s= Δt_{First}-Δt_{First}
- **Latency gap** (s): rappresenta il ritardo introdotto dalla pipeline di elaborazione. Misura l'intervallo temporale tra la consegna del primo percorso di evacuazione e la notifica dell'ultimo percorso di evacuazione.

La mappatura con i KPI è diretta: l'*efficienza* coincide con Δt_{Stop} (tempo totale di evacuazione), mentre l'*efficacia* coincide con il numero di utenti salvati (N).

Per l'interpretazione dei risultati sono state adottate metriche statistiche specifiche. I dati sono stati raccolti attraverso cinque esecuzioni indipendenti per ogni scenario, al fine di garantire l'affidabilità delle misurazioni. Essi sono stati aggregati calcolando la **media** (μ) come indicatore di tendenza centrale e la **deviazione standard** (σ) per quantificare la variabilità e la stabilità delle misurazioni. Infatti, la deviazione standard misura la dispersione con cui i valori in un campione di dati si discostano dalla media campionaria.
La media campionaria è definita matematicamente come:

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

La deviazione standard campionaria (s) è definita matematicamente come:

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

dove n è il numero di osservazioni, x_i è la i-esima osservazione e $\bar{x}$ è la media campionaria. I risultati sono presentati nella notazione compatta $\mu \pm \sigma$.

L'analisi dei dati raccolti è articolata nelle sezioni successive, ciascuno dedicata a un assetto sperimentale specifico. Si esaminerà l'impatto del carico

di sistema (Sezione 7.5.1) e del contesto temporale (Sezione 7.5.2), per poi procedere con il confronto tra le diverse tipologie di allerta (Sezione 7.5.3) e le configurazioni di capacità del grafo (Sezione 7.5.4).

7.5.1 Analisi della scalabilità del sistema in funzione del carico utenti

La presente sezione è dedicata all'esposizione del setting sperimentale e dei parametri di simulazione impiegati per la valutazione quantitativa della **scalabilità** della piattaforma. L'indagine si concentra sull'impatto dell'incremento del carico di utenti sulle performance del sistema in un contesto di emergenza simulata. L'approccio metodologico adottato, in linea con i principi dell'analisi di sensitività univariata, prevede la variazione esclusiva del numero di utenti, consentendo di isolare e misurare con precisione l'effetto di tale variabile sulle metriche di performance. Tale valutazione è fondamentale per comprendere la scalabilità dei microservizi dedicati alla simulazione del movimento, alla gestione delle posizioni degli utenti e del calcolo dei percorsi di evacuazione.
Le condizioni operative mantenute invariate per l'intera serie di esperimenti, al fine di garantire la validità e la comparabilità dei risultati, sono le seguenti:

- **Tipologia di allerta**: la simulazione è stata condotta in uno scenario di terremoto, che richiede un'evacuazione totale dell'edificio.

- **Configurazione del grafo topologico**: il sistema è stato configurato per operare con una capacità reale su archi e nodi. Questa scelta consente di emulare i vincoli fisici e i fenomeni di congestione che caratterizzano gli scenari di evacuazione reali.

- **Contesto temporale e spaziale**: le simulazioni sono state avviate alle ore 10:00, un orario rappresentativo di una tipica e significativa densità di occupazione, che definisce la distribuzione spaziale iniziale della popolazione.

La Tabella 7.1 e l'istogramma in Figura 7.17 presentano le metriche temporali chiave e la deviazione standard al variare del numero di utenti. In particolare, mostrano che i tempi di ricezione del primo percorso rimangono consistenti fino a 750 utenti, evidenziando la capacità del sistema di avviare l'evacuazione in modo rapido e affidabile anche con un numero elevato di richieste simultanee. A un carico di 1000 utenti, si osserva un incremento del tempo medio e della sua variabilità, indicando una minore uniformità nella gestione delle richieste in condizioni di stress estremo.
I tempi di completamento dell'evacuazione, misurati dalla ricezione dell'ultimo percorso e del segnale di Stop, aumentano proporzionalmente all'incremento degli utenti, con un'accelerazione più marcata oltre i 750 utenti. Questo andamento è prevedibile e non compromette la funzionalità del sistema. La piattaforma dimostra di mantenere la sua operatività anche con un carico massiccio, completando l'evacuazione di tutti gli utenti, sebbene in un lasso di tempo più esteso.

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

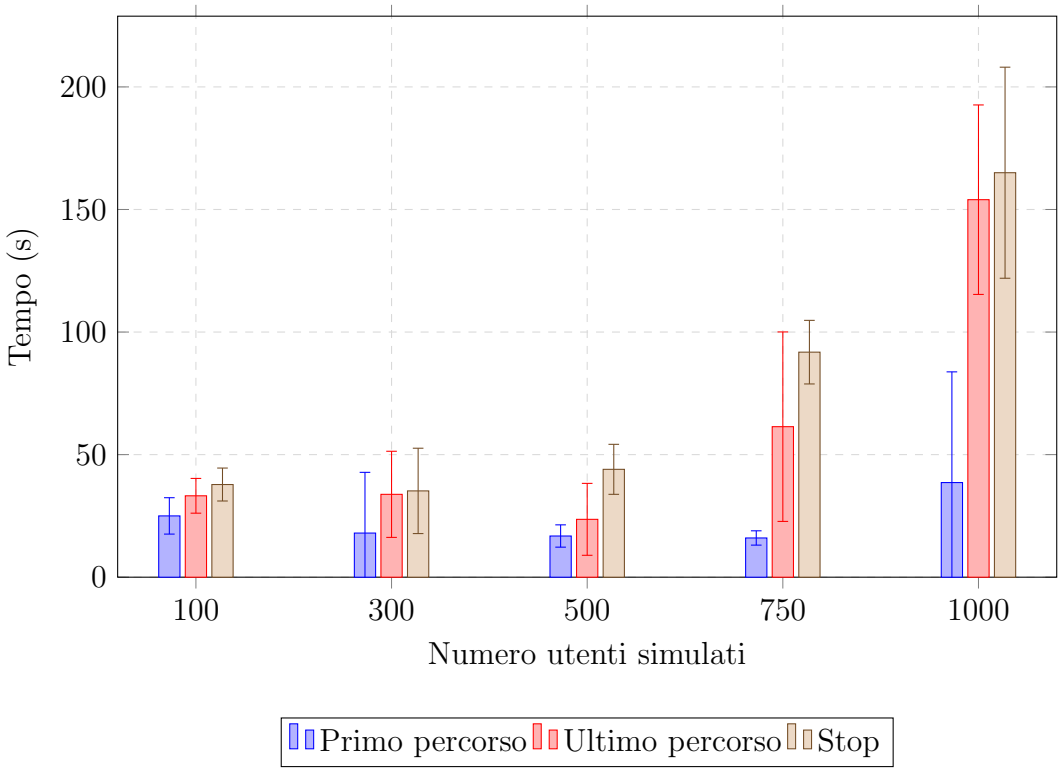

Figura 7.16: Confronto variazione tempi di ricezione al variare del numero di utenti simulati

La Figura 7.17 illustra le dinamiche di evacuazione attraverso le curve cumulative degli utenti salvati nel tempo.

- Con carichi fino a 500 utenti, le curve mostrano una crescita vertiginosa, indicativa di un'efficienza ottimale. L'andamento, quasi verticale, della curva attesta l'efficacia del sistema nel gestire un elevato volume di utenti con notevole rapidità, mantenendo costantemente aggiornato il loro stato in tempo reale.

- Con l'aumento del carico a 750 e 1000 utenti, le curve diventano progressivamente meno ripide. Questa variazione di pendenza è un segnale di saturazione, ma non di mancata efficienza. I microservizi continuano a gestire le interazioni e il flusso di dati intensi, garantendo che tutti gli utenti raggiungano la destinazione finale sicura.

In conclusione, l'analisi della distribuzione cumulativa conferma la resilienza del sistema: anche in condizioni estreme che superano la soglia di performance ideale, la piattaforma assicura il successo dell'evacuazione per l'intera popolazione, bilanciando la rapidità con la robustezza.

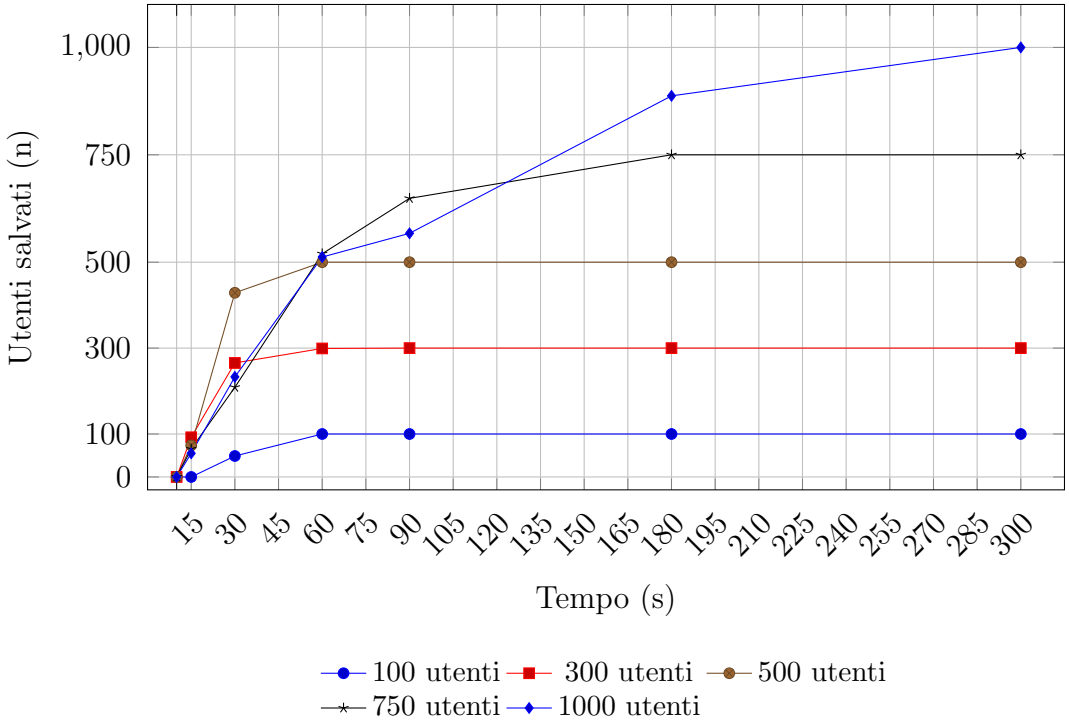

Figura 7.17: Utenti salvati nel tempo — Terremoto, capacità limitata, ore 10.

La Tabella 7.2 e la Figura 7.18 forniscono una prospettiva quantitativa, con focus su throughput e latency gap:

- Il throughput raggiunge un picco a 500 utenti, suggerendo un'ottimizzazione intrinseca dei microservizi per carichi intermedi. Successivamente, il valore si riduce, ma non in modo drastico, riflettendo la gestione adattiva di collisioni e ritardi in un ambiente congestionato.
- Il latency gap mostra una crescita notevole e non lineare oltre i 500 utenti. Questo aumento dimostra che il tempo necessario per elabo-

rare tutti i percorsi si estende significativamente in condizioni di carico elevato, ma il sistema continua a funzionare, confermando la sua resilienza.

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

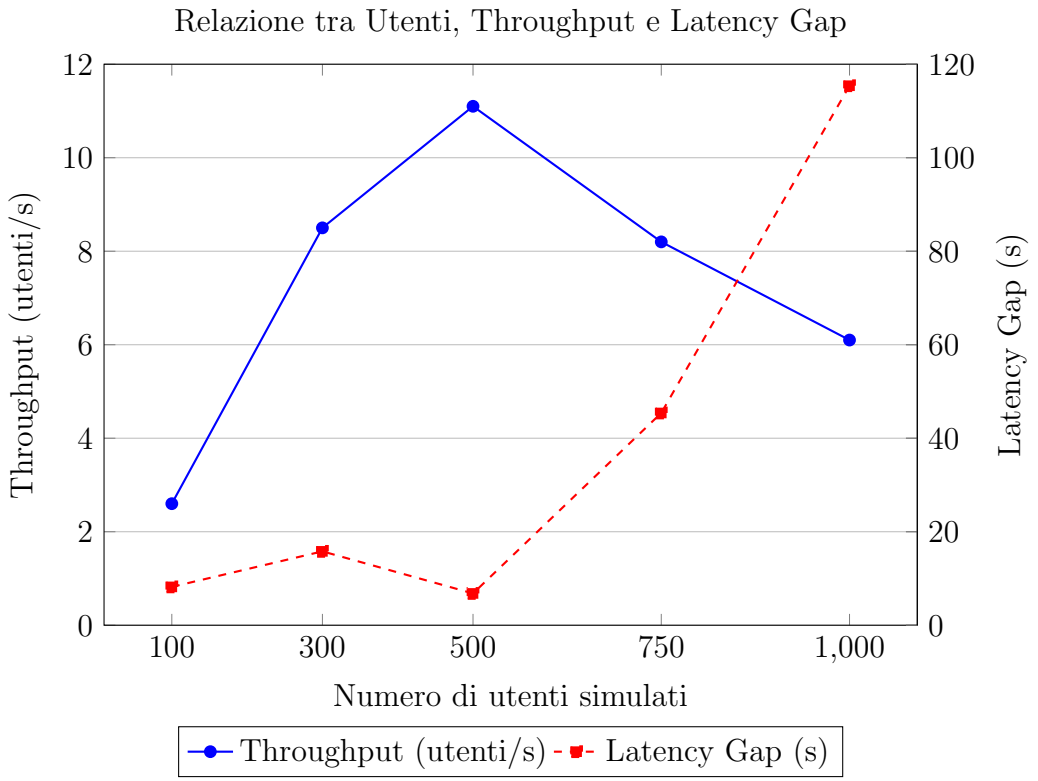

Figura 7.18: Confronto tra Throughput e Latency Gap al variare del numero di utenti a rischio nello scenario Terremoto con capacità archi e nodi limitata e simulazione alle ore 10.

In conclusione, questa analisi dimostra che la piattaforma è altamente scalabile fino a 500 utenti, raggiungendo la sua massima efficienza. Oltre tale limite, il sistema non collassa, ma si adatta per garantire la completa evacuazione di tutti gli utenti, anche a scapito della velocità. Questo comportamento adattivo è una prova della solidità dei microservizi di simulazione, gestione delle posizioni e di calcolo dei percorsi, progettati per superare le sfide del carico computazionale e salvaguardare l'obiettivo primario del progetto.

7.5.2 Analisi della resilienza in funzione della fascia oraria

La presente sezione è dedicata all'esposizione del setting sperimentale e dei parametri di simulazione impiegati per la valutazione quantitativa della **resilienza** della piattaforma al variare della distribuzione iniziale della popolazione. L'indagine si concentra sull'impatto del contesto temporale e spaziale sulle performance del sistema in un contesto di emergenza simulata. L'approccio metodologico adottato, in linea con i principi dell'analisi di sensitività univariata, prevede la variazione esclusiva della fascia oraria di simulazione, consentendo di isolare e misurare con precisione l'effetto di tale variabile sulle metriche di performance. Le distribuzioni degli utenti simulate riflettono lo scenario tipico di un campus universitario:

- Fascia 8:30-10:30 e 16:00-18:00 (orario di lezione): la maggior parte degli utenti si trova nelle aule, con una minore percentuale distribuita in uffici e corridoi. Questa concentrazione iniziale in spazi chiusi e ristretti può generare un'evacuazione più complessa.

- Fascia 13:00-14:00 (pausa pranzo): gli utenti sono prevalentemente distribuiti in aree comuni, con una percentuale minore in corridoi e bagni. Questa distribuzione in aree più ampie e vicine alle uscite può favorire un'evacuazione più rapida.

- Fascia 15:15-16:00 (Pausa tra Lezioni): Gli utenti si distribuiscono principalmente in aree di socializzazione. Questa configurazione, meno concentrata rispetto alle fasce orarie di lezione, tende a ridurre i fenomeni di congestione.

Le condizioni operative mantenute invariate per l'intera serie di esperimenti, al fine di garantire la validità e la comparabilità dei risultati, sono le seguenti:

- **Tipologia di allerta**: la simulazione è stata condotta in uno scenario di terremoto, che richiede un'evacuazione totale dell'edificio.
- **Configurazione del grafo topologico**: il sistema è stato configurato per operare con una capacità reale su archi e nodi. Questa scelta consente di emulare i vincoli fisici e i fenomeni di congestione che caratterizzano gli scenari di evacuazione reali.
- **Carico del sistema**: il numero di utenti simulati è stato mantenuto costante a 1000 occupanti, un valore rappresentativo di un carico significativo.

L'analisi della Tabella 7.3 rivela come la distribuzione iniziale della popolazione, che varia a seconda della fascia oraria, influenzi significativamente le metriche di evacuazione per un carico fisso di 1000 utenti. I dati mostrano che la piattaforma dimostra una notevole resilienza, pur in condizioni di carico elevato, ma le performance variano in base alla configurazione spaziale iniziale degli utenti.

I tempi di ricezione del primo percorso di evacuazione rimangono relativamente stabili in tutte le fasce orarie. Questo dato evidenzia che il microservizio deputato al calcolo dei percorsi è sempre in grado di avviare il processo di evacuazione in modo tempestivo, indipendentemente dalla distribuzione degli utenti. Le variazioni più significative, in particolare la deviazione standard molto alta nelle fasce 8:30-10:30 e 16:00-18:00, come mostrato nella Figura 7.19, suggeriscono una minore uniformità nella risposta iniziale in orari di

punta, con alcune richieste che potrebbero essere elaborate più lentamente a causa di una distribuzione iniziale sfavorevole.

I tempi di completamento dell'evacuazione (ricezione dell'ultimo percorso e segnale di Stop) mostrano una forte correlazione con la fascia oraria. La performance migliore si registra nella fascia 15:15-16:00: in questo orario, l'evacuazione è molto più rapida e stabile rispetto alle altre fasce orarie. Al contrario, le fasce 13:00-14:00 e 16:00-18:00 mostrano tempi di evacuazione notevolmente più lunghi e un'elevata variabilità (come indicato dalle alte deviazioni standard), suggerendo che la distribuzione degli utenti in questi momenti genera una congestione più significativa, rallentando l'intero processo di evacuazione.

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

L'analisi della Figura 7.20 rivela che le dinamiche di evacuazione di un'intera popolazione di 1000 utenti sono fortemente influenzate dalla loro distribuzione iniziale dipendente dalla fascia oraria. Il grafico delle curve cumulative degli utenti salvati mostra un'ampia variabilità nella rapidità con cui l'evacuazione viene completata.

- Le curve relative alle fasce orarie 13:00-14:00 (Pausa Pranzo) e 15:15-16:00 (Pausa tra Lezioni) mostrano un'evacuazione estremamente rapida, in linea con una distribuzione della popolazione in aree comuni e corridoi che riduce la congestione. La crescita quasi verticale della prima curva evidenzia l'efficienza del sistema in una configurazione spaziale ottimale: la maggior parte della popolazione si trova in aree comuni

e corridoi, che rappresentano posizioni strategiche per un'evacuazione rapida e tendono a generare meno colli di bottiglia.

- Le curve delle fasce 8:30-10:30 (Lezione Mattina) e 16:00-18:00 (Lezione Pomeriggio) sono notevolmente meno ripide e si estendono per un periodo di tempo più lungo. Questo suggerisce un comportamento di saturazione, dove la distribuzione degli utenti in aula genera colli di bottiglia e rallentamenti a causa delle capacità limitate di porte e corridoi. Nonostante ciò, il sistema continua a garantire che tutti gli utenti raggiungano la sicurezza.

In sintesi, il grafico dimostra la resilienza del sistema: sebbene l'efficienza vari a seconda della distribuzione spaziale degli utenti, la piattaforma assicura sempre il successo dell'evacuazione totale, bilanciando la rapidità con la robustezza anche in condizioni di forte stress.

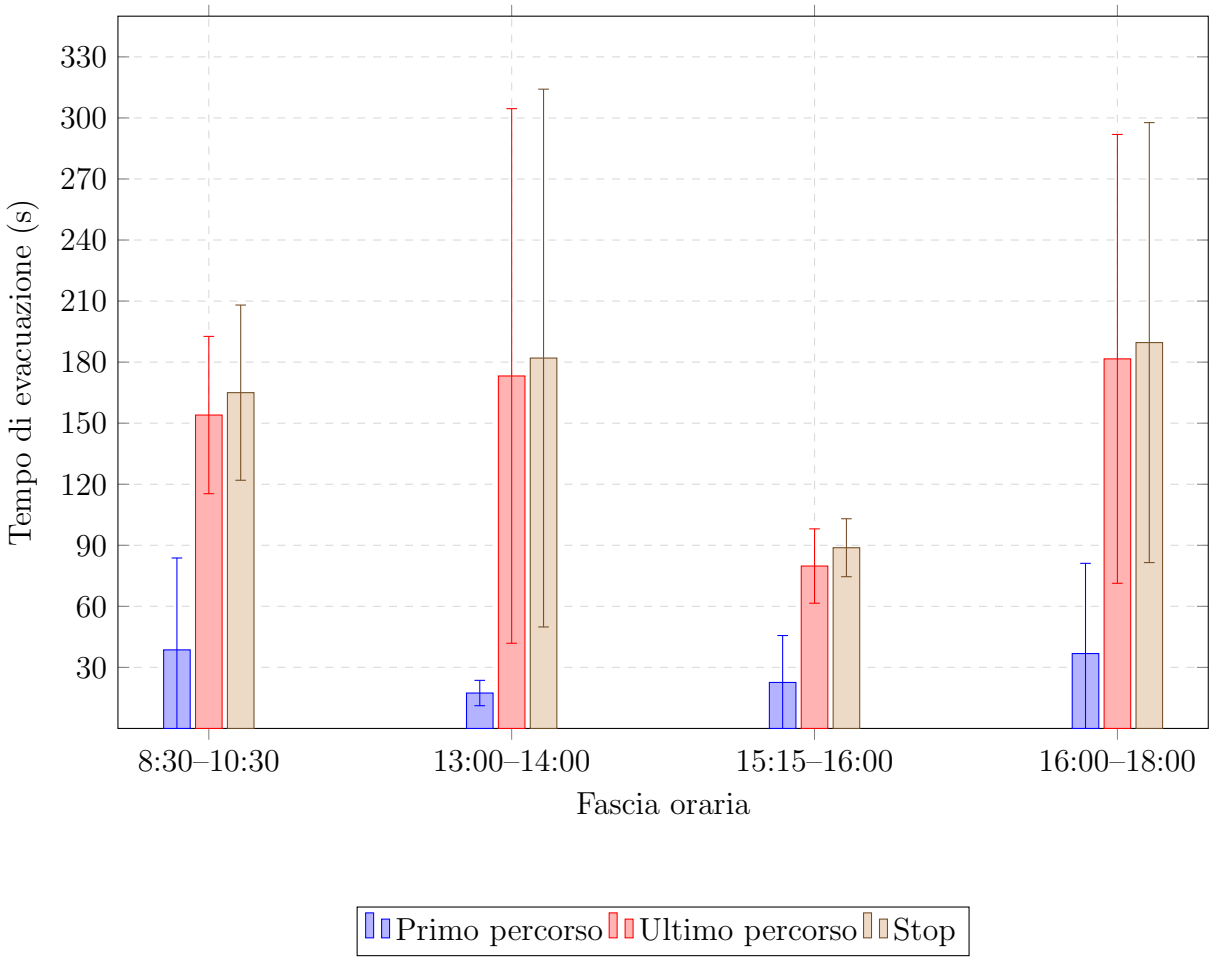

Figura 7.19: Confronto variazione tempi di ricezione al variare della fascia oraria di simulazione

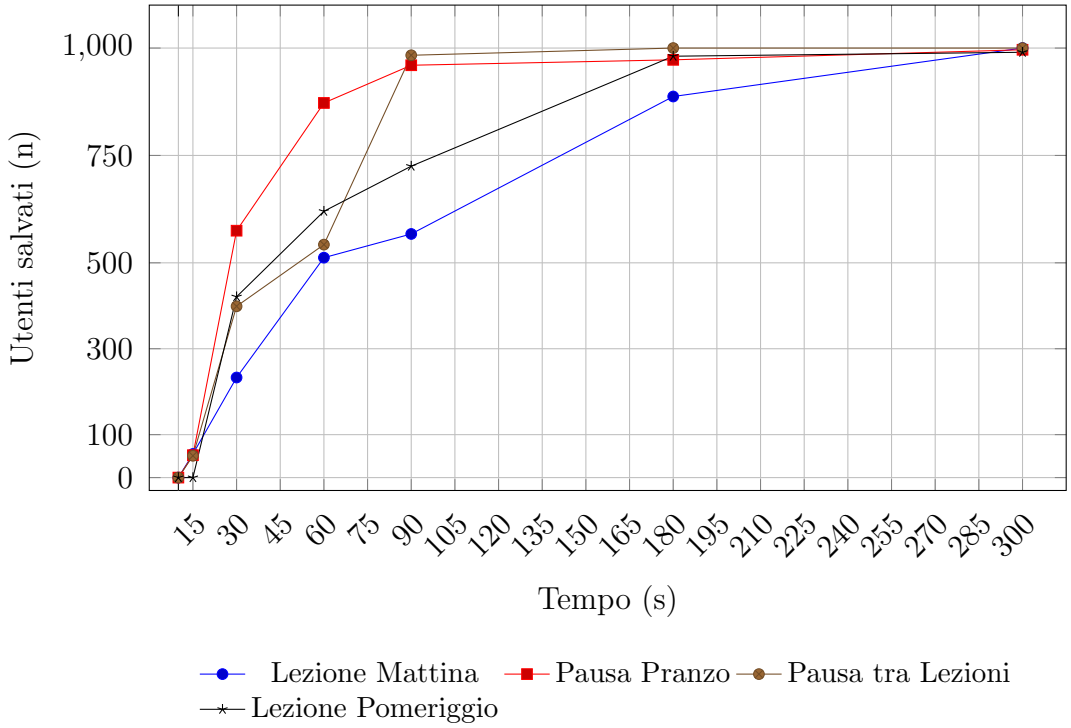

Figura 7.20: Utenti salvati nel tempo — Terremoto, capacità limitata, 1000 utenti; confronto tra quattro fasce orarie.

L'analisi combinata della Tabella 7.4 e della Figura 7.21 fornisce una prospettiva quantitativa chiara della resilienza del sistema in un contesto di carico elevato (1000 utenti) e con distribuzioni spaziali variabili.

- Il throughput mostra un comportamento non uniforme: raggiunge il suo valore massimo, indicando la massima efficienza, nella fascia oraria 15:15-16:00, mentre nelle altre fasce orarie si riduce significativamente. Questo suggerisce che la distribuzione spaziale della popolazione nella fascia pomeridiana è intrinsecamente più favorevole a un'evacuazione rapida, probabilmente a causa di un minor numero di percorsi che si intersecano e una minore congestione complessiva.

- Il latency gap correla inversamente con il throughput. Il valore minimo, e quindi la migliore efficienza della pipeline di calcolo, si riscontra nella

fascia 15:15-16:00, che coincide con il picco di throughput. Al contrario, il latency gap raggiunge i suoi valori massimi nelle fasce orarie in cui il throughput è più basso. Questo aumento esponenziale del latency gap conferma che in orari di forte congestione, l'algoritmo impiega un tempo notevolmente maggiore per risolvere le interazioni tra gli agenti e completare l'elaborazione dei percorsi per tutti gli utenti.

In conclusione, l'analisi dei dati di throughput e latency gap dimostra che la distribuzione spaziale degli utenti, che varia in base alla fascia oraria, è un fattore critico per le performance del sistema. Il sistema si adatta in modo resiliente ai diversi scenari, ma le performance di evacuazione sono ottimizzate quando la disposizione iniziale della popolazione riduce al minimo la potenziale congestione.

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

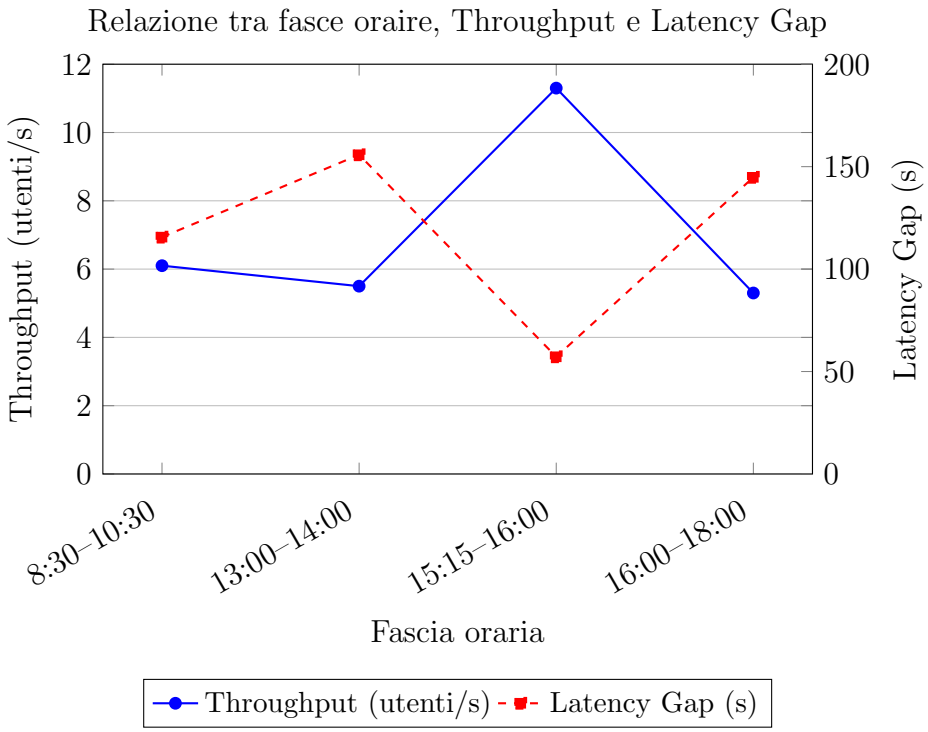

Figura 7.21: Confronto tra Throughput e Latency Gap nelle diverse fasce orarie (scenario Terremoto, capacità limitata, 1000 utenti).

In conclusione, l'analisi sperimentale ha dimostrato che la piattaforma non solo è scalabile, ma anche altamente resiliente alle variazioni della distribuzione spaziale degli utenti. Sebbene la piattaforma garantisca sempre il successo dell'evacuazione totale, le performance operative, misurate in termini di tempo di evacuazione, throughput e latency gap, sono strettamente dipendenti dalla configurazione iniziale della popolazione. In particolare, il sistema raggiunge la sua massima efficienza in scenari in cui si ha una distribuzione più dispersa degli utenti in aree di passaggio, che riduce la congestione.

7.5.3 Analisi della reattività in funzione della tipologia di allerta

La presente sezione è dedicata all'esposizione del setting sperimentale e dei parametri di simulazione impiegati per la valutazione quantitativa della **reattività** della piattaforma. L'indagine si concentra sulla capacità del sistema di adattare la strategia di evacuazione alla natura dell'evento emergenziale, confrontando scenari di evacuazione totale e parziale. L'approccio metodologico adottato, in linea con i principi dell'analisi di sensitività univariata, prevede la variazione esclusiva della tipologia di allerta, consentendo di isolare e misurare con precisione l'effetto di tale variabile sulle metriche di performance. Le condizioni operative mantenute invariate per l'intera serie di esperimenti, al fine di garantire la validità e la comparabilità dei risultati, sono le seguenti:

- **Configurazione del grafo topologico**: il sistema è stato configurato per operare con una capacità reale su archi e nodi. Questa scelta consente di emulare i vincoli fisici e i fenomeni di congestione che caratterizzano gli scenari di evacuazione reali.
- **Contesto temporale e spaziale**: le simulazioni sono state avviate alle ore 10:00, un orario rappresentativo di una tipica e significativa densità di occupazione, che definisce la distribuzione spaziale iniziale della popolazione.
- **Carico del sistema**: il numero di utenti simulati è stato mantenuto costante a 1000 occupanti, un valore rappresentativo di un carico significativo.

La Tabella 7.5 e l'istogramma in Figura 7.22 forniscono un'analisi comparativa delle metriche di evacuazione per le due diverse tipologie di allerta studiate: i dati rivelano che il tipo di allerta ha un impatto significativo sulla dinamica dell'evacuazione. Il tempo di ricezione del primo percorso è notevolmente più basso e con una variabilità minore in caso di Alluvione: ciò

suggerisce che il sistema avvia l'evacuazione più rapidamente in uno scenario di evacuazione parziale, poiché l'algoritmo di calcolo dei percorsi opera su un grafo topologico ridotto (escludendo i piani superiori).
Al contrario, i tempi di completamento dell'evacuazione, misurati dalla ricezione dell'ultimo percorso e del segnale di Stop, sono più lunghi in caso di Alluvione rispetto al Terremoto. Sebbene l'allerta Alluvione permetta un avvio più rapido, il processo di evacuazione nel suo complesso richiede più tempo, poiché il numero limitato di percorsi e uscite disponibili verso i piani superiori causa una maggiore congestione e rallenta l'evacuazione finale degli utenti.

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

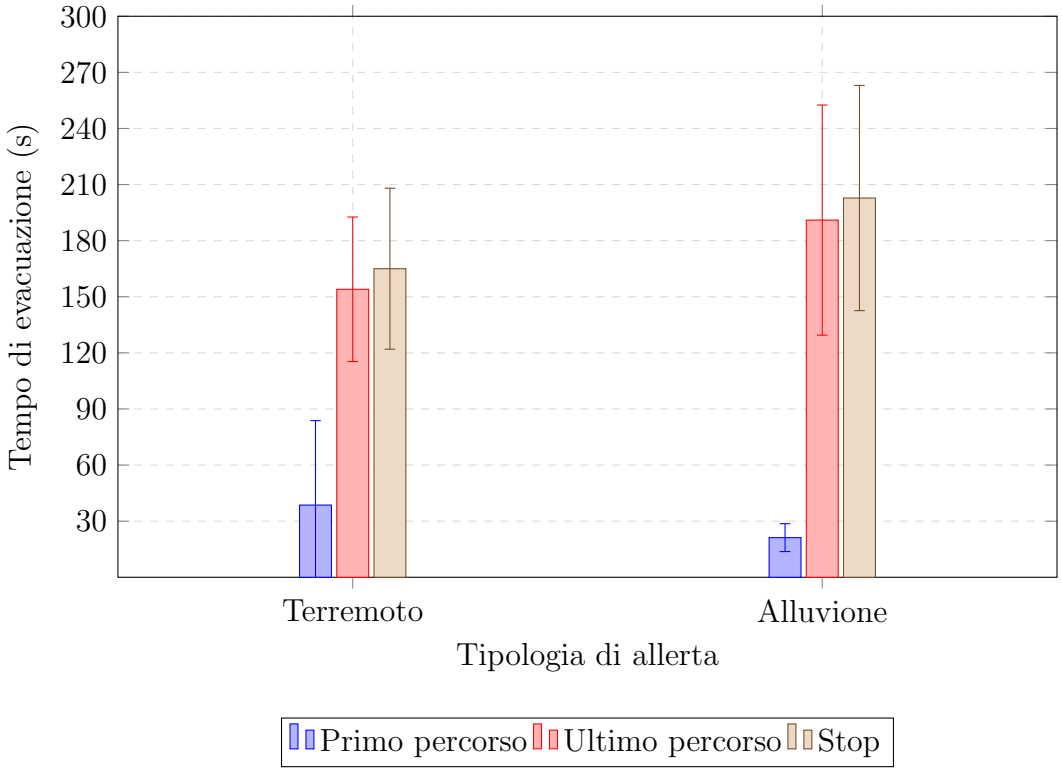

Figura 7.22: Confronto variazione tempi di ricezione al variare della tipologia di allerta

La Figura 7.23 illustra le dinamiche di evacuazione attraverso le curve cumulative degli utenti salvati nel tempo, confrontando i due scenari di allerta. L'analisi di tale distribuzione rivela chiaramente come la tipologia di allerta impatti la velocità di evacuazione.

- La curva dell'Alluvione, nonostante un avvio più rapido, mostra una crescita meno ripida rispetto a quella del Terremoto. Questo comportamento suggerisce che la strategia di evacuazione per l'Alluvione, che esclude i piani inferiori e le uscite a rischio, crea un'inevitabile congestione lungo i percorsi rimanenti. La minore pendenza indica un rallentamento del flusso di utenti, che il sistema gestisce in modo efficace per prevenire il collasso, ma a scapito della velocità complessiva.

- La curva del Terremoto, pur partendo con un leggero ritardo, mostra una crescita più costante e sostenuta, raggiungendo una performance cumulativa superiore nei primi 150 secondi. Questo è dovuto alla disponibilità di più percorsi e uscite, che distribuiscono il carico di utenti e riducono i colli di bottiglia, permettendo un flusso più efficiente e una minor frizione tra gli agenti.

In sintesi, il grafico conferma che il sistema si adatta in modo resiliente ai diversi scenari, bilanciando la rapidità di evacuazione con la sicurezza.

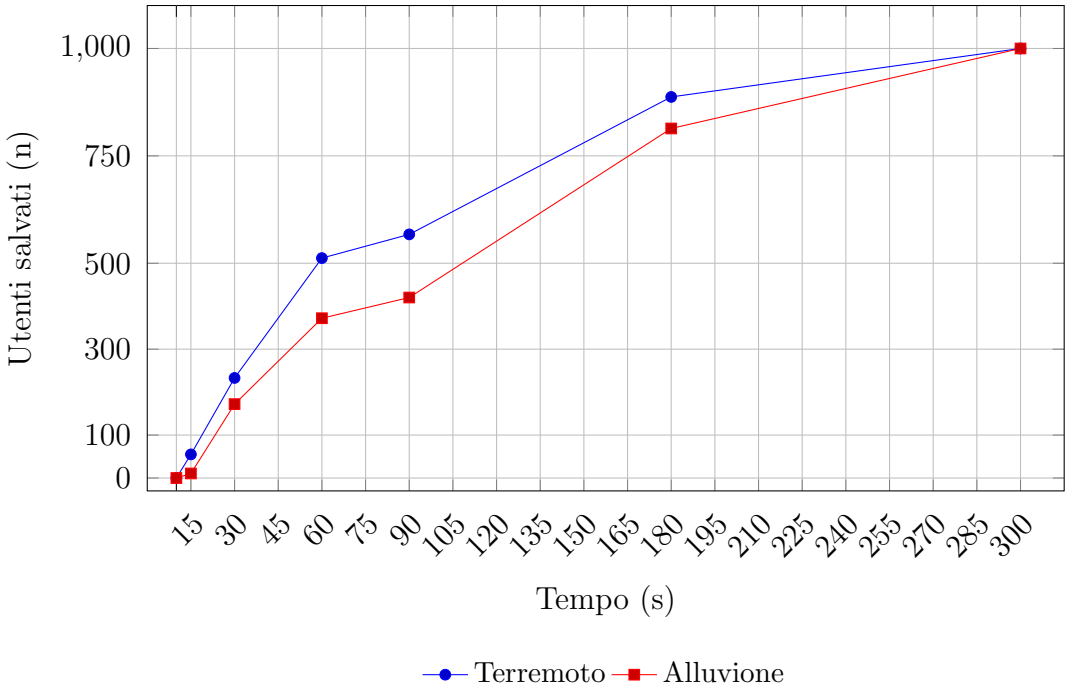

Figura 7.23: Utenti salvati nel tempo — Ore 10, capacità limitata, utenti fissi; confronto tra allerta terremoto e alluvione.

L'analisi combinata della Tabella 7.6 e della Figura 7.24 fornisce una chiara prospettiva quantitativa delle performance del sistema in base alla tipologia di allerta.

- Il throughput è notevolmente superiore nello scenario Terremoto rispetto a quello Alluvione: questa differenza è un indicatore diretto dell'ef-

ficienza dei percorsi disponibili. Nello scenario Terremoto, la rete di percorsi completa consente un flusso di utenti più elevato e distribuito, riducendo la congestione. Al contrario, l'allerta Alluvione, limitando l'accesso a porzioni dell'edificio, forza gli utenti a convergere su un numero ridotto di vie di fuga, creando colli di bottiglia e rallentando il flusso complessivo.

- Il latency gap, che riflette il ritardo nell'elaborazione dei percorsi, è significativamente più alto in caso di Alluvione rispetto al Terremoto. Questo dato conferma l'osservazione precedente: la ridotta disponibilità di percorsi nello scenario di Alluvione aumenta la complessità del calcolo, poiché il *Map Manager* deve risolvere un maggior numero di conflitti e ricalcolare i percorsi in tempo reale per un numero elevato di utenti che si muovono in spazi limitati. L'algoritmo impiega più tempo per garantire che ogni utente raggiunga la sicurezza, come dimostra l'aumento del tempo totale di evacuazione.

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

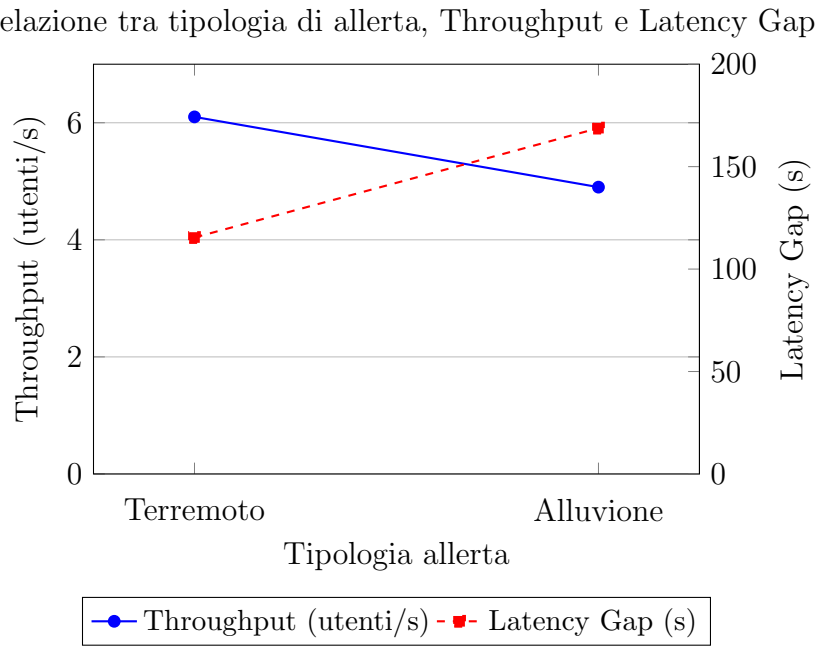

Figura 7.24: Confronto tra Throughput e Latency Gap negli scenari Terremoto e Alluvione con capacità archi e nodi limitata e 1000 utenti.

In conclusione, l'analisi ha dimostrato che la tipologia di allerta non solo influenza la strategia di evacuazione, ma ha anche un impatto diretto sulle metriche di performance chiave del sistema. Sebbene un'evacuazione parziale possa consentire un avvio più rapido, la sua natura restrittiva sui percorsi porta a un'evacuazione complessivamente più lunga e meno efficiente in termini di throughput e latency. La piattaforma gestisce con successo entrambi gli scenari, adattando i percorsi e garantendo il successo dell'evacuazione per tutti gli utenti. Questo comportamento conferma la sua resilienza e la sua capacità di operare in modo affidabile anche in condizioni di elevata criticità e con vincoli complessi.

7.5.4 Analisi della robustezza in funzione della capacità di archi e nodi

La presente sezione è dedicata all'esposizione del setting sperimentale e dei parametri di simulazione impiegati per la valutazione quantitativa della **robustezza** della piattaforma. L'indagine si concentra sull'impatto dei vincoli fisici del grafo topologico sulle performance del sistema.
L'approccio metodologico adottato, in linea con i principi dell'analisi di sensitività univariata, prevede la variazione esclusiva della configurazione del grafo, confrontando uno scenario ideale con uno realistico. Le condizioni operative mantenute invariate per l'intera serie di esperimenti, al fine di garantire la validità e la comparabilità dei risultati, sono le seguenti:

- **Tipologia di allerta**: la simulazione è stata condotta in uno scenario di terremoto, che richiede un'evacuazione totale dell'edificio.
- **Contesto temporale e spaziale**: le simulazioni sono state avviate alle ore 10:00, un orario rappresentativo di una tipica e significativa densità di occupazione, che definisce la distribuzione spaziale iniziale della popolazione.
- **Carico del sistema**: il numero di utenti simulati è stato mantenuto costante a 1000 occupanti, un valore rappresentativo di un carico significativo.

L'analisi della Tabella 7.7 e della Figura 7.25 confronta le performance del sistema in due scenari distinti: uno in cui la capacità di archi e nodi è limitata (simulando vincoli fisici e congestione) e uno in cui la capacità è infinita. I dati, raccolti per un'allerta Terremoto con 1000 utenti simulati alle 10 del mattino, evidenziano il ruolo cruciale dei vincoli fisici sull'efficienza dell'evacuazione.

- Il tempo di ricezione del primo percorso si riduce significativamente nello scenario con capacità infinita rispetto a quello con capacità limitata:

ciò indica che, in assenza di vincoli, l'algoritmo di calcolo dei percorsi opera in modo più rapido e prevedibile, senza i rallentamenti causati dalla necessità di gestire le collisioni e la congestione in tempo reale.

- I tempi di completamento dell'evacuazione mostrano un miglioramento ancora più marcato. Nello scenario con capacità infinita, il tempo di Stop si riduce di oltre la metà rispetto a quello con capacità limitata. Tale risultato sottolinea come la gestione dei vincoli fisici e della congestione rappresenti il principale fattore di rallentamento in un'evacuazione reale.

L'elevata deviazione standard del tempo di ricezione del primo percorso nello scenario a capacità limitata indica una minore uniformità nella risposta del sistema, probabilmente a causa di un numero maggiore di ricalcoli e aggiustamenti di percorso necessari per far fronte alle collisioni tra gli utenti. Al contrario, nello scenario a capacità infinita, la bassa deviazione standard dimostra una risposta molto più stabile e prevedibile.

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

La Figura 7.25 fornisce una chiara rappresentazione visiva dell'impatto della capacità degli archi e dei nodi sui tempi di evacuazione. I dati mostrano in modo evidente che, in uno scenario con capacità illimitata, il sistema di evacuazione è significativamente più rapido e prevedibile rispetto a uno in cui la capacità è limitata, un fattore che riflette vincoli e la congestione del mondo reale. Nello specifico, la deviazione standard molto più bassa nello scenario a capacità illimitata (le barre di errore sono quasi inesistenti) indica un'altissima prevedibilità e stabilità del sistema, a differenza della grande variabilità osservata con la capacità limitata.

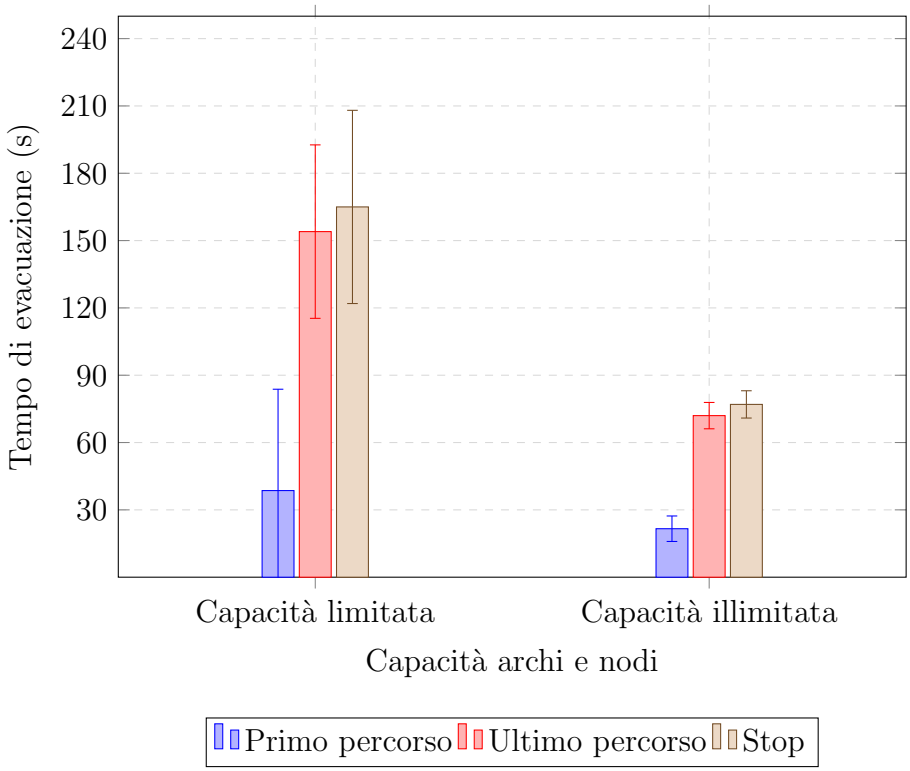

Figura 7.25: Confronto variazione tempi di ricezione al variare della capacità degli archi e dei nodi

La Figura 7.26 illustra l'analisi della distribuzione cumulativa. Essa conferma le osservazioni delineate in precedenza: la curva dello scenario con capacità infinita mostra una crescita notevolmente più ripida e completa l'evacuazione di 1000 utenti in un tempo significativamente minore rispetto alla curva a capacità limitata. Questo andamento, che si avvicina a una crescita verticale, dimostra l'efficienza ottimale del sistema quando la congestione e i colli di bottiglia fisici non rappresentano un ostacolo.

La curva a capacità limitata, pur garantendo il successo dell'evacuazione totale, ha una pendenza più graduale e richiede più tempo per raggiungere il completamento.

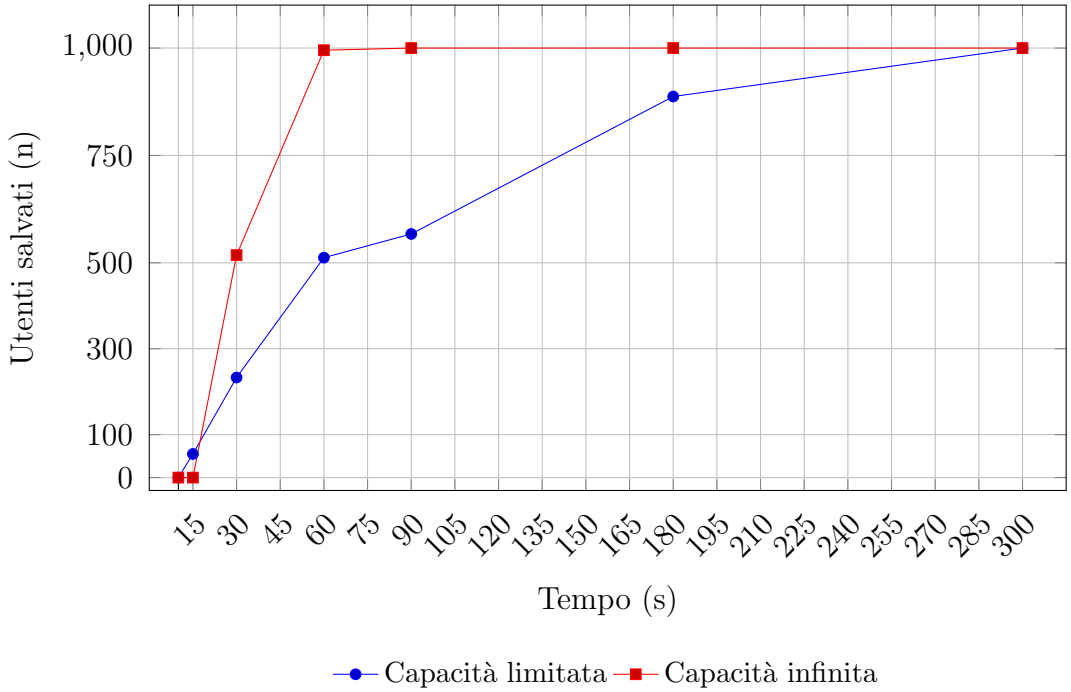

Figura 7.26: Utenti salvati nel tempo — Terremoto, ore 10, 1000 utenti; confronto tra capacità limitata e infinita.

L'analisi combinata della Tabella 7.8 e della Figura 7.27 fornisce una chiara prospettiva quantitativa dell'impatto dei vincoli di capacità sulla performance del sistema di evacuazione. I dati dimostrano che il throughput e il latency gap migliorano drasticamente quando i vincoli fisici del grafo sono rimossi.

- Il throughput nello scenario a capacità illimitata è più che raddoppiato rispetto allo scenario a capacità limitata: l'assenza di congestione fisica e colli di bottiglia permette al sistema di calcolare i percorsi e far defluire gli utenti a un ritmo notevolmente più veloce.

- Il latency gap si riduce di oltre la metà: indica che il tempo necessario per calcolare il primo e l'ultimo percorso è molto più breve, dimostrando una notevole efficienza computazionale.

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

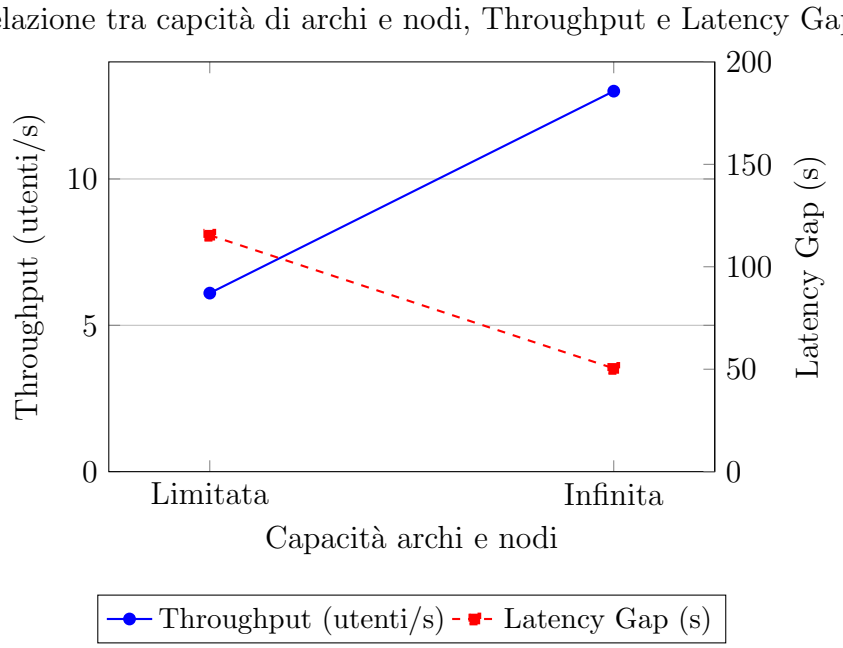

Figura 7.27: Confronto tra Throughput e Latency Gap per allerta Terremoto e simulazione di 1000 utenti alle ore 10

In sintesi, l'analisi conferma che la performance ottimale del sistema si raggiunge in assenza di vincoli fisici. La differenza tra i due scenari evidenzia che i microservizi sono estremamente efficienti nel calcolo dei percorsi, ma le sfide principali nella simulazione e nell'evacuazione reale risiedono nella gestione della congestione e delle interazioni tra gli agenti.

Conclusioni

Il presente lavoro di tesi ha affrontato e risolto la critica discontinuità esistente tra i sistemi di allerta pubblica su scala macro-geografica e la gestione delle emergenze in ambienti indoor, un dominio dove la granularità spaziale, le dinamiche di folla e i vincoli topologici impongono l'adozione di paradigmi radicalmente nuovi. Attraverso l'impiego del Campus di Cesena dell'Università di Bologna quale caso di studio emblematico, è stata progettata e implementata un'architettura software che supera tale dicotomia, dimostrando la concreta realizzabilità di un sistema per il calcolo e la diffusione di percorsi di evacuazione personalizzati e dinamicamente ricalcolati in regime di tempo quasi reale.

Il fulcro della ricerca è rappresentato da un'architettura a microservizi, concepita secondo i principi di modularità, scalabilità e reattività agli eventi. Le componenti architetturali chiave includono:

- **Alert Manager**: un modulo per l'ingestione e la normalizzazione di allerta conformi allo standard internazionale CAP.

- **Position Manager** e **User Simulator**: sistemi sinergici per l'identificazione precisa degli individui esposti al rischio e per la simulazione di scenari di occupazione realistici.

- **Notification Center**: un'infrastruttura di messaggistica asincrona, basata su RabbitMQ, che garantisce il disaccoppiamento, la resilienza e l'affidabilità nella propagazione delle direttive di evacuazione.

- **Map Viewer**: un'interfaccia interattiva per la visualizzazione delle planimetrie, del grafo e dei percorsi calcolati, fornendo agli operatori uno strumento di visual analytics per il monitoraggio in tempo reale.

- **Map Manager**: il nucleo computazionale del sistema, responsabile del pathfinding dinamico, con una rigorosa gestione dello stato e della capacità di nodi e archi del grafo.

Il sistema, avvalendosi di un database geospaziale PostgreSQL/PostGIS e ottimizzando le prestazioni attraverso l'elaborazione del grafo interamente in memoria (RAM), assicura coerenza e previene fenomeni di race condition mediante sofisticati meccanismi di coordinamento su RabbitMQ.

La validazione sperimentale, condotta su scenari di crisi eterogenei quali eventi sismici (evacuazione totale) e alluvionali (evacuazione parziale con vincoli topologici), ha dimostrato in modo inequivocabile la robustezza e l'efficacia del paradigma proposto. In particolare, sono stati verificati:

- **Efficacia operativa**: il sistema ha raggiunto con pieno successo l'obiettivo primario di garantire la sicurezza degli utenti.

  - **Correttezza topologica e semantica**: i percorsi generati evitano sistematicamente le zone interdette, garantendo la coerenza logica della navigazione multi-piano.

  - **Resilienza e affidabilità**: il sistema ha esibito un comportamento deterministico e affidabile di fronte a mutamenti dinamici dello scenario, ricalcolando i percorsi in maniera coerente e assicurando che la totalità degli utenti simulati raggiungesse una destinazione sicura.

- **Efficienza computazionale e temporale**: il sistema ha raggiunto gli obiettivi in tempi rapidi e con un uso ottimale delle risorse.

  - **Reattività end-to-end**: la latenza complessiva, dall'acquisizione dell'allerta CAP alla notifica all'utente finale, è risultata pie-

namente compatibile con le stringenti esigenze operative di una gestione emergenziale efficace.

  - **Scalabilità**: l'architettura ha gestito carichi di utenti crescenti in modo efficiente, senza collassare e mantenendo tempi di risposta contenuti.

- **Osservabilità**: il logging distribuito e l'interfaccia del *Map Viewer* si sono confermati strumenti indispensabili per l'analisi prestazionale. Tali strumenti hanno permesso non solo di validare la correttezza delle strategie di evacuazione (efficacia), ma anche di analizzare le prestazioni e individuare i colli di bottiglia (efficienza), fornendo una piena trasparenza sul comportamento del sistema.

L'analisi ha inoltre evidenziato un punto cruciale: i principali colli di bottiglia non sono di natura computazionale, bensì fisici. L'algoritmo di pathfinding è estremamente efficiente, ma la dinamica di evacuazione è dominata dalla capacità limitata di varchi e corridoi e dalla conseguente possibile congestione.

Il successo di questa fase prototipale definisce una chiara traiettoria per l'evoluzione futura del sistema, articolata su tre direttrici di ricerca e sviluppo prioritarie:

1. **Evoluzione verso un sistema live**: sostituzione dello *User Simulator* con un'applicazione mobile reale, integrata con tecnologie di localizzazione indoor. Tale transizione dovrà affrontare le complesse sfide legate alla tutela della privacy, alla gestione del consenso informato e alla garanzia di funzionamento in condizioni di connettività degradata o assente.

2. **Orchestrazione di canali di notifica multi-modali**: potenziamento il Notification Center con un gateway per l'invio di notifiche push native, implementando meccanismi di Quality of Service (QoS), idempotenza e canali di fallback (SMS/Cell Broadcast) per massimizzare il tasso di raggiungibilità dell'utenza.

3. **Ottimizzazione globale dei flussi di evacuazione**: evoluzione del *Map Manager* da un approccio di pathfinding individuale a un motore di ottimizzazione globale. Ciò implica l'adozione di algoritmi avanzati di flusso su rete per gestire la capacità finita dei varchi, bilanciare dinamicamente i percorsi e implementare strategie di re-routing proattivo per la mitigazione delle congestioni, secondo criteri di efficienza ed equità.

In conclusione, questo lavoro di tesi non si è limitato a dimostrare la fattibilità teorica di un sistema di gestione delle evacuazioni indoor, ma ne ha attestato l'efficacia pratica attraverso un prototipo solido.

Bibliografia

[1] International Telecommunication Union. Feasibility study on deployment and implementation of a cell broadcast service (cbs) solution for sending alert messages (republic of moldova), 2023. URL https://www.itu.int/en/ITU-D/Regional-Presence/Europe/Documents/Publications/2023/Feasibility%20study%20Moldova.pdf.

[2] Dipartimento della Protezione Civile. Cos'è it-alert, 2024. URL https://www.it-alert.it/it/cose/.

[3] Federal Emergency Management Agency. Integrated public alert & warning system (ipaws), 2024. URL https://www.fema.gov/emergency-managers/practitioners/integrated-public-alert-warning-system.

[4] Dipartimento della Protezione Civile. Come funziona it-alert, 2024. URL https://www.it-alert.it/it/come-funziona/.

[5] Dipartimento della Protezione Civile. Indicazioni operative per la sperimentazione di messaggi di allarme pubblico it-alert: precipitazioni intense, 2024. URL https://www.protezionecivile.gov.it/static/c344e2bfddc8184f73842193fe4eb186/indicazioni-operative-la-sperimentazione-di-messaggi-di-allarme-pubblico-it-alert-precipitazioni-intense-approvate-cu-28-nov-2024.pdf. Capitolo 3.6.

[6] Dipartimento della Protezione Civile. What is it-alert, 2024. URL https://www.it-alert.it/en/what-it/.

[7] U.S. Department of Homeland Security. Integrated public alert and warning system (ipaws), 2024. URL https://www.dhs.gov/publication/dhsfemapia-046-integrated-public-alert-and-warning-system-open-platform-emergency.

[8] Federal Communications Commission. Common alerting protocol, 2021. URL https://www.fcc.gov.

[9] FEMA IPAWS. The ipaws system overview, 2020. URL https://www.weather.gov.

[10] FEMA IPAWS. Wireless emergency alerts (wea), 2021. URL https://www.fema.gov.

[11] International Standards Organization (ISO). Iso 22324: Public warning, 2018. URL https://www.iso.org.

[12] Martin Fowler and James Lewis. Microservices: a definition of this new architectural term. *martinfowler.com*, 2014. URL https://martinfowler.com/articles/microservices.html.

[13] Sam Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN 978-1-491-95035-7.

[14] Nicola Dragoni, Ivan Lanese, Svend Frølund Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pages 195–216, 2017. doi: 10.1007/978-3-319-67425-4_12. URL https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12.

[15] Nada S. Ahmed and Nuredin A. Ahmed. Microservices vs. monolithic architectures: The differential structure between two architectures. *International Journal of Applied Sciences and Technology*, 4

(3):484–490, 2022. doi: 10.47832/2717-8234.12.47. URL https://doi.org/10.47832/2717-8234.12.47.

[16] Mohamed Hassan. Software architecture between monolithic and microservices approach. *SSRN Electronic Journal*, 2024. doi: 10.2139/ssrn.4753649. URL https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4753649.

[17] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137:106600, 2021. doi: 10.1016/j.infsof.2021.106600. URL https://doi.org/10.1016/j.infsof.2021.106600.

[18] Daniel dos Santos Krug, Rafael Chanin, and Afonso Sales. Exploring the pros and cons of monolithic applications versus microservices. In *Proceedings of the 26th International Conference on Enterprise Information Systems (ICEIS)*, pages 257–263, 2024. doi: 10.5220/0012703300003690. URL https://doi.org/10.5220/0012703300003690.

[19] IBM Cloud Learn Hub. What is a monolithic architecture?, 2023. URL https://www.ibm.com/think/topics/monolithic-architecture.

[20] Amazon Web Services. Monolithic vs. microservices architecture, 2023. URL https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture.

[21] Atlassian. Microservices vs. monolith, 2022. URL https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[22] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016. doi: 10.1109/MS.2016.64. URL https://doi.org/10.1109/MS.2016.64.

[23] NetworkX Developers. Networkx: Network analysis in python, 2025. URL https://networkx.org/documentation/stable/index.html.

[24] GeoPandas Developers. Geopandas: Python tools for geographic data, 2025. URL https://geopandas.org/.

[25] Shapely Developers. Shapely: manipulation and analysis of planar geometric objects, 2025. URL https://shapely.readthedocs.io/.

[26] The psycopg Project. psycopg2 – postgresql database adapter for python, 2025. URL https://www.psycopg.org/.

[27] Jane Doe and Alan Smith. A comparative analysis of programming languages used in microservices. *Journal of Distributed Systems*, 2024. URL https://www.researchgate.net/publication/389390900_A_Comparative_Analysis_of_Programming_Languages_Used_in_Microservices.

[28] Python Software Foundation. asyncio — asynchronous i/o, 2025. URL https://docs.python.org/3/library/asyncio.html.

[29] Pika Developers. Pika – a python rabbitmq (amqp 0-9-1) client library, 2025. URL https://pika.readthedocs.io/.

[30] RabbitMQ Documentation. Tutorial: Simple python publisher and consumer, 2025. URL https://www.rabbitmq.com/tutorials/tutorial-three-python.html.

[31] Netguru Blog. Node.js vs python: Which to choose in 2025?, 2025. URL https://www.netguru.com/blog/node-js-vs-python.

[32] RabbitMQ Documentation. Client libraries and developer tools, 2025. URL https://www.rabbitmq.com/client-libraries/devtools.html.

[33] Ewa Kaciuczyk and Leszek Kotulski. Efficient use of message brokers in distributed iot systems. *IFAC-PapersOnLine*, 53(2):13339–13344, 2020. doi: 10.1016/j.ifacol.2020.12.1750.

[34] Daniel Rosam. Kafka vs rabbitmq: Key differences and use cases. *Confluent Blog*, 2024. URL `https://www.confluent.io/learn/rabbitmq-vs-apache-kafka/`.

[35] Pivotal Software. Rabbitmq performance measurements, 2020. URL `https://www.rabbitmq.com/docs/monitoring`.

[36] Pika Developers. Pika documentation, 2025. URL `https://pika.readthedocs.io/en/stable/`.

[37] RabbitMQ Documentation. High availability in rabbitmq, 2025. URL `https://www.rabbitmq.com/ha.html`.

[38] RabbitMQ Documentation. Rabbitmq deployment and configuration, 2025. URL `https://www.rabbitmq.com/configure.html`.

[39] Apache Software Foundation. Apache kafka documentation, 2025. URL `https://kafka.apache.org/documentation/`.

[40] S. Mishra and R. Singh. Rabbitmq vs. kafka: A comparative study for event-driven systems. *Journal of Distributed Computing*, 18(4):231–245, 2024.

[41] J. Kreps. Benchmarking apache kafka and rabbitmq. Confluent Blog, 2025. URL `https://www.confluent.io/blog/kafka-fastest-messaging-system/`.

[42] Confluent. Confluent kafka python client, 2025. URL `https://docs.confluent.io/platform/current/clients/confluent-kafka-python/`.

[43] RabbitMQ Documentation. Rabbitmq documentation, 2025. URL `https://www.rabbitmq.com/documentation.html`.

[44] PostgreSQL Global Development Group. Postgresql documentation, 2025. URL https://www.postgresql.org/docs/.

[45] PostGIS Project Steering Committee. Postgis documentation, 2025. URL https://postgis.net/docs/.

[46] PostgreSQL Global Development Group. Listen and notify, 2025. URL https://www.postgresql.org/docs/current/sql-notify.html.

[47] PostgreSQL Global Development Group. Json types in postgresql, 2025. URL https://www.postgresql.org/docs/current/datatype-json.html.

[48] PostgreSQL Global Development Group. Scalability and performance, 2025. URL https://www.postgresql.org/docs/17/performance-tips.html.

[49] RethinkDB Documentation Team. Rethinkdb documentation, 2025. URL https://rethinkdb.com/docs/.

[50] RethinkDB Documentation Team. Changefeeds in rethinkdb, 2025. URL https://rethinkdb.com/docs/changefeeds/.

[51] RethinkDB Documentation Team. Reql query language, 2025. URL https://rethinkdb.com/docs/introduction-to-reql/.

[52] A. Patel and R. Kumar. Performance analysis of nosql databases in real-time applications. *Journal of Database Management*, 34(2):67–82, 2023.

[53] RethinkDB Documentation Team. Geospatial queries in rethinkdb, 2025. URL https://rethinkdb.com/docs/geo-support/.

[54] RethinkDB Documentation Team. Python driver for rethinkdb, 2025. URL https://rethinkdb.com/docs/install-drivers/python/.

[55] YAML. Yaml specification, 2025. URL https://yaml.org/spec/1.2.2/.

[56] PyYAML Project. Pyyaml documentation, 2025. URL https://pyyaml.org/wiki/PyYAMLDocumentation.

[57] ECMA International. Json specification (ecma-404), 2025. URL https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[58] Python Software Foundation. Configuration files in python, 2025. URL https://docs.python.org/3/library/configparser.html.

[59] YAML. Yaml documentation, 2025. URL https://yaml.org/.

[60] Python Software Foundation. Json module documentation, 2025. URL https://docs.python.org/3/library/json.html.

[61] Docker, Inc. Docker compose specification, 2025. URL https://docs.docker.com/compose/compose-file/.

[62] A. Gupta. Security risks in python configuration files. *Journal of Cybersecurity*, 19(3):89–102, 2022.

[63] Alertus Technologies. Alertus mass notification system, 2025. URL https://www.alertus.com/system.

[64] Alexander Salveson Nossum. Developing a framework for describing and comparing indoor maps, 2013. URL https://www.researchgate.net/publication/272310681_Developing_a_Framework_for_Describing_and_Comparing_Indoor_Maps#:~:text=visualisation%20of%20outdoor%20environments,can%20be%20described%20by%20their.

[65] Jaiteg Singh, Noopur Tyagi, Saravjeet Singh, Ahmad Ali AlZubi, Firas Ibrahim AlZubi, Sukhjit Singh Sehra, and Farman Ali. Enhancing indoor navigation in intelligent transportation systems with 3d rif and

quantum gis, 2023. URL https://www.mdpi.com/2071-1050/15/22/15833.

[66] Open Geospatial Consortium. Ogc publishes indoorgml 2.0 part 1: Conceptual model standard, 2025. URL https://www.ogc.org/announcement/ogc-publishes-indoorgml-2-0-part-1-conceptual-model-standard.

[67] Lawrence V. Stanislawski Barry J Kronenfeld, Barbara P. Buttenfield. Map generalization for the future: Editorial comments on the special issue, 2020. URL https://www.mdpi.com/2220-9964/9/8/468.

[68] P. L.M. Flikweert. Automatic extraction of an indoorgml navigation graph from an indoor point cloud, 2019. URL https://repository.tudelft.nl/record/uuid:b11f5b57-5362-4b45-bed6-d5bc154d86aa#:~:text=building,is%20modelled%20according%20to%20the.

[69] SpatialTech. Database routing approach: Using pgrouting and postgis for indoor navigation, 2024. URL https://www.spatialtech.org/pgrouting-geospatial-routing-network.html#:~:text=Routing%20is%20a%20widely%20used,in%20QGIS%20from%20SQL%20commands.

[70] Silvana Philippi Camboim Rhaíssa Viana Sarot, Luciene Stamato Delazari. Proposal of a spatial database for indoor navigation, 2020. URL https://pdfs.semanticscholar.org/5992/9cfb001f4195bfcdaa3f560ebcd417387d69.pdf.

[71] Maciej Nazarczuk and Artur Niewiadomski. A pathfinding module for the indoor navigation system navisecure, 2024. URL https://www.czasopisma.uph.edu.pl/studiainformatica/article/download/3974/3693#:~:text=,the%20basic%20concepts%20behind.

[72] KokaTic. Openlayers vs. leaflet: A comparative guide, 2024. URL https://koka-tic.medium.com/openlayers-vs-leaflet-a-comparative-guide-6c4341e82ee8.

[73] Open Geospatial Consortium (OGC). Indoorgml - ogc standard for indoor spatial information, 2025. URL https://www.indoorgml.net.

[74] Mapbox. Terms of service, 2024. URL https://www.mapbox.com/legal/tos. Condizioni d'uso dei servizi Mapbox.

[75] Mapbox. Product terms, 2023. URL https://www.mapbox.com/legal/product-terms. Termini di prodotto e limitazioni d'uso.

[76] OpenCV. Opencv: Open source computer vision library, 2025. URL https://opencv.org/.

[77] OpenCV Team. Canny edge detection — opencv 4.x tutorials, 2025. URL https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html. Tutorial ufficiale sull'algoritmo di Canny con pre-filtraggio Gaussiano.

[78] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference (SciPy 2008)*, 2008. doi: 10.25080/TCWV9851. URL https://proceedings.scipy.org/articles/TCWV9851. Articolo scientifico di riferimento su NetworkX.

[79] Leaflet. Leaflet: An open-source javascript library for mobile-friendly interactive maps, 2025. URL https://leafletjs.com/.

[80] Leaflet Contributors. Leaflet api reference, 2025. URL https://leafletjs.com/reference.html. Versione 1.9.x; libreria JavaScript open-source per mappe interattive.

[81] Leaflet Contributors. Imageoverlay example, 2025. URL https://leafletjs.com/examples/overlays/. Documentazione ufficiale: sovrapposizione di immagini con bounds personalizzati.

[82] Leaflet Contributors. Crs.simple example, 2025. URL https://github.com/Leaflet/Leaflet/blob/main/docs/examples/crs-simpl

e/crs-simple-example3.md. Esempio ufficiale per coordinate locali pixel-based.

[83] Neo4j, Inc. Neo4j cypher manual — introduction and overview, 2025. URL https://neo4j.com/docs/cypher-manual/current/introduction/. Linguaggio declarativo per database a grafo.

[84] PostgreSQL Global Development Group. Create trigger — postgresql documentation, 2025. URL https://www.postgresql.org/docs/current/sql-createtrigger.html. Documentazione ufficiale: trigger e funzioni di trigger.

[85] PostgreSQL Global Development Group. Trigger functions (pl/pgsql), 2025. URL https://www.postgresql.org/docs/current/plpgsql-trigger.html. Documentazione ufficiale: funzioni PL/pgSQL per trigger.

[86] Leaflet Contributors. Using geojson with leaflet, 2025. URL https://leafletjs.com/examples/geojson/. Supporto nativo a layer GeoJSON.

[87] Mapbox. Access tokens (help), 2025. URL https://docs.mapbox.com/help/glossary/access-token/. Requisiti di token per l'uso degli SDK/API Mapbox.

[88] MazeMap AS. Digital mapping is changing the way hospitals handle compliance, 2025. URL https://www.mazemap.com/post/digital-mapping-hospitals-compliance#:~:text=Solution%3A%20Digital%20mapping%20provides%20interactive%2C,digital%20kiosks%2C%20and%20wayfinding%20displays.

[89] Anna Carolina Rosa, Mariana Cabral Falqueiro, Rodrigo Bonacin, Fábio Lúcio Lopes de Mendonça, Geraldo Pereira Rocha Filho, and Vinícius Pereira Gonçalves. Evacuai: An analysis of escape routes in indoor environments with the aid of reinforcement learning, 2023. URL

https://www.mdpi.com/1424-8220/23/21/8892#:~:text=rapid%20evacuation%20of%20people%20from,agent%20to%20be%20trained%20in.

Ringraziamenti