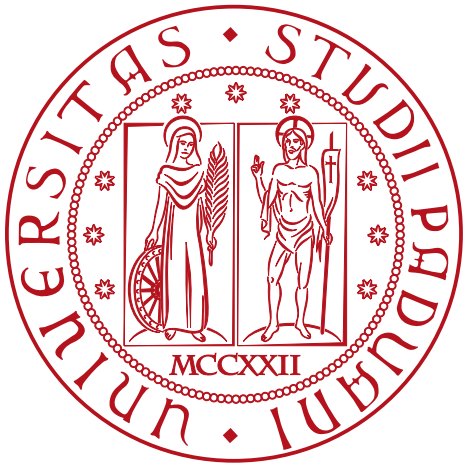
Università degli studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



VoIPDashboard - Strumento web per la visualizzazione e l'analisi di dati telefonici

Tesi di laurea triennale

Relatore

Prof. Marco Zanella

Laureando

Francesco Fragonas

Matricola 2076436

ANNO ACCADEMICO 2024-2025

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage curricolare, della durata di circa trecentoventi ore, dal laureando Francesco Fragonas presso l'azienda Cinquenet SRL. Lo stage è stato condotto sotto la supervisione del tutor aziendale Fabio Baraldo, mentre il prof. Prof. Marco Zanella ha ricoperto il ruolo di tutor accademico.

Ringraziamenti

TESTO RINGRAZIAMENTI

Padova, Dicembre 2025 *Francesco Fragonas*

Indice

Elenco delle Figure

Elenco delle Tabelle

Elenco dei Codici Sorgente

Capitolo 1

Introduzione

1.1 L'azienda

Cinquenet S.r.l. è un'azienda specializzata nel settore delle Information and Communication Technology (ICT) con sede a Cerea, in provincia di Verona. Fondata da un team di professionisti con oltre vent'anni di esperienza consolidata nel settore delle telecomunicazioni, l'azienda si distingue nel panorama delle soluzioni tecnologiche per la combinazione di competenza tecnica, passione e attenzione al dettaglio. La filosofia aziendale di Cinquenet si basa sullo sviluppo di soluzioni ICT all'avanguardia, progettate per garantire elevata affidabilità operativa e stabilità nel tempo. Questo approccio ha permesso all'azienda di costruire una solida reputazione nel territorio veneto e di espandere la propria presenza nel mercato delle telecomunicazioni e dei servizi informatici.

1.1.1 Aree di specializzazione

L'offerta di Cinquenet si articola in tre macro-aree di competenza, che riflettono un approccio integrato alle esigenze tecnologiche delle aziende moderne:

Soluzioni telefoniche e di rete:

Il core business dell'azienda comprende la progettazione e implementazione di infrastrutture di telecomunicazione avanzate. Tra i servizi principali figurano la realizzazione di reti in fibra ottica, collegamenti ADSL, linee professionali dedicate, centralini virtuali e impianti telefonici aziendali. Particolare expertise viene dedicata alle soluzioni wireless e ai link radio, tecnologie fondamentali per garantire connettività in contesti dove le infrastrutture tradizionali risultano insufficienti.

Servizi informatici e cloud:

Cinquenet offre una gamma completa di soluzioni per la gestione dell'infrastruttura IT aziendale, includendo servizi di hosting, server cloud, backup dei dati e consulenza informatica specializzata. L'azienda si è inoltre posizionata come partner strategico per l'implementazione di

tecnologie emergenti quali Internet of Things (IoT) e soluzioni basate su intelligenza artificiale.

Sicurezza e protezione:

Un'area di crescente importanza nel portfolio aziendale è rappresentata dalla cybersecurity, settore nel quale Cinquenet sviluppa strategie di protezione integrate per la sicurezza informatica. Parallelamente, l'azienda opera nel campo della sicurezza fisica attraverso l'installazione di sistemi di videosorveglianza e impianti antintrusione.

1.2 Motivazone del progetto

Ho scelto di svolgere lo stage presso Cinquenet srl per diverse ragioni che rendevano questa opportunità particolarmente interessante dal punto di vista formativo e professionale. Innanzitutto, conoscevo già l'azienda e il suo approccio lavorativo, il che mi ha permesso di valutare positivamente l'ambiente e le metodologie operative.
La decisione di non optare per un'azienda tradizionalmente focalizzata sulla programmazione è stata dettata dal desiderio di ampliare le mie competenze in un contesto più diversificato. Mi intrigava l'idea di lavorare in un'azienda operante nel mondo dell'informatica ma con un focus specifico su reti, connessioni e centralini virtuali PABX, settori che offrono prospettive di crescita professionale complementari allo sviluppo software puro.
Il progetto di stage consiste nello sviluppo di una dashboard web che consente ai clienti di accedere a tutte le statistiche e i dettagli delle chiamate effettuate e ricevute. La piattaforma offre funzionalità di filtraggio avanzate per interno, ring group, DID e periodi temporali, presentando i dati attraverso statistiche, grafici e tabelle dettagliate. Tutti i contenuti sono progettati per essere facilmente stampabili ed esportabili in formato PDF e CSV.
Questo progetto rappresenta un'opportunità ideale per combinare competenze di programmazione web con la conoscenza del mondo dei centralini virtuali e delle telecomunicazioni aziendali, offrendo un'esperienza formativa completa e multidisciplinare.

1.3 Struttura della tesi

Il presente documento è strutturato secondo la seguente organizzazione:

Il secondo capitolo presenta una descrizione dettagliata dello stage, includendo l'organizzazione del lavoro, il rapporto con l'azienda e

il tutor aziendale, la metodologia adottata e l'analisi preventiva dei rischi.

Il terzo capitolo contiene l'analisi approfondita dei requisiti del sistema, suddivisi per tipologia e priorità, insieme all'identificazione degli stakeholder e dei casi d'uso principali.

Il quarto capitolo fornisce un'introduzione teorica alle tecnologie e agli strumenti utilizzati, presentando le motivazioni alla base delle scelte architetturali e tecnologiche adottate.

Il quinto capitolo descrive nel dettaglio il lavoro svolto durante il periodo di stage, illustrando le problematiche incontrate, le soluzioni implementate e le funzionalità sviluppate.

Il sesto capitolo presenta le conclusioni dell'esperienza, valutando il raggiungimento degli obiettivi prefissati, le conoscenze acquisite e i possibili sviluppi futuri del progetto.

Convenzioni tipografiche:

Per la stesura del documento sono state adottate le seguenti convenzioni:

- Gli acronimi, le abbreviazioni e i termini tecnici specialistici sono definiti nel glossario posto al termine del documento;
- La prima occorrenza dei termini presenti nel glossario è evidenziata con la seguente notazione: termine$\hat{}$G$\hat{}$;
- I termini in lingua straniera e il gergo tecnico sono riportati in corsivo.

Capitolo 2

Descrizione stage

In questo capitolo viene approfondita l'organizzazione dello stage, descrivendo il progetto realizzato, la metodologia di lavoro adottata, il rapporto con l'azienda e l'analisi preventiva dei rischi.

2.1 Introduzione al progetto

Il progetto di stage si inserisce nel contesto dei servizi di telefonia aziendale offerti da Cinquenet ai propri clienti. L'azienda fornisce soluzioni di centralini virtuali PABX che gestiscono le comunicazioni telefoniche di numerose realtà aziendali. Tuttavia, mancava uno strumento che permettesse ai clienti finali di analizzare autonomamente i dati relativi alle proprie chiamate telefoniche.
L'obiettivo del progetto è stato lo sviluppo di una dashboard web che consentisse ai clienti di visualizzare statistiche dettagliate sulle chiamate effettuate e ricevute, con la possibilità di applicare filtri avanzati e generare report personalizzati. Il sistema doveva integrarsi con l'infrastruttura esistente dei centralini PABX, estraendo i dati dalle basi dati operative e presentandoli in formato accessibile e intuitivo.
La realizzazione è avvenuta completamente ex novo, progettando sia il backend per l'elaborazione dei dati che il frontend per la visualizzazione. La sfida principale è consistita nel comprendere la struttura complessa dei dati telefonici, identificare le informazioni rilevanti e progettare query efficienti per l'estrazione e l'aggregazione delle statistiche.
L'approccio scelto prevedeva la realizzazione di un MVP (Minimum Viable Product) funzionante nel minor tempo possibile, che includesse già le funzionalità base del sistema, anche se non ancora completamente ottimizzate. Questo prototipo iniziale sarebbe poi servito come base per successive iterazioni di miglioramento e ampliamento.

2.2 Organizzazione del lavoro

Per lo sviluppo del progetto è stato adottato il Modello di Sviluppo Evolutivo, una metodologia particolarmente adatta quando i requisiti

non sono completamente definibili a priori e si desidera ottenere rapidamente versioni utilizzabili del sistema.

2.2.1 Il Modello di Sviluppo Evolutivo

Il Modello Evolutivo è un approccio incrementale in cui gli incrementi successivi costituiscono versioni prototipali utilizzabili e valutabili dagli stakeholder. A differenza di modelli sequenziali rigidi, questo approccio permette di:

- Rispondere a bisogni non inizialmente preventivabili: durante lo sviluppo possono emergere nuovi requisiti o modifiche a quelli esistenti
- Produrre prototipi utilizzabili: ogni iterazione rilascia una versione funzionante del sistema che può essere testata e valutata
- Ammettere iterazioni multiple: ogni fase può essere riattraversata più volte per raffinamenti successivi
- Gestire l'incertezza: particolarmente utile quando la complessità del dominio applicativo richiede esplorazione e apprendimento progressivo

Schema generale del Modello Evolutivo:

Il processo di sviluppo si articola in tre fasi principali:

1. Analisi preliminare
Questa fase iniziale è dedicata all'identificazione dei requisiti fondamentali e alla definizione dell'architettura di base del sistema, progettata per essere modulare e facilitare future evoluzioni. Viene inoltre pianificato il percorso di sviluppo suddividendo il lavoro in passi incrementali, e si procede con uno studio approfondito del dominio applicativo, in particolare della struttura dati dei sistemi PABX e della logica delle chiamate telefoniche.

2. Analisi e realizzazione iterativa
Il sistema viene progressivamente costruito attraverso cicli iterativi in cui l'analisi viene continuamente raffinata in base alle conoscenze acquisite. Ogni iterazione comprende progettazione, codifica e testing delle funzionalità, seguita dall'integrazione dei componenti sviluppati. Al termine di ogni ciclo, il lavoro viene validato attraverso sessioni di feedback con il tutor aziendale per verificare l'aderenza ai requisiti e identificare eventuali necessità di miglioramento.

3. Rilascio di prototipi
Ogni iterazione produce una versione funzionante del sistema che viene valutata dal tutor aziendale e, nelle fasi più mature, dal cliente finale. I

feedback raccolti guidano le iterazioni successive, orientando lo sviluppo verso le reali esigenze degli utilizzatori. Questo processo ciclico continua fino al raggiungimento di un livello di maturità soddisfacente per l'accettazione finale.

2.2.2 Applicazione del modello al progetto

Il lavoro è stato organizzato in sprint settimanali, cicli di sviluppo della durata di una settimana ciascuno, strutturati secondo le seguenti fasi:

Inizio sprint - Riunione di pianificazione
Ogni sprint iniziava con un incontro con il tutor aziendale in cui veniva effettuata una revisione del lavoro svolto nella settimana precedente, identificando eventuali criticità o problematiche emerse. Successivamente si procedeva alla definizione degli obiettivi dello sprint corrente e alla pianificazione dettagliata delle attività da svolgere.

Durante lo sprint
La fase centrale dello sprint era dedicata allo sviluppo vero e proprio delle funzionalità pianificate, accompagnato da attività continue di testing e debug per garantire la qualità del codice. Durante questa fase erano frequenti confronti informali con il tutor per risolvere dubbi tecnici o richiedere chiarimenti.

Fine sprint - Riunione di review
Al termine dello sprint veniva organizzata una sessione di review in cui le funzionalità implementate venivano dimostrate al tutor aziendale. Durante questo incontro si raccoglievano feedback e suggerimenti, si valutava il raggiungimento degli obiettivi prefissati e si identificavano le priorità per lo sprint successivo.

Nella fase iniziale del progetto, il supporto del tutor aziendale è stato fondamentale per acquisire le conoscenze necessarie sul dominio applicativo. Il tutor ha dedicato tempo significativo a spiegare la logica di funzionamento dei centralini PABX, illustrare la struttura del database e definire le modalità corrette di estrazione e interpretazione dei dati telefonici. Con il progredire dello stage e l'acquisizione di maggiore autonomia operativa, il ruolo del tutor si è progressivamente evoluto da formativo a consulenziale, focalizzandosi principalmente sulla validazione delle scelte progettuali e sulla definizione di nuovi requisiti emergenti.

2.3 Vincoli

Lo sviluppo del progetto è stato soggetto a diversi vincoli:

Vincoli temporali
- Durata complessiva dello stage: 320 ore
- Necessità di produrre un prototipo dimostrabile entro le prime settimane
- Scadenze settimanali per il completamento degli sprint

Vincoli tecnologici
- Integrazione obbligatoria con l'infrastruttura esistente di Cinquenet
- Utilizzo del database già in uso per i centralini PABX
- Requisiti di performance per la gestione di grandi volumi di dati storici

Vincoli architetturali
- Necessità di un'architettura modulare per future estensioni
- Personalizzazione grafica per ciascun cliente finale

2.4 Pianificazione

La pianificazione iniziale del progetto ha seguito lo schema del Modello Evolutivo, suddividendo il lavoro in macro-fasi:

Settimane 1-2 : Analisi preliminare e setup
- Studio del dominio applicativo (telefonia PABX)
- Analisi della struttura del database esistente
- Definizione dei requisiti fondamentali
- Setup dell'ambiente di sviluppo

Settimane 3-5 : Primo prototipo (MVP)
- Progettazione dell'architettura del sistema
- Implementazione delle query base per l'estrazione dati
- Sviluppo dell'interfaccia utente minimale
- Prime funzionalità di filtraggio e visualizzazione
- Prima dimostrazione al cliente finale

Settimane 6-7 : Raffinamento e ottimizzazione
- Ottimizzazione delle performance delle query
- Miglioramento dell'accuratezza dei dati
- Ampliamento delle funzionalità di filtraggio
- Implementazione dell'esportazione dati

- Implementazione eventuali feedback ricevuti

Settimana 8 : Personalizzazione e finalizzazione
- Sviluppo del sistema di personalizzazione grafica
- Testing completo con dati reali
- Documentazione finale
- Preparazione della presentazione finale al cliente

2.5 Analisi preventiva dei rischi

Durante la fase iniziale del progetto è stata condotta un'analisi dei rischi per identificare le potenziali criticità che avrebbero potuto compromettere il successo dello stage. Per ciascun rischio sono stati valutati la probabilità di occorrenza, l'impatto sul progetto e le strategie di mitigazione adottate.

R1 - Non rispetto delle tempistiche
- **Descrizione:** Impossibilità di completare le funzionalità pianificate entro le 320 ore di stage.
- **Probabilità:** Media
- **Impatto:** Alto
- **Cause potenziali:**
  - Sottostima della complessità tecnica
  - Difficoltà impreviste nell'integrazione con sistemi esistenti
  - Scarsa familiarità iniziale con il dominio applicativo
- **Strategie di mitigazione adottate:**
  - Adozione del Modello Evolutivo per rilasciare versioni funzionanti già dalle prime settimane
  - Pianificazione di sprint brevi (1 settimana) per identificare rapidamente eventuali ritardi
  - Definizione chiara delle priorità: focus sulle funzionalità core (MVP) prima delle funzionalità secondarie
  - Confronti settimanali con il tutor per ricalibrazione tempestiva degli obiettivi

R2 - Inaccuratezza dei dati mostrati
- **Descrizione:** Visualizzazione di statistiche e dati non corretti o fuorvianti per l'utente finale.
- **Probabilità:** Alta
- **Impatto:** Alto
- **Cause potenziali:**
  - Errata interpretazione della logica dei dati telefonici
  - Query SQL non corrette o incomplete

  - Mancata gestione di casi particolari nel dominio PABX
  - Problemi di aggregazione e calcolo delle statistiche
- **Strategie di mitigazione adottate:**
  - Sessioni approfondite con il tutor per comprendere la semantica dei dati PABX
  - Validazione incrociata dei risultati con report esistenti o conteggi manuali
  - Testing incrementale con dataset reali forniti dall'azienda
  - Revisione frequente delle query SQL con il tutor aziendale

R3 - Difficoltà nel recupero dei dati
- **Descrizione:** Complessità tecnica nell'estrazione efficiente dei dati dal database PABX.
- **Probabilità:** Media
- **Impatto:** Alto
- **Cause potenziali:**
  - Struttura del database complessa e non documentata
  - Performance scarse con query su grandi volumi di dati storici
  - Necessità di aggregazioni complesse
- **Strategie di mitigazione adottate:**
  - Fase iniziale dedicata esclusivamente allo studio del database
  - Creazione di query di test incrementali per validare la comprensione della struttura
  - Utilizzo di indici e ottimizzazioni progressive delle query

R4 - Scarsa esperienza con le tecnologie utilizzate
- **Descrizione:** Limitata familiarità con alcuni strumenti, linguaggi o framework necessari per il progetto.
- **Probabilità:** Bassa
- **Impatto:** Medio
- **Cause potenziali:**
  - Tecnologie non approfondite durante il percorso universitario
  - Specificità degli strumenti utilizzati in azienda
  - Curva di apprendimento necessaria per essere produttivi
- **Strategie di mitigazione adottate:**
  - Studio autonomo preliminare delle tecnologie principali
  - Utilizzo di documentazione ufficiale e tutorial
  - Refactoring progressivo del codice man mano che la padronanza aumentava

R5 - Requisiti poco chiari o in evoluzione
- **Descrizione:** Cambiamenti frequenti o ambiguità nei requisiti funzionali richiesti.

- **Probabilità:** Media
- **Impatto:** Medio
- **Cause potenziali:**
  - Necessità del cliente finale non completamente definite a priori
  - Feedback emergenti durante la visualizzazione dei prototipi
  - Nuove esigenze identificate durante lo sviluppo
- **Strategie di mitigazione adottate:**
  - Scelta del Modello Evolutivo proprio per gestire l'incertezza sui requisiti
  - Architettura modulare e flessibile per facilitare modifiche
  - Validazione frequente con il tutor e raccolta sistematica di feedback
  - Prioritizzazione dei requisiti: implementazione immediata dei requisiti certi, posticipazione di quelli incerti

R6 - Problemi di performance del sistema

- **Descrizione:** Tempi di risposta inaccettabili per l'utente finale, specialmente con grandi volumi di dati.
- **Probabilità:** Media
- **Impatto:** Basso
- **Cause potenziali:**
  - Query SQL non ottimizzate
  - Mancanza di indici appropriati
  - Caricamento di troppi dati contemporaneamente
  - Elaborazioni pesanti lato client
- **Strategie di mitigazione adottate:**
  - Test di performance fin dalle prime versioni con dataset realistici
  - Ottimizzazione progressiva delle query più critiche
  - Implementazione di paginazione e lazy loading
  - Monitoring dei tempi di esecuzione delle query principali

Capitolo 3

Analisi dei requisiti

In questo capitolo vengono analizzati e approfonditi i requisiti individuati per la realizzazione del progetto di dashboard per l'analisi delle chiamate telefoniche.

3.1 Introduzione ai requisiti

I requisiti del sistema sono stati identificati attraverso un processo di analisi condotto in collaborazione con il tutor aziendale e, successivamente, validati con il cliente finale durante la presentazione del primo prototipo funzionante. Questi sono stati suddivisi in due macro categorie principali:

Requisiti funzionali Rappresentano le funzionalità che il sistema deve offrire per rispondere alle esigenze degli utenti finali, definendo le operazioni che la dashboard deve essere in grado di svolgere. Si suddividono in:

- **Obbligatori**: indispensabili per il corretto funzionamento del sistema e per soddisfare le necessità primarie degli utenti
- **Desiderabili**: non strettamente necessari, tuttavia se implementati garantiscono una migliore esperienza utente e una maggiore usabilità del software
- **Opzionali**: la loro aggiunta non è essenziale per il funzionamento base del sistema, vengono implementati se rimane tempo a disposizione al termine dello sviluppo delle funzionalità prioritarie

Requisiti non funzionali Di questa categoria fanno parte i requisiti qualitativi e quelli di vincolo. I primi garantiscono una qualità maggiore del software dal punto di vista delle prestazioni, usabilità, affidabilità e sicurezza. I requisiti di vincolo invece stabiliscono limitazioni o condizioni che il sistema deve rispettare, come tecnologie da utilizzare, standard aziendali o normative da seguire.

3.2 Tracciamento dei requisiti

Per garantire una classificazione chiara e sistematica, i requisiti raccolti sono stati categorizzati in base alla loro tipologia e priorità, utilizzando la seguente notazione:

Sigla	**Significato**
F	Funzionale
N	Non funzionale
Q	Qualitativo
V	Vincolo
O	Obbligatorio
D	Desiderabile
P	Opzionale

Tabella 1: Legenda per la classificazione dei requisiti

Ogni requisito è stato identificato secondo il seguente schema di codifica:

R-XY-N

dove:

- **R** indica che si tratta di un requisito
- **X** indica se il requisito è funzionale (F) o non funzionale (N)
- **Y** indica il livello di importanza se il requisito è funzionale, oppure la tipologia se è non funzionale:
  - se X = F, allora Y può assumere i valori O (obbligatorio), D (desiderabile) o P (opzionale)
  - se X = N, allora Y può assumere i valori Q (qualitativo) o V (vincolo)
- **N** identifica in maniera univoca il requisito all'interno della sua macro categoria

3.3 Requisiti funzionali

Di seguito vengono elencati i requisiti funzionali identificati per il sistema di dashboard.

Codice	Descrizione
R-FO-1	Il sistema deve implementare un sistema di autenticazione sicuro tramite username e password
R-FO-2	Il sistema deve permettere l'accesso alle funzionalità solo ad utenti autenticati
R-FO-3	Il sistema deve visualizzare una pagina Dashboard principale contenente i dati generali delle chiamate con possibilità di filtraggio per periodo temporale
R-FO-4	Il sistema deve fornire una sezione Utenti contenente l'elenco completo degli interni telefonici e pagine di dettaglio per ciascun utente con statistiche e filtri per periodo
R-FO-5	Il sistema deve fornire una sezione Ring Group contenente l'elenco completo dei gruppi e pagine di dettaglio per ciascun ring group con statistiche e filtri per periodo
R-FO-6	Il sistema deve fornire una sezione DID contenente l'elenco completo dei Direct Inward Dialing e pagine di dettaglio per ciascun DID con statistiche e filtri per periodo
R-FO-7	Il sistema deve presentare i dati attraverso grafici per facilitarne l'interpretazione visuale
R-FO-8	Il sistema deve visualizzare metriche aggregate tramite card informative (chiamate totali, ricevute, risposte, perse, effettuate, tempo totale)
R-FO-9	Il sistema deve presentare l'elenco dettagliato delle chiamate in entrata e in uscita tramite tabelle

Codice	Descrizione
R-FO-10	Il sistema deve permettere l'esportazione dei dati delle tabelle in formato PDF e CSV
R-FD-1	Aggiunta di filtri nella Dashboard per utenti, ring group e did
R-FD-2	Creazione di una pagina dedicata alla gestione degli account, con inserimento, modifica ed eliminazione
R-FD-3	Personalizzazione del sistema con modifica loghi e nome cliente
R-FP-1	Implementazione della crittografia delle password memorizzate nel database
R-FP-2	Implementazione della modifica della password degli utenti (solo per admin)
R-FP-3	Creazione utente superadmin non modificabile ed eliminabile

Tabella 2: Tracciamento dei requisti funzionali.

3.4 Requisiti non funzionali

I requisiti non funzionali definiscono gli aspetti qualitativi e i vincoli tecnici del sistema.

Codice	Descrizione
R-NQ-1	Il sistema deve garantire tempi di risposta rapidi anche con volumi elevati di dati
R-NQ-2	L'interfaccia deve essere intuitiva senza necessità di formazione specifica
R-NQ-3	I messaggi di errore devono essere chiari e comprensibili
R-NQ-4	Il sistema deve avere una gestione corretta degli errori senza perdita di dati
R-NQ-5	L'accesso deve essere protetto contro accessi non autorizzati
R-NQ-6	Il sistema deve essere accessibile da dispositivi mobili e desktop
R-NV-1	Separazione tra frontend e backend tramite API RESTful
R-NV-2	Accessibilità tramite browser web moderni senza plugin aggiuntivi o installazioni locali
R-NV-3	Il sistema deve poter essere distribuito in ambienti containerizzati come Docker
R-NV-4	Possibilità di integrazione con sistemi di autenticazione esterni in futuro (LDAP, OAuth)

Tabella 3: Tracciamento dei requisti non funzionali.

3.5 Riepilogo dei requisiti

La tabella seguente riporta il riepilogo quantitativo dei requisiti identificati durante la fase di analisi.

Tipologia	**Quantità**
Requisiti funzionali	16
- Obbligatori	10
- Desiderabili	3
- Opzionali	3
Requisiti non funzionali	10
- Qualitativi	6
- Di vincolo	4
Totale	**26**

Tabella 4: Riepilogo dei requisiti di progetto

Capitolo 4

Tecnologie

In questo capitolo vengono presentate le tecnologie e gli strumenti utilizzati per lo sviluppo del progetto. Le scelte tecnologiche sono state effettuate sulla base dell'analisi dei requisiti del capitolo precedente, con particolare attenzione ai vincoli imposti dall'azienda ospitante e alla compatibilità con l'infrastruttura esistente.
Per ogni tecnologia viene fornita una breve descrizione e vengono spiegate le motivazioni che hanno portato alla sua adozione nel contesto specifico del progetto.

4.1 Linguaggi e Framework

4.1.1 HTML, CSS e JavaScript

Figura 1: Logo HTML5

Figura 2: Logo CSS3

Figura 3: Logo JavaScript

HTML (HyperText Markup Language), CSS (Cascading Style Sheets) e JavaScript sono i linguaggi fondamentali per lo sviluppo di applicazioni web. HTML fornisce la struttura semantica dei contenuti, CSS gestisce la presentazione visuale e il layout, mentre JavaScript implementa la logica interattiva e il comportamento dinamico dell'applicazione.

Motivazioni della scelta
La decisione di utilizzare le tecnologie web native, senza framework moderni come React Angular o Vue.js, è stata guidata da specifici vincoli aziendali e caratteristiche del progetto:

- **Vincolo aziendale:** L'azienda ospitante ha espresso la necessità di evitare l'adozione di framework complessi com curve di apprendimento ripide e dipendenze esterne. I dipendenti dell'azienda hanno una familiarità consolidata con HTML, CSS e JavaScript, rendendo più agevole la manutenzione e l'evoluzione del codice nel tempo.
- **Semplicità dell'interfaccia:** L'applicazione sviluppata presenta un'interfaccia utente relativamente semplice, che non richiede le funzionalità avanzate offerte dai framework moderni. L'uso diretto di HTML, CSS e JavaScript consente di mantenere il codice leggero e facilmente comprensibile.
- **Manutenibilità:** Per semplificare la manutenzione e lo sviluppo, sono state create classi JavaScript modulari e riutilizzabili per la generazione di elementi comuni come tabelle e grafici, garantendo coerenza nel codice senza la complessità aggiuntiva dei framework.
- **Prestazioni e scalabilità:** L'approccio nativo offre dimensioni ridotte dell'applicazione e tempi di caricamento più rapidi. Le tecnologie web standard mantengono inoltre una migliore compatibilità retroattiva con i browser, riducendo il rischio di obsolescenza del codice nel tempo.

4.1.2 Node.js

Figura 4: Logo Node.js

Node.js è un ambiente di esecuzione JavaScript lato server che consente di sviluppare applicazioni scalabili e ad alte prestazioni. Viene utilizzato per gestire il backend dell'applicazione, inclusa la logica di business, la gestione delle richieste HTTP e l'interazione con il database.

Motivazioni della scelta
La scelta di Node.js per lo sviluppo del backend è stata motivata da diverse considerazioni tecniche e stategiche:

- **Architettura API-first:** L'obiettivo del progetto era creare un backend basato su API RESTful che permettesse una netta separazione tra frontend e backend. Questa architettura facilita eventuali integrazioni future con altri sistemi software aziendali, consentendo di esporre le funzionalità del sistema telefonico attraverso endpoint ben definiti. Un'architettura basata su API rende inoltre il sistema più flessibile e manutenibile nel tempo.
- **Coerenza linguistica:** Mantenere JavaScript come linguaggio principale sia per il frontend che per il backend semplifica lo sviluppo e la manutenzione del codice. Gli sviluppatori possono lavorare su entrambe le parti dell'applicazione senza dover imparare linguaggi diversi, riducendo la curva di apprendimento e migliorando la produttività del team.
- **Ecosistema npm:** Node.js beneficia di npm, uno dei registri di pacchetti più grandi e attivi al mondo. Questo ecosistema offre una vasta gamma di librerie e strumenti che accelerano lo sviluppo, consentendo di integrare funzionalità complesse con facilità.
- **Prestazioni per operazioni I/O:** Node.js è particolarmente adatto per applicazioni che richiedono un'elevata gestione delle operazioni di input/output, come le API RESTful. La sua architettura basata su eventi e il modello non bloccante consentono di gestire un gran numero di connessioni simultanee in modo efficiente.

4.2 Database Management System

4.2.1 MySQL Server

Figura 5: Logo MySQL

MySQL è un database management system (DBMS) relazionale open source tra i più diffusi e utilizzati al mondo. Supporta il linguaggio SQL standart per la gestione e l'interrogazione dei dati, offrendo funzionalità avanzate come gestione delle transazioni ACID, meccanismi di backup e recovery, replicazione dei dati e ottimizzazione delle query. E' particolarmente adatto per applicazioni web grazie alla sua scalabilità, affidabilità e facilità di integrazione con vari linguaggi di programmazione, incluso JavaScript tramite Node.js.

Motivazioni della scelta
La decisione di adottare MySQL Server come DBMS per il progetto è stata guidata principalmente da ragioni di continuità tecnologica e compatibilità con l'infrastruttura esistente:

- **Coerenza con il sistema esistente:** Il centralino telefonico venduto da Cinquenet srl utilizza già MySQL come database per la gestione dei dati operativi (chiamate, utenti, configurazioni, ecc.). Mantenere la stessa tecnologia garantisce uniformità nell'infrastruttura IT aziendale e semplifica notevolmente la gestione complessiva dei sistemi.
- **Competenze interne:** Il personale tecnico dell'azienda possiede già familiarità con MySQL, riducendo la necessità di formazione aggiuntiva e facilitando la manutenzione e l'ottimizzazione del database nel tempo. Il team può gestire autonomamente backup, ottimizzazioni e troubleshooting senza necessità di acquisire nuove competenze su altri DBMS.
- **Leggerezza e prestazioni:** MySQL è noto per la sua efficienza e capacità di gestire carichi di lavoro elevati, rendendolo adatto per applicazioni web che richiedono accessi frequenti al database. La sua architettura ottimizzata consente di ottenere buone prestazioni anche con risorse hardware limitate.

4.2.2 MySQL Workbench

Figura 6: Logo MySQL Workbench

MySQL Workbench è lo strumento ufficiale di amministrazione e sviluppo per MySQL sviluppato da Oracle. Offre un'interfaccia grafica intuitiva per la gestione dei database, consentendo agli sviluppatori e agli amministratori di eseguire operazioni come la progettazione dello schema del database, la scrittura e l'esecuzione di query SQL, la gestione degli utenti e dei permessi, nonché il monitoraggio delle prestazioni del server MySQL.

Motivazioni della scelta
L'adozione di MySQL Workbench come strumento di gestione del database è stata motivata da diversi fattori chiave:

- **Interfaccia grafica intuitiva:** Workbench permette di gestire il database attraverso un'interfaccia visuale user-friendly, semplificando operazioni complesse come la progettazione dello schema ER (Entity-Relationship), la creazione e modifica di tabelle, l'esecuzione di query e la visualizzazione dei risultati. Questo risulta particolarmente utile durante lo sviluppo per verificare rapidamente la struttura dei dati e testare query.
- **Strumento ufficiale:** Essendo lo strumento ufficiale sviluppato da Oracle, MySQL Workbench garantisce piena compatibilità con tutte le funzionalità di MySQL. Questo assicura che tutte le operazioni eseguite tramite Workbench siano supportate e ottimizzate per il server MySQL.

4.3 Strumenti di Sviluppo

4.3.1 Postman

Figura 7: Logo Postman

Postman è una piattaforma completa per lo sviluppo e testing di API che consente di progettare, testare, documentare e monitorare interfacce REST atraverso un'interfaccia intuitiva. E' diventato lo standard de facto per il testing di API grazie alla sua semplicità d'uso e alle sue funzionalità avanzate.

Motivazioni della scelta
Postman è stato scelto come strumento principale per il testing delle API sviluppate nel progetto per diverse ragioni:

- **Testing efficiente delle API:** Durante lo sviluppo del backend basato su API REST, era fondamentale poter testare rapidamente gli endpoint senza dover sviluppare prima il frontend. Postman permette di inviare richieste HTTP (GET, POST, PUT, DELETE) con parametri personalizzati, headers e body in formato JSON, visualizzando immediatamente le risposte del server. Questo ha accelerato significativamente il ciclo di sviluppo e debug.
- **Gestione delle collections:** Postman consente di organizzare le richieste API in collezioni, facilitando la gestione e il riutilizzo dei test. Durante lo sviluppo, sono state create collezioni specifiche per ogni risorsa API, permettendo di eseguire test ripetitivi in modo strutturato.
- **Debugging efficiente:** La visualizzazione dettagliata delle risposte, inclusi status code, headers, body e tempi di risposta, facilita l'identificazione rapida di problemi e l'ottimizzazione delle performance delle API.

4.3.2 Suite JetBrains: IntelliJ IDEA e WebStorm

Figura 8: Logo JetBrains

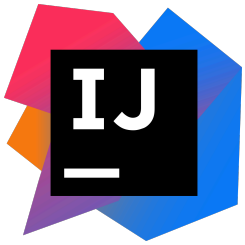


Figura 9: Logo IntelliJ IDEA

Figura 10: Logo WebStorm

JetBrains offre una suite di IDE (Integrated Development Enviroment) professionali specifici per linguaggi e tecnologie. IntelliJ IDEA è ottimizzato per lo sviluppo Java, ma supporta anche JavaScript e Node.js tramite plugin. WebStorm è un IDE specializzato per lo sviluppo web front-end e back-end con supporto nativo per HTML, CSS e JavaScript.

Motivazioni della scelta
L'adozione degli IDE JetBrains per lo sviluppo del progetto è stata motivata da diversi vantaggi chiave:

- **Intelligent code completion:** Gli IDE JetBrains offrono funzionalità avanzate di completamento del codice basate su analisi statica, che accelerano la scrittura del codice riducendo gli errori di sintassi e migliorando la produttività degli sviluppatori.
- **Refactoring avanzato:** Le potenti funzionalità di refactoring consentono di ristrutturare il codice in modo sicuro e efficiente, facilitando la manutenzione e l'evoluzione del progetto nel tempo.
- **Integrazione con strumenti:** Integrazione nativa con Git per il version control, npm per la gestione dei pacchetti, terminale integrato e supporto per l'esecuzione diretta di script Node.js. Questo centralizza il workflow di sviluppo in un'unica applicazione, evitando il continuo cambio di contesto tra diversi strumenti.
- **Specializzazione per contesto:** Utilizzare IntelliJ IDEA per il backend Node.js e WebStorm per il frontend web garantisce strumenti ottimizzati per ciascun stack tecnologico, con funzionalità, suggerimenti e plugin specifici per il tipo di sviluppo in corso.
- **Licenza accademica gratuita:** Come studente universitario, è possibile ottenere gratuitamente licenze educational per tutti i prodotti JetBrains, rendendo accessibile questa suite professionale senza costi aggiuntivi. Questo ha permesso di utilizzare strumenti di qualità enterprise durante lo sviluppo del progetto.

4.4 Versionamento del Codice

4.4.1 Git e GitHub

Figura 11: Logo Git

Figura 12: Logo GitHub

Git è un sistema di controllo versione distribuito che traccia le modifiche al codice sorgente durante lo sviluppo software. GitHub è una piattaforma di hosting per repository Git basata su cloud che aggiunge funzionalità collaborative, gestione progetti e strumenti di integrazione continua.

Motivazioni della scelta
Git e GitHub sono stati scelti come strumenti di versionamento del codice per diverse ragioni fondamentali:

- **Standard de facto:** Git è lo standard industriale per il version control, adottato dalla maggioranza dei progetti software moderni. La sua conoscenza è fondamentale per qualsiasi sviluppatore e la sua adozione garantisce compatibilità con praticamente qualsiasi workflow aziendale.
- **Tracciamento completo delle modifiche:** Ogni commit mantiene uno snapshot completo del progetto con metadata dettagliati (autore, data, messaggio descrittivo). Questo permette di ripercorrere l'intera evoluzione del software, comprendere le motivazioni dietro ogni modifica e identificare quando e dove sono stati introdotti eventuali bug.
- **Branching e sviluppo parallelo:** Il modello di branching di Git permette di lavorare su nuove funzionalità o correzioni in branch isolati senza interferire con il codice principale (branch main/master). Questo consente di sperimentare in sicurezza e di mantenere sempre una versione stabile del codice pronta per il deployment.
- **Backup distribuito:** La natura distribuita di Git garantisce che ogni clone del repository sia un backup completo della storia del progetto. Questo previene perdite di dati e permette di continuare a lavorare anche offline, sincronizzando le modifiche successivamente.

4.5 Documentazione

4.5.1 Typst

Figura 13: Logo Typst

Typst è un sistema di typesetting moderno, progettato come alternativa contemporanea a LaTeX. Utilizza una sintassi più intuitiva e leggera, tempi di compilazione significativamente più rapidi e un'architettura pensata per semplificare la creazione di documenti tecnici di alta qualità tipografica.

Motivazioni della scelta
Typst è stato scelto come strumento di documentazione per il progetto per diverse ragioni chiave:

- **Sintassi intuitiva:** La sintassi di Typst è progettata per essere più leggibile e facile da imparare rispetto a LaTeX. Questo ha permesso di concentrarsi maggiormente sul contenuto del documento piuttosto che sulla complessità della formattazione, accelerando il processo di scrittura.
- **Compilazione rapida:** Typst offre tempi di compilazione molto più veloci rispetto a LaTeX, consentendo di vedere rapidamente le modifiche apportate al documento. Questo consente un workflow iterativo più fluido con preview istantaneo delle modifiche, facilitando la correzione di errori di formattazione e l'aggiustamento del layout in tempo reale.
- **Facilità di apprendimento:** Per chi non ha esperienza pregressa con LaTeX, Typst risulta molto più accessibile e immediato da utilizzare. La documentazione è chiara e moderna, con esempi pratici che permettono di iniziare a produrre documenti professionali rapidamente.

4.6 Containerizzazione

4.6.1 Docker

Figura 14: Logo Docker

Docker è una piattaforma di containerizzazione che permette di pacchettizzare applicazioni con tutte le loro dipendenze in container isolati e portabili. I container sono ambienti di esecuzione leggeri e autosufficienti che garantiscono che l'applicazione funzioni allo stesso modo su qualsiasi sistema che supporti Docker.

Motivazioni della scelta
L'adozione di Docker per il progetto è stata motivata da diversi vantaggi significativi offerti dalla containerizzazione:

- **Ambiente consistente e riproducibile:** Docker elimina il classico problema del «funziona sulla mia macchina» garantendo che l'applicazione giri esattamente allo stesso modo in sviluppo, test e produzione. Questo riduce drasticamente i problemi legati a differenze di configurazione tra ambienti diversi.
- **Isolamento delle dipendenze:** Ogni componente dell'applicazione (backend Node.js, database MySQL, eventuali servizi aggiuntivi) può essere containerizzato separatamente con le proprie dipendenze specifiche, evitando conflitti tra versioni di librerie e semplificando la gestione complessiva del sistema.
- **Deployment semplificato:** Un'immagine Docker contiene tutto il necessario per eseguire l'applicazione: codice, runtime, librerie di sistema e configurazioni. Questo rende il processo di deployment consistente, ripetibile e molto più semplice rispetto all'installazione manuale di tutte le dipendenze su ogni macchina.
- **Facilitazione dello sviluppo:** Docker Compose permette di orchestrare facilmente servizi multipli (backend + database) con una semplice configurazione YAML, semplificando il setup dell'ambiente di sviluppo locale. Nuovi sviluppatori possono avviare l'intero stack con un singolo comando.

Capitolo 5

Progettazione del Sistema

In questo capitolo viene descritta l'architettura del sistema sviluppato durante lo stage, illustrando le scelte progettuali, l'implementazione dei componenti e le problematiche affrontate. La trattazione parte da una visione generale dell'architettura per poi approfondire database, backend e frontend, concludendo con l'analisi delle principali difficoltà e soluzioni adottate.

5.1 Architettura generale del sistema

Il sistema adotta un'architettura three-tier, pattern consolidato che separa le responsabilità in tre tier (livelli) distinti: presentazione (frontend), applicazione (backend) e dati (database). Questa scelta risponde ai requisiti funzionali e non funzionali identificati in fase di analisi.
La separazione garantisce manutenibilità del codice: modifiche all'interfaccia non richiedono interventi sul backend o sul database, aspetto rilevante considerando la necessità aziendale di manutenibilità da parte di personale non coinvolto nello sviluppo iniziale. L'architettura facilita inoltre la scalabilità: ogni componente può essere ottimizzato, sostituito o replicato indipendentemente, adattando il sistema a carichi crescenti senza riprogettazione completa.

5.1.1 Tier presentazione

Il presentation tier consiste in un'applicazione web client-side sviluppata con tecnologie web standard HTML5, CSS3 e JavaScript ES6+. L'interfaccia è stata progettata seguendo principi di usabilità e accessibilità, rispondendo al requisito R-NQ-2 sulla utilizzabilità senza formazione specifica. Il design responsive permette utilizzo su dispositivi diversi (desktop, tablet, smartphone) tramite media queries CSS che adattano layout e dimensioni.

Il frontend comunica con il backend esclusivamente tramite API REST con pattern asincrono basato su Promise e async/await, garantendo reattività nell'esperienza utente. Le chiamate di rete non bloccano l'interfaccia: durante operazioni lunghe, indicatori visivi informano l'utente dello stato, e l'interfaccia resta responsiva permettendo navigazione ad altre sezioni. La scelta di non utilizzare framework frontend pesanti come React, Angular o Vue è stata dettata da considerazioni **pragmatiche**: accessibilità del codice per il team aziendale con competenze JavaScript basilari, riduzione della complessità del progetto evitando build tool complessi e gestione dipendenze pesanti, tempi di caricamento ottimizzati per l'utente finale, e manutenibilità a lungo termine senza dipendenza da evoluzioni rapide di framework esterni.
Questa scelta non ha compromesso la qualità dell'interfaccia: adottando pattern moderni come componenti riutilizzabili e gestione stato centralizzata, è stato possibile sviluppare un'applicazione strutturata e manutenibile pur rimanendo su vanilla JavaScript.

5.1.2 Tier applicazione

L'application tier è implementato mediante server Node.js con API RESTful sviluppate usando il framework Express, soluzione consolidata nell'ecosistema JavaScript. Questo strato intermedio costituisce il nucleo elaborativo del sistema e gestisce tutte le operazioni di business logic. Le responsabilità principali includono autenticazione degli utenti tramite verifica credenziali e generazione token JWT, autorizzazione verificando permessi per operazioni richieste in base al ruolo utente, elaborazione delle richieste provenienti dal frontend con parsing e validazione parametri, aggregazione e trasformazione dei dati secondo necessità business (calcoli statistici, formattazioni, conversioni), gestione centralizzata degli errori con logging strutturato e risposte appropriate, e validazione approfondita degli input per prevenire attacchi injection e dati inconsistenti.
La decisione di adottare un'architettura API-first, dove il backend espone esclusivamente API REST senza occuparsi direttamente della presentazione, risponde al vincolo R-NV-1 e offre vantaggi strategici significativi. L'approccio facilita future integrazioni con altri sistemi aziendali: un eventuale sistema di reportistica automatica che genera PDF giornalieri, un'applicazione mobile nativa per iOS/Android, o un sistema di notifiche real-time, potrebbero tutti interfacciarsi con le stesse API utilizzate dal frontend web senza necessità di duplicare la logica applicativa. Questa architettura supporta anche pattern di sviluppo moderno come microfrontend, dove team diversi possono sviluppare porzioni indipendenti dell'interfaccia utilizzando le stesse API centrali. Durante lo sviluppo, la separazione netta ha permesso di definire prima le interfacce API e poi procedere parallelamente su frontend e backend,

riducendo le dipendenze tra i componenti e consentendo cicli di testing indipendenti per ciascun layer.

5.1.3 Tier dati

Il data tier è rappresentato da un database relazionale MySQL 8.0, DBMS consolidato che offre robustezza, performance, supporto transazionale ACID, e ecosistema ricco di strumenti. Il database gestisce due categorie distinte di dati: dati operativi dell'applicazione includendo utenti, sessioni, configurazioni di sistema, personalizzazioni per utente, e log di sistema; e dati telefonici estratti dal centralino PABX con dettagli completi delle chiamate (CDR), informazioni su interni e gruppi, e statistiche pre-aggregate per query frequenti.
Una decisione progettuale fondamentale è stata mantenere una replica locale completa dei Call Detail Record anziché interrogare direttamente il database del centralino ad ogni richiesta utente. Questa architettura, sebbene introduca complessità con necessità di sincronizzazione periodica e gestione consistenza dati, offre vantaggi critici giustificanti la scelta. Le query complesse per calcolo statistiche (aggregazioni su migliaia di record, join multipli, calcoli su window functions) non impattano il sistema telefonico in produzione che rimane dedicato alla sua funzione primaria di gestione chiamate real-time. I tempi di risposta sono ottimizzati tramite creazione di indici specializzati per le query di reporting che non sarebbe possibile implementare sul database vendor-managed dove non sono stati forniti privilegi DDL. Il sistema rimane operativo anche in caso di temporanea indisponibilità del centralino per manutenzione o problemi di rete, garantendo continuità del servizio di reporting. È possibile implementare trasformazioni e arricchimenti dei dati senza modificare la sorgente: calcoli derivati, normalizzazioni, categorizzazioni possono essere applicati sui dati locali.
Il meccanismo di sincronizzazione è stato progettato per minimizzare l'impatto sul centralino e garantire consistenza. Le sincronizzazioni avvengono con frequenza configurabile (di default ogni giorno a mezzanotte, ma aumentabile fino a pochi minuti in caso di monitoraggio real-time). Il processo è incrementale: solo record nuovi o modificati vengono trasferiti, identificati confrontando timestamp. L'operazione è transazionale: commit avviene solo se l'intera sincronizzazione completa con successo, prevenendo stati parziali. Gestione robusta degli errori con retry automatico e alerting permette identificare rapidamente problematiche di connettività o configurazione.

5.1.4 Flusso di comunicazione tra i livelli

Il flusso di comunicazione tra i livelli segue uno schema ben definito che garantisce consistenza e tracciabilità. Quando un utente interagisce con l'interfaccia web (ad esempio selezionando un periodo temporale per visualizzare statistiche chiamate), viene innescata una sequenza di operazioni: il frontend costruisce una richiesta HTTP contenente i parametri di filtro serializzati in formato appropriato (query parameters per GET, JSON body per POST), allega il token JWT di autenticazione nell'header `Authorization: Bearer <token>`, e invia la richiesta tramite chiamata AJAX alla specifica route API.
Il backend riceve la richiesta e la elabora attraverso una pipeline di middleware: il middleware di logging registra la richiesta per auditing, il middleware di autenticazione verifica il token JWT decodificandolo e validando la firma con la chiave segreta, se il token è valido estrae le informazioni utente (ID, ruolo) e le allega all'oggetto request, il middleware di autorizzazione verifica se l'utente ha i permessi per l'operazione richiesta (es. solo admin possono gestire utenti), il middleware di validazione verifica che i parametri rispettino lo schema atteso.
Superati i middleware, la richiesta raggiunge il controller appropriato che: estrae i parametri validati dalla richiesta, determina quale operazione business deve essere eseguita, invoca uno o più metodi dei model con parametri appropriati, i model costruiscono le query SQL necessarie utilizzando il Query Builder per prevenire SQL injection, eseguono le query sul database MySQL ottenendo result set, elaborano e trasformano i risultati secondo le necessità business (aggregazioni, calcoli, join logici), e restituiscono i dati processati al controller.
Il controller riceve i dati dai model, costruisce la risposta HTTP appropriata con codice di stato semanticamente corretto (200 per successo, 201 per creazione, 400 per errori client, 500 per errori server), serializza i dati in formato JSON, e invia la risposta al frontend. Se si verifica un errore in qualsiasi punto della pipeline, il middleware di gestione errori lo intercetta e restituisce una risposta di errore strutturata al client con messaggio user-friendly.
Il frontend riceve la risposta, verifica il codice di stato HTTP, se la richiesta è riuscita deserializza il JSON e processa i dati ricevuti, aggiorna lo stato dell'applicazione con i nuovi dati, triggera il rendering dei componenti UI che dipendono da quei dati (grafici vengono generati con Chart.js, tabelle con DataTables, cards con componenti custom), e nasconde eventuali indicatori di loading mostrando i dati aggiornati. Se la risposta indica un errore, il frontend mostra un messaggio appropriato all'utente, differenziando tra errori correggibili (es. validazione) che suggeriscono azioni correttive, ed errori di sistema che suggeriscono di riprovare o contattare supporto.

5.2 Progettazione del database

Figura 15: Schema database MySQL con tabelle e relazioni principali.

Il database MySQL si compone di sette tabelle con responsabilità ben definite.

La tabella `cdr` (Call Detail Record) costituisce il nucleo informativo, replicando localmente la tabella del PABX. Memorizza il registro dettagliato delle chiamate: identificativi univoci (`uniqueid`, `linkedid`), timestamp di inizio (`calldate`) e fine, numeri chiamante (`src`) e chiamato (`dst`), durata conversazione (`billsec`) e durata totale chiamata (`duration`), stato finale (`disposition` con valori ANSWERED, NO ANSWER, BUSY, FAILED), tipo dispositivo utilizzato (`channel`, `dstchannel`), informazioni di routing (`context`, `exten`), e campi aggiuntivi per funzionalità avanzate del PABX.
La struttura riflette la complessità del dominio telefonico. I campi `channel` e `dstchannel` descrivono i canali di comunicazione utilizzati,

permettendo di identificare la tecnologia (SIP, IAX2, DAHDI) e il dispositivo specifico coinvolto. Il campo `accountcode` può essere utilizzato per associare chiamate a specifici account o progetti per scopi di billing. La replica locale, sebbene richieda sincronizzazione, previene impatti su performance del centralino e permette indici ottimizzati per il reporting. Un campo particolarmente rilevante è `linkedid`, che raggruppa chiamate logicamente correlate. Nel sistema telefonico, una singola «chiamata» dal punto di vista dell'utente può generare multiple entry nel CDR. Quando un cliente chiama un DID che attiva un ring group, si creano: una entry per la chiamata in ingresso al DID, una entry per ogni interno del ring group che viene fatto squillare (anche se non risponde), entry aggiuntive se la chiamata viene trasferita o messa in attesa, entry per interazioni con sistemi IVR o voicemail. Il `linkedid` permette di correlare tutte queste entry separate, ricostruendo il percorso completo della chiamata attraverso il sistema. Questa capacità è stata sfruttata per implementare la funzionalità di analisi dettagliata delle chiamate (requisito R-DF-2), che mostra all'utente tutti i passaggi di una chiamata complessa con timeline visuale e dettagli su ogni hop.

La tabella `users` gestisce autenticazione e autorizzazione degli utenti. Memorizza username univoco utilizzato per il login, password hashata con bcrypt, ruolo utente (enum con valori "superadmin", "admin" e "user") che determina i permessi, email per comunicazioni e recupero password, e flag per indicare account attivi/disabilitati. Il sistema di ruoli implementa un modello RBAC (Role-Based Access Control) semplificato ma efficace: gli amministratori possono gestire utenti (creare, modificare, eliminare), configurare impostazioni di sistema, e accedere a tutte le funzionalità, mentre gli utenti standard possono visualizzare dati secondo i loro filtri personalizzati, personalizzare le proprie dashboard, e accedere solo alle funzionalità di reporting. Questa separazione risponde al requisito R-OF-5 sulla gestione differenziata delle autorizzazioni e previene accessi non autorizzati a funzionalità sensibili. **INSERIRE SUPERADMIN**

La tabella `settings` memorizza configurazioni globali del sistema in formato chiave-valore, un pattern comune per impostazioni che possono evolvere nel tempo senza modifiche allo schema. Le configurazioni attualmente implementate includono: il nome dell'azienda (visualizzato nell'interfaccia utente), l'identificativo del server PABX utilizzato per recuperare i dati tramite API, il tenant ID per filtrare e prelevare dal database solo i dati della tenant correttamente, e l'intervallo di sincronizzazione automatica espresso in minuti. Queste impostazioni permettono flessibilità nella gestione del sistema senza necessità di interventi sul

database o sul codice, rispondendo al requisito R-NQ-4 sulla facilità di configurazione. **CONTROLLARE**

La tabella `sync_log` traccia cronologia delle sincronizzazioni con il PABX, un requisito fondamentale per monitoring e troubleshooting. Ogni sincronizzazione genera un record contenente timestamp inizio e fine operazione permettendo calcolo durata, numero di record importati dalla tabella CDR del centralino, numero di record effettivamente inseriti (può differire se alcuni record erano già presenti), e stato finale (success, partial_success, failed). Durante lo sviluppo e testing, questa tabella si è rivelata indispensabile per debugging del meccanismo di sincronizzazione.
FINIRE TABELLE

5.3 Sicurezza nella progettazione

La sicurezza è stata considerata fin dalle prime fasi progettuali, seguendo principi defense in depth e least privilege.

5.3.1 Autenticazione e autorizzazione

L'autenticazione è basata su JWT con:
- Password hashate con bcrypt (salt factor 10, computazionalmente costoso per prevenire brute force)
- Token firmati con chiave segreta (HMAC-SHA256)
- Scadenza configurabile (default 24h, bilanciamento sicurezza/UX)
- Refresh token per estensione sessioni senza re-login frequenti

L'autorizzazione implementa RBAC con verifica permessi a livello middleware: ogni endpoint protetto verifica ruolo richiesto prima di eseguire operazione.

5.3.2 Protezione da attacchi comuni

SQL Injection: prevenuta tramite prepared statement con binding parametrico, mai concatenazione diretta di input utente in query
XSS (Cross-Site Scripting): input utente sanitizzato prima del rendering, uso Content Security Policy header
CSRF (Cross-Site Request Forgery): token CSRF per operazioni state-changing, verifica origin/referer header
Validazione input: schema validation stringente con Joi per ogni parametro, whitelist di valori ammissibili
Rate limiting: middleware per limitare numero richieste per IP su endpoint critici (es. login)

5.3.3 Gestione credenziali

Credenziali sensibili (database password, JWT secret, API keys) sono:
- Memorizzate in variabili ambiente, mai versionate
- Diverse per ogni ambiente (development, staging, production)
- Ruotate periodicamente secondo policy aziendale
- Accessibili solo a processi autorizzati con least privilege

5.4 Considerazioni su scalabilità e performance

La progettazione considera requisiti di scalabilità futuri pur mantenendo semplicità appropriata per volumetrie attuali.

5.4.1 Scalabilità orizzontale

L'architettura stateless del backend (JWT anziché session server-side) permette scaling orizzontale: multiple istanze backend possono operare dietro load balancer senza necessità di session affinity o shared state.
Il frontend, essendo completamente client-side, scala naturalmente: può essere servito da CDN o web server statici replicati.
Il database rappresenta il potenziale collo di bottiglia: inizialmente deploy single-instance, ma architettura permette future implementazioni di read replicas per distribuire carico query analitiche.

5.4.2 Ottimizzazioni performance progettuali

Query ottimizzate: progettate per minimizzare join complessi, uso di viste materializzate per calcoli pesanti
Indici database: progettati analizzando query frequenti durante design, bilanciando performance lettura vs overhead scrittura
Paginazione: progettata a livello API con cursori per evitare offset pesanti su grandi dataset

5.4.3 Monitoring e observability

La progettazione include punti di osservabilità:
- Logging strutturato a livelli (debug, info, warning, error)
- Metriche temporali per ogni query database
- Trace ID propagato attraverso request pipeline per correlazione log
- Health check endpoint per monitoring esterno

Questi elementi facilitano identificazione proattiva di problemi performance e debugging in produzione.

Capitolo 6

Implementazione del Sistema

In questo capitolo viene descritta l'implementazione concreta del sistema, illustrando i pattern architetturali e di design utilizzati, le tecniche di sviluppo adottate, e come sono stati realizzati i requisiti funzionali. La trattazione copre l'organizzazione del codice, i meccanismi implementativi chiave, e le problematiche affrontate durante lo sviluppo con relative soluzioni.

6.1 Organizzazione del codice backend

Il backend è strutturato secondo pattern MVC (Model-View-Controller) adattato per API REST, dove controller gestiscono richieste HTTP, model astraggono l'accesso al database, e non esistono view tradizionali ma risposte JSON.

6.1.1 Struttura delle cartelle

Il codice backend è organizzato in moduli con responsabilità ben definite seguendo il principio di Separation of Concerns:

Routes (`/routes`): contiene file che definiscono endpoint API, mappando combinazioni URL + metodi HTTP a controller. Ogni risorsa principale (users, calls, stats, config) ha il proprio file di routes, facilitando navigazione e manutenzione:

```javascript
// routes/extRoutes.js
const express = require("express");
const router = express.Router();
const ExtController = require("../controllers/
extController");
```

```
5  const { authenticate, authorize } = require('../middleware/auth');
6  const { validateRequest } = require('../middleware/validation');
7  const { extensionSchemas } = require('../validators/extensionSchemas');
8
9  // GET /api/extensions/list - Lista tutte le estensioni
10 router.get(
11   "/list",
12   authenticate,
13   ExtController.getExtensions
14 );
15
16 // GET /api/extensions/info?id=xxx - Dettaglio estensione per id
17 router.get(
18   "/info",
19   authenticate,
20   validateRequest(extensionSchemas.getById),
21   ExtController.getExtensionById
22 );
23
24 module.exports = router;
```

Controllers (`/controllers`): contiene business logic per ciascun endpoint. I controller sono funzioni async che ricevono oggetti request/response, estraggono parametri, invocano model appropriati, processano dati, costruiscono risposte HTTP, e gestiscono errori delegando al middleware centralizzato. Ogni controller è focalizzato su un'operazione specifica, mantenendo funzioni concise (< 50 righe) e testabili.

```
1  // controllers/extController.js                    JavaScript
2  const ExtModel = require("../models/ExtModel");
3
```

```
4   const ExtController = {
5       async getExtensions(req, res) {
6           try {
7               const extensions = await
                ExtModel.getExts();
8
9               if (!extensions) {
10                  return res.status(404).json({
11                      error: 'Nessun interno trovato'
12                  });
13              }
14
15              res.json({
16                  success: true,
17                  data: extensions,
18                  count: extensions.length
19              });
20
21          } catch (error) {
22              console.error('Errore getExtensions:',
                error);
23              res.status(500).json({
24                  error: 'Errore nel recupero degli
                    interni',
25                  details: error.message
26              });
27          }
28      },
29
30      async getExtensionById(req, res) {
31          try {
32              const { ext } = req.query;
33
34              // Validazione parametro
35              if (!ext || ext.trim() === '') {
```

```
36                 return res.status(400).json({
37                     error: 'Parametro ext
                       obbligatorio'
38                 });
39             }
40
41             const extension = await
               ExtModel.getExtById(ext.trim());
42
43             if (!extension) {
44                 return res.status(404).json({
45                     error: `Interno ${ext} non
                       trovato`
46                 });
47             }
48
49             res.json({
50                 success: true,
51                 data: extension
52             });
53
54         } catch (error) {
55             console.error('Errore
               getExtensionByNumber:', error);
56             res.status(500).json({
57                 error: 'Errore nel recupero
                   dell\'interno',
58                 details: error.message
59             });
60         }
61     },
62 };
63
64 module.exports = ExtController;
```

Models (`/models`): contiene classi che incapsulano interazione con database. Ogni model corrisponde a una o più tabelle correlate e fornisce metodi per CRUD e query complesse. I model astraggono completamente i dettagli SQL dai controller: un controller non costruisce mai query SQL direttamente, ma invoca metodi del model con parametri business-logic. Esempio:

JavaScript

```
// models/ExtModel.js
const BaseModel = require('./BaseModel');

/**
 * ExtModel - Classe che estende BaseModel
 */
class ExtModel extends BaseModel {
    constructor() {
        super('ext');
    }

    /**
     * Trova tutti gli interni
     * @returns {Promise<Object|null>}
     */
    async getExts() {
        try {
            const query = this.query()
                .orderBy('ext')

            const results = await
            this.execute(query);
            return results.length > 0 ? results :
            null;
        } catch (error) {
            throw new Error(`Errore nel recupero
            degli interni: ${error.message}`);
        }
    }
```

```
27
28      /**
29       * Trova interno per ext
30       * @param {string} ext
31       * @returns {Promise<Object|null>}
32       */
33      async getExtById(ext) {
34          try {
35              const query = this.query()
36                  .where('ext', '=', ext);
37
38              const results = await
                this.execute(query);
39              return results.length > 0 ?
                results[0] : null;
40          } catch (error) {
41              throw new Error(`Errore nel recupero
                dell'interno: ${error.message}`);
42          }
43      }
44  }
45
46  module.exports = new ExtModel();
```

Middleware (`/middleware`): contiene funzioni di elaborazione intermedia che operano tra ricezione richiesta ed esecuzione controller. Middleware implementati:

- `authenticate`: verifica token JWT ed estrae informazioni utente
- `authorize(role)`: verifica permessi per operazioni specifiche
- `validateRequest(schema)`: valida parametri contro schema Joi
- `rateLimiter`: previene abusi limitando richieste per IP
- `errorHandler`: gestisce errori centralizzata

Utils (`/utils`): contiene funzioni di utilità riutilizzabili per parsing e formattazione date, conversioni tra formati, validazioni comuni, hashing e crittografia. Centralizzano logica condivisa evitando duplicazione. Inoltre contiene il Query Builder per costruzione dinamica di query SQL complesse.

Questa organizzazione segue il principio Single Responsibility: ogni modulo ha un'unica ragione di cambiamento. Facilita testing unitario (ogni componente testabile isolatamente con mock), debugging (problemi localizzabili rapidamente), e onboarding (struttura chiara permette comprensione rapida).

6.2 Pattern Architetturali e di Design

L'implementazione del sistema adotta diversi pattern consolidati che migliorano manutenibilità, testabilità ed estensibilità del codice. I pattern descritti non sono stati applicati per ragioni teoriche, ma emergono da esigenze concrete emerse durante lo sviluppo e dalla necessità di risolvere problemi specifici mantenendo il codice pulito e manutenibile.

6.2.1 Builder Pattern

Il Builder Pattern è stato implementato nel backend per la costruzione dinamica di query SQL complesse. La necessità emerge dalla grande varietà di filtri applicabili alle query (periodo temporale, interni specifici, gruppi, DID, direzione chiamata, durata, esito) la cui combinazione genererebbe migliaia di query statiche con approccio impraticabile dal punto di vista della manutenibilità.
Il `QueryBuilder` accumula clausole progressivamente, permettendo composizione flessibile attraverso method chaining:

```javascript
// utils/QueryBuilder.js
class QueryBuilder {
  constructor(baseTable) {
    this.baseTable = baseTable;
    this.selectFields = ['*'];
    this.joins = [];
    this.whereConditions = [];
    this.groupByConditions = []
    this.orderByFields = [];
    this.limitValue = null;
    this.offsetValue = null;
    this.params = [];
  }

```

```
  select(fields) {
    if (Array.isArray(fields)) {
        this.selectFields = fields;
    } else if (typeof fields === 'string') {
        this.selectFields = fields.split(',').map(f => f.trim());
    }
    return this;
  }

  join(table, condition, type = 'INNER') {
    this.joins.push(`${type} JOIN ${table} ON ${condition}`);
    return this;
  }

  where(field, operator, value) {
    if (value !== undefined && value !== null && value !== '') {
        this.whereConditions.push(`${field} ${operator} ?`);
        this.params.push(value);
    }
    return this;
  }

  whereIn(field, values) {
    if (Array.isArray(values) && values.length > 0) {
        const placeholders = values.map(() => '?').join(',');
        this.whereConditions.push(`${field} IN (${placeholders})`);
        this.params.push(...values);
    }
```

```
43      return this;
44    }
45
46    whereBetween(field, startValue, endValue) {
47      if (startValue !== undefined && startValue !==
        null && startValue !== '') {
48        this.where(field, '>=', startValue);
49      }
50      if (endValue !== undefined && endValue !== null
        && endValue !== '') {
51        this.where(field, '<=', endValue);
52      }
53      return this;
54    }
55
56    groupBy(field) {
57      if (field !== undefined && field !== null &&
        field !== '') {
58          this.groupByConditions.push(`${field}`);
59      }
60      return this;
61    }
62
63    /*
64    ... altri metodi come whereLike, orderBy, limit,
    offset
65    */
66
67    build() {
68      let query = `SELECT ${this.selectFields.join(',
        ')} FROM ${this.baseTable}`;
69      if (this.joins.length > 0) {
70        query += ` ${this.joins.join(' ')}`;
71      }
72      if (this.whereConditions.length > 0) {
```

```
73          query += ` WHERE
            ${this.whereConditions.join(' AND ')}`;
74        }
75        if (this.groupByConditions.length > 0) {
76          query += ` GROUP BY
            ${this.groupByConditions.join(' ,')}`;
77        }
78        if (this.orderByFields.length > 0) {
79          query += ` ORDER BY
            ${this.orderByFields.join(', ')}`;
80        }
81        if (this.limitValue !== null) {
82          query += ` LIMIT ${this.limitValue}`;
83        }
84        if (this.offsetValue !== null) {
85          query += ` OFFSET ${this.offsetValue}`;
86        }
87        return {
88          query: query,
89          params: this.params
90        };
91      }
92  }
```

Il pattern garantisce sicurezza e manutenibilità:

- **Prevenzione SQL Injection**: uso esclusivo di prepared statement con binding parametrico. I valori utente non vengono mai concatenati direttamente nella query
- **Leggibilità**: il method chaining rende esplicita la logica di costruzione della query
- **Flessibilità**: aggiungere nuovi filtri richiede solo aggiungere metodi al builder senza modificare codice esistente
- **Testabilità**: il builder può essere testato unitariamente verificando che generi query corrette
- **Riutilizzo**: lo stesso builder è utilizzabile in context diversi (model, report, export)

6.2.2 Repository Pattern

Il Repository Pattern astrae l'accesso ai dati fornendo un'interfaccia orientata al dominio business che nasconde completamente i dettagli di persistenza. I model implementano questo pattern fungendo da intermediari tra i controller e il database, esponendo metodi semantici che esprimono operazioni di business anziché dettagli SQL.
L'obiettivo principale è separare la **logica di business** (cosa fare con i dati) dalla **logica di accesso ai dati** (come recuperare/memorizzare i dati). I controller operano con concetti di dominio senza conoscere tabelle, join, indici o dialetti SQL specifici.
I model sono organizzati per entità di dominio, ciascuno responsabile dell'accesso ai dati della propria area:

```javascript
// models/RingGroupModel.js
const BaseModel = require('./BaseModel');

class RingGroupModel extends BaseModel {
  constructor() {
    super('ring_group');
  }

  async getRingGroups() {
    try {
      const query = this.query()
        .select([
          'ring_group.ext AS ext',
          'ring_group.nome AS nome',
          'GROUP_CONCAT(rgd.ext ORDER BY rgd.ext SEPARATOR \'-\') AS destinazioni'
        ])
        .leftJoin('ring_group_destination rgd', 'ring_group.id = rgd.id_gruppo')
        .groupBy('ring_group.id')
        .orderBy('ring_group.ext');

      const results = await this.execute(query);
      return results.length > 0 ? results : null;
```

```
23      } catch (error) {
24        throw new Error(`Errore nel recupero delle
          ring group: ${error.message}`);
25      }
26    }
27
28    async getRingGroup(ext) {
29      try {
30        const query = this.query()
31          .where('ext', '=', ext)
32
33        const results = await this.execute(query);
34        return results.length > 0 ? results[0] :
          null;
35      } catch (error) {
36        throw new Error(`Errore nel recupero della
          ring group: ${error.message}`);
37      }
38    }
39
40    async getRingGroupById(id) {
41      try {
42        const query = this.query()
43          .where('id', '=', id)
44
45        const results = await this.execute(query);
46        return results.length > 0 ? results[0] :
          null;
47      } catch (error) {
48        throw new Error(`Errore nel recupero della
          ring group: ${error.message}`);
49      }
50    }
51  }
52
```

```
53  module.exports = new RinGroupModel();
```

6.2.3 Chain of Responsibility

Il Chain of Responsibility è un pattern comportamentale che permette di passare richieste lungo una catena di handler, dove ogni handler decide se processare la richiesta o passarla al successivo. Nel contesto di Express.js, questo pattern è implementato nativamente attraverso il sistema di **middleware**, dove ogni middleware nella catena ha l'opportunità di elaborare la richiesta HTTP, arricchirla con informazioni aggiuntive, terminare la catena restituendo una risposta, oppure delegare al middleware successivo.

Architettura della Pipeline

Express.js implementa il pattern attraverso una sequenza ordinata di funzioni middleware che ricevono tre parametri: l'oggetto `request`, l'oggetto `response`, e la funzione `next()` che permette di passare il controllo al middleware successivo. Ogni middleware può modificare gli oggetti request/response condividendo stato lungo la catena, oppure interrompere l'elaborazione restituendo una risposta al client.
La definizione delle route mostra esplicitamente la catena di responsabilità:

```
1   // routes/accountRoutes.js                    JS JavaScript
2   const express = require('express');
3   const router = express.Router();
4   const AccountController = require("../controllers/
    accountController");
5   const verifyToken = require('../middleware/
    verifyToken');
6   const authorizeRoles = require('../middleware/
    authorizeRoles');
7   const accountSchemas } = require('../validators/
    accountSchemas');
8
9   // Chain completa per endpoint protetto con
    validazione
10  router.get(
```

```
    '/list',
    verifyToken,
    accountSchemas(accountSchemas.getList),
    AccountController.getAccountList
);

// Chain con autorizzazione per operazioni amministrative
router.post(
    '/new',
    verifyToken,
    authorizeRoles('admin'),
    accountSchemas(accountSchemas.insertAccount),
    AccountController.insertAccount
);

module.exports = router;
```

Ogni route definisce una pipeline specifica componendo middleware riutilizzabili. L'ordine è cruciale: l'autenticazione deve precedere l'autorizzazione, la validazione deve precedere la business logic.

6.3 Implementazione dell'autenticazione

L'autenticazione è implementata tramite JSON Web Token (JWT), standard industriale per gestione sessioni in applicazioni distribuite.

6.3.1 Processo di login

Al login, la sequenza è:
1. Client invia POST a `/auth/login` con username e password.
2. Controller autentica credenziali contro database usando bcrypt per confronto hash.
3. Se valide, genera JWT contenente user ID, nome, username e ruolo.
4. Token firmato con chiave segreta (HMAC-SHA256) memorizzata in variabile ambiente.
5. Token restituito al client con scadenza configurabile (default 30 giorni).

Implementazione:

```javascript
const AuthController = {
  login: async (req, res) => {
    try {
      const { username, password } = req.body;

      const user = await userModel.getUserByUsername(username);
      if (!user) {
        return res.status(401).json({ error: "Credenziali non valide" });
      }

      const passwordValid = await bcrypt.compare(password, user.password);
      if (!passwordValid) {
        return res.status(401).json({ error: "Credenziali non valide" });
      }

      if (!user.autorizzato) {
        return res.status(401).json({ error: "Utente non autorizzato" });
      }

      const token = AuthController._generateToken(user);
      AuthController._setTokenCookie(res, token);

      res.json({
        message: 'Login effettuato',
        user: {
          id: user.id,
          nome: user.nome,
          username: user.username,
          ruolo: user.tipo_utente
```

```
30          }
31        });
32      } catch (err) {
33        console.error('Login error:', err);
34        res.status(500).json({ error: "Errore nel
          login" });
35      }
36    },
37
38    _generateToken: (user) => {
39      const payload = {
40        id: user.id,
41        nome: user.nome,
42        username: user.username,
43        ruolo: user.tipo_utente,
44      };
45      return jwt.sign(payload, jwtConfig.secret, {
46        expiresIn: jwtConfig.expiresIn
47      });
48    },
49
50    _setTokenCookie: (res, token) => {
51      res.cookie('token', token, {
52        httpOnly: true,
53        maxAge: 30 * 24 * 60 * 60 * 1000,
54        secure: process.env.NODE_ENV ===
          'production',
55        sameSite: 'Lax'
56      });
57    }
58  };
```

6.4 Implementazione della gestione errori

Gestione errori implementa middleware centralizzato che intercetta eccezioni da qualsiasi punto dell'applicazione.

6.4.1 Classificazione errori

Errori classificati in categorie con trattamento differenziato:
Errori validazione (400): input non conforme, parametri mancanti/ malformati. Dettagli specifici su validazione fallita per correzione immediata.
Errori autenticazione (401): token mancante, scaduto, invalido. Istruzione di effettuare nuovo login.
Errori autorizzazione (403): utente autenticato ma senza permessi. Distingue da «non sei loggato».
Errori risorsa non trovata (404): risorsa richiesta non esiste. Distingue tra errore client e server.
Errori database (500): problemi connessione, violazioni vincoli, timeout. Loggati con priorità alta.
Errori interni (500): eccezioni non anticipate, bug. Loggati con stack trace completo.

6.5 Implementazione delle API REST

Le API seguono principi REST con rigore architetturale.

6.5.1 Uso semantico dei metodi HTTP

- **GET**: operazioni sola lettura senza side-effect
- **POST**: creazione nuove risorse, non idempotente
- **PUT**: aggiornamento completo risorse esistenti, idempotente
- **DELETE**: rimozione risorse, idempotente

Esempio:

```javascript
// GET /api/users - lista utenti
// GET /api/users/:id - dettaglio utente
// POST /api/users - crea nuovo utente
// PUT /api/users/:id - aggiorna utente completo
// DELETE /api/users/:id - elimina utente
```

6.5.2 URL gerarchici

Risorse identificate da URL che riflettono relazioni del dominio:

```
/api/users/{userId}/extensions - interni associati a utente
/api/calls/{callId}/linked - chiamate correlate
/api/stats/hourly?start=...&end=... - statistiche orarie per periodo
```

6.5.3 Codici di stato HTTP

Risposte usano codici semanticamente corretti:

- **200 OK**: operazione riuscita con contenuto
- **201 Created**: risorsa creata (include header Location)
- **204 No Content**: operazione riuscita senza contenuto di ritorno
- **400 Bad Request**: errori validazione
- **401 Unauthorized**: mancata autenticazione
- **403 Forbidden**: mancata autorizzazione
- **404 Not Found**: risorsa non trovata
- **409 Conflict**: conflitto stato (es. username già esistente)
- **500 Internal Server Error**: errori server

6.5.4 Formato risposte

Tutti i payload in JSON con strutture consistenti:
Successo:

```json
{
  "data": { ... },
  "meta": {
    "timestamp": "2024-01-15T10:30:00Z",
    "requestId": "req_abc123"
  }
}
```

Errore:

```json
{
  "error": {
    "message": "Descrizione errore",
    "code": 400,
```

```
    "details": { "field": "username", "issue": "già esistente" },
    "requestId": "req_abc123"
  }
}
```

6.6 Implementazione frontend

Frontend implementa pattern SPA (Single Page Application) dove navigazione aggiorna dinamicamente contenuto senza reload pagina.

6.6.1 Organizzazione modulare

Codice organizzato per modularità e riutilizzo:
HTML: struttura semantica con elementi HTML5 appropriati (header, nav, main, section, article, footer)
CSS: metodologia BEM per nominazione classi garantendo specificità chiara. Organizzato in file tematici:

- `base.css`: reset e stili globali
- `layout.css`: griglia e posizionamento
- `components.css`: componenti riutilizzabili
- `pages.css`: stili specifici pagina
- `responsive.css`: media queries e adattamenti mobile

JavaScript: moduli ES6 con componenti riutilizzabili. Ogni componente esporta funzione o classe che incapsula markup, stile e logica. Esempio:

```javascript
// components/DashboardCard.js
export function createDashboardCard({ title, value, icon, trend }) {
  const card = document.createElement('div');
  card.className = 'dashboard-card';
  card.innerHTML = `
    <div class="card__icon">${icon}</div>
    <div class="card__content">
      <h3 class="card__title">${title}</h3>
      <p class="card__value">${value}</p>
      <span class="card__trend ${trend > 0 ? 'positive' : 'negative'}">
```

```
            ${trend > 0 ? '↑' : '↓'}
            ${Math.abs(trend)}%
        </span>
      </div>
    `;
    return card;
}
```

6.6.2 Routing client-side

Router client-side implementato per navigazione SPA mantenendo funzionalità browser (back/forward, bookmarking):

JavaScript

```
class Router {
    constructor(routes) {
      this.routes = routes;
      this.init();
    }

    init() {
      // Intercetta click su link interni
      document.addEventListener('click', (e) => {
        if (e.target.matches('[data-link]')) {
          e.preventDefault();
          this.navigate(e.target.href);
        }
      });

      // Gestisce navigazione browser
      window.addEventListener('popstate', () => {
        this.loadRoute(window.location.pathname);
      });

      this.loadRoute(window.location.pathname);
    }

```

```
24    navigate(url) {
25      history.pushState(null, null, url);
26      this.loadRoute(url);
27    }
28
29    loadRoute(path) {
30      const route = this.routes.find(r =>
        r.path.test(path));
31      if (route) {
32        route.component();
33      }
34    }
35  }
```

Supporta path parametrici (`/call/:callId`) con estrazione parametri e query parameters per filtri preservati in URL.

6.6.3 Service layer

Astrae comunicazione con backend esponendo funzioni JavaScript semantiche:

```
1   class CallService {                                    JavaScript
2     static async getCallDetails(callId) {
3       const response = await fetch(`/api/calls/
        ${callId}`, {
4         headers: {
5           'Authorization': `Bearer ${getToken()}`,
6           'Content-Type': 'application/json'
7         }
8       });
9
10      if (!response.ok) {
11        throw new ApiError(response.status, await
          response.json());
12      }
13
14      return response.json();
```

```
  }

  static async getCallsByPeriod(filters) {
    const params = new URLSearchParams(filters);
    const response = await fetch(`/api/calls?${params}`, {
      headers: { 'Authorization': `Bearer ${getToken()}` }
    });

    if (!response.ok) {
      throw new ApiError(response.status, await response.json());
    }

    return response.json();
  }
}
```

Service layer gestisce automaticamente autenticazione (inserimento token JWT da localStorage), parsing risposte, e gestione errori lanciando eccezioni tipizzate intercettabili dai componenti.

6.6.4 Gestione stato

Implementato pattern Observer semplificato con oggetto globale `AppState`:

```javascript
class AppState {
  constructor() {
    this.state = {};
    this.listeners = {};
  }

  setState(path, value) {
    setNestedProperty(this.state, path, value);
    this.notifyListeners(path);
  }
```

```
11
12    getState(path) {
13      return getNestedProperty(this.state, path);
14    }
15
16    subscribe(path, callback) {
17      if (!this.listeners[path]) {
18        this.listeners[path] = [];
19      }
20      this.listeners[path].push(callback);
21    }
22
23    notifyListeners(path) {
24      const listeners = this.listeners[path] || [];
25      listeners.forEach(callback =>
        callback(this.getState(path)));
26    }
27  }
```

Componenti si sottoscrivono a cambiamenti stato specifici reagendo con aggiornamenti UI. Centralizza stato prevenendo inconsistenze e facilita debugging tracciando modifiche in unico punto.

6.6.5 Librerie di visualizzazione

Chart.js per grafici: supporta tutte tipologie necessarie (linee, barre, torta, doughnut), API chiara, performance buone, personalizzazione ampia. Grafici implementati:

- Linee per trend temporali (volumi chiamate, durate medie)
- Barre per distribuzioni (chiamate per interno, per direzione)
- Torta per composizioni percentuali (distribuzione per esito)
- Combinati con assi multipli per correlazioni

DataTables per tabelle complesse: fornisce sorting multi-colonna, filtering full-text e per-colonna, paginazione, export (CSV, Excel, PDF), responsive design. Richiede jQuery come dipendenza, inizialmente considerato limite ma benefici hanno superato perplessità.

Entrambe librerie mature, ampiamente adottate, ben documentate, attivamente mantenute. Decisione di usarle ha risparmiato settimane sviluppo permettendo focus su funzionalità business-specific.

6.7 Implementazione sincronizzazione CDR

Sincronizzazione dati dal PABX è processo critico implementato con attenzione a robustezza e performance.

6.7.1 Processo incrementale

Sincronizzazione avviene periodicamente (configurabile, default ogni giorno a mezzanotte). Meccanismo:

1. Recupera timestamp ultima sincronizzazione riuscita da `sync_log`
2. Interroga database PABX per record CDR con `calldate > lastSyncTimestamp`
3. Trasferisce record in batch (1000 alla volta per bilanciare memoria e numero query)
4. Inserisce record in database locale verificando duplicati tramite `uniqueid`
5. Registra risultato in `sync_log` con metriche (record importati, durata, errori)
6. Aggiorna timestamp ultima sincronizzazione riuscita

```javascript
async function syncCDR() {
  const startTime = Date.now();
  const lastSync = await getLastSuccessfulSync();

  try {
    const newRecords = await fetchCDRFromPABX(lastSync.timestamp);
    let imported = 0;

    for (const batch of chunk(newRecords, 1000)) {
      await db.transaction(async (trx) => {
        for (const record of batch) {
          await trx('cdr').insert(record).onConflict('uniqu
          imported++;
        }
      });
    }

```

```
18      await logSync({
19        status: 'success',
20        recordsImported: newRecords.length,
21        recordsInserted: imported,
22        duration: Date.now() - startTime
23      });
24
25    } catch (error) {
26      await logSync({
27        status: 'failed',
28        error: error.message,
29        duration: Date.now() - startTime
30      });
31      throw error;
32    }
33  }
```

6.7.2 Gestione transazioni

Inserimenti avvengono in transazioni per atomicità: o tutti record batch vengono importati o nessuno, prevenendo stati parziali. Gestione errori robusta:

- Errori connessione: retry con backoff esponenziale (3 tentativi, intervalli 1s, 2s, 4s)
- Errori parsing: singolo record invalido loggato ma non blocca importazione altri
- Violazioni vincoli: gestite con `ON CONFLICT IGNORE` per duplicati

6.7.3 Monitoring e alerting

Sistema traccia sincronizzazioni con metriche:

- Durata media sincronizzazioni (per identificare degradazioni)
- Tasso errori (spike indicano problemi)
- Volume dati importati (variazioni anomale segnalano problemi PABX)

Alert configurabili:

- 3 sincronizzazioni consecutive fallite → email admin
- Durata sincronizzazione > 10x media → warning performance
- Nessun nuovo record per > 24h → possibile problema raccolta dati

6.8 Problematiche affrontate e soluzioni

6.8.1 Gestione asincronicità

Natura asincrona JavaScript e chiamate di rete ha richiesto attenzione. Promise e async/await adottati sistematicamente per leggibilità.
Loading states: implementati indicatori visivi (spinner, progress bar) durante operazioni lunghe, cruciale per UX con elaborazioni di grandi volumi.
Gestione errori di rete: retry automatico con backoff esponenziale per richieste fallite per problemi transitori (timeout, errori 5xx). Sistema tenta fino a 3 volte con intervalli crescenti prima di presentare errore all'utente.
Timeout: configurabili (default 30s) per evitare attese indefinite. Dopo timeout, richiesta cancellata e utente informato.
Race condition management: quando utente cambia rapidamente filtri, richieste multiple possono completare in ordine diverso. Soluzione con request cancellation tramite AbortController:

JavaScript

```
let currentRequest = null;

async function loadData(filters) {
  // Cancella richiesta pendente
  if (currentRequest) {
    currentRequest.abort();
  }

  currentRequest = new AbortController();

  try {
    const data = await fetch(url, { signal:
    currentRequest.signal });
    renderData(data);
  } catch (error) {
    if (error.name !== 'AbortError') {
      handleError(error);
    }
  }
}
```

Garantisce solo risultati ultima richiesta vengano visualizzati.

6.8.2 Visualizzazione grandi dataset

Applicazione gestisce dataset enormi (anni di storico chiamate). Visualizzazione diretta impraticabile per performance e usabilità.
Campionamento automatico per grafici: backend determina granularità appropriata in base a range temporale:
- Poche ore → bucket 5 minuti
- Giorni → bucket orari
- Settimane/mesi → bucket giornalieri
- Anni → bucket mensili

Numero punti resta entro limiti gestibili (< 300) indipendentemente da ampiezza periodo.
Paginazione server-side per tabelle: frontend richiede pagina specifica (default 50 righe), backend restituisce solo quella porzione con conteggio totale. Controlli usabilità avvisano quando filtro produce risultati eccessivi (> 10.000), suggerendo raffinamento pur permettendo proseguire.

6.8.3 Ottimizzazione performance

Performance preoccupazione costante, rispondendo a R-NQ-1 sui tempi di risposta rapidi.
Profiling query: backend strumentato per registrare tempo esecuzione ogni query. Query lente analizzate con EXPLAIN di MySQL.
Esempio concreto: query per calcolare volume chiamate per interno in mese impiegava 8s con 500K record. EXPLAIN rivelava full table scan. Creazione indice composito su `(calldate, src, disposition)` ha ridotto tempo a 0.3s (miglioramento 25x).
Altri indici creati:
- `linkedid` per chiamate correlate (5s → 0.2s)
- `dst` per statistiche numero chiamato
- Compositi su combinazioni frequenti

Indici migliorano letture introducendo overhead scritture. Nel contesto (sincronizzazioni periodiche, letture dominanti), trade-off favorevole. Analisi ha mostrato indici rallentano sincronizzazioni 15% ma migliorano query 10-30x.
Riduzione volume dati trasferiti: query selezionano esplicitamente solo colonne necessarie. Per grafici, backend calcola aggregazioni anziché trasferire dati grezzi. Grafico volume orario restituisce 24 valori aggregati anziché migliaia di record (payload da MB a KB).
Performance rendering frontend: limite massimo 500 punti per grafico, backend reaggrega se necessario. Paginazione server-side per tabelle garantisce max 100 righe nel DOM.

6.8.4 Limitazioni accesso dati PABX

Problematica ricorrente: limitazione accesso dati centralino. Vendor fornisce solo tabella CDR e alcune API REST, non accesso completo database per sicurezza e supporto.
Funzionalità avanzate richieste impossibili:

- Correlazione con ticket supporto o dati CRM
- Analisi qualità audio (metriche jitter, packet loss, MOS score)
- Correlazione automatica con registrazioni audio e playback
- Tracking dettagliato trasferimenti con tempi per ogni stato

Soluzione pragmatica: concentrarsi su funzionalità realizzabili, implementandole eccellentemente. In alcuni casi, funzionalità semplificate che fornivano comunque valore. Esempio: sistema mostra se chiamata registrata e fornisce identificativo, permettendo recupero manuale da PABX quando necessario.
Limitazioni comunicate trasparentemente agli stakeholder. Quando funzionalità richiesta, analizzati dati necessari, verificata disponibilità, e se non disponibili spiegato chiaramente. Comunicazione proattiva con dimostrazioni concrete: mostrare quali tabelle servirebbero, quali informazioni contengono, perché vendor non le espone.
Funzionalità reingegnerizzate per usare dati disponibili. Esempio: analisi distribuzione geografica chiamate, inizialmente concepita per località precise tramite dati non disponibili, reimplementata usando prefissi telefonici nei CDR, fornendo insight utili su provenienza chiamate.
Esperienza ha evidenziato importanza comunicazione trasparente vincoli tecnici. Quando limitazioni spiegate con esempi concreti e linguaggio comprensibile, correlando richieste a dati necessari e spiegando inaccessibilità, stakeholder hanno compreso, apprezzato onestà, e collaborato a riprioritizzazione. Dialogo aperto ha mantenuto aspettative realistiche, evitato frustrazioni, e spesso stimolato creatività nell'identificare soluzioni alternative fornenti valore comparabile.

Glossario

Bibliografia