Università degli Studi di Padova

GlitchHub TEAM

Specifica Tecnica

Versione **0.5.0**

Stato	Bozza
Distribuzione	GlitchHub Team Prof. Vardanega Tullio Prof. Cardin Riccardo

Registro Modifiche

Ver.	Data	Autore	Verificatore	Descrizione
0.5.0	02/04/2026	Riccardo Graziani	Jaume Bernardi	Aggiornamento tecnologie frontend, descrizione pattern frontend, ampliata Sezione 3.1.3.3
0.4.1	17/03/2026	Jaume Bernardi	Alessandro Dinato	Applicazione correzioni nate dalla verifica
0.4.0	17/03/2026	Jaume Bernardi	Alessandro Dinato	Prima stesura aggiornamento tecnologie e versioni
0.3.1	17/03/2026	Jaume Bernardi	Elia Ernesto Stellin	Applicazione modifiche richieste in verifica
0.3.0	16/03/2026	Jaume Bernardi	Elia Ernesto Stellin	Prima stesura dei paragrafi di architettura System Context, Container, Component e Deployment
0.2.0	03/03/2026	Siria Salvalaio	Alessandro Dinato	Inizio stesura sezione Sezione 3.3
0.1.0	02/03/2026	Siria Salvalaio	Alessandro Dinato	Piccole modifiche suggerite dal verificatore e stesura sezione Sezione 3.2
0.0.1	01/03/2026	Siria Salvalaio	Alessandro Dinato	Bozza struttura documento, sezioni Sezione 1, Sezione 1.3 e inizio Sezione 2

Indice

Indice delle tabelle

Indice delle figure

1. Introduzione

1.1. Scopo del documento

Il presente documento definisce in modo analitico l'architettura del sistema software, offrendo una scomposizione accurata delle sue componenti, delle logiche di interazione e della loro distribuzione nel sistema. Esso è il sostegno progettuale per la fase di realizzazione, garantendo continuità con i risultati ottenuti nel **PoC**$_G$ e introducendo accorgimenti necessari per elevare il grado di maturità e robustezza architetturale.

In particolare il documento si occupa di:

- **Architettura Logica e Deployment**: analisi delle componenti, delle loro interconnessioni e della loro collocazione in fase di esecuzione.
- **Pattern e Idiomi**: descrizione dei criteri di design utilizzati e delle soluzioni a basso livello scelte per ottimizzare il codice.
- **Dettaglio Progettuale**: approfondimento delle scelte tecniche volte a massimizzare la comprensione e la manutenibilità del sistema.

1.2. Glossario

Il **glossario** è un documento redatto dal gruppo e aggiornato nell'arco del progetto didattico, con lo scopo di fornire definizioni coerenti per i termini tecnici e quelli specifici relativi al corso di Ingegneria del Software.

Questo documento è fondamentale per garantire una comprensione uniforme della documentazione prodotta ai lettori esterni dal gruppo e per definire un riferimento interno al gruppo, riducendo possibili ambiguità interpretative.

Per indicare che la definizione di una parola o di un concetto è disponibile, si è deciso di utilizzare la seguente notazione: **definizione nel glossario**$_G$.

1.3. Riferimenti

1.3.1. Riferimenti normativi

- **Capitolato d'appalto C7**:
  - https://www.math.unipd.it/~tullio/IS-1/2025/Progetto/C7.pdf
  - **Ultimo accesso**: 01/03/2026
- **Norme di Progetto v1.3.0**:
  - https://glitchhub-team.github.io/pdf/RTB/DocumentiInterni/NormeProgetto.pdf
  - **Ultimo accesso**: 01/03/2026

1.3.2. Riferimenti informativi

- **Modello C4**:
  - https://c4model.com/
  - **Ultimo accesso**: 01/03/2026
- **Documentazione linguaggio Go**$_G$:
  - https://go.dev/doc/
  - **Ultimo accesso**: 01/03/2026
- **Documentazione linguaggio Typescript**$_G$:
  - https://www.typescriptlang.org/docs/
  - **Ultimo accesso**: 01/03/2026
- **Documentazione Angular**$_G$:

- https://angular.dev/overview
- **Ultimo accesso**: 02/04/2026
- **Documentazione Gin**$_G$:
- https://gin-gonic.com/en/docs/
- **Ultimo accesso**: 01/03/2026
- **Documentazione TimescaleDB**:
- https://docs.timescale.com/
- **Ultimo accesso**: 01/03/2026
- **Documentazione NATS**$_G$:
- https://docs.nats.io/
- **Ultimo accesso**: 01/03/2026
- **Documentazione Docker**$_G$:
- https://docs.docker.com/
- **Ultimo accesso**: 01/03/2026
- **Documentazione Grafana**$_G$:
- https://grafana.com/docs/
- **Ultimo accesso**: 01/03/2026
- **Documentazione Prometheus**$_G$:
- https://prometheus.io/docs/introduction/overview/
- **Ultimo accesso**: 01/03/2026

2. Tecnologie

Per lo sviluppo del sistema abbiamo scelto uno stack tecnologico moderno e solido, selezionando ogni strumento con l'obiettivo di supportare bene un'architettura a microservizi che sia facile da gestire e capace di crescere nel tempo. Le nostre scelte sono state condotte dalla necessità di creare un'infrastruttura per la gestione dei dati IoT che funzioni bene anche sotto carico, garantendo che il flusso di informazioni dai sensori BLE sia sempre veloce e affidabile.

Di seguito si trovano l'elenco dei componenti scelti, con breve spiegazione delle loro caratteristiche principali.

2.1. Linguaggi e ambienti di programmazione

Tecnologia	Versione	Descrizione
Go	1.26.0	**Go**$_G$ è un linguaggio compilato e staticamente tipizzato che combina semplicità sintattica e prestazioni di alto livello in contesti distribuiti. La sua gestione nativa della concorrenza permette di eseguire numerosi processi simultanei con un consumo minimo di risorse hardware. Nel progetto viene utilizzato per realizzare il **Gateway**$_G$ simulato, in cui diverse **goroutine**$_G$ operano in parallelo per emulare il comportamento simultaneo dei sensori **BLE**$_G$. È inoltre impiegato per lo sviluppo del Data consumer, incaricato di processare i flussi dati in ingresso, e per il Backend della dashboard, garantendo velocità e scalabilità nella gestione delle richieste **API**$_G$ del **Cloud**$_G$ Layer.
Typescript	5.9.2	**TypeScript**$_G$ è un superset tipizzato di **JavaScript**$_G$ che introduce il controllo statico dei tipi, migliorando drasticamente la leggibilità e la manutenibilità del codice. Compila in **JavaScript**$_G$ standard, garantendo piena compatibilità con l'ecosistema **Node.js**$_G$ e i browser moderni. Viene adottato per lo sviluppo dei microservizi e del frontend **Angular**$_G$, poiché permette di rilevare errori in fase di scrittura, assicurando la robustezza e la qualità del codice necessarie per superare i test di validazione richiesti dal capitolato
Node.js	20.19.0	**Node.js**$_G$ è un ambiente di runtime open-source e multipiattaforma basato sul motore V8 di Google, progettato per l'esecuzione di codice **JavaScript**$_G$ lato server. Grazie al suo modello di I/O non bloccante e orientato agli eventi, garantisce un'elevata efficienza nella gestione di connessioni simultanee con un overhead minimo. Nel progetto viene utilizzato come base per l'esecuzione dei microservizi e per la gestione delle dipendenze tramite **npm**$_G$, assicurando un ambiente stabile e scalabile per l'integrazione dei diversi componenti del sistema e la comunicazione con il **Cloud**$_G$ Layer.
npm	10.8.2	npm (Node Package Manager) è il gestore di pacchetti predefinito per l'ecosistema **Node.js**$_G$. Fornisce un registro pubblico di librerie e strumenti JavaScript e un client da riga di comando che permette di installare, aggiornare, pubblicare e gestire dipendenze all'interno di un progetto.

Tabella 1: Linguaggi e ambienti di programmazione.

2.2. Framework per la codifica per il backend della dashboard

Tecnologia	Versione	Descrizione
Gin	1.11.0	**Gin**$_G$ è un framework web HTTP ad alte prestazioni per Go che sfrutta un router ottimizzato per garantire velocità di esecuzione superiori e un utilizzo ridotto di risorse. Offre funzionalità predefinite per la gestione di middleware e la validazione dei dati, accelerando lo sviluppo di interfacce affidabili. È stato scelto per implementare le **API**$_G$ e la logica backend del **Cloud**$_G$ Layer, assicurando risposte rapide alle richieste degli utenti e un'integrazione fluida tra i vari servizi del sistema.

Tabella 2: Framework per la codifica per il backend della dashboard.

2.3. Framework per lo sviluppo del frontend

Tecnologia	Versione	Descrizione
Angular	21.1.0	**Angular**$_G$ è un framework open-source per lo sviluppo di applicazioni web Single Page **SPA**$_G$. Utilizza un'architettura basata su componenti che garantisce modularità e manutenibilità del codice. Viene impiegato per la realizzazione dell'interfaccia utente della dashboard, permettendo la visualizzazione in tempo reale dei dati acquisiti dai sensori e la configurazione semplificata di nuovi gateway o sensori simulati.

Tabella 3: Framework per lo sviluppo del frontend.

2.4. Tecnologie per la gestione di dati temporali

Tecnologia	Versione	Descrizione
TimescaleDB	2.25.2-pg18	**TimescaleDB**$_G$ è un database open-source per serie temporali costruito su **PostgreSQL**$_G$, che ottimizza l'archiviazione e la velocità di interrogazione di dati indicizzati nel tempo. La sua architettura permette di gestire grandi volumi di informazioni mantenendo la piena compatibilità con il linguaggio SQL. È stato scelto per archiviare i dati provenienti dai sensori **BLE**$_G$, garantendo l'ingestione rapida e la persistenza necessarie per il monitoraggio in tempo reale e le analisi storiche previste dal sistema.
SQLite	3.51.2	SQLite è un motore di database relazionale leggero, autonomo e basato su file, progettato per essere incorporato direttamente nelle applicazioni senza richiedere un server esterno. Utilizza un singolo

		file come archivio dati e implementa gran parte dello standard SQL, offrendo un database transazionale, affidabile e altamente portabile.
PostgreSQL	18.3	**PostgreSQL**$_G$ è un sistema di gestione di database relazionali open-source e progettato per garantire affidabilità, estensibilità e conformità ACID. È noto per la sua robustezza, la ricchezza delle funzionalità e la capacità di gestire carichi di lavoro complessi in ambienti enterprise, nonché dati complessi e query analitiche avanzate. Nel progetto viene utilizzato come motore di persistenza principale nel **Cloud**$_G$ Layer, sfruttando la sua architettura robusta per garantire l'integrità dei dati e la scalabilità necessaria a gestire l'archiviazione a lungo termine delle informazioni.

Tabella 4: Tecnologie per la gestione di dati temporali.

2.5. Tecnologie per la comunicazione e messaggistica

Tecnologia	Versione	Descrizione
NATS	2.12.5	**NATS**$_G$ è un message broker e sistema di messaggistica distribuito ad alte prestazioni, progettato per lo scambio di dati rapido e affidabile tra servizi indipendenti. La sua architettura leggera supporta modelli di comunicazione asincrona, garantendo bassa latenza e alta scalabilità nel transito delle informazioni. È stato selezionato per orchestrare il flusso di dati tra i microservizi del **Cloud**$_G$ Layer, assicurando uno smistamento efficiente dei pacchetti provenienti dai **gateway**$_G$ verso i sistemi di persistenza e monitoraggio.
nsc	2.12.0	nsc è lo strumento ufficiale per la gestione delle identità, delle autorizzazioni e delle configurazioni di sicurezza nell'ecosistema NATS. Permette di creare e amministrare operatori, account, utenti, chiavi e JWT utilizzati dal sistema di autenticazione e autorizzazione di NATS

Tabella 5: Tecnologie per la comunicazione e messaggistica.

2.6. Tecnologie per la virtualizzazione e deployment

Tecnologia	Versione	Descrizione
Docker		**Docker**$_G$ è una piattaforma di containerizzazione che permette di pacchettizzare i microservizi e le loro dipendenze in unità isolate e portatili. Garantisce che il software funzioni in modo identico in ogni ambiente, eliminando i problemi di configurazione tra sviluppo e produzione. È stato scelto per semplificare il deployment dell'infrastruttura **Cloud**$_G$ e facilitare l'orchestrazione dei servizi, assicurando la scalabilità e la manutenibilità richieste dal capitolato.

Ubuntu	24.04 LTS	Ubuntu è una distribuzione **Linux**$_G$ basata su Debian, scelta come immagine di base per la containerizzazione dei microservizi del sistema. Grazie al supporto a lungo termine, garantisce un ambiente di esecuzione estremamente stabile, sicuro e costantemente aggiornato. Nel progetto viene utilizzata per creare container leggeri e standardizzati, assicurando che i vari componenti operino su una base software coerente, riducendo le discrepanze tra l'ambiente di sviluppo locale e il deployment finale.

Tabella 6: Tecnologie per la virtualizzazione e deployment.

2.7. Tecnologie per il monitoraggio dei microservizi

Tecnologia	Versione	Descrizione
Grafana		**Grafana**$_G$ è una piattaforma open-source per l'analisi e la visualizzazione di dati che permette di creare dashboard dinamiche e interattive. Supporta la rappresentazione grafica di metriche complesse attraverso pannelli altamente personalizzabili. È stata scelta per fornire agli utenti e agli amministratori uno strumento intuitivo per monitorare i dati raccolti dai sensori **BLE**$_G$ in tempo reale. Consente di visualizzare immediatamente lo stato del sistema e identificare eventuali anomalie operative.
Prometheus		**Prometheus**$_G$ è un sistema di monitoraggio e allerta specializzato nella raccolta di metriche sotto forma di serie temporali. Utilizza un modello di recupero dati ottimizzato per le architetture a microservizi e ambienti cloud. Viene adottato per osservare le prestazioni dell'infrastruttura e i volumi di traffico gestiti dai **gateway**$_G$. La sua integrazione permette di generare alert automatici qualora un componente o un gateway smetta di funzionare correttamente.

Tabella 7: Tecnologie per il monitoraggio dei microservizi.

2.8. Librerie

Tecnologia	Versione	Descrizione
gomock	0.6.0	gomock è un framework di mocking per il linguaggio **Go**$_G$, utilizzato per facilitare la scrittura di unit test isolati e affidabili. Permette di generare automaticamente implementazioni fittizie di interfacce complesse, consentendo agli sviluppatori di simulare e verificare le interazioni tra i diversi componenti del sistema, come la comunicazione tra il **Gateway**$_G$ e i servizi di backend, senza la necessità di dipendenze esterne attive durante la fase di test.

GORM	1.31.1	GORM è una libreria di ORM (Object-Relational Mapping) per il linguaggio **Go**$_G$, progettata per astrarre e semplificare l'interazione tra il codice applicativo e i database relazionali. Fornisce un livello di astrazione sopra SQL, permettendo di definire modelli, eseguire query, gestire relazioni e operazioni CRUD attraverso un'API idiomatica e coerente con lo stile di Go.
GORM PostgresSQL Driver	1.6.0	GORM PostgresSQL Driver è l'interfaccia tra il framework GORM e il database **PostgreSQL**$_G$. Nel progetto svolge il ruolo fondamentale di tradurre le operazioni definite nel codice **Go**$_G$ in dialetto SQL compatibile con **TimescaleDB**$_G$.
jwt	4.5.2	jwt è una versione della libreria Go dedicata alla creazione, firma, validazione e gestione dei JSON Web Token. Fornisce un'implementazione conforme agli standard RFC 7519, permettendo di generare token sicuri per l'autenticazione e l'autorizzazione nelle applicazioni scritte in **Go**$_G$.
GoDotEnv	1.5.1	GoDotEnv è una libreria per il linguaggio **Go**$_G$ che permette di caricare variabili d'ambiente da file .env all'interno dell'applicazione. Replica il comportamento del pacchetto dotenv diffuso in altri ecosistemi, facilitando la gestione di configurazioni esterne al codice sorgente.
google/uuid	1.6.0	uuid è una libreria per il linguaggio **Go**$_G$ che fornisce un'implementazione completa e conforme agli standard RFC 4122 per la gestione degli UUID (Universally Unique Identifier), ovvero identificatori univoci a livello globale. Può effettuare la loro generazione il parsing da stringa, la validazione, la serializzazione e la manipolazione in vari formati.
Uber Fx	1.24.0	Fx è un framework di dependency injection per il linguaggio **Go**$_G$, progettato fornire una struttura modulare e dichiarativa per inizializzare, configurare e orchestrare i diversi elementi di un'applicazione in modo sicuro, scalabile e facilmente testabile. Nel progetto viene utilizzato per orchestrare l'avvio e lo spegnimento dei vari servizi del backend, come il server **Gin**$_G$, la connessione a **TimescaleDB**$_G$ e il client **NATS**$_G$.
Uber Zap	1.27.1	Zap è una libreria di logging ad alte prestazioni per applicazioni scritte in **Go**$_G$. È progettata per offrire log strutturati, veloci e a basso overhead, rendendola particolarmente adatta a sistemi ad alta scalabilità, microservizi e applicazioni con requisiti di osservabilità rigorosi.
NATS Client	1.49.0	NATS Client è la libreria ufficiale per il linguaggio **Go**$_G$ che permette l'interazione tra i servizi applicativi e il sistema

		di messaggistica **NATS**$_G$. Nel progetto viene impiegata per implementare i pattern di comunicazione asincrona, permettendo al **Gateway**$_G$ simulato di pubblicare i dati dei sensori e al Data consumer di sottoscriversi ai flussi in ingresso.
RxJS	7.8.0	RxJS è una libreria per la programmazione reattiva in **JavaScript**$_G$ e **TypeScript**$_G$, che consente di gestire flussi di dati asincroni e basati su eventi attraverso l'uso di *Observable*. Nel progetto viene utilizzata principalmente nel frontend **Angular**$_G$ per orchestrare la gestione dei dati in tempo reale provenienti dai sensori **BLE**$_G$, facilitando la sincronizzazione tra le interfacce utente e i servizi backend.
tslib	2.3.0	tslib è una libreria per il linguaggio **TypeScript**$_G$ che fornisce helper runtime per le funzionalità del linguaggio, come l'estensione delle classi, la gestione dei decoratori e altre operazioni comuni. Nel progetto viene utilizzata per ottimizzare il codice **TypeScript**$_G$ compilato e ridurre la duplicazione di codice tra i moduli.
Chart.js	4.5.1	Chart.js è una libreria **JavaScript**$_G$ per la creazione di grafici interattivi e personalizzabili. Nel progetto viene utilizzata nel frontend **Angular**$_G$ per visualizzare i dati dei sensori in tempo reale e storici attraverso grafici dinamici e intuitivi.
ng2-charts	10.0.0	ng2-charts è una libreria *wrapper* che fornisce componenti per integrare Chart.js nelle applicazioni **Angular**$_G$, facilitando la creazione di grafici reattivi e personalizzabili.
jsdom	27.1.0	jsdom è una libreria **JavaScript**$_G$ che fornisce un'implementazione del DOM e di altre API web standard in ambiente **Node.js**$_G$, permettendo di eseguire test e manipolazioni del DOM senza la necessità di un browser reale. Nel progetto viene utilizzata principalmente nei test di unità del frontend **Angular**$_G$ per simulare l'ambiente del browser e verificare il comportamento dei componenti che interagiscono con il DOM.

Tabella 8: Librerie.

2.9. Tecnologie per analisi statica

Tecnologia	Versione	Descrizione

Golangci-lint	2.10.1	Golangci-lint è un aggregatore di **linter**$_G$ per il linguaggio **Go**$_G$, progettato per analizzare il codice sorgente alla ricerca di errori sintattici, problemi di stile e potenziali bug logici. Tra le altre funzionalità, aiuta anche nell'identificare tempestivamente porzioni di codice inefficienti o non sicure.
gofumpt	0.9.2	gofumpt è uno strumento di formattazione del codice per il linguaggio **Go**$_G$, progettato come una versione più rigorosa e coerente di gofmt. Applica regole aggiuntive di stile e formattazione per garantire un codice più uniforme, leggibile e conforme a linee guida più restrittive rispetto allo standard. Rimane pienamente compatibile con gofmt.
Gopls	0.21.1	Gopls è il language server ufficiale per il linguaggio **Go**$_G$, sviluppato dal team Go per fornire funzionalità avanzate agli editor tramite il protocollo LSP (Language Server Protocol). Offre servizi come autocompletamento, navigazione del codice, refactoring, analisi statica, suggerimenti in tempo reale e gestione intelligente dei moduli Go.
ESLint	10.0.2	ESLint è uno strumento di analisi statica per il linguaggio **JavaScript**$_G$ e **TypeScript**$_G$, progettato per identificare e correggere problemi di stile, errori di sintassi e potenziali bug nel codice.
Prettier	3.8.1	Prettier è un formattatore di codice per **JavaScript**$_G$, **TypeScript**$_G$ e altri linguaggi, progettato per applicare uno stile di codifica coerente e uniforme. Automatizza la formattazione del codice secondo regole predefinite, migliorando la leggibilità e riducendo le discussioni sullo stile all'interno del team di sviluppo.

Tabella 9: Tecnologie per analisi statica.

2.10. Tecnologie per analisi dinamica

Tecnologia	Versione	Descrizione
Delve	1.26.0	Delve è il debugger ufficiale per il linguaggio **Go**$_G$ progettato per fornire un'ispezione profonda dello stato interno delle applicazioni durante la loro esecuzione. A differenza dei debugger generici, è ottimizzato per comprendere le strutture dati native di **Go**$_G$, come le **goroutine**$_G$, i canali e le mappe.
NATS CLI	2.12.0	NATS Command Line Interface è lo strumento da riga di comando ufficiale per interagire con un server o un cluster NATS. Permette di gestire connessioni, pubblicare e ricevere messaggi, amministrare account e utenti, monitorare lo stato del sistema e lavorare con funzionalità avanzate.
Vitest	4.0.8	Vitest è un framework di test per **JavaScript**$_G$ e **TypeScript**$_G$, progettato per essere veloce e leggero, con supporto per test unitari, test di integrazione e test end-to-end. Offre funzionalità come il mocking,

		la copertura del codice e l'esecuzione parallela dei test, rendendolo adatto a progetti di qualsiasi dimensione. Nel progetto viene utilizzato principalmente per i test del frontend **Angular**$_G$, garantendo la qualità e l'affidabilità del codice che gestisce l'interfaccia utente e le interazioni con i servizi backend.

Tabella 10: Tecnologie per analisi dinamica.

3. Architettura

L'architettura del sistema è basata su un modello a **microservizi**, in cui ogni componente funzionale viene eseguito come un'unità indipendente e isolata per garantire la massima resilienza dell'intero ecosistema.

3.1. Architettura logica

L'architettura logica del sistema è documentata seguendo il modello C4, utile per descrivere il software su diversi livelli di astrazione e da molteplici punti di vista fornendo la scomposizione dell'applicativo in container, componenti, relazioni tra gli elementi e tra gli utenti.

3.1.1. Context

L'analisi dell'architettura logica inizia con il diagramma di System Context, che definisce il perimetro del progetto. In questa fase, definita dal livello di astrazione più alto, non si analizzano le tecnologie interne o implementative ma ci si focalizza esclusivamente sulle interazioni tra i componenti interni del sistema e le interazioni coi componenti esterni e utenti umani.

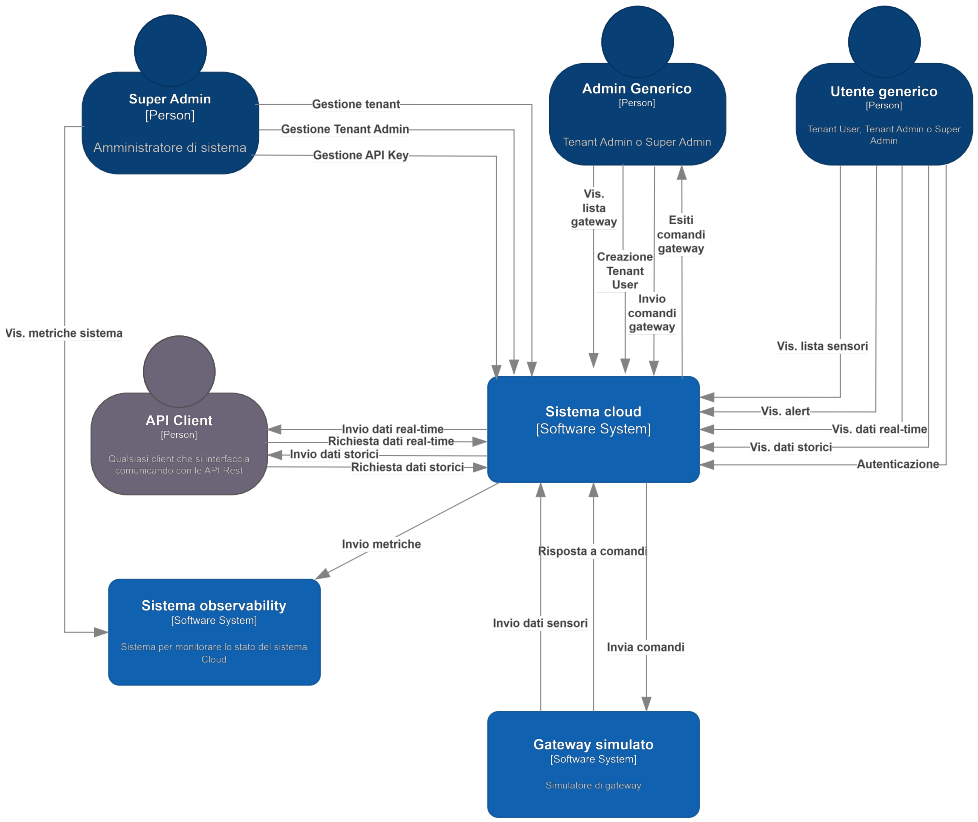

Figura 1: System Context diagram

Il Sistema Cloud è il fulcro dell'intero ambiente in quanto principale fornitore dei servizi del software, quali la ricezione e memorizzazione dei dati di qualsiasi natura (nel database più opportuno) e il loro invio verso la dashboard, nonché la definizione dei perimetri di sicurezza e correlata autenticazione degli utenti. ed interagisce con tutti gli altri utenti ed elementi presenti, ovvero:

- Super Admin, tipo di utente con poteri di amministrazione globale su tutti i tenant che hanno accettato la clausola d'impersonificazione.
- Admin generico, opera all'interno del perimetro di un singolo tenant, gestendo gli utenti finali e coordinando la comunicazione con i gateway assegnati.
- Utente generico, abilitato alla consultazione dei dati storici e in tempo reale e alla ricezione degli alert. Non può quindi influenzare l'ambiente ma ha solo i permessi per osservarne una porzione.
- API Client, un attore non umano che interagisce con il sistema tramite interfacce REST per uno scambio di informazioni automatizzato.
- Sistema observability, un componente esterno dedicato alla raccolta di metriche e log provenienti dal Cloud, che permette al Super Admin di verificare che lo stato di salute e le prestazioni del sistema siano nei parametri ottimali.
- Gateway simulato, entità che simula il flusso di dati proveniente dai sensori, generandoli internamente e riportandoli al Cloud. Rimane inoltre in ascolto per ricevere comandi.

3.1.2. Container

In questo contesto, un container è inteso come una parte del sistema o data store (ad esempio un database) che necessita di rimanere in esecuzione perché l'ecosistema complessivo funzioni correttamente. Un diagramma di questo tipo mostra l'architettura del software ad alto livello, definendo anche la distribuzione delle responsabilità, le scelte tecnologiche infrastrutturali principali e le scelte relative alla comunicazione tra i container.

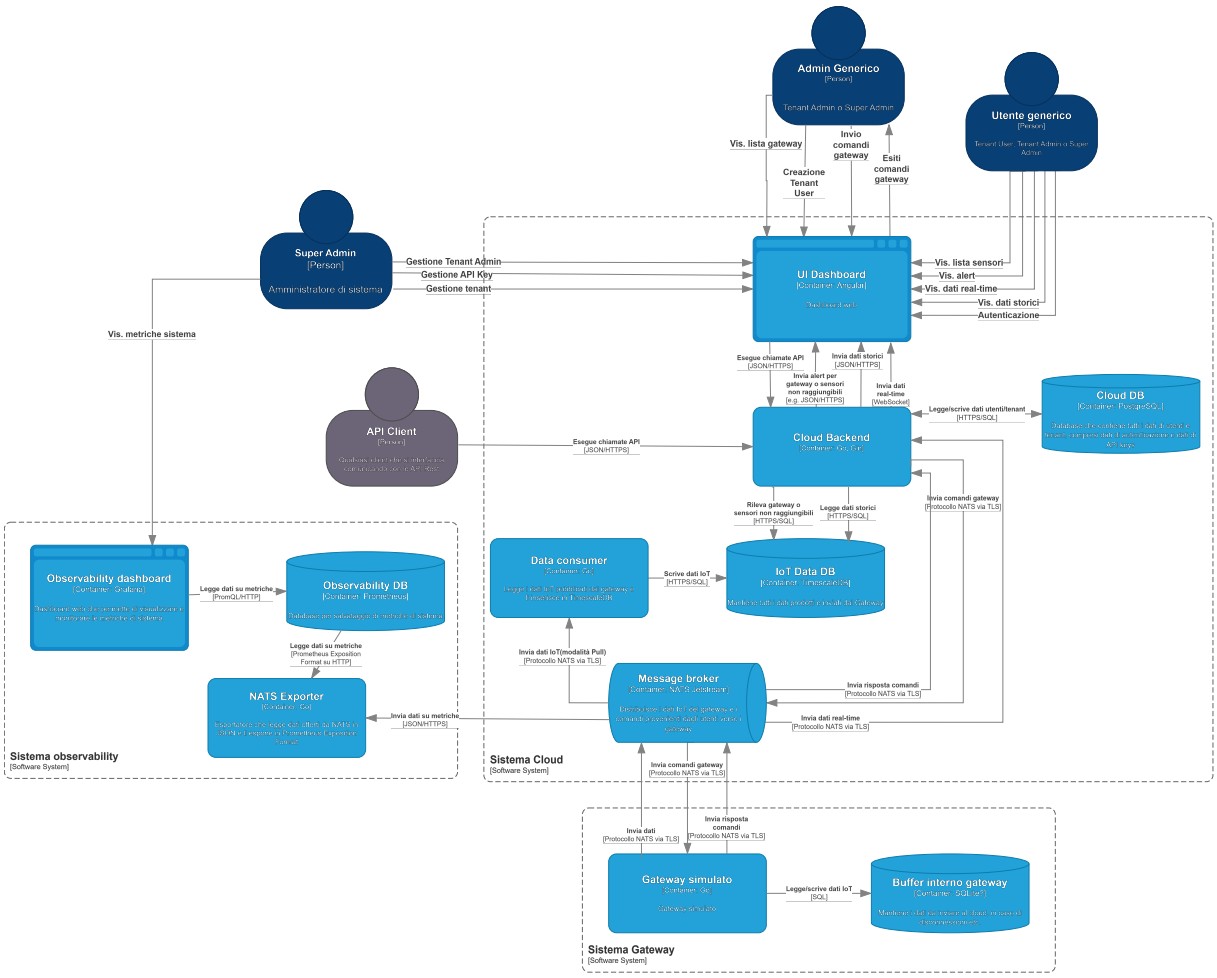

Figura 2: Container diagram

Qui viene definito con più dettaglio il contenuto di alcuni componenti presenti nel Context, il livello di astrazione precedente.

3.1.2.1. Sistema Cloud

Sono ora rappresentati:

- due database, il IoT Data DB e il CloudDB, il primo per i dati prodotti dai sensori simulati e il secondo per tutte le altre informazioni utili, ad esempio dati di tenant o API keys.
- il frontend dell'applicazione, ovvero UI Dashboard (in Angular), con il compito di fornire un'interfaccia per le funzionalità offerte dal Cloud Backend ad utenti umani.
- Message Broker, che permette una corretta gestione del flusso di dati IoT e comandi destinati al gateway.
- Data Consumer ha il compito di ricevere i valori generati dai sensori, leggerli e formattarli prima di inserirli nel database.
- Cloud Backend (in Go e Gin) che rimane il fulcro dell'applicazione. Come riportato sopra, emette i principali servizi del software.

3.1.2.2. Il Sistema observability

Ora presenta un container NATS Exporter che raccoglie le metriche di sistema e le inoltra a Observability DB (Prometheus) per il monitoraggio, con visualizzazione finale fornita tramite Observability Dashboard (Grafana).

3.1.2.3. Il Gateway simulato

Comprende un database a scopo di buffer. Interagisce con il Message Broker del Sistema Cloud tramite protocolli NATS per inviare i dati prodotti e ricevere e rispondere a comandi.

3.1.3. Component

Un diagramma Component rappresenta l'ultimo livello di astrazione dell'architettura logica prima di scendere nel dettaglio del codice sorgente. Un componente è inteso come un raggruppamento di funzionalità correlate esposte tramite un'interfaccia definita, che risiede all'interno di un container. Rispetto al livello precedente del modello C4, in questo strato si descrivono le responsabilità interne, le dipendenze e le scelte implementative dei container principali che orchestrano il sistema.

I container principali descritti in questa sezione sono:

- Gateway simulato
- Data Consumer
- Angular SPA
- Cloud Backend

3.1.3.1. Gateway

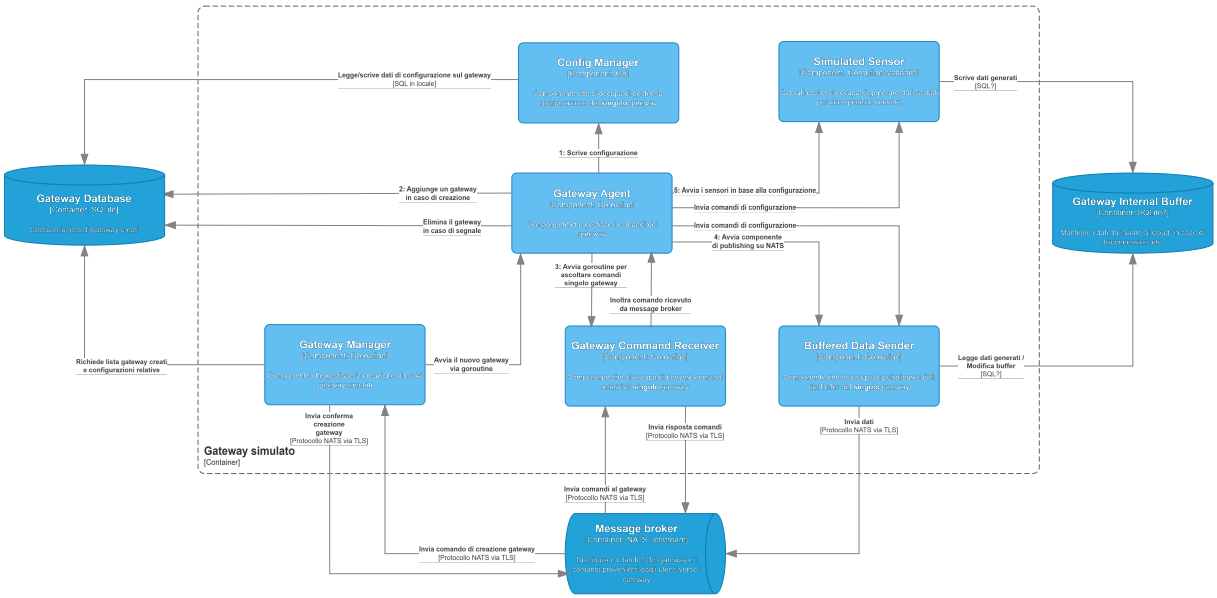

Figura 3: Gateway Component diagram

Come riportato sopra, questa parte del software ha il compito di simulare più gateway e i sensori ad essi associati, gestendo, per entrambi, le variabili di configurazione e condividendo i dati quando richiesto. Ogni Simulated Sensor genera dati specifici che vengono salvati temporaneamente in un Gateway Internal Buffer per prevenire perdite in caso di disconnessione o altri casi eccezionali. Il Gateway Manager orchestra le istanze dei gateway simulati tramite **goroutine**$_G$, mentre il Gateway Agent coordina l'avvio dei sensori in base alla configurazione gestita dal Config Manager, ascolta i comandi ricevuti dal Gateway Command Receiver e inizia a inviare i dati del singolo gateway simulato verso il Message Broker, il cui onere ricade sul Buffered Data Sender.

3.1.3.2. Data Consumer

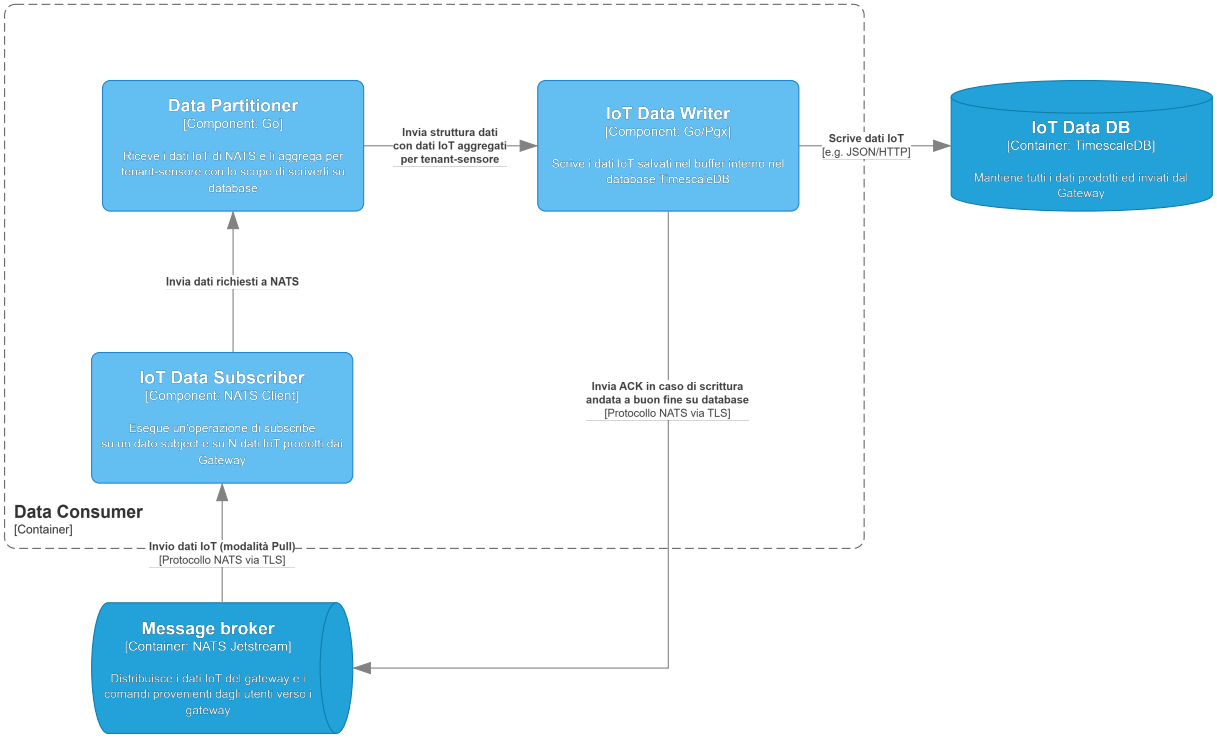

Figura 4: Data Consumer Component diagram

Il **Data Consumer$_G$** è un **microservizio** specializzato con l'unico scopo di gestire l'arrivo di dati IoT dal message broker (**NATS$_G$**), validarne il contenuto e assicurarne la persistenza su **TimescaleDB$_G$**. Il processo inizia con l'IoT Data Subscriber, un client NATS che si iscrive in modalità Pull sui dati prodotti dai gateway. Da qui il Data Partitioner (tramite Go) riceve i dati grezzi e li aggrega per tenant e sensore, inserendoli in strutture dati ottimizzate per il sistema di persistenza in TimescaleDB. L'IoT Data Writer riceve la struttura generata e scrive i dati aggregati nel database TimescaleDB, inviando un segnale di conferma (ACK) al broker per confermare la corretta ricezione degli stessi.

3.1.3.3. Angular

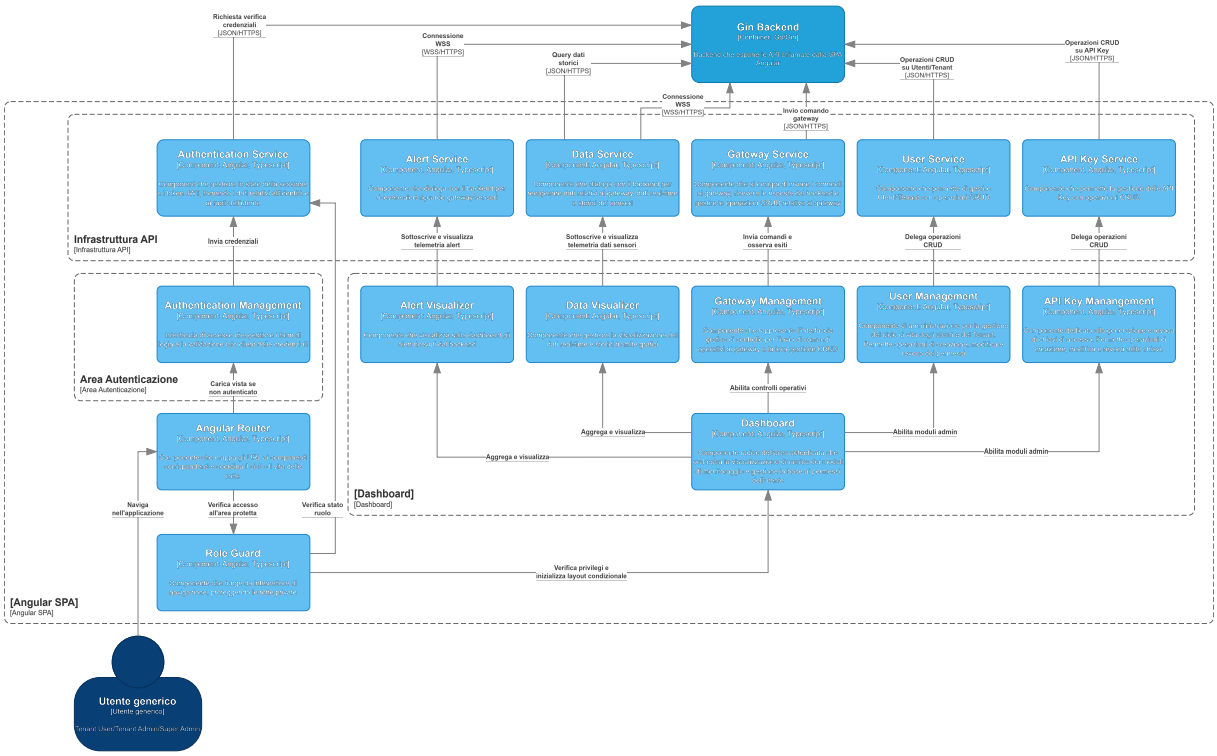

Figura 5: Angular Component diagram

Il componente Angular SPA ha lo scopo di fornire un'interfaccia visiva che permette di visualizzare i dati raccolti dai gateway e i relativi alert e gestire tenant, gateway, sensori e chiavi API di accesso. Il cuore della navigazione è affidato all'Angular Router, che coordina il caricamento dei componenti in base all'URL e permette all'utente di autenticarsi tramite il componente Autenthication Management. La sicurezza è garantita dal Role Guard, che verifica i privilegi dell'utente (Admin o utente generico) prima di inizializzare il layout. La struttura del frontend è organizzata in moduli funzionali, ognuno dedicato a una specifica area di funzionalità:

3.1.3.3.1. Dashboard

Il modulo dashboard rappresenta la schermata principale della SPA Angular, in cui l'utente può visualizzare:

- dati in tempo reale: grafici dinamici che mostrano i valori dei sensori aggiornati in tempo reale, grazie alla sottoscrizione a flussi di dati via webSocket;
- dati storici: grafici e tabelle che permettono di esplorare i dati raccolti nel tempo, con filtri per intervalli temporali, tipi di sensori e gateway specifici;
- elenco dei gateway e dei sensori: una panoramica dei dispositivi attivi, con la possibilità di visualizzare dettagli specifici e lo stato di salute di ciascuno.

3.1.3.3.2. Gestione utenti

Il modulo gestione utenti consente agli amministratori di creare, modificare e cancellare utenti, assegnare loro ruoli e gestire i tenant. Include funzionalità per:

- creazione di nuovi utenti con specifici privilegi;
- eliminazione di utenti esistenti.

3.1.3.3.3. Gestione tenant

Il modulo gestione tenant permette agli amministratori di creare e configurare nuovi tenant e assegnare gateway e sensori a ciascun tenant. Le funzionalità includono:

- creazione di nuovi tenant con configurazioni personalizzate;
- assegnazione di gateway e sensori a tenant specifici.

3.1.3.3.4. Gestione gateway e sensori

Il modulo gestione gateway e sensori consente agli amministratori di configurare i gateway e i sensori simulati, definendo le caratteristiche dei dati generati e le modalità di comunicazione. Le funzionalità includono:

- configurazione dei gateway simulati, inclusa la definizione dei sensori associati e delle caratteristiche dei dati generati.
- possibilità di inviare comandi specifici ai gateway simulati, ad esempio per riavviare o reimpostare tali gateway.

3.1.3.4. Backend

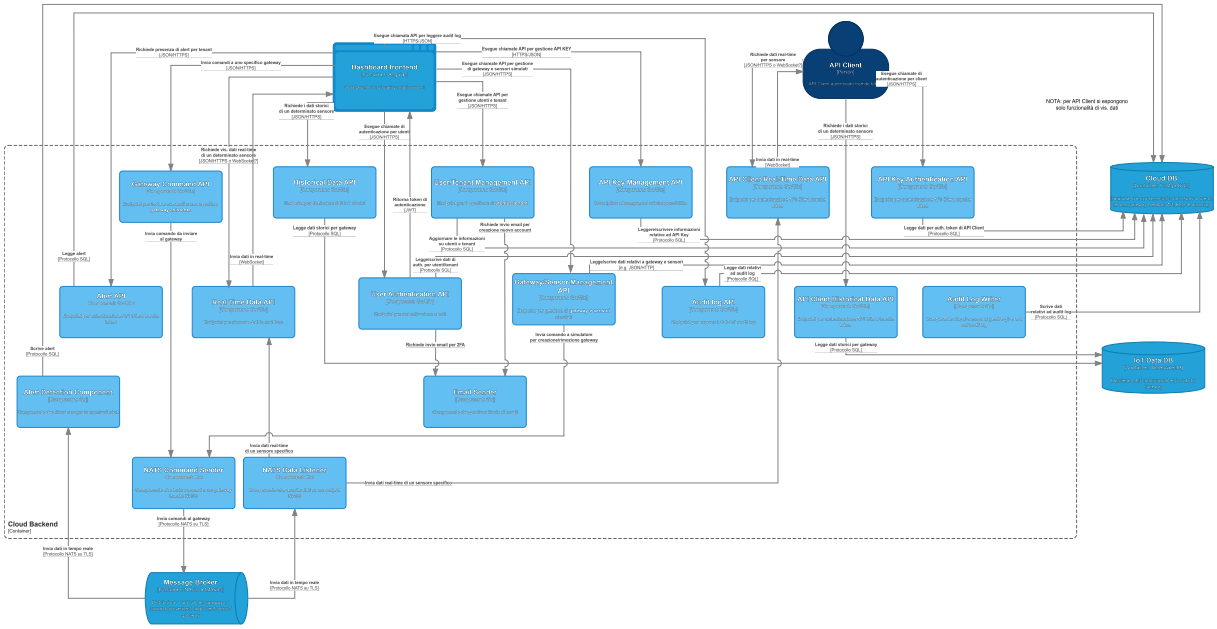

Figura 6: Backend Component diagram

Il Cloud Backend (Control & Management Layer) rappresenta il motore decisionale e l'orchestratore del sistema. Sviluppato in Go usando il framework Gin, è progettato per gestire un carico elevato di richieste contemporanee. Il suo scopo è esporre le **API**$_G$ per la consultazione dei dati e la gestione del sistema, tramite un sistema di controllo degli accessi gerarchico. Questi servizi vengono forniti agli utenti finali tramite l'interfaccia front-end descritta prima, la quale ha la responsabilità di chiamare e gestire le risposte del componente presente.

La comunicazione tra i microservizi è mediata da NATS, che funge da **message broker**$_G$ asincrono, permettendo un disaccoppiamento efficace tra la fase di ricezione dei dati e quella di elaborazione. Infine, il sistema prevede la segregazione logica dei dati per supportare una gestione multi-tenant, assicurando che ogni cliente possa accedere esclusivamente alle proprie risorse.

Le componenti interne possono essere suddivise in macro-aree funzionali:

3.1.3.4.1. Gestione degli accessi

Questa area gestisce il ciclo di vita degli utenti, il loro accesso alla piattaforma, la creazione di nuovi profili e la configurazione dei tenant, garantendo che ogni operazione sia isolata nel perimetro del cliente corretto. Vi sono component dedicati alla generazione e alla validazione di

chiavi di accesso per gli API Client esterni, il che permette l'integrazione sicura di sistemi terzi che possono consultare i dati senza passare dall'interfaccia web.

3.1.3.4.2. Gestione dati e flussi real-time

Il backend si posiziona come strato di mediazione tra il frontend e i database di persistenza. Ciò include sia la capacità di estrarre i dati storici dei sensori dal database IoT Data DB per rispondere alle query di visualizzazione di dati, servendoli in formato JSON tramite HTTPS, sia la capacità di ricevere i dati appena pubblicati dal Message Broker e inoltrarli in tempo reale ai client connessi via WebSocket, minimizzando la latenza di visualizzazione. Il NATS Command Server è il componente utile ad inviare comandi ricevuti dagli utenti verso un gateway o sensore specifico, tramite il Message Broker.

3.1.3.4.3. Servizi di background

Sono inoltre presenti componenti che operano indipendentemente dalle richieste dirette degli utenti per mantenere l'integrità e la sicurezza del sistema, ad esempio:

- Alert Detection Component, utilizzato per rilevare l'assenza di dati durante la comunicazione con un gateway o sensore specifico.
- Audit Log Writer, che registra ogni operazione critica (modifica utenti, invio comandi, login, etc.) sul Cloud DB attraverso l'Audit Log API, garantendo la tracciabilità completa delle azioni amministrative.

Questa scelta progettuale garantisce un'elevata scalabilità orizzontale, permettendo di potenziare o aggiornare singole parti del sistema senza compromettere la stabilità dell'intera infrastruttura. Ogni microservizio è containerizzato tramite **Docker**$_G$, assicurando la portabilità tra i diversi ambienti di esecuzione e semplificando le procedure di manutenzione.

3.2. Architettura di deployment

3.2.1. Diagramma di deployment

Il Deployment Diagram illustra come le istanze dei container siano effettivamente distribuite sull'infrastruttura fisica o virtuale all'interno di un determinato ambiente. Questo livello permette di mappare i componenti logici su nodi di deployment, i quali rappresentano le risorse computazionali dove il software viene eseguito, quali infrastrutture fisiche, macchine virtuali, container Docker o ambienti di esecuzione specifici (come database server o browser web).

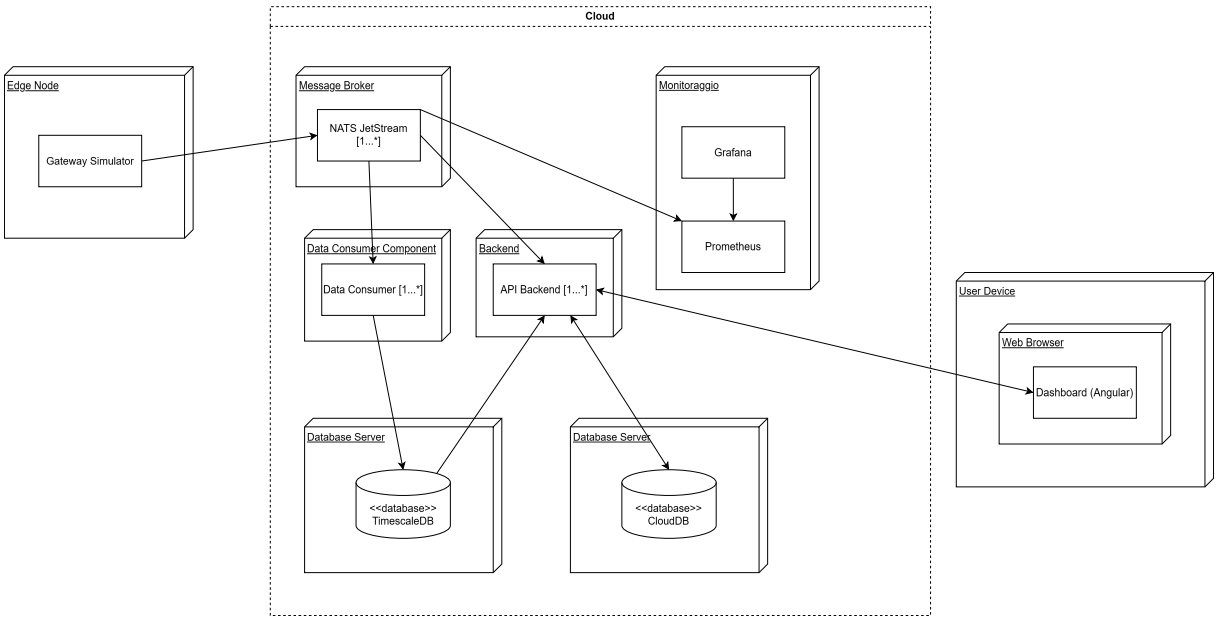

Figura 7: Deployment diagram

Le responsabilità sono separate tra l'Edge Node, il Cloud e l'interfaccia utente (User Device). Nel primo risiede l'istanza del Gateway Simulator, che viene rappresentata come un'entità autonoma al di fuori del nodo centrale, in modo da rappresentare in modo più realistico l'infrastruttura su cui si dovrebbe basare il sistema. Qualora si decidesse di implementare quest'ultima in una successiva iterazione del progetto, ciò impedirebbe di causare modifiche a cascata obbligatorie verso altre aree dell'architettura. L'infrastruttura Cloud rimane l'ambiente di esecuzione principale dove vivono i diversi **microservizi**, per lo più in nodi indipendenti. Oltre ai database per la persistenza dei dati e al nodo di monitoraggio (che ospita Prometheus e Grafana), sono presenti nodi con componenti multi-istanza, ovvero:

- Message Broker, che ospita istanze di NATS JetStream.
- API Backend e Data Consumer, dedicati all'esecuzione dei container di business logic.

User Device rappresenta l'ambiente di esecuzione lato client. Qui il nodo è il Web Browser, all'interno del quale viene distribuita ed eseguita l'istanza della Dashboard Angular.

3.3. Design Pattern

I design pattern sono stati selezionati per garantire che l'architettura a microservizi sia flessibile e scalabile, rispettando gli obiettivi di manutenibilità definiti nel capitolato.

3.3.1. Strategy

3.3.1.1. Descrizione

Il pattern Strategy è un design pattern comportamentale che permette di definire una famiglia di algoritmi, incapsularli in strutture separate e renderli intercambiabili a runtime. Questo approccio consente di variare il comportamento di un componente senza modificarne la logica di controllo principale.

3.3.1.2. Motivi per la scelta

Nel nostro sistema, il **Gateway$_G$** simulato deve gestire l'invio di dati **IoT$_G$** provenienti da sensori con caratteristiche profondamente diverse. L'uso dello Strategy permette di isolare la logica di generazione del dato di ogni specifico sensore, rendendo il gateway scalabile e pronto a supportare nuovi profili BLE senza dover riscrivere il core del simulatore.

3.3.1.3. Utilizzo nel progetto

3.3.2. Command

3.3.2.1. Descrizione

Il pattern *Command* è un design pattern comportamentale che incapsula una richiesta come un oggetto, permettendo così di parametrizzare i client con diverse richieste, instradarle o metterle in coda. Questo approccio separa l'oggetto che invoca l'operazione da quello che sa come eseguirla, facilitando la gestione di operazioni complesse e asincrone.

3.3.2.2. Motivi per la scelta

La scelta del pattern *Command* è fondamentale per gestire la comunicazione tra il **Cloud$_G$** Layer e l'Edge Layer. Poiché l'invio di istruzioni ai gateway avviene tramite un Message broker (NATS) in modalità asincrona, il pattern permette di trattare ogni comando come un'entità autonoma. Ciò garantisce la tracciabilità delle operazioni, la possibilità di gestire i log degli esiti e assicura che il Cloud Backend rimanga reattivo senza dover attendere l'esecuzione immediata sul gateway fisico o simulato.

3.3.2.3. Utilizzo nel progetto

3.3.3. Adapter

3.3.3.1. Descrizione

L'Adapter è un design pattern strutturale che funge da intermediario tra due componenti con interfacce incompatibili. Agisce come un wrapper (involucro) che traduce i dati o le chiamate di un «fornitore» nel formato atteso dal «ricevente», permettendo loro di collaborare senza dover modificare il codice originale delle parti coinvolte.

3.3.3.2. Motivi per la scelta

La scelta di questo pattern è dettata dalla necessità di gestire l'eterogeneità dei sensori fisici simulati. Poiché ogni sensore può esporre dati secondo profili **BLE$_G$** differenti o protocolli specifici, l'Adapter permette di uniformare queste informazioni prima che entrino nel cuore del sistema. Questo garantisce che il **Cloud$_G$** Layer sia completamente agnostico rispetto alla sorgente fisica del dato, semplificando la manutenzione e l'aggiunta di nuovi dispositivi.

3.3.3.3. Utilizzo nel progetto

3.3.4. Dependency injection

3.3.4.1. Descrizione

Il pattern Dependency InjectionG è un design pattern strutturale che consente di rendere esplicite le dipendenze di un oggetto. Invece di creare direttamente le dipendenze all'interno delle classi o dei componenti, queste possono essere fornite dall'esterno: in questo modo, un componente dichiara le sue dipendenze senza doversi preoccupare di istanziarle, permettendo dunque una maggiore modularità tra i diversi componenti del Sistema. Esistono principalmente due tipi di dependency injectionG: • Constructor Injection: le dipendenze vengono passate attraverso il costruttore; • Setter Injection: le dipendenze vengono impostate tramite metodi setter. Nel progetto viene utilizzata la Constructor Injection.

La *Dependency injection* è un design pattern che permette di rendere esplicite le dipendenze di un oggetto, favorendo l'inversione del controllo (IoC). Invece di creare e gestire le dipendenze internamente a una classe o a un componente, queste vengono fornite dall'esterno. In questo modo,

un modulo dichiara semplicemente «cosa gli serve» per funzionare, senza doversi occupare di come istanziarlo o configurarlo, garantendo un elevato grado di modularità e disaccoppiamento tra le componenti del sistema.

Esistono principalmente due modalità di implementazione:
- **Constructor Injection**: le dipendenze vengono passate attraverso il costruttore al momento della creazione dell'oggetto.
- **Setter Injection**: le dipendenze vengono impostate tramite metodi specifici (setter) dopo l'istanziazione.

Nel progetto è stato usato il ... .

3.3.4.2. Motivi per la scelta

Per quanto riguarda il **frontend**, **Angular$_G$** offre un sistema di *dependency injection* integrato e altamente efficiente, che consente di gestire le dipendenze tra i componenti in modo dichiarativo e modulare, evitando la necessità di istanziare manualmente i servizi o di gestire le dipendenze in modo esplicito all'interno dei componenti.

3.3.4.3. Utilizzo nel progetto

Nel **frontend Angular$_G$**, la *dependency injection* è utilizzata per gestire le dipendenze tra i componenti, i servizi e altri elementi dell'applicazione. Tramite il metodo `inject()` è infatti possibile iniettare le dipendenze necessarie direttamente nei componenti. Ad esempio, i servizi che gestiscono la comunicazione con il backend o la gestione dello stato dell'applicazione vengono iniettati nei componenti che ne hanno bisogno, garantendo un'architettura modulare e facilmente testabile.

3.3.5. Observer

3.3.5.1. Descrizione

Il pattern *Observer* è un design pattern comportamentale che definisce una relazione di dipendenza uno-a-molti tra oggetti, in cui un oggetto (il *subject*) mantiene una lista di dipendenti (*observers*) e notifica loro automaticamente ogni cambiamento di stato. Questo approccio consente di implementare un sistema di comunicazione efficiente e flessibile, in cui i componenti possono reagire dinamicamente agli eventi senza essere strettamente accoppiati.

3.3.5.2. Motivi per la scelta

I **signals** di **Angular$_G$** sono stati scelti per implementare il pattern *Observer* all'interno del frontend poiché offrono un meccanismo reattivo e performante per gestire lo stato dell'applicazione. Questo pattern è particolarmente utile per sincronizzare la visualizzazione dei dati dai *services* ai *components*, garantendo che ogni cambiamento nei dati venga automaticamente riflesso nell'interfaccia utente senza la necessità di interventi manuali o di gestione complessa dello stato.

3.3.5.3. Utilizzo nel progetto

Nel progetto, il pattern *Observer* è implementato principalmente attraverso i *signals* di **Angular$_G$**, che permettono ai componenti di sottoscriversi a flussi di dati provenienti dai servizi. Tutti i dati provenienti dal backend (come ad esempio i dati dei sensori, l'elenco dei gateway/sensori, i dati storici, etc.) vengono salvati e gestiti attraverso *signals*, creati con il metodo `signal()`. Tali dati sono poi esposti dai servizi attraverso relative istanze *readonly*.

I componenti dell'interfaccia utente possono quindi sottoscriversi a tale istanze per ricevere aggiornamenti in tempo reale, garantendo una sincronizzazione efficiente e reattiva tra il backend e la visualizzazione dei dati.

3.3.6. Altro Pattern

3.3.6.1. Descrizione

3.3.6.2. Motivi per la scelta

3.3.6.3. Utilizzo nel progetto

3.4. Microservizi sviluppati

4. Stato dei requisiti funzionali

4.1. Stato per requisito

4.2. Grafici riassuntivi

Jaume Bernardi

Firma del revisore interno