Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Agenti AI per web app enterprise: analisi della telemetria e automazione del testing

Tesi di laurea triennale

Relatore
Prof. Francesco Ranzato

Laureando
Alessio Zanella
Matricola 2101079

Anno Accademico 2025-2026

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, presso l'azienda Sia Informatica s.r.l.
Il progetto nasce da due esigenze concrete legate a *Sia.Core*, la piattaforma condivisa da tutti i prodotti aziendali, distribuita su decine di installazioni cliente: da un lato, la necessità di una rete di sicurezza che permetta di evolvere il *framework* senza rompere ciò che già funziona; dall'altro, la necessità di rendere visibile il comportamento dell'applicazione in produzione, dove un malfunzionamento può manifestarsi lontano dalla sua causa reale e con un ritardo difficile da imputare.
La risposta a queste esigenze si articola in due filoni paralleli. Il primo riguarda la **telemetria e l'osservabilità**: è stata progettata un'architettura completa basata su *OpenTelemetry* per la raccolta distribuita di metriche, *trace* e *log*, con *Grafana* come piattaforma unificata di visualizzazione e analisi. Il secondo riguarda il **testing automatico**: è stata introdotta un'infrastruttura di test articolata su *unit test*, *integration test* e test *end-to-end*, volta a garantire la correttezza del *framework* al variare del codice.
Un elemento trasversale a entrambi i filoni è l'utilizzo strutturato dell'intelligenza artificiale come assistente allo sviluppo: agenti AI specializzati per modulo, guidati da *skill* procedurali e connessi direttamente agli strumenti di sviluppo tramite Model Context Protocol (MCP), hanno reso la propagazione di telemetria e test ai nuovi moduli un processo replicabile e omogeneo, riducendo le allucinazioni e aumentando la qualità del codice generato.

"Happiness is your current status minus envy and expectations"

— Jimmy Carr

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Francesco Ranzato, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ringrazio i miei fratelli per aver reso più leggeri e spensierati (non sempre) questi anni, anche al di fuori dell'ambito accademico.

Dedico un grazie anche ai miei colleghi che, con la loro simpatia, le mille battute e le interminabili sfide a calcetto all'ultimo sangue, hanno sempre saputo unire professionalità e simpatia.

Infine, desidero ringraziare i miei amici per tutto il fantastico percorso universitario passato insieme e per le innumerevoli avventure, e talvolta disavventure, che abbiamo condiviso.

Padova, Luglio 2026 Alessio Zanella

Indice

1 Introduzione **1**
- 1.1 L'azienda . . . 1
- 1.2 L'idea . . . 1
- 1.3 Impiego dell'AI durante lo stage . . . 2
- 1.4 Implementazioni realizzate . . . 3
- 1.5 Organizzazione del testo . . . 3

2 Descrizione dello stage **5**
- 2.1 Introduzione al progetto . . . 5
- 2.2 Obiettivi . . . 5
- 2.3 Analisi preventiva dei rischi . . . 6
- 2.4 Pianificazione . . . 8
  - 2.4.1 Discrepanze rispetto alla pianificazione iniziale . . . 8
  - 2.4.2 Pianificazione settimanale . . . 8

3 Analisi dei requisiti **11**
- 3.1 Analisi del sistema . . . 11
- 3.2 Casi d'uso . . . 11
  - 3.2.1 Attori . . . 12
  - 3.2.2 Elenco casi d'uso . . . 12
- 3.3 Requisiti . . . 26
  - 3.3.1 Requisiti funzionali . . . 26
  - 3.3.2 Requisiti non funzionali . . . 27
  - 3.3.3 Tracciamento dei requisiti . . . 27

4 Tecnologie adottate **28**
- 4.1 Studio e scelta delle tecnologie . . . 28
- 4.2 Tecnologie e strumenti . . . 28

5 Progettazione **32**
- 5.1 Osservabilità nei sistemi distribuiti . . . 32
- 5.2 Lo standard OpenTelemetry . . . 33
- 5.3 Architettura della telemetria . . . 33
  - 5.3.1 Produzione dei dati di telemetria . . . 33
  - 5.3.2 Integrazione con OpenTelemetry Collector ed ecosistema Grafana 34
  - 5.3.3 Protocolli e canali di comunicazione . . . 35
  - 5.3.4 Scelte architetturali . . . 36

5.4 Modello dei segnali e utilizzo degli attributi . . . 37
5.4.1 Modello dei log . . . 37
5.4.2 Modello delle trace . . . 37
5.4.3 Modello delle metriche . . . 37
5.4.4 Tassonomia degli attributi . . . 38
5.4.5 Tracciamento dell'identità utente . . . 39
5.4.6 Exemplar: collegamento diretto da metrica a trace . . . 39
5.5 Design pattern applicati . . . 40
5.5.1 Pattern Adapter per il counter di errore dei controller . . . 40
5.5.2 Pattern Factory . . . 41
5.6 Diagrammi delle classi del sistema di telemetria . . . 41
5.7 Implementazione dei test automatici . . . 42
5.7.1 Contesto . . . 42
5.7.2 Strategia di testing adottata . . . 42
5.7.3 Vantaggi ottenuti . . . 45

6 Metodologia assistita da agenti AI 46
6.1 Contesto e motivazione . . . 46
6.2 Ambiti di utilizzo . . . 46
6.2.1 Studio delle tecnologie . . . 46
6.2.2 Prove di implementazione . . . 47
6.2.3 Propagazione della telemetria agli altri moduli . . . 47
6.2.4 Propagazione dei test automatici . . . 47
6.3 Workflow Agenti AI . . . 48
6.4 Confronto pianificazione vs esecuzione . . . 48
6.5 Limiti e mitigazioni . . . 49
6.6 Model Context Protocol . . . 49
6.6.1 Il problema che MCP risolve . . . 49
6.6.2 Dettagli del protocollo . . . 50
6.6.3 Applicazione nel progetto . . . 50
6.7 Considerazioni conclusive . . . 51

7 Codifica 52
7.1 Organizzazione del codice . . . 52
7.2 Pipeline OpenTelemetry . . . 52
7.3 Classi di diagnostica per modulo . . . 53
7.4 Pattern `RecordMetricError` con Adapter . . . 53
7.5 Middleware di arricchimento utente . . . 54
7.6 Configurazione degli exemplars . . . 54
7.7 Log Serilog con sink OTLP . . . 54
7.8 Dashboard Grafana e versionamento . . . 54
7.9 Telemetria delle performance hardware . . . 56
7.9.1 Contesto e obiettivo . . . 56
7.9.2 Scelte progettuali . . . 56
7.9.3 Segnali prodotti . . . 57
7.9.4 Dashboard Grafana . . . 57

8 Conclusioni **59**
8.1 Obiettivi raggiunti e consuntivo finale 59
8.1.1 Integrazione della telemetria in produzione 61
8.2 Conoscenze acquisite e analisi del lavoro svolto 62
8.3 Scenari di applicabilità e sviluppi futuri 64
8.4 Valutazione personale 65

A Tracciamento completo dei requisiti **67**

B Diagrammi delle classi del sistema di telemetria **70**
B.1 Pipeline condiviso in `BaseApplicationBuilder` 70
B.2 Pattern `Sia{Modulo}Diagnostics` e Adapter 72
B.3 Arricchimento del contesto utente 73

C Listati di codice rappresentativi **74**
C.1 Pattern Adapter applicato alla diagnostica del modulo CRUD 74

Acronimi e abbreviazioni **76**

Glossario **77**

Bibliografia **80**

Elenco delle figure

1.1 Logo Sia Informatica 1

3.1 UC1: Generazione assistita di unit test backend 14
3.2 UC2: Generazione assistita di integration test su DB 15
3.3 UC3: Generazione assistita di unit test e E2E frontend 17
3.4 UC4: Esecuzione della suite di test e produzione del report di copertura 18
3.5 UC5: Propagazione della telemetria a un nuovo modulo 19
3.6 UC6: Visualizzazione di metriche, trace e log di un modulo 21
3.7 UC7: Diagnosi di una richiesta lenta tramite exemplar 22
3.8 UC8: Identificazione dell'utente coinvolto in una trace 23
3.9 UC9: Creazione o modifica di una dashboard tramite agente AI e MCP 24

4.1 Logo Angular 29
4.2 Logo .NET 29
4.3 Logo OpenTelemetry 30
4.4 Logo Grafana 30
4.5 Logo xUnit 30
4.6 Logo Playwright 31
4.7 Logo Vitest 31
4.8 Logo Claude 31

5.1 Architettura OTel Collector 35

7.1 Overview e durata richieste della dashboard di *Sia.Base* 55
7.2 Sezione richieste per `${bucket}` della dashboard *Sia.Base* 55
7.3 Sezione trace e log della dashboard di *Sia.Grid* 55
7.4 Sezione risorse della dashboard di telemetria hardware 58

B.1 Diagramma delle classi della Pipeline OpenTelemetry 71
B.2 Diagramma delle classi del pattern di diagnostica 72
B.3 Diagramma delle classi dell'arricchimento trace con dati utente . . . 73

Elenco delle tabelle

7.1 ObservableGauge del modulo `Sia.HealthChecks.Servers` 57

8.1 Endpoint HTTP più lenti osservati in 7 giorni (durata $> 2\,\mathrm{s}$) 61

A.1 Tracciamento dei requisiti funzionali 67
A.2 Tracciamento dei requisiti qualitativi 68
A.3 Tracciamento dei requisiti di vincolo 69

Capitolo 1

Introduzione

Il presente capitolo introduce il contesto aziendale in cui si è svolto lo stage, l'idea da cui il progetto ha preso forma, i contributi originali del lavoro presentato e l'organizzazione del documento.

1.1 L'azienda

Sia Informatica S.r.l. è una società con sede a Vigonza (PD) che, dal 1996, opera nel settore dello sviluppo software e della consulenza informatica per aziende di diversi comparti produttivi. Il *core business* consiste nella progettazione e realizzazione di soluzioni tecnologiche personalizzate, costruite su misura delle esigenze operative dei clienti. L'offerta comprende sistemi Enterprise Resource Planning (ERP) verticali, declinati per i settori del *packaging*, dell'arredamento e della moda con il marchio *Job3 ERP*, servizi web tra cui un Customer Relationship Management (CRM) (*Sales Connect*) e strumenti di gestione logistica, soluzioni di gestione documentale (fatturazione elettronica, firma digitale, conservazione sostitutiva), nonché tecnologie di integrazione avanzata come Radio-Frequency Identification (RFID) e un assistente basato sull'intelligenza artificiale denominato *Kitsune*.
Con oltre vent'anni di esperienza, l'azienda è oggi un partner tecnologico consolidato per le imprese del Nord-Est italiano che cercano soluzioni integrate, scalabili e sostenibili nel tempo.

Figura 1.1: Logo Sia Informatica

1.2 L'idea

Tutti i prodotti aziendali condividono una stessa base tecnologica, *Sia.Core*: un'applicazione *web enterprise* che cresce di funzionalità mese dopo mese, distribuita su

decine di installazioni di clienti diversi. Una piattaforma di queste dimensioni vive di un compromesso delicato: ogni nuova funzionalità deve poter essere introdotta rapidamente, ma senza intaccare la stabilità di ciò che già funziona.
A questo compromesso si aggiunge una difficoltà tipica delle applicazioni distribuite: quando in produzione qualcosa non va come previsto, capire cosa sta accadendo e perché non è affatto immediato. Un malfunzionamento può avere origine in un punto del sistema e manifestarsi in un punto completamente diverso, con un ritardo difficile da imputare. Senza strumenti adatti, lo sviluppatore si trova a indagare a posteriori, con informazioni frammentarie e dipendendo dalla memoria di chi ha installato il software.
L'idea alla base dello stage nasce da queste due esigenze parallele:

- dare al team di sviluppo una rete di sicurezza che permetta di intervenire sul codice del framework con la confidenza che nulla di già funzionante venga rotto, ovvero un sistema di *testing* automatico integrato e ripetibile;
- dare a chi si occupa di assistenza e diagnosi uno strumento che renda visibile il comportamento dell'applicazione in tempo reale, ovvero un sistema di telemetria che osservi in modo sistematico ciò che il software sta facendo e ne registri le prestazioni e le anomalie.

1.3 Impiego dell'AI durante lo stage

L'intelligenza artificiale ha rappresentato uno degli elementi centrali dell'esperienza di stage, venendo impiegata come supporto alle attività di analisi, progettazione, sviluppo e validazione del software. Più che come semplice strumento di generazione automatica di codice, gli agenti AI sono stati utilizzati come veri e propri assistenti allo sviluppo, capaci di accelerare attività ripetitive, facilitare l'esplorazione di basi di codice complesse e supportare l'apprendimento di nuove tecnologie.
L'adozione di questi strumenti si è rivelata particolarmente utile nel contesto di un framework enterprise di grandi dimensioni, caratterizzato da numerosi moduli, convenzioni architetturali consolidate e una vasta base di codice preesistente. In tale scenario, gli agenti AI hanno contribuito sia alla comprensione dell'architettura del sistema sia alla propagazione di modifiche su componenti diversi, riducendo significativamente il tempo necessario per attività altrimenti molto onerose.
Per poter sfruttare al meglio le potenzialità di questo strumento, e per limitare il più possibile allucinazioni e generazione di codice incoerente con il resto della *repository*, sono stati creati ed usati alcuni artefatti, tra i quali:

- nuovi agenti AI specializzati, configurati per operare su specifici moduli del sistema e guidati tramite prompt strutturati e contesto controllato, con convenzioni del *framework* in modo che la propagazione della telemetria e dei test ai vari moduli avvenga in modo uniforme;
- un insieme di *skill* procedurali: piccoli regolamenti scritti per l'agente, per la specializzazione degli agenti in determinati compiti, come: lo sviluppo dei vari segnali di telemetria, la creazione di dashboard per la visualizzazione di dati, la scrittura di test di vario tipo, ecc.;

- l'uso del MCP, uno standard recente che consente all'agente di interrogare direttamente i sistemi coinvolti (*Grafana*, il database, gli strumenti di test) anziché lavorare su assunzioni potenzialmente errate.

1.4 Implementazioni realizzate

Il lavoro svolto durante il periodo di stage ha prodotto diversi contributi originali, sintetizzabili come segue:

- la **progettazione completa di un'architettura di osservabilità** per *Sia.Core*, dalla scelta dello stack (*OpenTelemetry* + *Grafana LGTM*) all'integrazione con l'infrastruttura esistente, passando alla standardizzazione del *pipeline*, in modo che ogni nuovo modulo possa aderirvi senza conoscerne i dettagli, terminando con la presentazione dei dati raccolti in dashboard apposite.
- la definizione di una **tassonomia coerente** di metriche, *trace* e attributi per i moduli centrali del *framework* (autenticazione, singoli componenti *Angular* come griglie e form, prestazioni dell'hardware, interrogazione del database, ecc.), pensata per evitare l'esplosione della cardinalità lato Prometheus e per consentire il confronto omogeneo tra moduli e clienti diversi;
- l'introduzione di due meccanismi per il **miglioramento dell'analisi della telemetria**: l'arricchimento automatico di ogni *trace* con l'identità dell'utente autenticato e l'uso degli *exemplar* per collegare metriche aggregate alla *trace* che le ha prodotte, che rendono possibile una catena diagnostica completa: dal grafico Grafana alla richiesta specifica, passando per l'utente coinvolto e i *log* emessi nel suo contesto;
- l'introduzione, da zero, di un'**infrastruttura di testing automatico** nel *framework* backend, articolata su *unit test* per la logica di business e *integration test* per la verifica dell'attendibilità dei dati presenti nei database, delle procedure SQL utilizzate e delle prospettive presenti per descrivere la struttura dei vari componenti;
- lo sviluppo di **test anche lato frontend**, comprendenti sia *unit test* tradizionali per la verifica della logica applicativa, sia test di validazione *end-to-end*, finalizzati a verificare il corretto comportamento e la corretta visualizzazione dell'applicazione dal punto di vista dell'utente finale;
- come già approfondito nella sezione precedente (1.3), la **creazione di agenti AI e *skill*** per la replicabilità e la propagazione di telemetria e test a contesti di sviluppo analoghi nel *framework*.

1.5 Organizzazione del testo

Il secondo capitolo descrive gli obiettivi del periodo di stage, la pianificazione del lavoro e l'analisi preventiva dei rischi.

Il terzo capitolo approfondisce le funzionalità del prodotto, presentando i requisiti e i casi d'uso individuati.

Il quarto capitolo presenta le tecnologie e gli strumenti adottati durante lo stage.

Il quinto capitolo illustra le scelte progettuali e architetturali del sistema di telemetria, con rinvio in appendice ai dettagli tecnici di ampia dimensione.

Il sesto capitolo descrive in dettaglio la metodologia di lavoro AI-assistita, le *skill* sviluppate, il MCP e le sue applicazioni pratiche al progetto.

Il settimo capitolo espone gli aspetti implementativi salienti, mantenendo nel corpo principale solo le descrizioni di alto livello e relegando in appendice i frammenti di codice e i dettagli più estesi.

L'ottavo capitolo riassume i risultati raggiunti, le conoscenze acquisite e gli sviluppi futuri più promettenti.

Riguardo alla stesura del testo, sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune sono definiti nel glossario, situato alla fine del presente documento;
- alla prima occorrenza di un termine riportato nel glossario è utilizzata la nomenclatura "parola (abbreviazione)", mentre nelle occorrenze successive si utilizza solamente l'abbreviazione;
- i termini in lingua straniera o appartenenti al gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

Il presente capitolo descrive gli obiettivi del progetto di stage, la pianificazione delle attività in termini di ore di lavoro e l'analisi preventiva dei principali rischi che potrebbero insorgere nel corso del suo svolgimento.

2.1 Introduzione al progetto

Il progetto di stage si è sviluppato lungo due linee di lavoro parallele, entrambe rivolte al *framework Sia.Core*, base tecnologica condivisa da tutti i prodotti aziendali. La prima linea ha riguardato la progettazione e l'implementazione di un sistema di telemetria completo, basato sullo standard OpenTelemetry (OTel) e sullo stack *Grafana LGTM*, in grado di raccogliere e correlare *trace*,*metriche* e *log* per offrire una visione unificata del comportamento delle applicazioni in produzione. La seconda linea ha riguardato l'introduzione di un'infrastruttura di *testing* automatico nel *framework*, articolata su *unit test* e *integration test* lato *server* e su *unit test* e test *end-to-end* lato *client*. In contemporanea, è stato strutturato un agente di intelligenza artificiale per l'assistenza allo sviluppo, con la creazione di agenti specializzati, di *skill* procedurali e l'integrazione del MCP per consentire all'agente di interrogare direttamente i sistemi coinvolti (*Grafana*, database, strumenti di test). I fondamenti teorici dell'osservabilità e della metodologia AI-assistita sono trattati in dettaglio rispettivamente nei capitoli 5 e 6.

2.2 Obiettivi

1. Analizzare i requisiti di osservabilità dell'ecosistema *Sia.Core* e identificare le soluzioni tecnologiche adeguate, scegliendo lo stack di riferimento (*OpenTelemetry + Grafana LGTM*).

2. Progettare un'architettura di telemetria basata sullo standard OTel, integrandola con il *framework .NET 9* esistente in modo che ogni nuovo modulo possa aderirvi senza conoscerne i dettagli interni.

3. Definire una tassonomia coerente di metriche, *trace* e attributi per i moduli centrali del *framework* (autenticazione, componenti *Angular* come griglie e

form, prestazioni hardware, accesso al database), evitando l'esplosione della cardinalità lato Prometheus.

4. Definire le strategie di campionamento delle *trace* e di gestione delle metriche in ottica di scalabilità verso ambienti di produzione.
5. Implementare meccanismi di arricchimento automatico delle *trace* con l'identità dell'utente autenticato e di collegamento tra metriche aggregate e *trace* tramite *exemplar*.
6. Configurare lo stack *Grafana LGTM* e produrre dashboard in grado di rendere navigabile la catena diagnostica dal grafico alla singola richiesta.
7. Introdurre da zero un'infrastruttura di *testing* automatico nel *framework backend*, articolata su *unit test* per la logica di business e *integration test* sull'attendibilità dei dati nei database, sulle procedure SQL e sulle prospettive di configurazione dei componenti.
8. Sviluppare test lato *frontend* (*Angular*), comprendenti *unit test* per la logica applicativa e test di validazione *end-to-end* per il comportamento dell'applicazione dal punto di vista dell'utente finale.
9. Realizzare agenti AI specializzati, *skill* procedurali e integrazioni MCP per garantire la replicabilità della propagazione di telemetria e test ai diversi moduli del *framework*.

2.3 Analisi preventiva dei rischi

Durante la fase iniziale di analisi, sono stati individuati i principali rischi che sarebbero potuti emergere nel corso dello stage. Per ciascun rischio identificato, è stata definita una possibile strategia di mitigazione, al fine di ridurne l'impatto e garantire il corretto avanzamento delle attività previste.

1. Tecnologie sconosciute
Descrizione: alcune delle tecnologie previste nel progetto risultano totalmente o parzialmente sconosciute, con il rischio di rallentare le attività nelle fasi iniziali dello stage.
Soluzione: pianificazione di un'attività strutturata di studio e approfondimento, attraverso l'utilizzo di documentazione ufficiale, tutorial online, corsi forniti dall'azienda e lo sviluppo di piccoli progetti di consolidamento, al fine di acquisire rapidamente le competenze necessarie.

2. Gestione del tempo
Descrizione: la pianificazione delle attività potrebbe risultare non accurata, soprattutto a causa della presenza di tecnologie nuove e della componente sperimentale legata all'uso dell'intelligenza artificiale, con il rischio di ritardi rispetto alle scadenze previste.
Soluzione: adottare il modello agile SCRUM per definire obiettivi intermedi settimanali, monitoraggio continuo dello stato di avanzamento e revisione periodica della pianificazione, in modo da adattare le attività in base alle difficoltà riscontrate.

3. Allucinazioni Large Language Model (LLM)
Descrizione: l'agente di codifica *Claude Code*, utilizzato come *pair engineer* per accelerare lo studio delle tecnologie e la propagazione della telemetria ai moduli del framework, può produrre risultati non corretti o non aderenti alle specifiche (fenomeno noto come "allucinazione"), compromettendo l'affidabilità del codice generato.
Soluzione: adozione di tecniche di verifica e validazione dei risultati prodotti, affiancate da un'attenta progettazione delle istruzioni fornite all'agente e dall'utilizzo di specifiche strutturate. Iterazioni successive e revisioni manuali del codice consentono di garantire la correttezza della strumentazione introdotta nei moduli applicativi.

4. Integrazione tra tecnologie
Descrizione: l'integrazione tra *OpenTelemetry*, il framework *.NET 9* di *Sia.Core* e la piattaforma di visualizzazione *Grafana* (con i relativi backend Prometheus, Tempo e Loki) può presentare difficoltà legate a compatibilità, interfacciabilità e correlazione dei segnali, con il rischio di rallentare lo sviluppo o compromettere il corretto flusso dei dati telemetrici.
Soluzione: utilizzo di un approccio incrementale all'integrazione, partendo da configurazioni di base della pipeline OTel e aumentando progressivamente la complessità. Consultazione della documentazione, utilizzo di ambienti di sviluppo locali per la validazione end-to-end dei segnali, e confronto con il team aziendale per la risoluzione di eventuali criticità.

5. Impatto della telemetria sulle prestazioni
Descrizione: La strumentazione capillare dei moduli applicativi e la raccolta di trace, metriche e log potrebbero introdurre un overhead non trascurabile in produzione, degradando i tempi di risposta delle Application Program Interface (API) o generando un volume eccessivo di dati telemetrici.
Soluzione: adozione di strategie di campionamento configurabili per sorgente, attenta progettazione degli attributi di metriche per evitare cardinalità elevata in Prometheus, e possibilità di disattivare la telemetria tramite *appsettings* senza modifiche al codice. Validazione dell'overhead in ambiente di staging prima del rilascio in produzione.

6. Differenze di comportamento tra ambiente di sviluppo e produzione
Descrizione: il sistema di telemetria potrebbe comportarsi in modo differente tra ambiente di sviluppo, *staging* e produzione, a causa di differenze nella configurazione del collector OTel, nei volumi di traffico o nell'infrastruttura. Ciò può portare a malfunzionamenti non rilevati durante le fasi di sviluppo.
Soluzione: utilizzo di ambienti di *staging* il più possibile allineati alla produzione, configurazione differenziata per ambiente tramite *appsettings*, e validazione end-to-end del flusso dall'evento applicativo alla dashboard *Grafana* prima del rilascio.

7. Qualità dei test generati automaticamente
Descrizione: i test generati tramite l'agente di codifica *Claude Code* potrebbero risultare incompleti, non corretti o con copertura insufficiente, compromettendo l'efficacia del processo di *testing*.

Soluzione: introduzione di attività di validazione e revisione dei test generati, utilizzo di metriche di copertura del codice e iterazione sulle istruzioni fornite all'agente per migliorare progressivamente la qualità dei risultati.

2.4 Pianificazione

L'attività di stage è stata pianificata su una durata complessiva di 8 settimane, per un totale di 300 ore.
La pianificazione è stata strutturata in modo progressivo, suddividendo il lavoro in fasi settimanali, ciascuna con obiettivi specifici e attività mirate, al fine di garantire un apprendimento graduale e un'integrazione efficace delle tecnologie coinvolte.

2.4.1 Discrepanze rispetto alla pianificazione iniziale

Il piano di lavoro prevedeva una fase di confronto comparativo tra più modelli di intelligenza artificiale (GPT-4o, Claude, Gemini) e l'adozione di *Spec-Kit* per formalizzare le specifiche affidate all'AI. Nella pratica, *Claude* è stato scelto fin dalle prime settimane sulla base di un'esperienza pregressa e di un'indicazione aziendale, senza condurre il confronto comparativo pianificato; analogamente, *Spec-Kit* non è stato adottato, sostituito nei fatti dalle *skill* procedurali e dagli agenti specializzati, che hanno svolto una funzione analoga nel fissare le convenzioni del *framework*.
La motivazione è stata pragmatica: *Claude* si è dimostrato fin dai primi tentativi in grado di produrre risultati di qualità sufficiente da rendere un confronto sistematico poco conveniente rispetto al monte ore disponibile. Il tempo risparmiato è stato reinvestito nell'estensione della telemetria a più moduli del previsto, nello sviluppo di *skill* e agenti riutilizzabili non pianificati, in una suite di test più ampia e profonda, e nell'introduzione di funzionalità di telemetria avanzate (arricchimento utente, Exemplar) non previste dal piano originale.

2.4.2 Pianificazione settimanale

Settimane 1 e 2: Studio dell'architettura del sistema

Le prime due settimane sono state dedicate all'analisi dell'architettura dell'applicazione Sia.Core, con particolare attenzione alla separazione tra componente *client-side* e *server-side*. In questa fase sono stati studiati la struttura del progetto *Angular* e del backend *.NET 9*, i principali pattern architetturali adottati dal framework e i flussi di comunicazione tra frontend, API e database. L'attività ha incluso anche la revisione del codice esistente insieme al team aziendale e la mappatura preliminare dei componenti più critici candidati all'integrazione di telemetria e test automatici.

Settimana 3: Studio degli strumenti e della metodologia AI-assistita

La terza settimana è stata dedicata all'apprendimento degli strumenti chiave del progetto: lo stack *OpenTelemetry* e *Grafana LGTM*, gli strumenti di *testing* per *.NET* e *Angular*, e la metodologia di lavoro basata su agenti di intelligenza artificiale. Sono stati approfonditi i concetti fondamentali di osservabilità distribuita, come *trace*, metriche e *log*, oltre alla configurazione di un ambiente locale completo basato

sullo stack LGTM. Parallelamente sono stati studiati i framework di testing adottati nel progetto e il modello di lavoro AI-assistito basato su agenti specializzati, *skill* procedurali e integrazioni MCP per l'accesso diretto ai sistemi di monitoraggio e ai database.

Settimana 4: Progettazione dell'architettura di telemetria e strategia di testing

Durante la quarta settimana è stata progettata l'architettura del sistema di telemetria ed è stata definita in parallelo la strategia di *testing* per le due linee di lavoro. L'attività ha riguardato l'identificazione dei moduli del framework da strumentare, la definizione della pipeline OTel e la progettazione di una tassonomia coerente di metriche, *trace* e attributi condivisi. Particolare attenzione è stata dedicata al controllo della cardinalità delle metriche lato Prometheus e alla definizione delle diverse tipologie di test da implementare: *unit test*, *integration test* e test *end-to-end*. In parallelo è stato definito il flusso operativo AI-assistito, stabilendo convenzioni, criteri di validazione e modalità di revisione del codice generato dagli agenti.

Settimana 5: Implementazione della pipeline OTel e prima suite di test

La quinta settimana è stata dedicata all'implementazione del *pipeline OpenTelemetry* nel *framework .NET 9*, alla prima strumentazione dei moduli core e alla realizzazione di una prima suite di test con il supporto degli agenti AI. Sono stati configurati il *pipeline* di raccolta telemetrica, gli *exporter* OpenTelemetry Protocol (OTLP) e la correlazione tra *log* e *trace* tramite `TraceId` e `SpanId`. In questa fase è stato inoltre introdotto il pattern condiviso `SiaXxxDiagnostics` per uniformare l'esposizione delle metriche e delle *trace* nei diversi moduli del framework. Parallelamente sono stati sviluppati i primi *unit test* backend e frontend, successivamente revisionati e validati manualmente.

Settimana 6: Propagazione della telemetria, dashboard e integration test

Nella sesta settimana il lavoro si è concentrato sulla propagazione della telemetria a ulteriori moduli del *framework*, sulla configurazione delle dashboard e sull'introduzione degli *integration test* e dei test *end-to-end* frontend. Sono stati sviluppati strumenti e *skill* procedurali per standardizzare la strumentazione dei moduli applicativi, inclusi componenti *Angular*, API e moduli di accesso ai dati. Contestualmente sono state create dashboard *Grafana* parametrizzabili per servizio e cliente, sfruttando il protocollo MCP per l'interrogazione diretta dello stack di osservabilità. Sul fronte del testing sono stati introdotti *integration test* basati su dati reali provenienti dal database e test *end-to-end* orientati alla validazione dell'esperienza utente.

Settimana 7: Validazione del sistema integrato, tuning delle dashboard e suite di test

La settima settimana ha previsto la validazione complessiva del sistema di telemetria su ambiente di *staging*, l'ottimizzazione delle dashboard e l'esecuzione della suite completa di test. In particolare sono stati verificati il corretto funzionamento del

flusso telemetrico *end-to-end*, l'overhead introdotto dalla telemetria e la cardinalità effettiva delle metriche raccolte in Prometheus. Parallelamente è stata eseguita l'intera suite di test automatizzati, analizzando eventuali fallimenti e migliorando la copertura del codice.

Settimana 8: Messa in produzione e documentazione finale

L'ultima settimana è stata dedicata alla messa in produzione della soluzione di telemetria, al consolidamento della suite di test e alla redazione della documentazione finale. Le attività hanno incluso il *deploy* della configurazione *OpenTelemetry* e delle dashboard *Grafana LGTM*, l'integrazione stabile dei test automatici nel processo di *build* e la produzione della documentazione tecnica relativa all'architettura di osservabilità, alle convenzioni di strumentazione e agli agenti AI sviluppati durante lo stage.

Capitolo 3

Analisi dei requisiti

Il presente capitolo descrive le funzionalità del prodotto, formalizzate sotto forma di requisiti di progetto, e relativi casi d'uso associati.

3.1 Analisi del sistema

Il prodotto sviluppato durante il periodo di stage mette a disposizione dello sviluppatore un sistema che permette la creazione automatizzata di test e il monitoraggio dello stato del sistema *Sia.Core* tramite dashboard.
Dal punto di vista funzionale, la soluzione si compone di due macro-sottosistemi:

- un sottosistema di *testing* automatico, integrato nel *framework Sia.Core* e articolato su *unit test* e *integration test* lato *server* (logica di business, attendibilità dei dati nel database, procedure SQL e prospettive di configurazione) e su *unit test* e test *end-to-end* lato *client* (logica applicativa e validazione dell'esperienza utente);
- un sottosistema di *osservabilità*, basato sullo standard OTel e sullo stack *Grafana LGTM* (*Loki*, *Grafana*, *Tempo*, *Mimir/Prometheus*), che raccoglie *trace*, metriche e *log* prodotti dall'applicazione e ne permette la visualizzazione e la correlazione tramite dashboard.

Trasversalmente ai due sottosistemi, è stato adottato un agente di intelligenza artificiale come assistente allo sviluppo in modo strutturato, con la creazione di agenti specializzati, di *skill* procedurali e l'integrazione del MCP per consentire all'agente di interrogare direttamente *Grafana*, il database e gli strumenti di test. I due sottosistemi sono indipendenti sul piano tecnologico ma complementari sul piano metodologico: la telemetria fornisce il *feedback* che guida il miglioramento iterativo dei test, mentre i test supportano la validazione continua della strumentazione.

3.2 Casi d'uso

La presente sezione illustra i casi d'uso analizzati, corredati dai relativi diagrammi. I diagrammi dei casi d'uso (in inglese *Use Case Diagram*) costituiscono una tipologia

di diagrammi Unified Modeling Language (UML) finalizzati alla rappresentazione delle funzionalità e dei servizi offerti da un sistema, dal punto di vista degli attori che interagiscono con esso.

3.2.1 Attori

Gli attori nei casi d'uso rappresentano entità esterne al sistema, quali utenti o altri servizi, che interagiscono con esso. In merito al sistema sviluppato si possono individuare come attori:

- **Sviluppatore**: è l'attore principale che utilizza il sistema, sia per generare e revisionare i test sia per consultare le dashboard di telemetria durante le attività di diagnosi e *tuning*.
- **Agente AI**: agente di codifica basato su un modello linguistico di grandi dimensioni (LLM), invocato dall'ambiente di sviluppo come *pair engineer*. Opera tramite *skill* procedurali e MCP per generare test, propagare la telemetria ai moduli del *framework* e interrogare *Grafana*, database e strumenti di test.
- **Sia.Core**: applicazione *full-stack* oggetto dell'osservazione e del *testing*, sorgente dei segnali di telemetria e obiettivo della suite di test.

3.2.2 Elenco casi d'uso

- UC1 — Generazione assistita di *unit test* backend tramite agente AI.
  - UC1.1 — Identificazione del componente da coprire.
  - UC1.2 — Invocazione dell'agente AI tramite *skill* dedicata.
  - UC1.3 — Analisi del codice e generazione dei test.
  - UC1.4 — Revisione e integrazione nella suite.
- UC2 — Generazione assistita di *integration test* su database.
  - UC2.1 — Selezione dell'oggetto da validare.
  - UC2.2 — Interrogazione del database e delle prospettive via MCP.
  - UC2.3 — Generazione dell'*integration test*.
  - UC2.4 — Revisione e integrazione nella suite.
- UC3 — Generazione assistita di *unit test* e test *end-to-end frontend*.
  - UC3.1 — Identificazione del componente o del flusso utente da coprire.
  - UC3.2 — Generazione di *unit test* per la logica del componente.
  - UC3.3 — Generazione del test *end-to-end*.
  - UC3.4 — Revisione e integrazione nella suite.
- UC4 — Esecuzione della suite di test e produzione del report di copertura.
  - UC4.1 — Avvio della suite di test.

  - UC4.2 — Esecuzione dei test integrata nel processo di *build*.
  - UC4.3 — Generazione del report di copertura del codice.
  - UC4.4 — Gestione dei test falliti (estensione).

- UC5 — Propagazione della telemetria a un nuovo modulo del *framework* tramite *skill* dedicate.
  - UC5.1 — Identificazione del modulo da strumentare.
  - UC5.2 — Invocazione della *skill* di propagazione della telemetria.
  - UC5.3 — Introduzione della classe di diagnostica e registrazione della sorgente nel *pipeline* OTel.
  - UC5.4 — Definizione di metriche con cardinalità controllata.
  - UC5.5 — Validazione dei segnali sullo stack *Grafana LGTM*.

- UC6 — Visualizzazione delle metriche, *trace* e *log* di un modulo tramite dashboard *Grafana LGTM*.
  - UC6.1 — Apertura della dashboard del modulo.
  - UC6.2 — Selezione delle variabili *template*.
  - UC6.3 — Consultazione correlata di metriche, *trace* e *log*.

- UC7 — Diagnosi di una richiesta lenta a partire da un pannello percentili, tramite *exemplar* verso la *trace* e navigazione ai *log* correlati.
  - UC7.1 — Individuazione del campione anomalo sul pannello percentili.
  - UC7.2 — Navigazione all'*exemplar* associato al campione.
  - UC7.3 — Apertura della *trace* corrispondente in *Tempo*.
  - UC7.4 — Navigazione ai *log* correlati tramite `TraceId`.

- UC8 — Identificazione dell'utente coinvolto in una *trace* tramite l'arricchimento automatico degli *span*.
  - UC8.1 — Apertura della *trace* di interesse.
  - UC8.2 — Lettura degli attributi `enduser.id` e `enduser.name` dello *span* radice.
  - UC8.3 — Filtro di ulteriori *trace* e *log* per lo stesso utente.

- UC9 — Creazione o modifica di una dashboard *Grafana* tramite agente AI e MCP.
  - UC9.1 — Specifica delle esigenze di *monitoring* all'agente AI.
  - UC9.2 — Validazione delle query PromQL/TraceQL/LogQL via MCP.
  - UC9.3 — Pubblicazione o aggiornamento della dashboard sull'istanza *Grafana*.
  - UC9.4 — Gestione di una query non valida (estensione).

UC1: Generazione assistita di unit test backend

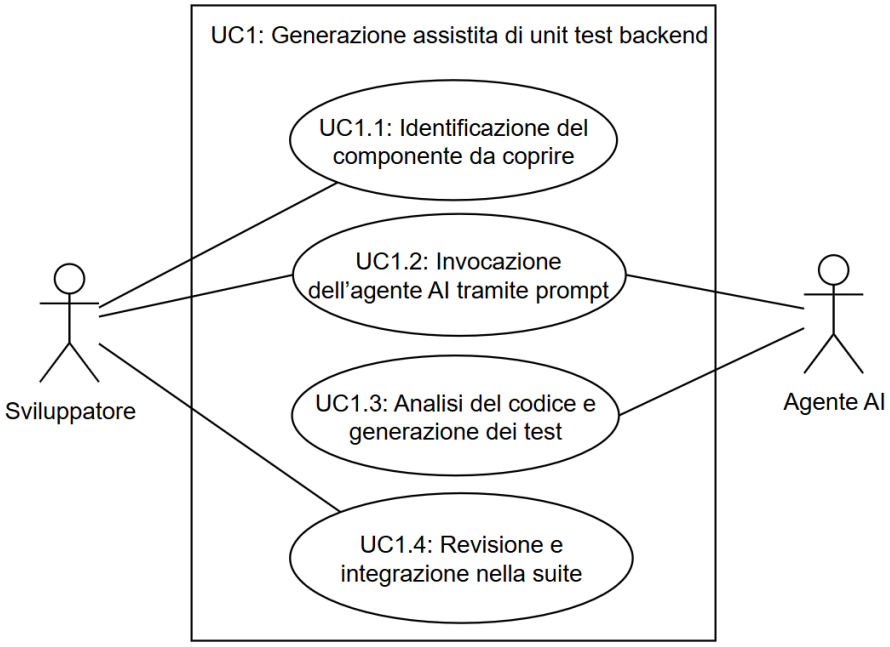

Figura 3.1: UC1: Generazione assistita di unit test backend

Attori Principali: Sviluppatore, Agente AI.

Precondizioni: L'ambiente di sviluppo del *framework Sia.Core* è aperto, l'agente AI è configurato con la *skill* dedicata agli *unit test backend* e nel progetto è presente il *test runner .NET*.

Descrizione: Lo sviluppatore vuole introdurre *unit test* su un servizio o un *repository* del *framework backend* privo di copertura adeguata, avvalendosi dell'agente AI.

Postcondizioni: Nuovi *unit test* sono presenti nel progetto, eseguibili nella *build*, e contribuiscono all'aumento della copertura del codice.

Scenario Principale:

1. Lo sviluppatore identifica il componente da coprire (UC1.1);
2. Lo sviluppatore invoca l'agente AI tramite *prompt* (UC1.2);
3. L'agente analizza il codice del componente e genera gli *unit test* (UC1.3);
4. Lo sviluppatore revisiona i test e li integra nella suite (UC1.4).

UC1.1: Identificazione del componente da coprire

Descrizione: Lo sviluppatore individua un servizio o un *repository* privo di copertura adeguata, anche basandosi sul report di copertura prodotto da una precedente esecuzione della suite.

UC1.2: Invocazione dell'agente AI tramite prompt

Descrizione: Lo sviluppatore richiama l'agente AI passando il riferimento al componente, l'agente riconosce il contesto del *prompt* e recupera la *skill* di generazione *unit test*.

UC1.3: Analisi del codice e generazione dei test

Descrizione: L'agente analizza il codice del componente, ne deduce i comportamenti attesi e i casi limite, e produce *unit test* che esercitano la logica di business.

UC1.4: Revisione e integrazione nella suite

Descrizione: Lo sviluppatore valuta la correttezza e la pertinenza dei test, li adatta se necessario e li integra nella suite del progetto.

UC2: Generazione assistita di integration test su DB

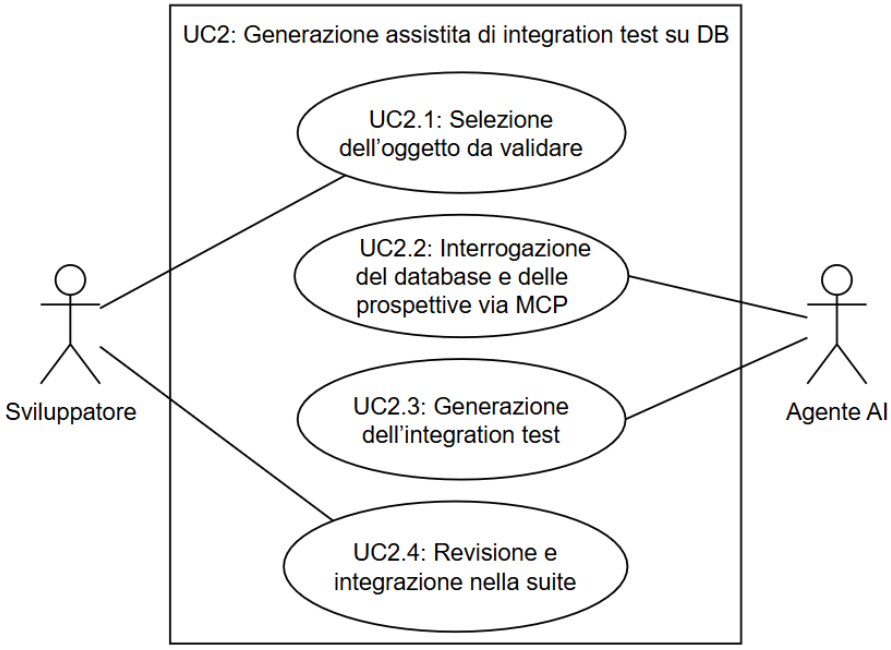

Figura 3.2: UC2: Generazione assistita di integration test su DB

Attori Principali: Sviluppatore, Agente AI.

Precondizioni: Il MCP è configurato verso il database del *framework* e l'agente AI dispone delle *skill* dedicate agli *integration test*.

Descrizione: Lo sviluppatore vuole verificare l'attendibilità dei dati nel database, il corretto comportamento di una procedura SQL o la coerenza di una prospettiva di configurazione di un componente, sfruttando l'agente AI e l'accesso diretto via MCP.

Postcondizioni: La suite di *integration test* verifica in modo automatizzato la consistenza dell'oggetto selezionato.

Scenario Principale:

1. Lo sviluppatore seleziona l'oggetto da validare (UC2.1);
2. L'agente AI ispeziona via MCP schema, dati e procedure coinvolte (UC2.2);
3. L'agente genera l'*integration test* (UC2.3);
4. Lo sviluppatore revisiona e integra i test nella suite (UC2.4).

UC2.1: Selezione dell'oggetto da validare

Descrizione: Lo sviluppatore identifica l'oggetto target (tabella, vista, *stored procedure*, prospettiva) e il criterio di validità che deve essere verificato.

UC2.2: Interrogazione del database e delle prospettive via MCP

Descrizione: L'agente AI interroga il database e il filesystem delle prospettive per ricostruire schema, dati e dipendenze dell'oggetto target.

UC2.3: Generazione dell'integration test

Descrizione: L'agente produce un *integration test* che esegue le operazioni necessarie e ne verifica la conformità al criterio di validità formalizzato.

UC2.4: Revisione e integrazione nella suite

Descrizione: Lo sviluppatore valuta il test, lo adatta al contesto specifico e lo integra nella suite.

UC3: Generazione assistita di unit test e E2E frontend

Attori Principali: Sviluppatore, Agente AI.
Precondizioni: L'ambiente di sviluppo *Angular* del *framework* è aperto, l'agente AI dispone delle *skill* dedicate al *testing frontend* e gli strumenti di *unit test* e di test *end-to-end* sono installati.
Descrizione: Lo sviluppatore vuole introdurre *unit test* sulla logica di un componente *Angular* e/o test *end-to-end* su un flusso utente, avvalendosi dell'agente AI.
Postcondizioni: Sono disponibili *unit test* e test *end-to-end* nella suite del *frontend*.
Scenario Principale:

1. Lo sviluppatore identifica il componente o il flusso utente (UC3.1);
2. L'agente genera gli *unit test* per la logica del componente (UC3.2);
3. L'agente genera il test *end-to-end* corrispondente (UC3.3);
4. Lo sviluppatore revisiona e integra i test nella suite (UC3.4).

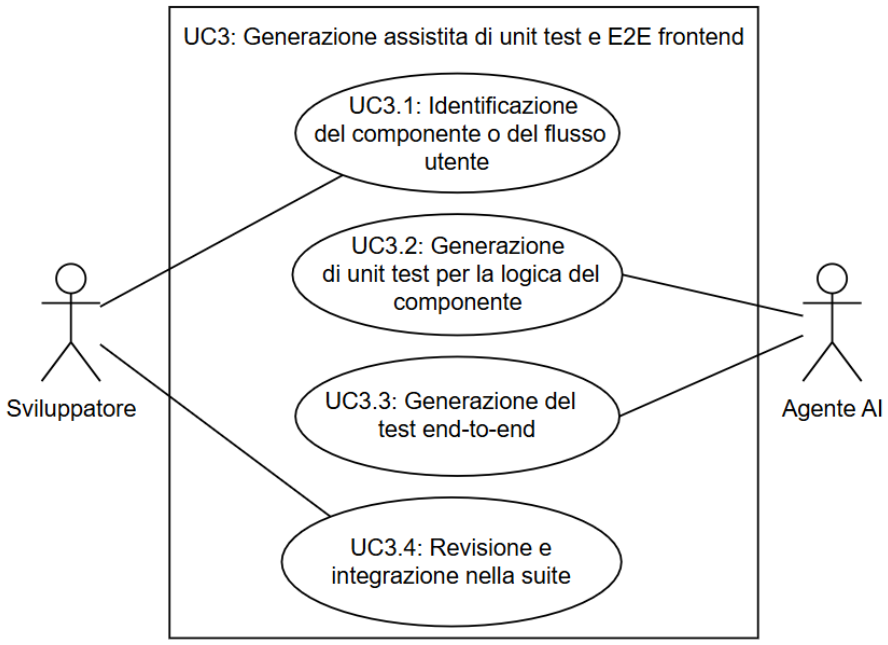

Figura 3.3: UC3: Generazione assistita di unit test e E2E frontend

UC3.1: Identificazione del componente o del flusso utente

Descrizione: Lo sviluppatore individua un componente da coprire con *unit test* e/o un flusso utente da coprire con un test *end-to-end*.

UC3.2: Generazione di unit test per la logica del componente

Descrizione: L'agente analizza la logica del componente *Angular*, dei servizi iniettati e dei *template binding*, e genera *unit test* con i *mock* delle dipendenze.

UC3.3: Generazione del test end-to-end

Descrizione: L'agente genera un test *end-to-end* che simula l'interazione dell'utente sull'applicazione, verificando le transizioni di stato e gli elementi attesi nelle varie schermate.

UC3.4: Revisione e integrazione nella suite

Descrizione: Lo sviluppatore esegue i test in locale, ne valuta l'esito e li integra nella suite *frontend*.

UC4: Esecuzione della suite di test e produzione del report di copertura

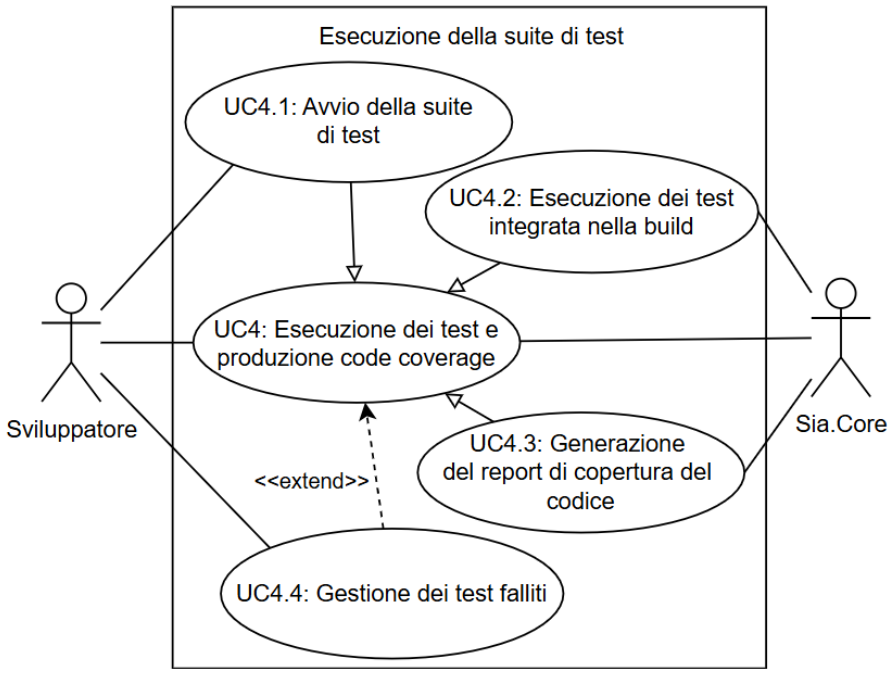

Figura 3.4: UC4: Esecuzione della suite di test e produzione del report di copertura

Attori Principali: Sviluppatore, Sia.Core.

Precondizioni: La suite di test (*unit*, *integration* ed *end-to-end*) è integrata nel processo di *build*.

Descrizione: Lo sviluppatore lancia l'esecuzione completa della suite per ottenere un esito complessivo e un report di copertura aggiornato del codice.

Postcondizioni: Lo sviluppatore dispone di un esito complessivo della suite e di un report di copertura.

Scenario Principale:

1. Lo sviluppatore avvia la suite di test (UC4.1);
2. Il sistema esegue tutti i test integrati nella *build* (UC4.2);
3. Il sistema produce il report di copertura del codice (UC4.3).

Estensioni:

1. In presenza di test falliti, lo sviluppatore avvia la procedura di gestione (UC4.4).

UC4.1: Avvio della suite di test

Descrizione: Lo sviluppatore lancia il comando o l'azione di avvio della suite.

UC4.2: Esecuzione dei test integrata nella build

Descrizione: Il sistema esegue in sequenza *unit test*, *integration test* ed *end-to-end test* riportando l'esito di ciascuno.

UC4.3: Generazione del report di copertura del codice

Descrizione: Il sistema raccoglie i dati di copertura dei test e produce un report consultabile dallo sviluppatore.

UC4.4: Gestione dei test falliti

Descrizione: Lo sviluppatore consulta l'esito del test fallito, individua la causa nel codice sorgente e procede alla correzione del codice o del test.

UC5: Propagazione della telemetria a un nuovo modulo

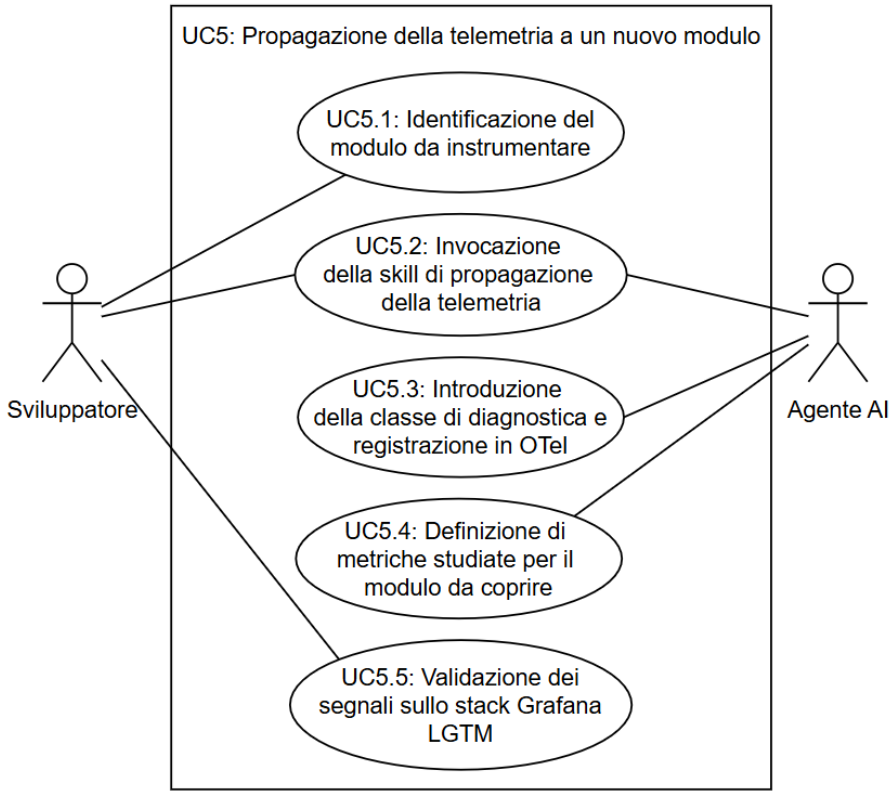

Figura 3.5: UC5: Propagazione della telemetria a un nuovo modulo

Attori Principali: Sviluppatore, Agente AI.

Precondizioni: Esiste un modulo del *framework* non ancora coperto da telemetria, sono disponibili le *skill* per la propagazione di *trace*, metriche e *log*, e lo stack *Grafana LGTM* è raggiungibile.

Descrizione: Lo sviluppatore vuole estendere la copertura della telemetria a un nuovo modulo del *framework*, avvalendosi delle *skill* dedicate dell'agente AI per garantire l'aderenza alle convenzioni.

Postcondizioni: Il modulo emette segnali di telemetria conformi alle convenzioni del *framework* e visibili nello stack *Grafana LGTM*.

Scenario Principale:

1. Lo sviluppatore identifica il modulo da strumentare (UC5.1);
2. Lo sviluppatore invoca la *skill* di propagazione della telemetria (UC5.2);
3. L'agente introduce il pattern `SiaXxxDiagnostics` e registra la sorgente nel *pipeline* OTel (UC5.3);
4. L'agente definisce metriche studiate per il modulo da coprire (UC5.4);
5. Lo sviluppatore valida i segnali sullo stack *Grafana LGTM* (UC5.5).

UC5.1: Identificazione del modulo da strumentare

Descrizione: Lo sviluppatore individua un modulo non ancora coperto e ne valuta gli aspetti rilevanti per la telemetria (operazioni critiche, dipendenze esterne, prestazioni).

UC5.2: Invocazione della skill di propagazione della telemetria

Descrizione: Lo sviluppatore richiama l'agente AI passando il riferimento al componente, l'agente riconosce il contesto del *prompt* e recupera la *skill* di propagazione della telemetria.

UC5.3: Introduzione della classe di diagnostica e registrazione in OTel

Descrizione: L'agente introduce nel modulo il pattern `SiaXxxDiagnostics`, definisce `ActivitySource` e `Meter` e registra la nuova sorgente nel *pipeline* OTel del *framework*.

UC5.4: Definizione di metriche studiate per il modulo da coprire

Descrizione: L'agente definisce le metriche del modulo applicando le convenzioni del *framework* sui tag, in modo da evitare l'esplosione della cardinalità lato Prometheus.

UC5.5: Validazione dei segnali sullo stack Grafana LGTM

Descrizione: Lo sviluppatore esegue il modulo in sviluppo e verifica che *trace*, metriche e *log* compaiano correttamente in *Tempo*, *Prometheus* e *Loki*.

UC6: Visualizzazione di metriche, trace e log di un modulo

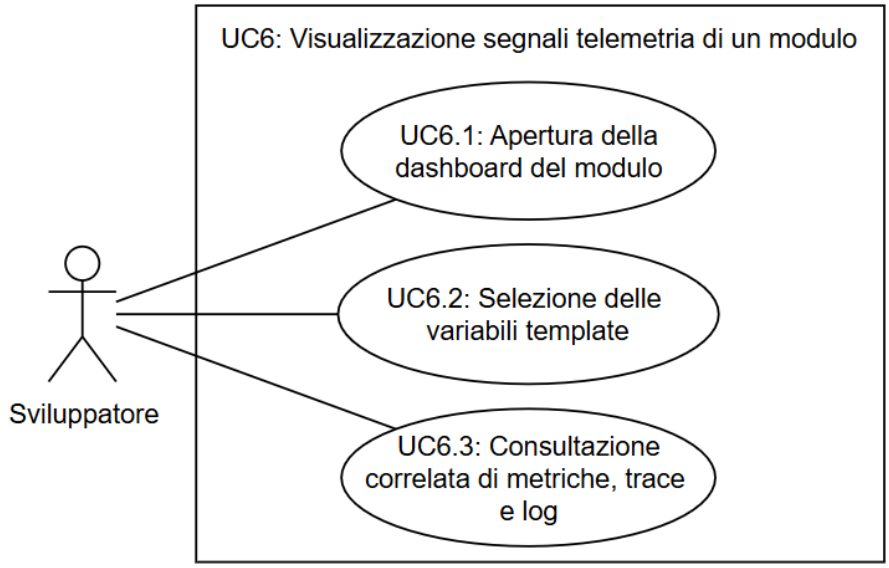

Figura 3.6: UC6: Visualizzazione di metriche, trace e log di un modulo

Attori Principali: Sviluppatore.

Precondizioni: Lo stack *Grafana LGTM* è raggiungibile, il modulo di interesse è strumentato e produce segnali, e la dashboard del modulo è disponibile.

Descrizione: Lo sviluppatore consulta in modo correlato metriche, *trace* e *log* di un modulo per ottenerne una visione unificata in un periodo di tempo.

Postcondizioni: Lo sviluppatore dispone di una visione unificata del comportamento del modulo nel periodo selezionato.

Scenario Principale:

1. Lo sviluppatore apre la dashboard del modulo (UC6.1);
2. Lo sviluppatore seleziona servizio, installazione e cliente tramite le variabili *template* (UC6.2);
3. Lo sviluppatore consulta in modo correlato metriche, *trace* e *log* (UC6.3).

UC6.1: Apertura della dashboard del modulo

Descrizione: Lo sviluppatore apre la dashboard associata al modulo dalla cartella di organizzazione del *framework*.

UC6.2: Selezione delle variabili template

Descrizione: Lo sviluppatore seleziona dai *dropdown* delle variabili *template* servizio, installazione e cliente per circoscrivere la visualizzazione.

UC6.3: Consultazione correlata di metriche, trace e log

Descrizione: Lo sviluppatore osserva i pannelli di metriche, *trace* e *log* relativi al modulo nel periodo selezionato, sfruttando i collegamenti reciproci messi a disposizione dalla dashboard.

UC7: Diagnosi di una richiesta lenta tramite exemplar

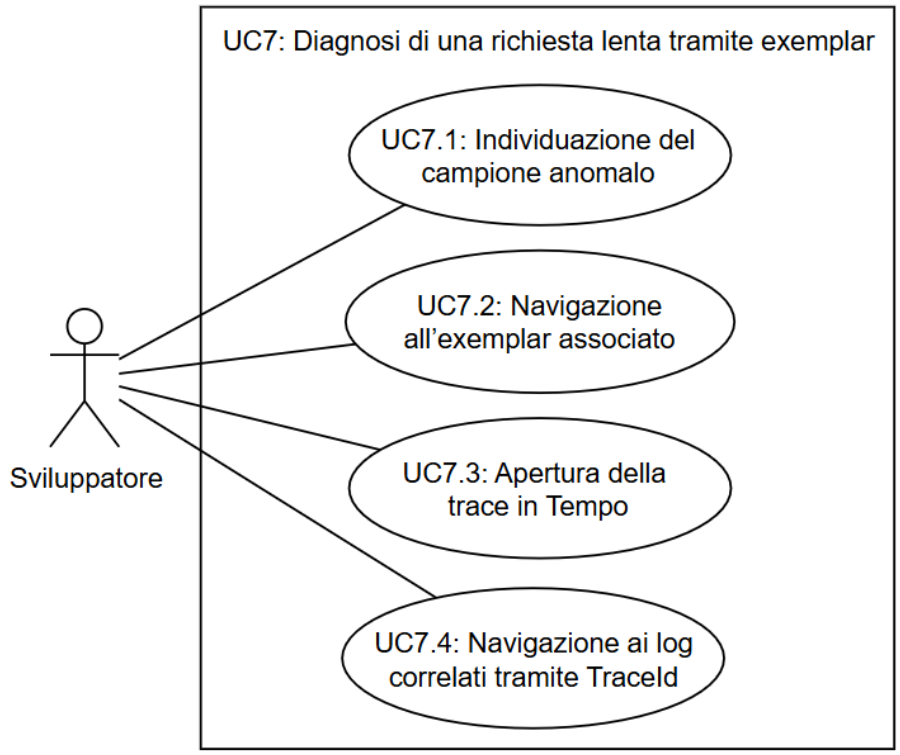

Figura 3.7: UC7: Diagnosi di una richiesta lenta tramite exemplar

Attori Principali: Sviluppatore.

Precondizioni: Una dashboard *Grafana* mostra un pannello percentili (es. p95/p99) di una metrica di tipo *Histogram* con *exemplar* abilitati e i *log* sono indicizzati in *Loki* con `TraceId`.

Descrizione: Lo sviluppatore vuole ricostruire la catena diagnostica completa di una richiesta lenta a partire da un campione anomalo sul pannello percentili.

Postcondizioni: Lo sviluppatore ha ricostruito la catena diagnostica della singola richiesta lenta, dall'aggregato al *log* di dettaglio.

Scenario Principale:

1. Lo sviluppatore individua il campione anomalo sul pannello percentili (UC7.1);
2. Lo sviluppatore naviga all'*exemplar* associato (UC7.2);
3. Lo sviluppatore apre la *trace* corrispondente in *Tempo* (UC7.3);
4. Lo sviluppatore naviga ai *log* correlati tramite `TraceId` (UC7.4).

UC7.1: Individuazione del campione anomalo

Descrizione: Lo sviluppatore identifica il punto temporale e il valore del campione anomalo sul pannello.

UC7.2: Navigazione all'exemplar associato

Descrizione: Lo sviluppatore clicca sull'*exemplar* associato al campione, rappresentato come marker sul pannello.

UC7.3: Apertura della trace in Tempo

Descrizione: Lo sviluppatore apre la *trace* in *Tempo*, esamina lo *span* radice e i suoi figli, individuando il punto in cui si concentra la latenza.

UC7.4: Navigazione ai log correlati tramite TraceId

Descrizione: Lo sviluppatore segue il link verso *Loki*, filtrato per `TraceId`, e consulta i *log* emessi nel contesto della richiesta.

UC8: Identificazione dell'utente coinvolto in una trace

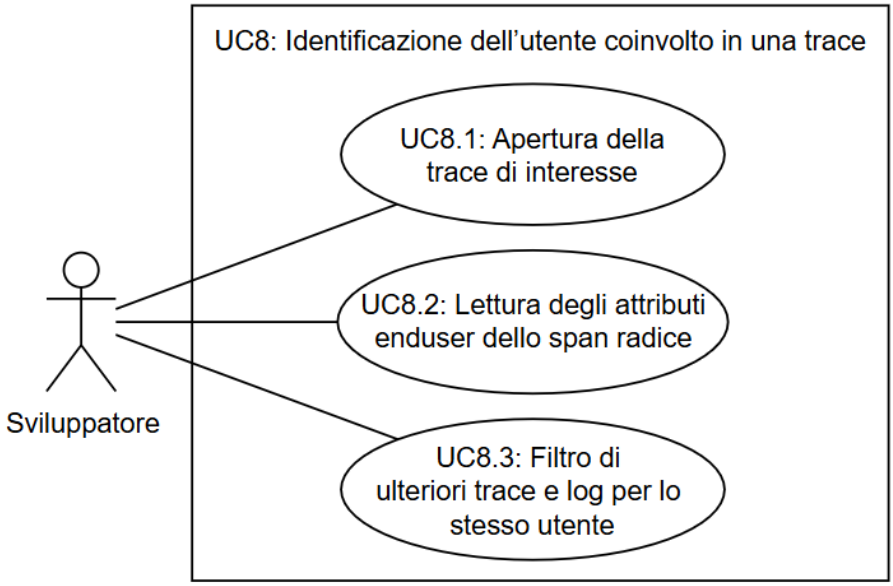

Figura 3.8: UC8: Identificazione dell'utente coinvolto in una trace

Attori Principali: Sviluppatore.

Precondizioni: La telemetria del *framework* arricchisce automaticamente ogni *span* con gli attributi `enduser.id` e `enduser.name`, e *log* e *trace* sono indicizzati su tali attributi.

Descrizione: Lo sviluppatore vuole risalire all'utente che ha originato una richiesta tracciata e analizzare il suo intero contesto.

Postcondizioni: Lo sviluppatore conosce l'identità dell'utente coinvolto e può proseguire l'indagine su tutto il suo contesto.

Scenario Principale:

1. Lo sviluppatore apre la *trace* di interesse (UC8.1);
2. Lo sviluppatore legge gli attributi `enduser.id` e `enduser.name` dello *span* radice (UC8.2);
3. Lo sviluppatore filtra ulteriori *trace* e *log* per lo stesso utente (UC8.3).

UC8.1: Apertura della trace di interesse

Descrizione: Lo sviluppatore apre la *trace* in *Tempo*.

UC8.2: Lettura degli attributi enduser dello span radice

Descrizione: Lo sviluppatore esamina gli attributi dello *span* radice e legge `enduser.id` e `enduser.name`.

UC8.3: Filtro di ulteriori trace e log per lo stesso utente

Descrizione: Lo sviluppatore esegue ricerche in *Tempo* e *Loki* filtrando per `enduser.id`, ottenendo l'intero contesto delle attività dell'utente.

UC9: Creazione o modifica di una dashboard tramite agente AI e MCP

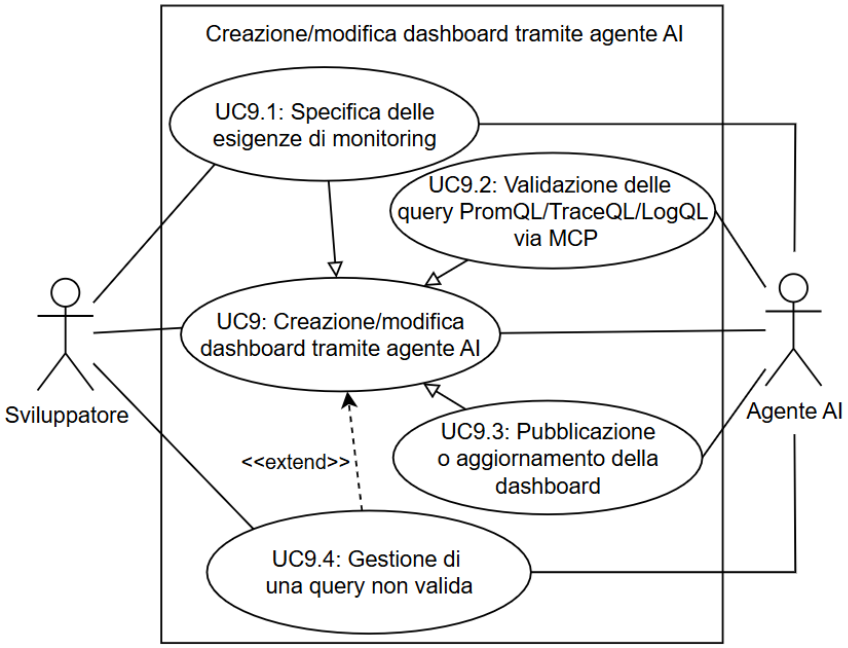

Figura 3.9: UC9: Creazione o modifica di una dashboard tramite agente AI e MCP

Attori Principali: Sviluppatore, Agente AI.

Precondizioni: Lo stack *Grafana LGTM* è raggiungibile e l'agente AI dispone delle credenziali MCP verso *Grafana* e verso le sorgenti dati.

Descrizione: Lo sviluppatore vuole creare o modificare una dashboard delegando all'agente AI sia la formulazione delle query sia la pubblicazione finale.

Postcondizioni: La dashboard è disponibile in *Grafana* con i pannelli e le query richiesti, validati sui dati reali.

Scenario Principale:

1. Lo sviluppatore specifica all'agente le esigenze di *monitoring* (UC9.1);
2. L'agente formula e valida le query via MCP (UC9.2);
3. L'agente pubblica o aggiorna la dashboard sull'istanza *Grafana* (UC9.3).

Estensioni:

1. In presenza di una query non valida, l'agente avvia la procedura di gestione (UC9.4).

UC9.1: Specifica delle esigenze di monitoring

Descrizione: Lo sviluppatore descrive in linguaggio naturale i pannelli desiderati, le grandezze da osservare e i filtri di interesse.

UC9.2: Validazione delle query PromQL/TraceQL/LogQL via MCP

Descrizione: L'agente costruisce le query PromQL/TraceQL/LogQL necessarie e le esegue via MCP contro le sorgenti dati per verificarne la sintassi e la pertinenza dei risultati.

UC9.3: Pubblicazione o aggiornamento della dashboard

Descrizione: L'agente, tramite MCP, crea o aggiorna la dashboard direttamente sull'istanza *Grafana*, applicando convenzioni di naming e di organizzazione in cartelle.

UC9.4: Gestione di una query non valida

Descrizione: L'agente analizza l'errore o l'anomalia, riformula la query e ritenta; in caso di esito negativo dopo più tentativi, riporta il problema allo sviluppatore richiedendo chiarimenti sulla specifica.

3.3 Requisiti

3.3.1 Requisiti funzionali

RF1. Il sistema deve tracciare il percorso completo di ogni richiesta HTTP in ingresso, incluse le operazioni di accesso al database.

RF2. Le operazioni Create, Read, Update, Delete (CRUD) verso il database devono essere riconoscibili tramite attributi che espongono il metodo e la *repository* da cui è stata scatenata la richiesta.

RF3. Il sistema deve esporre metriche operative per i moduli centrali del *framework* (autenticazione, componenti *Angular* come griglie e *form*, prestazioni hardware, accesso al database).

RF4. I *log* devono essere correlabili alle *trace* tramite `TraceId` e `SpanId`.

RF5. Il sistema deve essere attivabile e disattivabile tramite configurazione, senza richiedere modifiche al codice.

RF6. Deve essere possibile configurare indipendentemente il campionamento per ciascuna sorgente di *trace*.

RF7. Deve essere possibile visualizzare *trace* e metriche raggruppate in dashboard *Grafana* costruite sullo stack *LGTM*.

RF8. Le dashboard di *Grafana* devono raccogliere i dati da più servizi e installazioni di clienti diversi, e permettere di filtrare per gli stessi.

RF9. Il sistema deve arricchire automaticamente ogni *span* con l'identità dell'utente autenticato (`enduser.id`, `enduser.name`) senza richiedere modifiche ai singoli moduli applicativi.

RF10. Le metriche di tipo *Histogram* devono esporre *exemplar*, ovvero puntatori al `TraceId` del campione, per abilitare la navigazione dalla metrica aggregata alla *trace* corrispondente.

RF11. Devono essere prodotti *unit test* per la logica di business dei moduli core del *framework backend*, eseguibili in modo automatizzato.

RF12. Devono essere prodotti *integration test* per la verifica dell'attendibilità dei dati presenti nel database, delle procedure SQL e delle prospettive di configurazione dei componenti.

RF13. Devono essere prodotti *unit test* lato *frontend* per la verifica della logica applicativa dei componenti *Angular*.

RF14. Devono essere prodotti test *end-to-end* lato *frontend* per verificare il corretto comportamento e la corretta visualizzazione dell'applicazione dal punto di vista dell'utente finale.

RF15. La suite di test deve essere integrabile nel processo di *build* e fornire un report di copertura del codice.

RF16. Devono essere realizzati agenti AI specializzati e *skill* procedurali per propagare in modo uniforme telemetria e test ai diversi moduli del *framework*.

3.3.2 Requisiti non funzionali

RNF1. La telemetria non deve degradare sensibilmente le prestazioni in produzione.

RNF2. Le metriche devono essere progettate per non generare un numero eccessivo di serie temporali in Prometheus; in particolare, i tag ad alta variabilità (es. identificativi utente o di record) non devono essere usati come attributi di metriche.

RNF3. I segnali prodotti devono essere compatibili con qualsiasi backend OTel-compatibile.

RNF4. Le impostazioni devono essere modificabili per ambiente (sviluppo, *staging*, produzione) tramite *appsettings*.

3.3.3 Tracciamento dei requisiti

Dall'analisi dei requisiti e degli *use case* è stata sviluppata una tabella di tracciamento che pone in relazione ciascun requisito con i casi d'uso che ne dipendono. Per distinguere le diverse classi di requisito si è adottato un codice identificativo nella forma R(F/Q/V)(O/D), dove:

- la prima lettera (sempre R) indica che si tratta di un requisito;
- la seconda lettera ne indica la natura: F (funzionale), Q (qualitativo) o V (di vincolo);
- la terza lettera ne indica la priorità: O (obbligatorio) o D (desiderabile).

Per non appesantire il corpo del capitolo, le tabelle complete del tracciamento sono raccolte nell'appendice A, alle quali si rimanda i lettori interessati al dettaglio.

Capitolo 4

Tecnologie adottate

Il presente capitolo descrive le tecnologie adottate per lo sviluppo del prodotto e ne illustra le principali caratteristiche.

4.1 Studio e scelta delle tecnologie

Una parte fondamentale dello stage è stata dedicata all'attività di ricerca, analisi e selezione delle tecnologie più adeguate per lo sviluppo del progetto. Questa fase iniziale ha rivestito un ruolo cruciale, in quanto le scelte tecnologiche influenzano in modo significativo sia la qualità del prodotto finale sia la sostenibilità dello sviluppo nel lungo periodo. Il processo di selezione è iniziato con una prima fase di scrematura delle soluzioni disponibili sul mercato, effettuata sulla base dei requisiti e delle esigenze espresse dall'azienda. In particolare, sono stati considerati diversi criteri di valutazione, tra cui le funzionalità offerte dalle tecnologie, i costi associati al loro utilizzo e il grado di integrazione con il sistema già esistente. Successivamente, le tecnologie ritenute più promettenti sono state sottoposte a una fase di sperimentazione pratica attraverso lo sviluppo di Proof of Concept (PoC). Questa attività ha permesso di valutare concretamente le prestazioni, l'efficacia e la compatibilità delle soluzioni selezionate, fornendo elementi utili per una scelta consapevole e motivata delle tecnologie da adottare nel progetto.

4.2 Tecnologie e strumenti

Di seguito è fornita una descrizione generale delle tecnologie e degli strumenti utilizzati per lo sviluppo del progetto.

Angular

Framework open-source per lo sviluppo di applicazioni *web front-end*, basato su *TypeScript* e mantenuto da *Google*.

- **Architettura a componenti**: *Angular* promuove una struttura modulare basata su componenti riutilizzabili, facilitando la manutenzione e la scalabilità dell'applicazione.

- **Strumenti integrati**: offre soluzioni native per *routing*, gestione dello stato, form e comunicazione con API, riducendo la necessità di librerie esterne.
- **Supporto alle SPA**: è progettato per lo sviluppo di Single Page Application (SPA), garantendo un'esperienza utente fluida e dinamica grazie all'aggiornamento asincrono dei contenuti.

Figura 4.1: Logo Angular

.NET

Piattaforma di sviluppo *software open-source* e multipiattaforma sviluppata da *Microsoft*, utilizzata per la creazione di applicazioni web, desktop, mobile e cloud.

- **Elevate prestazioni e scalabilità**: .NET garantisce alte performance e una gestione efficiente delle risorse, risultando adatto ad applicazioni *enterprise* e ad alto carico.
- **Architettura strutturata e flessibile**: supporta *pattern* architetturali come *Clean Architecture* e *Dependency Injection*, favorendo la manutenibilità e l'organizzazione del codice.
- **Ampio ecosistema e integrazione**: mette a disposizione numerose librerie e strumenti per lo sviluppo di API, gestione dei dati, sicurezza e integrazione con servizi esterni.

Figura 4.2: Logo .NET

OpenTelemetry

Insieme di strumenti, API e librerie *open-source* per la raccolta, generazione ed esportazione di dati di telemetria, quali metriche, *log* e *trace*, provenienti da applicazioni software. È utilizzato per implementare sistemi di osservabilità distribuita, permettendo il monitoraggio del comportamento e delle prestazioni di un sistema in tempo reale.

Figura 4.3: Logo OpenTelemetry

Grafana

Piattaforma *open-source* per la visualizzazione e l'analisi di dati, utilizzata principalmente per il monitoraggio di sistemi e applicazioni. Consente di creare dashboard interattive a partire da diverse fonti di dati (come *OpenTelemetry*), facilitando l'interpretazione delle metriche e l'individuazione di anomalie o criticità. Nel progetto è stato adottato lo stack *LGTM* (Loki per i *log*, Grafana per la visualizzazione, Tempo per le *trace*, Prometheus/Mimir per le *metriche*).

Figura 4.4: Logo Grafana

xUnit

Framework di testing unitario *open-source* per applicazioni .NET, ampiamente adottato nell'ecosistema Microsoft per la scrittura di test unitari e di integrazione. Supporta pattern come: *data-driven test*, l'esecuzione parallela dei test e l'integrazione con librerie complementari quali *NSubstitute* per il mocking e *FluentAssertions* per asserzioni espressive.

Figura 4.5: Logo xUnit

Playwright

Framework *open-source* per l'automazione dei browser e il testing end-to-end di applicazioni web. Permette di simulare il comportamento reale degli utenti attraverso interazioni come navigazione, compilazione di form e verifica dei contenuti, supportando diversi browser (Chromium, Firefox, WebKit).

Figura 4.6: Logo Playwright

Vitest

Framework di testing unitario moderno per applicazioni JavaScript e TypeScript, progettato per integrarsi con strumenti di sviluppo basati su Vite. Offre esecuzione rapida dei test, e funzionalità avanzate come mocking, snapshot testing e coverage del codice.

Figura 4.7: Logo Vitest

Claude Code

Claude Code è l'agente di codifica basato su modelli linguistici di grandi dimensioni (LLM) utilizzato durante lo stage come *pair engineer* strutturato per lo studio delle tecnologie, la realizzazione di prototipi e la propagazione della telemetria e test ai moduli del framework.

Figura 4.8: Logo Claude

Capitolo 5

Progettazione

In questo capitolo si esporranno le caratteristiche tecniche della soluzione prodotta, descrivendo l'architettura e le scelte progettuali adottate per la telemetria e per i test

5.1 Osservabilità nei sistemi distribuiti

L'osservabilità è la proprietà di un sistema software che permette di comprenderne lo stato interno esaminando esclusivamente i dati prodotti in output. Nel contesto delle applicazioni web distribuite, questa proprietà è resa possibile dalla produzione sistematica di tre categorie di segnali:

- **Logs**: rappresentano eventi discreti generati dal sistema durante la sua esecuzione. Contengono informazioni dettagliate su operazioni svolte, errori, stati e messaggi diagnostici, risultando fondamentali per attività di debugging e analisi puntuale di anomalie.
- **Traces**: descrivono il flusso di esecuzione di una richiesta all'interno del sistema, tracciandone il percorso attraverso i vari componenti e servizi. Sono particolarmente utili in architetture distribuite, in quanto permettono di analizzare le dipendenze tra servizi e individuare eventuali colli di bottiglia o punti di errore.
- **Metriche**: rappresentano misurazioni numeriche aggregate nel tempo, utilizzate per monitorare lo stato e le prestazioni del sistema. Si distinguono in:
  - *Metriche di performance*: raccolte automaticamente tramite strumenti come OpenTelemetry, includono informazioni quali tempi di risposta, utilizzo delle risorse, throughput e latenza, utili per valutare le prestazioni del sistema.
  - *Metriche di business*: definite a livello applicativo, riguardano aspetti funzionali del dominio, come numero di operazioni effettuate, utenti attivi o eventi specifici, e consentono di monitorare l'utilizzo e il valore del sistema dal punto di vista aziendale.

La forza di un sistema di osservabilità moderno non risiede nell'utilizzo isolato di questi segnali, bensì nella loro correlazione: poter passare da una metrica anomala alla trace corrispondente, e da questa ai log emessi durante quella specifica richiesta, costituisce uno strumento diagnostico di grande efficacia.

5.2 Lo standard OpenTelemetry

OTel è un framework open-source, oggi standard de facto nel settore, che fornisce un modello unificato e vendor-neutral per la produzione, la raccolta e l'esportazione di segnali di osservabilità. È il risultato della fusione tra i progetti *OpenCensus* e *OpenTracing*, avvenuta nel 2019 sotto l'egida della Cloud Native Computing Foundation (CNCF).
I principi fondamentali di OTel rilevanti per questo progetto sono:

- **Separazione tra produzione ed esportazione.** Le API applicative sono indipendenti dal backend di storage scelto: lo stesso codice può inviare dati a Grafana, Jaeger, Zipkin o qualsiasi altro sistema compatibile.
- **Strumentazione automatica.** Per i framework più diffusi (ASP.NET Core, HttpClient, SqlClient), OTel fornisce librerie che non richiedono modifiche al codice applicativo.
- **Estensibilità.** È possibile aggiungere attributi custom, definire metriche applicative proprie e configurare Sampler e processor personalizzati.

5.3 Architettura della telemetria

Il sistema di telemetria sviluppato si basa su una distinzione tra due macro-componenti principali: la **produzione dei dati** lato applicazione e la **raccolta e visualizzazione** lato infrastruttura.
L'applicazione backend, sviluppata in ambiente *.NET*, utilizza lo standard open-source OpenTelemetry per generare i tre principali segnali di osservabilità: *trace*, *metriche* e *log*. Questi dati vengono successivamente inviati a un'infrastruttura centralizzata basata sullo stack Grafana, che ne consente l'analisi e la visualizzazione.

5.3.1 Produzione dei dati di telemetria

La generazione dei dati avviene direttamente all'interno dell'applicazione backend tramite diverse tecnologie integrate.
Le **trace** vengono prodotte utilizzando OpenTelemetry, che si appoggia alle API native di .NET (`ActivitySource` e `Activity`) per rappresentare le operazioni eseguite dal sistema sotto forma di *span*. Ogni richiesta viene quindi tracciata lungo tutto il suo ciclo di vita, includendo chiamate HTTP, operazioni sui database e logica applicativa.
Le **metriche** vengono generate attraverso due approcci distinti:

- metriche di *performance*, raccolte automaticamente da OpenTelemetry (ad esempio durata delle richieste HTTP o utilizzo delle risorse);

- metriche di *business*, definite manualmente tramite le API `Meter` di .NET, per monitorare aspetti specifici del dominio applicativo.

I **log** vengono invece gestiti tramite la libreria Serilog, che consente la produzione di log strutturati e la loro esportazione verso l'infrastruttura di osservabilità. Grazie all'integrazione con OpenTelemetry, i log risultano automaticamente correlati alle trace tramite identificativi condivisi (*TraceId* e *SpanId*).

5.3.2 Integrazione con OpenTelemetry Collector ed ecosistema Grafana

Per la gestione e la visualizzazione dei dati di telemetria è stata adottata un'architettura basata su OpenTelemetry Collector e sullo stack Grafana (LGTM: Loki, Grafana, Tempo e Prometheus). I dati di telemetria generati dall'applicazione backend, vengono inizialmente prodotti tramite le librerie OpenTelemetry, .NET Meters e Serilog, e poi vengono esportati in formato compatibile OpenTelemetry. Tutti questi dati vengono inviati a un componente intermedio, l'*OpenTelemetry Collector*, tramite protocollo OTLP (OpenTelemetry Protocol). Il Collector svolge un ruolo centrale nell'architettura di osservabilità, fungendo da punto di raccolta e smistamento dei dati. Le sue principali responsabilità includono:

- **Ricezione dei dati**: acquisizione di trace, metriche e log dall'applicazione tramite endpoint OTLP;
- **Elaborazione**: possibilità di applicare trasformazioni, filtri o arricchimenti ai dati raccolti;
- **Esportazione**: inoltro dei dati verso i sistemi di destinazione tramite specifici *exporter*.

Nel progetto, il Collector è configurato per instradare i diversi tipi di dati verso componenti distinti dello stack Grafana:

- le **trace** vengono esportate verso Tempo, permettendo l'analisi dei flussi di esecuzione e delle dipendenze tra servizi;
- le **metriche** vengono inviate a Prometheus, rendendo possibile il monitoraggio delle prestazioni e la creazione di dashboard;
- i **log** vengono esportati verso Loki, consentendo la consultazione centralizzata e la correlazione con trace e metriche.

Grafana rappresenta il punto di accesso unificato a queste informazioni, offrendo strumenti per la visualizzazione e l'analisi dei dati di telemetria. In particolare, permette di costruire dashboard personalizzate basate su metriche, esplorare i log e navigare le trace, abilitando una visione completa e correlata del comportamento del sistema. Questa architettura consente di separare la raccolta dei dati dalla loro visualizzazione, migliorando la scalabilità e la flessibilità del sistema di monitoraggio, oltre a facilitare eventuali estensioni future.

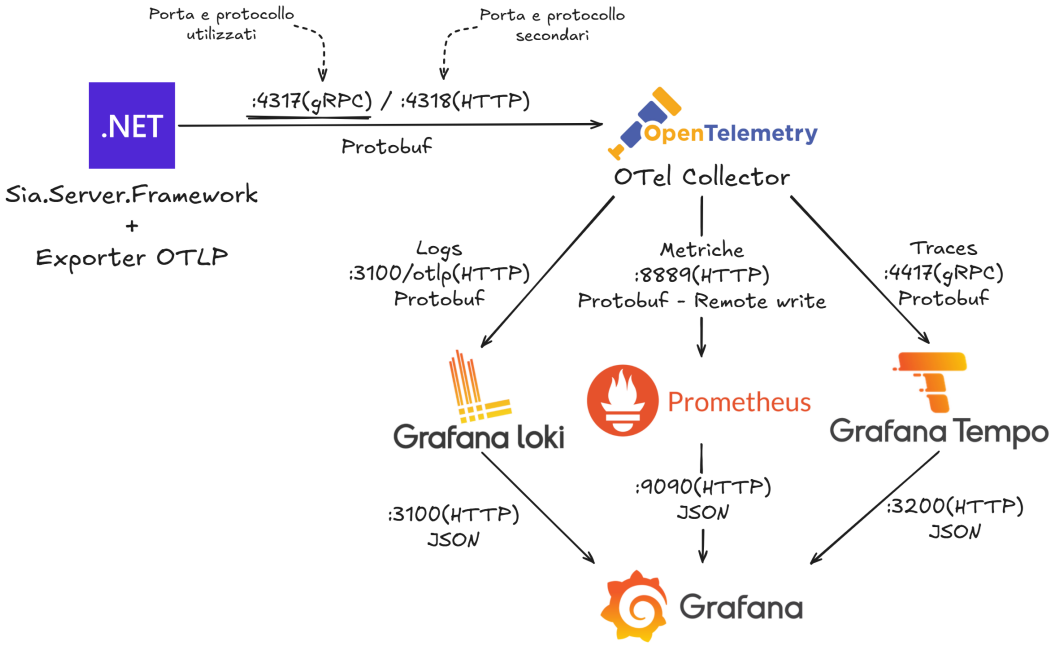

Figura 5.1: Architettura OTel Collector

5.3.3 Protocolli e canali di comunicazione

Ogni segnale telemetrico percorre un canale distinto, scelto in base alle caratteristiche del dato e ai requisiti di affidabilità del trasporto.
L'applicazione .NET invia tutti i segnali all'OTel Collector attraverso il protocollo OTLP, che utilizza *Protobuf* come formato di serializzazione. *Protobuf* è un formato binario: a parità di informazione trasportata, occupa una frazione dello spazio di un formato testuale come JSON, riducendo la banda consumata e il tempo di serializzazione. Per il trasporto, OTLP supporta due alternative: gRPC sulla porta 4317, che offre multiplexing delle richieste su una singola connessione persistente ed è il canale primario del progetto; HTTP sulla porta 4318, mantenuto come canale secondario per ambienti in cui gRPC è bloccato da proxy o firewall.
Raccolti dal Collector, i tre segnali vengono instradati verso database separati con protocolli differenziati. I *log* raggiungono Grafana Loki sulla porta 3100 tramite OTLP HTTP con serializzazione *Protobuf*. Le *trace* raggiungono Grafana Tempo sulla porta 4417 tramite gRPC con *Protobuf*. Le *metriche* seguono invece un percorso diverso: il Collector le espone sulla porta 8889 tramite *remote write* HTTP con *Protobuf*, e Prometheus le raccoglie con *scrape* periodico. Questa scelta riflette la natura delle metriche, dati aggregati e periodici, non eventi discreti, per i quali il modello *pull* di Prometheus è più efficiente di una connessione persistente.
Grafana si connette infine ai tre database tramite HTTP con formato JSON: un protocollo testuale leggibile, appropriato per le query interattive delle dashboard dove la latenza di serializzazione è trascurabile rispetto ai tempi di risposta dell'utente.

5.3.4 Scelte architetturali

5.3.4.1 Adozione di OpenTelemetry

La scelta di OTel, come richiesto dal requisito RVD-1, permette flessibilità nella scelta del backend di visualizzazione. Le alternative considerate, soluzioni proprietarie come Azure Application Insights o Datadog APM, avrebbero introdotto un accoppiamento stretto con un singolo fornitore, limitando la flessibilità nella scelta del backend e rendendo difficoltosa la migrazione futura.

5.3.4.2 Architettura opt-in

L'intero sistema è progettato come *opt-in*: la telemetria è attiva solo se `OpenTelemetry:Enabled = true` in *appsettings*, rispettando così il requisito RFO-5. Questo consente di abilitarla esclusivamente negli ambienti che la richiedono, evitando *overhead* in sviluppo locale.

5.3.4.3 Centralizzazione della configurazione

Tutta la configurazione OTel è centralizzata in `BaseApplicationBuilder`, componente condiviso da tutti i prodotti *Sia.Web*. I singoli moduli si limitano a registrare il proprio `ActivitySource` e `Meter` nel provider già costruito, senza conoscere i dettagli della pipeline.

5.3.4.4 Campionamento

Il campionamento è gestito da un Sampler custom, che combina una logica *ratio-based* per le *root span* con ereditarietà della decisione per le *child span*.
Se lo span ha un parent valido (`TraceId` e `SpanId` non nulli), il Sampler eredita la decisione del parent senza ricalcolo. Questo garantisce la coerenza interna della trace: se la root viene campionata, tutte le child span, vengono raccolte integralmente.
Per le root span, il Sampler risolve il ratio da applicare tramite `ResolveRatio`: il nome dell'activity (es. `sia.grid.page`) viene confrontato con i prefissi derivati dai nomi source configurati in appsettings (es. `Sia.Grid` → `sia.grid.`). I prefissi sono ordinati per lunghezza decrescente, così il match più specifico ha sempre precedenza su quello più generale. Se nessun prefisso corrisponde, si applica il ratio globale `SamplingRatio`.

5.3.4.5 Arricchimento automatico del contesto utente

Per soddisfare il requisito RFD-9, il sistema arricchisce automaticamente ogni span con l'identità dell'utente autenticato senza richiedere modifiche ai singoli moduli applicativi. Il meccanismo è descritto nel dettaglio nella sezione 5.4.5.

5.3.4.6 Correlazione metrica-trace tramite *exemplars*

Per soddisfare il requisito RFO-10, le metriche di tipo *Histogram* espongono degli *exemplars*: campioni rappresentativi a cui è associato il `TraceId` della richiesta che li ha generati. Questo collegamento abilita la navigazione dal pannello di una metrica aggregata alla trace specifica che ha prodotto il campione anomalo, rendendo

immediata la correlazione tra l'anomalia osservata e la richiesta che ne è all'origine. Il meccanismo è descritto nel dettaglio nella sezione 5.4.6.

5.3.4.7 Pattern architetturali applicati

La soluzione si appoggia a due *design pattern* classici per risolvere problemi ricorrenti: il pattern *Adapter*, utilizzato per riconciliare i `Counter` per-modulo con un contratto comune (`IControllerDiagnostics`), e il pattern *Factory*, utilizzato per costruire gli oggetti della pipeline OTel a partire dalla configurazione *appsettings*. I dettagli sono illustrati nella sezione 5.5.

5.4 Modello dei segnali e utilizzo degli attributi

5.4.1 Modello dei log

I log sono prodotti da Serilog con l'interfaccia `ILogger<T>` e arricchiti automaticamente con `TraceId` e `SpanId` quando emessi nel contesto di una richiesta tracciata. Questo abilita la navigazione bidirezionale tra log e trace in Grafana.

5.4.2 Modello delle trace

Una *trace* è l'insieme completo di un'operazione end-to-end, identificata da un unico `trace_id`. È composta da una gerarchia di *span*, ciascuno con un proprio `span_id` e, ad eccezione della root, un `parent_span_id`.
Nel sistema, le trace sono prodotte da tre tipologie di sorgenti: strumentazione automatica HTTP (richieste in ingresso via AspNetCore, chiamate in uscita via HttpClient); strumentazione automatica database (query SQL Server via SqlClient, query PostgreSQL via Npgsql); strumentazione manuale dei moduli di Sia.Core tramite le classi `Sia[NomeModulo]Diagnostics`.

5.4.3 Modello delle metriche

Sono impiegati quattro tipi di metriche, offerti dall'API `System.Diagnostics.Metrics` di *.NET*:

Counter Contatore monotono crescente per eventi cumulativi (es. login riusciti, query eseguite). In Prometheus acquisisce il suffisso `_total`.

Histogram Distribuzione statistica per latenze e percentili (p50, p95, p99). In Prometheus genera tre serie: `_bucket`, `_sum`, `_count`.

UpDownCounter Contatore non monotono, che può essere incrementato o decrementato, utilizzato per rappresentare grandezze "in corso" (es. numero di operazioni attualmente in esecuzione).

ObservableGauge Strumento campionato tramite *callback* periodica, utilizzato per valori istantanei non cumulativi (es. numero di connessioni attive).

La progettazione delle metriche ha seguito due criteri guida. Il primo è la bassa cardinalità: i tag con alta variabilità (es. identificatori numerici di record) sono

applicati solo agli span, mai alle metriche, per evitare un'esplosione del numero di serie temporali in Prometheus. Il secondo è la pre-aggregazione: le metriche vengono aggregate lato applicazione prima dell'esportazione, con overhead nettamente inferiore rispetto alla raccolta di trace complete.

5.4.4 Tassonomia degli attributi

5.4.4.1 Resource attributes (comuni a tutti i segnali)

Tutti i segnali telemetrici prodotti dal framework, indipendentemente dal modulo di provenienza, condividono un insieme di attributi descrittivi definiti a livello di risorsa. Tra i più importanti vi sono `service.name`, che identifica l'installazione applicativa, `sia.product_type`, che distingue la linea di prodotto, e `sia.customer_name`, che identifica il cliente a cui appartiene l'istanza monitorata. Questi attributi consentono di filtrare e aggregare metriche, *trace* e *log* provenienti da installazioni differenti mantenendo una vista unificata dell'intero ecosistema applicativo.

5.4.4.2 Attributi del modulo Sia.Base

Il modulo `Sia.Base`, responsabile delle operazioni CRUD e dell'accesso ai dati, utilizza una tassonomia comune per metriche e *trace*. Gli attributi principali descrivono il tipo di operazione eseguita (`sia.operation`), il metodo infrastrutturale coinvolto (`sia.method`), il metodo business che ha originato la chiamata (`sia.caller_method`), l'entità interessata (`sia.entity`) e il repository utilizzato (`sia.repository`). Ad esempio, una richiesta di inserimento di un ordine potrebbe essere identificata dagli attributi:

- `sia.operation = insert`;
- `sia.entity = OrderEntity`;
- `sia.repository = OrdersRepository`;
- `sia.caller_method = CreateOrder`.

In caso di errore viene inoltre registrato l'attributo `exception.type`, che permette di distinguere rapidamente problematiche quali `SqlException` o `TimeoutException`.

5.4.4.3 Attributi del modulo Sia.Auth

Nel modulo dedicato all'autenticazione, le metriche utilizzano esclusivamente attributi a bassa cardinalità per evitare la proliferazione di serie temporali. Un esempio è il tag `reason`, associato ai tentativi di autenticazione falliti. Le *trace* contengono invece informazioni più dettagliate sul processo di autenticazione. Tra gli attributi principali figurano `auth.method`, che identifica il meccanismo utilizzato per il login, e `auth.result`, che descrive l'esito dell'operazione con valori quali `success`, `unauthorized`, `error` o `two_factor_required`. Quando viene richiesta l'autenticazione a due fattori, ulteriori attributi specificano la tipologia adottata, ad esempio `email` oppure `totp`.

5.4.4.4 Attributi del modulo Sia.Grid

Nel modulo `Sia.Grid` il tag centrale è `componentId`, presente sia sulle trace sia sulle metriche di tutte le operazioni: identifica quale griglia specifica ha generato il segnale e permette di confrontare le prestazioni tra componenti diversi. A seconda dell'operazione eseguita, possono essere presenti ulteriori attributi specifici. Ad esempio, le operazioni di caricamento pagina includono informazioni sulla presenza di procedure SQL associate, mentre le operazioni di eliminazione massiva riportano il nome della procedura utilizzata. Questo approccio permette di correlare rapidamente eventuali rallentamenti o errori a uno specifico componente dell'interfaccia utente.

5.4.5 Tracciamento dell'identità utente

Ogni trace generata da una chiamata autenticata riporta automaticamente l'identità dell'utente tramite i tag `enduser.id` ed `enduser.name`, senza che i singoli moduli debbano iniettarli manualmente (RFD-9).
Il meccanismo si articola in due componenti trasparenti al codice applicativo:

un middleware ASP.NET Core (`EnrichActivityWithUserMiddleware`), registrato dopo `UseAuthentication()`, legge gli attributi `userId` e `userName` da `HttpContext.User` e li pubblica nel Baggage OTel.

un `ActivityProcessor` custom (`BaggageActivityProcessor`) intercetta l'hook `OnStart` di ogni nuova `Activity` e copia le entry con prefisso `enduser.` come tag sullo span. Poiché il Baggage viene ereditato dalle child span, tutte le query SQL generate nell'ambito della stessa richiesta portano gli stessi `enduser.*` della root.

Gli attributi `enduser.*` non vengono mai aggiunti come tag delle metriche: farlo farebbe crescere il numero di serie temporali di Prometheus linearmente con il numero di utenti, violando RQO-2. La correlazione *metrica* → *utente* si ottiene invece tramite gli exemplar: dalla metrica si naviga alla trace, che porta già `enduser.*`.

5.4.6 Exemplar: collegamento diretto da metrica a trace

Le metriche aggregate dicono quanto e quando qualcosa è andato storto, ma non quale richiesta specifica ne è responsabile. Un picco sul percentile p_{99} della latenza, ad esempio, non rivela di per sé la chiamata incriminata. Gli *exemplar* colmano questo divario: sono campioni concreti allegati ai dati aggregati di una metrica che trasportano il `TraceId` della richiesta che li ha prodotti. In Grafana, si materializzano come piccoli diamanti sovrapposti ai grafici istogramma; un click su uno di essi apre direttamente la trace corrispondente in Tempo, dalla quale si può proseguire verso i log correlati. Il risultato è una catena di navigazione completa (metrica → trace → utente → log).
L'utilizzo degli *exemplar* è possibile grazie ad una feature di *Prometheus* chiamata *exemplar-storage*. Quest'ultima è stata affiancata al filtro `TraceBased` che risolve il problema dei puntatori orfani, ovvero quei puntatori che puntano ad una trace scartata dal sampling. Questa fusione garantisce dunque che un exemplar venga prodotto soltanto quando la trace a cui punta è effettivamente disponibile, azzerando l'overhead sulle richieste non campionate.

5.5 Design pattern applicati

Oltre ai pattern architetturali generali citati nella sezione 5.3.4.7, la soluzione utilizza due *design pattern* della letteratura classica[1] per risolvere problemi ricorrenti della strumentazione.

5.5.1 Pattern Adapter per il counter di errore dei controller

Contesto
In un'architettura a moduli come quella di *Sia.Web*, ogni modulo possiede una propria classe di diagnostica statica con i propri `Counter` e `Histogram`, registrati sul `Meter` specifico del modulo. Questo garantisce isolamento e flessibilità, ma introduce un rischio di divergenza: la logica di registrazione degli errori nei controller, che richiede l'*unwrap* di `SiaLoggerException.OriginalException` e l'applicazione dei *tag* `exception.type` e `sia.operation`, rischierebbe di essere replicata in modo leggermente diverso in ciascun controller, rendendo le metriche non omogenee e difficili da confrontare in Grafana.

Problema
Le classi di diagnostica dei moduli sono `static` e non possono implementare direttamente un'interfaccia di istanza. È necessario riconciliare il contratto astratto fornito dal *framework* con la pluralità di implementazioni concrete, senza duplicare la logica di *unwrap* nei singoli controller.

Soluzione
Il *framework* espone un'interfaccia `IControllerDiagnostics` con un unico membro `Counter<long> ControllerErrors` e un metodo condiviso `RecordMetricError(IControllerDiagnostics, Exception, string)` che incapsula la logica di *unwrap* e di applicazione dei *tag*. Per ciascun modulo viene definita una classe annidata `private sealed class ControllerDiagnosticsAdapter`: l'*Adapter* adatta il *counter* statico del modulo al contratto comune. La classe `Sia{Modulo}Diagnostics` espone, accanto ai *field* statici esistenti, un singleton `public static readonly IControllerDiagnostics AsControllerDiagnostics`; il controller invoca sempre `RecordMetricError(Sia{Modulo}Diagnostics.AsControllerDiagnostics, ex, nameof(Metodo))` senza conoscere l'implementazione concreta.
Il listato canonico, riferito al modulo `Sia.Devextreme.Crud`, è riportato nell'appendice C.1 (listati C.1 e C.2).

Vantaggi.

- contratto unico a livello *framework*, *counter* distinti per modulo (cardinalità controllata);
- logica di *unwrap* e applicazione dei *tag* scritta una sola volta;
- zero ripetizione nel controller: ogni `catch` diventa una singola riga;

[1] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, MA, USA: Addison-Wesley Professional, 1994. ISBN: 9780201633610.

- piena retrocompatibilità con i chiamanti diretti dei *field* statici esistenti, che rimangono invariati.

5.5.2 Pattern Factory

Il pattern *Factory* è applicato in più punti della pipeline OTel per mantenere in un unico luogo la traduzione da configurazione *appsettings* a oggetti del runtime OpenTelemetry.

Costruzione del *sampler*
La creazione di `SiaParentBasedSampler` avviene in un metodo *factory* che, letti il valore globale `SamplingRatio` e i *ratio* per-*source* da *appsettings*, restituisce l'istanza del Sampler configurata correttamente. L'invocazione avviene una sola volta in `BaseApplicationBuilder`; il resto della pipeline consuma un oggetto già pronto.

Registrazione di `Meter` e `ActivitySource`
Le classi `Sia{Modulo}Diagnostics` agiscono come *static factory* centralizzata per gli strumenti di osservabilità del modulo: istanziano al *type loading* il `Meter`, gli `ActivitySource`, e i `Counter`/`Histogram` esposti come membri statici pubblici. Il codice applicativo non istanzia mai direttamente uno strumento OTel, ma accede sempre all'istanza fornita dalla classe `Diagnostics` del proprio modulo.

Costruzione del `ResourceBuilder`
Gli attributi di *resource* (`service.name`, `sia.product_type`, `sia.customer_name`) vengono letti da *appsettings* e applicati a un unico `Resource` condiviso da *trace*, metriche e *log*, garantendone la coerenza cross-segnale. Anche questa costruzione avviene in un metodo *factory* interno a `BaseApplicationBuilder`.

Vantaggi

- la configurazione da *appsettings* viene tradotta in oggetti OTel in un solo punto, semplificando i test e le variazioni per ambiente;
- i moduli vedono solo istanze pronte all'uso, senza conoscere la pipeline;
- è possibile sostituire la *factory* con una variante *mock* per i test senza toccare il codice di business.

5.6 Diagrammi delle classi del sistema di telemetria

A complemento delle descrizioni narrative dei paragrafi precedenti, la struttura complessiva del sistema di telemetria è formalizzata in tre diagrammi delle classi in stile UML semplificato:

- il *pipeline* OTel centralizzato in `BaseApplicationBuilder`, con i tre *provider* (`TracerProvider`, `MeterProvider`, `LoggerProvider`) che condividono lo stesso `ResourceBuilder` ed esportano via `OtlpExporter` (figura B.1);

- la struttura generica del pattern `Sia{Modulo}Diagnostics` con il `ControllerDiagnosticsAdapter` annidato (figura B.2);
- il flusso di arricchimento automatico delle `Activity` con l'identità utente, dal *middleware* ASP.NET Core al `BaggageActivityProcessor` (figura B.3).

Per non interrompere il flusso narrativo del capitolo, i tre diagrammi e le relative descrizioni di dettaglio sono raccolti nell'appendice B.

5.7 Implementazione dei test automatici

5.7.1 Contesto

Prima dell'intervento, la validazione del comportamento applicativo avveniva principalmente tramite verifiche manuali dell'interfaccia utente. Questo approccio rendeva complesso modificare i moduli centrali del framework con sufficiente sicurezza, poiché ogni cambiamento introduceva il rischio di regressioni difficili da individuare in modo sistematico.
L'assenza di una suite automatizzata comportava inoltre:

- la mancanza di una rete di sicurezza durante attività di *refactoring*;
- tempi elevati per la validazione delle modifiche;
- difficoltà nell'individuazione precoce di difetti;
- assenza di una *baseline* condivisa sul comportamento atteso del sistema.

L'obiettivo dello stage è stato quindi introdurre una metodologia di testing riproducibile e integrabile nei processi di sviluppo, capace di verificare sia la correttezza della logica applicativa sia il comportamento dell'interfaccia utente in scenari realistici.

5.7.2 Strategia di testing adottata

La strategia implementata è stata suddivisa in più livelli, ciascuno dedicato alla validazione di aspetti differenti del sistema. La strategia adottata distingue quattro tipologie di test con responsabilità complementari: Unit test frontend, test *end-to-end* frontend, unit test backend e integration test backend. L'introduzione di più livelli di test ha permesso di ottenere una copertura più ampia del sistema, combinando verifiche veloci e isolate con controlli completi sul comportamento integrato dell'applicazione, arrivando allo sviluppo di circa 600 test tra frontend e backend.

5.7.2.1 Unit test backend

Gli *unit test* backend hanno l'obiettivo di verificare in isolamento la logica applicativa implementata lato server, senza coinvolgere componenti esterni come database reali, servizi remoti o infrastrutture di rete.
Questo tipo di test consente di validare rapidamente il comportamento delle singole unità software, riducendo il rischio di regressioni durante le modifiche al codice e

facilitando le attività di manutenzione e *refactoring*.
Nel progetto sviluppato durante lo stage, tali test sono stati utilizzati per validare il comportamento dei servizi applicativi del framework e delle API utilizzate dal frontend. In particolare, sono state verificate la corretta gestione delle richieste, la propagazione delle eccezioni, il comportamento dei repository e la logica associata ai moduli strumentati con OpenTelemetry.
L'utilizzo di dipendenze simulate (*mock*) ha permesso di eseguire i test in modo deterministico e veloce, senza richiedere l'avvio dell'intera infrastruttura applicativa. I moduli backend che sono stati strumentati con questo tipo di test sono 2 per un totale di 51 test. Anche se in seguito, tramite l'uso di regole assegnate agli agenti AI developer, sono stati creati nuovi test per ogni modifica effettuata al codice applicativo.

5.7.2.2 Integration test backend

I test di integrazione backend verificano il corretto funzionamento congiunto dei diversi componenti dell'applicazione: accesso al database, persistenza dei dati, serializzazione, servizi applicativi e infrastruttura del framework. A differenza degli *unit test*, questi test non isolano le dipendenze tramite *mock*, ma eseguono operazioni reali su un ambiente controllato, così da riprodurre scenari più vicini al comportamento effettivo del sistema in produzione.
Nel progetto sono state adottate tre tipologie principali di test di integrazione. La prima riguarda i test CRUD, chiamati **lifecycle test**, come suggerito dal nome, verificano l'intero ciclo di vita di un'entità sul database, controllando che tutte e quattro le operazioni vadano a termine correttamente.
La seconda categoria, **smoke test**, ha lo scopo di esercitare ogni singola configurazione per componente presente in produzione, intercettando *stored procedure* malformate, *connection name* non risolvibili o configurazioni errate di colonne. Iterando su tutti i componenti presenti nel database e per ciascuno effettua una chiamata reale all'*endpoint* di lettura (es. `POST /api/grid-core/page` o `GET /api/CrudData/page`), asserendo che la risposta sia `200 OK` con *payload* JSON valido.
Infine, sono presenti **test summary** specifici per i componenti griglia, durante i quali vengono testati gli *endpoint* responsabili del calcolo aggregato (`sum`, `avg`, `count`, `min`, `max`), verificando che le colonne configurate (tramite *perspective* JSON) con funzionalità di aggregazione, funzionino correttamente.
L'utilizzo di dati reali provenienti dal database consente di validare non solo la correttezza del codice applicativo, ma anche l'affidabilità e la consistenza dei dati presenti nel sistema. Questo approccio permette di individuare problemi che difficilmente emergerebbero tramite semplici *mock*, come incongruenze nei dati, errori di mapping, vincoli non rispettati o differenze tra schema atteso e stato effettivo del database. In questo caso il numero di test sviluppati è molto maggiore, ci aggiriamo intorno ai 200, maggior parte facenti parte di *smoke test* che testano gli endpoint di griglie e CRUD (con una copertura del 100%) presenti nel DB.

5.7.2.3 Unit test frontend

Gli *unit test* sono scritti con **Vitest** e hanno l'obiettivo di verificare in isolamento le singole unità logiche dell'applicazione frontend, senza coinvolgere componenti esterni come il browser, il *backend* o il *rendering* reale dell'interfaccia grafica. Questo tipo di test permette di validare il comportamento di funzioni, servizi e componenti applicativi in modo rapido e deterministico, riducendo il rischio di regressioni durante l'evoluzione del software.
In generale, gli *unit test* verificano aspetti quali:

- la correttezza della logica applicativa;
- la gestione dello stato interno dei componenti;
- la trasformazione e validazione dei dati;
- la gestione degli errori e dei casi limite;
- il corretto comportamento di metodi puri e funzioni di utilità.

Nel progetto sviluppato durante lo stage, i test hanno riguardato principalmente la logica frontend implementata in TypeScript, includendo servizi applicativi, componenti Angular e direttive personalizzate. In particolare, sono stati verificati la gestione dello stato dei componenti, i meccanismi di raccolta e filtraggio dei dati, oltre al comportamento di funzionalità interattive come selezione, ordinamento e aggiornamento dinamico dell'interfaccia.
Questi test non richiedono né un browser reale né servizi esterni attivi, risultando quindi molto veloci nell'esecuzione e facilmente integrabili nei processi automatici di validazione del software. I moduli che risultavano essere più importanti da testare erano 4 (Grid, Calendar, Shared, Kanban) per un totale di circa 300 test.

5.7.2.4 Test end-to-end con Playwright

I test *end-to-end* (*E2E*) sono stati sviluppati tramite **Playwright** e hanno lo scopo di verificare il comportamento complessivo dell'applicazione simulando l'interazione di un utente reale con l'interfaccia web.
A differenza degli unit test, i test E2E eseguono l'applicazione in un browser reale e validano l'integrazione tra frontend, backend e servizi coinvolti. Questo approccio permette di intercettare problemi che emergono solo durante l'esecuzione completa del sistema, come errori di rendering, problemi di sincronizzazione, regressioni dell'interfaccia o malfunzionamenti nei flussi utente.
Come esposto precedentemente i test E2E sono utilizzati per verificare l'esperienza utente, in questa categoria ricadono requisiti come: la corretta navigazione tra le pagine, il comportamento delle principali funzionalità applicative, la corretta visualizzazione degli elementi grafici, la persistenza e l'aggiornamento dei dati nell'interfaccia grafica. Nel progetto, tali test sono stati utilizzati per validare il comportamento reale dei componenti più complessi del framework, come griglie dati, form dinamiche, operazioni CRUD e flussi di autenticazione. Sono stati inoltre utilizzati per verificare la corretta propagazione delle modifiche introdotte dagli agenti AI, riducendo il rischio di regressioni funzionali nei moduli condivisi. Il costo

da pagare è la velocità di esecuzione: ogni test richiede secondi, non millisecondi e la dipendenza dallo stato del sistema. Per questo i test E2E si concentrano sui flussi utente strategici, lasciando la copertura della logica di dettaglio agli *unit test*. Per questa caratteristica, i test E2E creati, sono stati semplicemente 29, suddivisi in: 5 per l'autenticazione, 7 per il calendario, 3 per componenti crud, 9 per le griglie e 5 per la visualizzazione di tipo kanban.

5.7.3 Vantaggi ottenuti

L'impatto più immediato dell'introduzione dei test è stata la scoperta di difetti latenti che non sarebbero emersi prima di un incidente in produzione. Per gli sviluppi futuri invece, l'automazione dei test ha reso possibile l'integrazione delle verifiche all'interno del flusso di sviluppo quotidiano, riducendo il numero di regressioni e migliorando la qualità complessiva del processo di sviluppo. L'esecuzione automatica delle suite ha ridotto significativamente il tempo necessario alla validazione delle modifiche e ha aumentato la confidenza nei refactoring dei moduli centrali del framework. Inoltre, la disponibilità di test automatici si è rivelata particolarmente utile nel contesto dello sviluppo AI-assistito: gli agenti di codifica potevano infatti produrre modifiche su più moduli, mentre le suite automatiche fornivano una validazione immediata del comportamento atteso.

Capitolo 6

Metodologia assistita da agenti AI

Il presente capitolo descrive in che modo è stata impiegata l'AI durante lo stage, incentrato sull'uso di un agente di codifica basato su un LLM come pair engineer *strutturato. Vengono illustrati gli ambiti di utilizzo, il workflow adottato, i benefici in termini di tempo e le mitigazioni adottate per i limiti noti.*

6.1 Contesto e motivazione

La pianificazione dello stage prevedeva una durata di 300 ore distribuite su 8 settimane, a fronte di un ambito ampio: apprendimento di tecnologie nuove (OpenTelemetry Software Development Kit (SDK) .NET, stack *Grafana* LGTM, PromQL/TraceQL/LogQL, Serilog OTLP, *Playwright*), realizzazione di prototipi, strumentazione di più moduli del framework *Sia.Web* e produzione della relativa documentazione. Per colmare il gap di conoscenza in merito alle tecnologie e lo studio della soluzione *Sia.Web*, si è fatto ricorso all'utilizzo di agenti di intelligenza artificiale. Questo approccio ha permesso di velocizzare le attività ripetitive e di replicare pattern già validati su nuovi moduli. L'introduzione di agenti di codifica, come *Claude Code*, ha reso più efficiente l'introduzione di test automatici e sistemi di telemetria all'interno di un'applicazione di grandi dimensioni, permettendo di creare, e soprattutto propagare, codesti processi di supporto, senza sottrarre eccessive risorse agli sviluppi principali.

6.2 Ambiti di utilizzo

L'agente è stato utilizzato in tre ambiti distinti, corrispondenti a quattro momenti differenti dello stage.

6.2.1 Studio delle tecnologie

Nelle prime fasi, dedicate allo studio delle tecnologie, l'agente è stato impiegato per sintetizzare la documentazione ufficiale delle varie tecnologie, e per confrontare le

possibili soluzioni da adottare. Tale attività ha permesso di ridurre in modo sensibile il tempo stimato di *onboarding*, concentrando lo studio sui soli punti critici per il progetto invece che sulla documentazione integrale di ciascuna tecnologia.

6.2.2 Prove di implementazione

Durante la fase di progettazione, l'agente è stato usato per produrre rapidamente varianti di PoC su scelte ancora da validare: che moduli testare, quale fosse il carico computazionale introdotto, l'ambiente di hosting più conveniente (*Docker* o servizi Windows). Avere a disposizione codice compilabile in pochi minuti ha consentito di provare e scartare alternative, invece di decidere a priori sulla base della sola documentazione.

6.2.3 Propagazione della telemetria agli altri moduli

L'ambito in cui l'uso dell'agente si è rivelato più produttivo è stato la propagazione della strumentazione ai moduli del framework (*Sia.Base*, *Sia.Auth*, *Sia.Grid*, *Sia.Crud*, ...). Per evitare che la replicazione del pattern di strumentazione su moduli diversi introducesse incoerenze (naming, cardinalità, attributi di *resource*), sono state definite *Claude skill* dedicate:

- `sia-telemetry-setup`: configurazione della pipeline OTel, Sampler, *exporter*, *appsettings*;
- `sia-telemetry-traces`: *span*, *tag*, *helper* `TrackQuery`;
- `sia-telemetry-metrics`: metriche, *naming*, cardinalità, pattern `RecordMetricError`;
- `sia-telemetry-logs`: Serilog OTLP, correlazione *log-trace*;
- `sia-telemetry-grafana`: *dashboard* Grafana, query PromQL/TraceQL/LogQL, strumenti MCP.

Ogni *skill* è un contratto procedurale, non un semplice *prompt*, che descrive convenzioni, *anti-pattern* e *template* di codice per uno specifico aspetto. L'agente carica la *skill* appropriata quando riceve un task pertinente, garantendo che moduli strumentati in momenti diversi rispettino le stesse convenzioni.

6.2.4 Propagazione dei test automatici

La stessa metodologia è stata replicata per l'introduzione e la propagazione dei test automatici, sia lato *backend* sia lato *frontend*. In entrambi i casi è stato definito un agente dedicato che codifica la struttura della *suite*, lo stack tecnologico di riferimento e le convenzioni di *naming* e organizzazione dei file, così che ogni nuovo test risulti coerente con quelli già esistenti.
Sul *backend* l'agente copre le categorie di test unitari, *smoke*, *lifecycle* e *summary*, ed è affiancato da due *skill* specifiche per la creazione di nuovi *lifecycle test*, essendo quest'ultimo il pattern più ripetitivo della *suite*. Una *skill* è orientata al modulo CRUD, nella quale viene spiegato all'LLM a cosa serve quel test, vincoli specifici dei test e dove raccogliere i dati necessari per la scrittura dei test; un'altra è specifica

del modulo Grid e ricopre le stesse funzioni della *skill* precedente ma relative ai componenti griglia. Sul *frontend* l'agente è invece focalizzato sui test di usabilità end-to-end basati su *Playwright*, sull'organizzazione dei *page object* e sulla scrittura di scenari resilienti rispetto al rendering asincrono dei componenti.
Il beneficio osservato è analogo a quello registrato sulla telemetria: una volta validato il pattern su un primo caso di riferimento, l'estensione a un nuovo componente diventa una richiesta semantica all'agente, che in pochi minuti produce i file di scenario e le dichiarazioni necessarie, senza che lo sviluppatore debba ricordare convenzioni distribuite tra più punti del progetto.

6.3 Workflow Agenti AI

Per evitare i rischi tipici della generazione automatica di codice (allucinazioni, *overfitting* al primo caso visto), è stato adottato un *workflow* strutturato in tre fasi.

1. **Plan.** Per ogni task non banale, l'agente produce un file `plan.md` che descrive obiettivo, componenti coinvolti, modifiche pianificate file per file, assunzioni e domande aperte. Nessun codice viene prodotto in questa fase.

2. **Confirm.** Lo sviluppatore revisiona il `plan.md`, può editarlo direttamente, integrarlo o rifiutarlo. L'agente si ferma *esplicitamente* in attesa di approvazione.

3. **Implement.** Solo dopo approvazione l'agente scrive un `task.md`, una *checklist* operativa, nella quale vengono tracciati in ordine, i singoli interventi eseguiti dall'agente.

Il *workflow* produce un artefatto tracciabile per ogni intervento: `plan.md` e `task.md` restano nel repository e fungono da registro decisionale.

6.4 Confronto pianificazione vs esecuzione

L'uso dell'agente ha prodotto un beneficio misurabile soprattutto nelle fasi a forte componente ripetitiva (studio, PoC, propagazione del pattern di strumentazione), e un beneficio meno marcato in quelle a forte componente decisionale (scelte architetturali, analisi di cardinalità). Nel complesso, il tempo risparmiato nelle fasi assistite è stato riinvestito nelle attività non automatizzabili: revisione semantica del codice prodotto, validazione delle *dashboard* contro Prometheus, affinamento iterativo delle *skill*.
Per dare un riferimento quantitativo a questo bilancio, lo studio delle tecnologie e lo sviluppo dei PoC erano stati preventivati attorno alle 80 ore: con l'uso degli agenti, il tempo effettivo si è ridotto a circa 30 ore. Una riduzione ancora più marcata si è osservata nella propagazione dei test, ambito per il quale la pianificazione iniziale non prevedeva nemmeno la copertura di tutti i moduli poi effettivamente strumentati: una stima approssimativa indica che, senza il supporto degli LLM, il lavoro avrebbe richiesto circa un mese, condensato invece in una settimana e mezza.

6.5 Limiti e mitigazioni

L'uso di un agente LLM comporta rischi noti che sono stati mitigati in modo sistematico.

Allucinazioni su metriche/label. L'agente tende a inventare nomi di metriche o *label* plausibili ma inesistenti. **Mitigazione**: regola obbligatoria di validazione tramite MCP Grafana, verificando l'esistenza di metriche e label in Grafana prima di chiudere ogni modifica delle dashboard.

Overfitting del prompt. Le *skill* iniziali, scritte a partire dal primo modulo strumentato (`Sia.Base`), contenevano riferimenti troppo specifici (nomi di metriche, *tag*, stored procedure). **Mitigazione**: riscrittura delle *skill* in termini generali, con placeholder `{modulo}` e riferimenti ai moduli canonici solo come esempi.

Perdita di contesto su sessioni lunghe. La *context window* dell'LLM è finita; su task lunghi, l'agente perde traccia di decisioni prese in precedenza. **Mitigazione**: memorizzazione persistente tramite `MEMORY.md` per progetto e per agente, `plan.md` e `task.md` come registro decisionale per ogni task.

Divergenza dei pattern tra moduli. Sviluppatori diversi, o sessioni diverse, possono produrre variazioni del pattern di strumentazione. **Mitigazione**: centralizzazione dei pattern obbligatori nell'agente `sia-telemetry-developer` (es. counter errori controller via `IControllerDiagnostics` con Adapter, uso obbligatorio della *template variable* `bucket` nelle dashboard).

Per superare gran parte di queste limitazioni e arricchire i processi di ragionamento degli agenti AI con contenuti e contesto aggiuntivi, è stato adottato il protocollo MCP. Considerato inoltre il crescente impatto di questo strumento nell'ecosistema LLM, si è ritenuto opportuno dedicargli una sezione specifica di approfondimento.

6.6 Model Context Protocol

Il MCP è il protocollo, introdotto nel 2024, che consente a un agente basato su modelli linguistici di interagire con servizi esterni seguendo uno standard, ottenendo dati dinamici e contestuali.

6.6.1 Il problema che MCP risolve

Un modello linguistico di grandi dimensioni LLM è, nella sua forma base, un sistema chiuso: conosce ciò che è stato scritto fino alla sua data di addestramento e risponde in linguaggio naturale, ma non può aprire un file, interrogare un database, chiamare un'API o agire nel mondo esterno. Questa limitazione non dipende dall'intelligenza del modello, dipende dall'assenza di un canale strutturato attraverso cui il modello possa ricevere dati aggiornati e restituire azioni concrete.
Prima dell'avvento di MCP, ogni integrazione tra un LLM e un sistema esterno richiedeva un connettore ad hoc: codice scritto una volta per quella specifica combinazione di modello e strumento, da riscrivere ogni volta che cambiava uno dei due. Con la proliferazione dei modelli e degli strumenti, il costo di manutenzione di

questa matrice di integrazioni punto-a-punto è diventato rapidamente insostenibile. Il MCP, introdotto da Anthropic nel novembre 2024 e oggi standard open-source governato dalla Linux Foundation sotto l'egida dell'Agentic AI Foundation, risolve il problema della standardizzazione definendo un'interfaccia universale che qualsiasi modello e qualsiasi strumento possono implementare una volta sola per comunicare con l'intero ecosistema.

6.6.2 Dettagli del protocollo

MCP è un protocollo client-server basato su *JSON-RPC 2.0* con connessioni *stateful*, articolato su tre ruoli: l'**host** (l'applicazione che ospita l'LLM, responsabile del consenso dell'utente e del controllo degli accessi), il **client** (il componente che gestisce la connessione verso un server specifico) e il **server MCP** (il servizio, locale o remoto, che espone dati e funzionalità all'LLM). Prima di qualsiasi scambio applicativo, client e server negoziano le proprie capacità in una fase di *handshake*, garantendo la retrocompatibilità tra versioni diverse del protocollo.
Sul piano funzionale, il server può esporre tre tipi di primitive: le **Resources** (dati consultabili, identificati da URI), i **Tools** (funzioni invocabili dal modello per agire nel mondo, che richiedono consenso esplicito dell'utente) e i **Prompts** (template predefiniti per workflow ricorrenti). Simmetricamente, il client può offrire al server tre capacità: il **Sampling** (che delega all'utente il controllo su quali prompt vengono inviati al modello), i **Roots** (i confini del filesystem o degli URI entro cui il server può operare) e l'**Elicitation** (la possibilità per il server di richiedere informazioni aggiuntive in modo strutturato).
Data l'ampiezza dei permessi che un server MCP può richiedere, la specifica affronta la sicurezza attraverso quattro principi: consenso esplicito dell'utente prima di ogni invocazione di tool (con cautela verso le descrizioni provenienti da server non verificati, vettore noto di *prompt injection*); privacy dei dati, che non devono essere trasmessi ai server senza consenso; controllo sul sampling, con visibilità dell'utente sui prompt inviati al modello; e separazione dei contesti, per limitare l'esposizione di informazioni confidenziali. Ricerche indipendenti hanno comunque evidenziato vulnerabilità concrete a *tool poisoning* e *prompt injection* in implementazioni permissive, da cui l'indicazione di adottare meccanismi di autorizzazione granulari.

6.6.3 Applicazione nel progetto

Nel contesto del progetto *Sia.Core*, MCP è stato impiegato per collegare l'agente di sviluppo direttamente a Grafana, a *Playwright* e a *mssql*, eliminando il passaggio manuale di schemi, nomi di metriche e strutture dati tra l'utente e l'agente.
L'impatto più evidente si è registrato nella generazione delle dashboard: prima dell'integrazione, l'agente produceva widget plausibili ma spesso errati, metriche inesistenti, attributi di filtraggio non presenti nelle serie temporali, tipi di grafico inadatti alla natura del dato. Connettere l'agente a Grafana tramite MCP ha ridotto sensibilmente questi errori, perché il modello poteva interrogare direttamente l'istanza e ricevere come contesto i nomi reali delle metriche, le label disponibili e la struttura delle trace esistenti in Tempo.
Nonostante il miglioramento, si è resa necessaria una fase di *prompt engineering* mirata a consolidare le linee guida operative dell'agente. In particolare, sono state

introdotte direttive esplicite su tre fronti: quali tipi di grafico associare a ciascuna categoria di metrica (es. *time series* per latenze e rate, *stat* per contatori cumulativi, *histogram* per distribuzioni di percentili); quali variabili di template definire a livello di dashboard per abilitare il filtraggio per `service_name`, `sia_repository` e `sia_operation`; come organizzare la griglia dei widget per mantenere leggibilità e coerenza visiva tra moduli diversi.
Un beneficio analogo si è ottenuto con l'integrazione di *Playwright* e *mssql*: in entrambi i casi, l'accesso diretto agli strumenti tramite MCP ha permesso all'agente di operare su dati reali anziché su assunzioni, riducendo il numero di iterazioni necessarie per ottenere un risultato corretto. L'accesso al database ha permesso all'LLM di apportare modifiche direttamente a tabelle, viste e *stored procedure*, oltre a recuperare dati utili per il *debugging* o per ricostruire la struttura delle entità coinvolte, condizione necessaria per scrivere query corrette senza doverla descrivere manualmente a ogni richiesta. Il collegamento di *Playwright* tramite MCP ha reso possibile all'LLM non solo di eseguire direttamente i test E2E, ma anche di aprire schede del browser per validare visivamente le modifiche apportate e ispezionare bug grafici, comprendendone la causa e intervenendo di conseguenza.

6.7 Considerazioni conclusive

L'utilizzo di un agente AI come supporto allo sviluppo, insieme all'adozione del protocollo Model Context Protocol, ha contribuito in modo significativo al miglioramento della qualità del prodotto, alla riduzione dei tempi di sviluppo e al superamento delle iniziali lacune tecniche.
Tuttavia, il ruolo dell'agente AI non deve essere interpretato come sostitutivo di quello dello sviluppatore. Il lavoro eseguito dall'agente, che fosse codice prodotto, scelte architetturali o propagazione degli sviluppi, ha spesso richiesto interventi di indirizzamento, correzione e validazione da parte dello sviluppatore risultando quindi un processo collaborativo piuttosto che autonomo.

Capitolo 7

Codifica

Il presente capitolo illustra l'organizzazione del codice prodotto e i dettagli implementativi della soluzione di telemetria, riprendendo le scelte progettuali descritte nel capitolo 5 e mostrando come si traducono nella struttura effettiva del framework.

7.1 Organizzazione del codice

Il codice di telemetria è distribuito su due livelli del framework:

- un livello **condiviso** (`Application.Shared.Telemetry` e `BaseApplicationBuilder`), che configura la pipeline OTel, registra Sampler, *processor* e *exporter*, e definisce le convenzioni comuni (attributi di *resource*, *middleware* di arricchimento utente);
- un livello **per-modulo** (`Sia.{Modulo}/Diagnostics/`), che contiene la classe statica `Sia{Modulo}Diagnostics` con `ActivitySource`, `Meter`, `Counter`, `Histogram`, e la dashboard Grafana `{nome}.dashboard.json` versionata insieme al codice.

Questa suddivisione garantisce che un nuovo modulo, per integrare la telemetria, debba solo creare la propria classe `Diagnostics` e registrare il proprio `Meter` e `ActivitySource` nella pipeline già costruito, senza conoscere i dettagli del Sampler o degli *exporter*.

7.2 Pipeline OpenTelemetry

La pipeline è configurato in `BaseApplicationBuilder.ConfigureBuilder` tramite l'API `AddOpenTelemetry()` dell'estensione ufficiale. I pacchetti NuGet utilizzati sono:

- `OpenTelemetry` e `OpenTelemetry.Extensions.Hosting` per l'integrazione con il *generic host* di .NET;
- `AspNetCore` e `Http` di `OpenTelemetry.Instrumentation` per le *trace* automatiche delle richieste HTTP in ingresso e in uscita;

- `OpenTelemetry.Instrumentation.SqlClient` per le *trace* automatiche delle query su SQL Server;
- `OpenTelemetry.Exporter.OpenTelemetryProtocol` per l'esportazione OTLP verso il *Collector*.

La configurazione dichiara nell'ordine: il `ResourceBuilder` con gli attributi comuni (`service.name`, `sia.product_type`, `sia.customer_name`), il provider di *trace* (`WithTracing`) con Sampler e *processor* custom, il provider di metriche (`WithMetrics`) con il filtro exemplar `TraceBased`, e il provider di *log* (`WithLogging`) integrato con Serilog.
Nell'appendice B.1 è presentato il diagramma delle classi relativo alla pipeline OpenTelemetry.

7.3 Classi di diagnostica per modulo

Ciascun modulo strumentato espone una classe statica `Sia{Modulo}Diagnostics`. La classe definisce come membri statici `readonly`:

- l'`ActivitySource` del modulo, con nome `Sia.{Modulo}` e versione;
- il `Meter` del modulo, con lo stesso nome e versione;
- i `Counter`/`Histogram`/`UpDownCounter` custom, con *naming* secondo la convenzione `sia.{modulo}.{operazione}.{segnale}`;
- il `Counter` `ControllerErrors` (`sia.{modulo}.controller.errors`) e il singleton `AsControllerDiagnostics` dell'Adapter verso `IControllerDiagnostics`, per i moduli con controller.

Nel codice applicativo, l'istrumentazione manuale di un'operazione avviene con: `Sia{...}Diagnostics.ActivitySource.StartActivity("sia.{...}.operazione");` seguito da `activity.SetTag(...)` per gli attributi specifici. Per le durate si usa un *helper* `TrackQuery` che combina in un unico blocco la creazione della *span*, la misurazione del tempo e la registrazione dell'`Histogram` corrispondente.

7.4 Pattern `RecordMetricError` con Adapter

L'implementazione del pattern descritto nella sezione 5.5 è la stessa in ogni modulo con controller. Il framework fornisce:

- l'interfaccia `Sia.Base.Diagnostics.IControllerDiagnostics`, con il contratto `Counter<long> ControllerErrors { get; }`
- `BaseApiController.RecordMetricError(IControllerDiagnostics, Exception, string operation)`, metodo che effettua l'*unwrap* di `SiaLoggerException.OriginalException` e applica i *tag* `exception.type` e `sia.operation`.

Nel modulo (esempio riferito a `Sia.Devextreme.Crud`) la classe `Diagnostics` si presenta, nei suoi elementi essenziali, come segue: C.1

7.5 Middleware di arricchimento utente

Il *middleware* `Application.Shared.Telemetry.EnrichActivityWithUserMiddleware` è registrato in `BaseApplicationBuilder` subito dopo `app.UseAuthentication()`. La sua responsabilità è leggere gli attributi `userId` e `userName` dal `HttpContext.User` e pubblicarle nel Baggage di OTel.
Un `ActivityProcessor` custom (`BaggageActivityProcessor`) intercetta poi l'*hook* `OnStart` di ogni `Activity` e copia ogni *entry* di *baggage* con prefisso `enduser.` come *tag* sull'`Activity`. Nell'appendice B.3 è possibile consultare il diagramma delle classi dell'arricchimento utente.

7.6 Configurazione degli exemplars

L'istruzione `SetExemplarFilter(ExemplarFilterType.TraceBased)` presente nel provider `Meter` della pipeline, istruisce OTel ad allegare un *exemplar* a un campione di metrica solo quando esiste una trace *sampled* nel contesto corrente. L'*exemplar* riporta il `TraceId` (e il `SpanId`) del campione, permettendo in Grafana la navigazione diretta dalla metrica alla trace.

7.7 Log Serilog con sink OTLP

Il *sink* OTLP (`Serilog.Sinks.OpenTelemetry`) è registrato in sostituzione dei *sink* precedenti, mantenendo il formato strutturato dei *log*. Serilog legge il contesto di `Activity.Current` e arricchisce automaticamente ogni evento con `TraceId` e `SpanId`, soddisfacendo il requisito RFO-4 di correlazione *log-trace*. Il livello minimo di *log* è configurabile per ambiente tramite *appsettings*.

7.8 Dashboard Grafana e versionamento

Ogni modulo strumentato espone una dashboard Grafana nella cartella `Sia.{Modulo}/Diagnostics/{nome}.dashboard.json`. Il file JSON è versionato nel repository insieme al codice di strumentazione, in modo da garantire il sincronismo tra le modifiche di istrumentazione e i pannelli di visualizzazione. Ogni dashboard definisce:

- le *template variable* `service_name`, `enduser_id`, `enduser_name` e `bucket`;
- pannelli di latenza con percentili *p50*, *p95*, *p99* calcolati via `histogram_quantile` (Figura 7.1);
- un pannello *stat* "Operazioni ultimo `${bucket}`" basato sull'`increase` del *counter* principale (Figura 7.2);
- pannelli di errori con *break-down* per `exception.type` (Figura 7.3);
- un pannello *logs* su Loki filtrato per `service.name` (Figura 7.3).

Il riferimento canonico è la dashboard `sia-base.dashboard.json` del modulo `Sia.Base`, replicata con le dovute personalizzazioni in ogni modulo successivo.

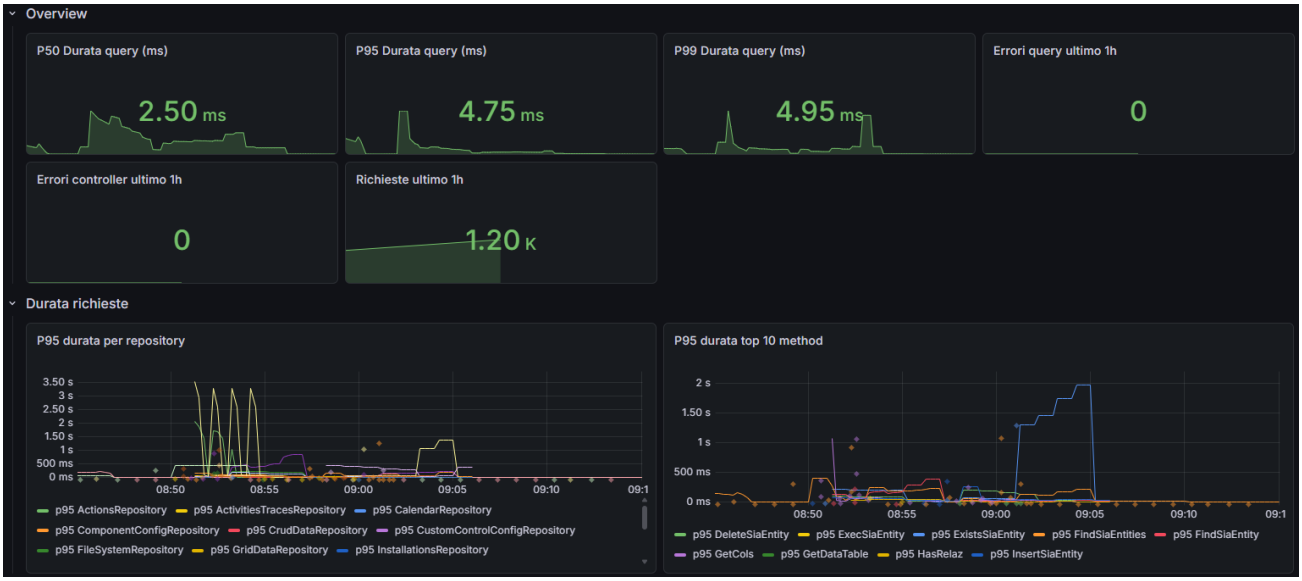

Figura 7.1: Overview e durata richieste della dashboard di *Sia.Base*

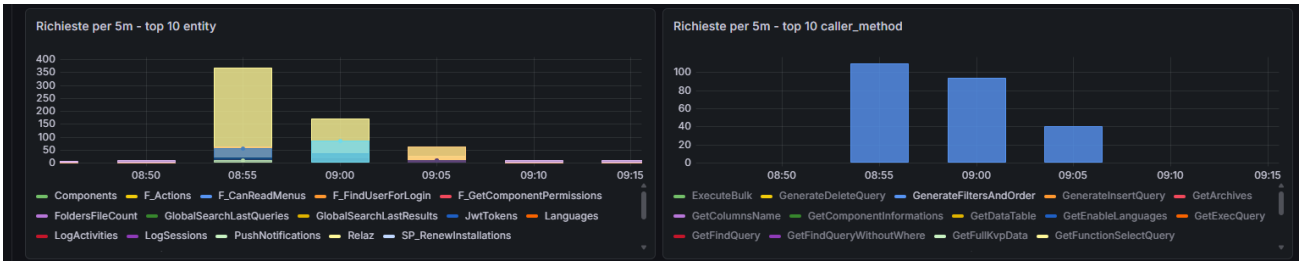

Figura 7.2: Sezione richieste per `${bucket}` della dashboard *Sia.Base*

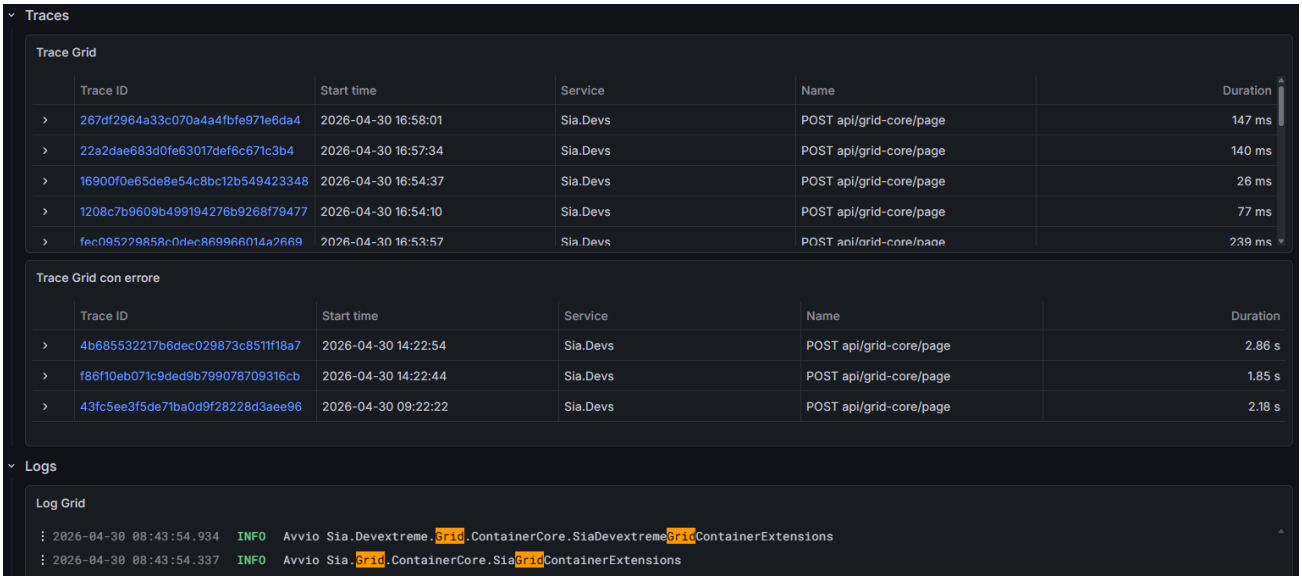

Figura 7.3: Sezione trace e log della dashboard di *Sia.Grid*

7.9 Telemetria delle performance hardware

7.9.1 Contesto e obiettivo

Il modulo `Sia.HealthChecks.Servers` è parte del framework *Sia.Server.Framework* ed è responsabile della raccolta periodica di indicatori sullo stato del server che ospita l'applicazione, con successiva notifica ad un *manager* centrale delle installazioni. Prima dell'intervento di telemetria, il modulo esponeva già i dati tramite:

- cinque implementazioni di `IHealthCheck` (`CpuUsageHealthCheck`, `CpuTemperatureHealthCheck`, `RamHealthCheck`, `DiskHealthCheck`, `SqlHealthCheck`), invocate dall'endpoint `/health` di ASP.NET Core;
- un `BackgroundService` (`HealthCheckServerHostedService`) che, ad intervallo configurabile (`HealthNotifyDelayMinutes`), aggrega i risultati e invia il report al *manager* tramite un `HttpClient` dedicato (`HealthCheckServerToServerApi`).

7.9.2 Scelte progettuali

Il modulo presenta due caratteristiche che lo distinguono dagli altri moduli strumentati.
La prima è l'assenza di un controller: gli `IHealthCheck` sono invocati dall'infrastruttura ASP.NET Core, non da endpoint applicativi. Di conseguenza il servizio di *HealthChecks* non implementa `IControllerDiagnostics` e non espone il counter `ControllerErrors`.
La seconda è la natura istantanea dei dati raccolti. CPU%, temperatura, GB di RAM liberi e spazio su disco non sono grandezze cumulative: l'uso di `Counter` o `Histogram` sarebbe improprio. Per rappresentare questo tipo di dato è stato utilizzato l'`ObservableGauge`, che a differenza di altri strumenti metrici, rappresenta uno stato istantaneo e non un aggregato. La sua caratteristica fondamentale è l'inversione del controllo: non è l'applicazione a inviare attivamente il dato, ma è il sistema di telemetria a recuperarlo periodicamente tramite una funzione di *callback (polling)* al momento della raccolta. E poiché le letture hardware avvengono solo quando l'endpoint `/health` viene interrogato, le gauge leggono un valore in cache, aggiornato dagli `IHealthCheck` alla loro esecuzione, evitando di esporre dati stantii.

7.9.3 Segnali prodotti

7.9.3.1 Trace

Il modulo produce tre span organizzate gerarchicamente:

- una span padre (`...cycle`) per ogni iterazione del ciclo di notifica che racchiude tutto il percorso per notificare un nuovo dato raccolto dal servizio di *HealthCheck*;
- una span figlia per ciascun controllo hardware eseguito (`...check`) che espone quale componente hardware è stato verificato e il suo stato attuale;
- infine, la span (`...send`) misura la durata dell'invio del report al *manager*

Le trace HTTP generate automaticamente da `HttpClient` restano figlie di `...send`: nella stessa trace è quindi possibile ispezionare sia la logica applicativa sia il dettaglio della chiamata HTTP verso il *manager*, senza dover correlare manualmente segnali separati.

7.9.3.2 Metriche

Accanto alle metriche cumulative standard (cycle, check, send), il modulo espone sei `ObservableGauge` per i valori hardware istantanei.

Tabella 7.1: ObservableGauge del modulo `Sia.HealthChecks.Servers`

Metrica	Unità	Descrizione
`...cpu.usage`	%	Percentuale di utilizzo della CPU al momento dell'ultima lettura
`...cpu.temperature`	°C	Temperatura del processore rilevata dai sensori hardware
`...ram.available`	GByte	Memoria RAM libera disponibile per i processi
`...ram.total`	GByte	Memoria RAM fisica totale installata nel server
`...disk.free`	GByte	Spazio libero sul disco, distinto per `sia.drive`
`...disk.total`	GByte	Capacità totale del disco, distinta per `sia.drive`

I tag hanno domini piccoli e chiusi (`sia.check` $\in \{cpu,\ cpu\_temperature,\ ram,\ disk,\ sql\}$, `sia.status` $\in \{Healthy, Unhealthy\}$), mantenendo la cardinalità sotto controllo indipendentemente dal numero di installazioni monitorate.

7.9.4 Dashboard Grafana

La dashboard segue lo schema standard del progetto, con l'aggiunta di una riga *Risorse* dedicata alle gauge hardware: andamento di CPU% e temperatura nel

tempo, RAM disponibile confrontata a soglia, spazio disco libero per drive e una *state timeline* delle connessioni SQL che visualizza a colpo d'occhio gli intervalli in cui una connessione è risultata *down*. I pannelli trace e log consentono, a partire da un picco anomalo su qualsiasi metrica hardware, di navigare alla trace del ciclo di notifica corrispondente e ai log emessi in quell'iterazione, chiudendo il percorso diagnostico metrica → trace → log.

Figura 7.4: Sezione risorse della dashboard di telemetria hardware

Capitolo 8

Conclusioni

Il presente capitolo riporta le conclusioni del lavoro svolto durante lo stage, analizzando i risultati ottenuti e gli obiettivi raggiunti, le conoscenze acquisite, i possibili sviluppi futuri degli ambiti applicativi toccati dal progetto e dando una valutazione personale dell'esperienza maturata.

8.1 Obiettivi raggiunti e consuntivo finale

L'oggetto di questo tirocinio è stato lo studio e l'implementazione, all'interno del *framework Sia.Core*, di un sistema di osservabilità completo basato sullo standard OTel e sullo stack *Grafana LGTM*, affiancato dall'introduzione *ex novo* di un'infrastruttura di *testing* automatico articolata su *unit test* e *integration test* lato *server* e su *unit test* e test *end-to-end* lato *client*. Le due linee di lavoro, parallele sul piano tecnologico, si sono dimostrate complementari sul piano metodologico: la telemetria ha fornito il *feedback* necessario a guidare il miglioramento iterativo dei test, mentre i test hanno supportato la validazione continua della strumentazione introdotta.
Trasversalmente alle due linee, il lavoro ha adottato un agente di intelligenza artificiale, attraverso la creazione di agenti specializzati, di *skill* procedurali e l'integrazione del MCP per consentire all'agente di interrogare direttamente *Grafana*, il database e gli strumenti di test.
Il progetto si è sviluppato sulle 8 settimane pianificate, per un totale di 300 ore. Il monte ore è stato complessivamente rispettato, con una redistribuzione interna: le fasi di studio, ricerca e prototipazione hanno richiesto un numero di ore inferiore al preventivato grazie all'uso degli agenti AI, permettendo di liberare tempo per l'approfondimento di funzionalità non inizialmente preventivate, descritte nel seguito.
In merito al raggiungimento degli obiettivi prefissati nel capitolo 2, si può confermare la soddisfazione di ciascuno. Più nello specifico, sono stati raggiunti correttamente i seguenti obiettivi:

- **Analisi dei requisiti di osservabilità e selezione delle tecnologie**: l'analisi ha portato all'adozione dello stack OTel + *Grafana LGTM*, motivata dai criteri descritti nel capitolo 4;

- **Progettazione dell'architettura di telemetria**: l'architettura è stata progettata e centralizzata in `BaseApplicationBuilder`, in modo che ogni nuovo modulo possa aderirvi semplicemente registrando il proprio `ActivitySource` e `Meter`, senza conoscere i dettagli interni del *pipeline*;
- **Definizione di una tassonomia coerente di metriche, *trace* e attributi**: la tassonomia è stata stesa per i moduli centrali del *framework* (autenticazione, componenti *Angular* come griglie e *form*, prestazioni hardware, accesso al database, ecc.), con regole esplicite che evitano l'esplosione della cardinalità lato Prometheus;
- **Definizione delle strategie di campionamento**: sono state realizzate a due livelli (Sampler globale e *processor* per-*source*), con *ratio* configurabili per ambiente tramite *appsettings*;
- **Arricchimento automatico delle *trace* ed *exemplar***: è stato implementato un *processor* OTel centralizzato che intercetta ogni *span* radice e ne propaga `enduser.id` ed `enduser.name`, e sono stati abilitati gli *exemplar* sulle metriche di tipo *Histogram*;
- **Configurazione di Grafana LGTM e produzione delle dashboard**: lo stack è stato configurato con i relativi *backend* di *storage* (Tempo, Loki, Prometheus) e le dashboard sono state versionate nel *repository* tramite *provisioning* dichiarativo, rendendo navigabile la catena diagnostica dal grafico alla singola richiesta;
- **Infrastruttura di *testing* backend**: è stata introdotta da zero una suite comprensiva di *unit test* sui *controller*, *integration test* di tipo *smoke*, *lifecycle* (CRUD/Grid) e *summary*; la sola messa in funzione della prima *suite* ha portato alla luce alcuni difetti applicativi latenti che si sarebbero manifestati esclusivamente presso un cliente specifico, in produzione;
- **Test *frontend***: sono stati prodotti *unit test* con *Vitest* per la logica dei componenti *Angular* e test *end-to-end* con *Playwright* per i flussi utente principali, integrati nel processo di *build*;
- **Agenti AI e *skill* procedurali**: è stata prodotta una suite di *Claude skill* e agenti AI che codificano in forma riutilizzabile le convenzioni del *framework*, garantendo la replicabilità della propagazione di telemetria e test ai diversi moduli.

Oltre a quanto previsto in fase di pianificazione, il tempo liberato grazie all'agente AI ha permesso l'approfondimento di funzionalità non precedentemente preventivate:

- L'arricchimento automatico delle *trace* con l'identità dell'utente;
- L'integrazione degli *exemplar*, che collegano ogni campione di metrica al `TraceId` della richiesta che lo ha generato;
- Grazie alle *skill* e agli agenti AI sviluppati, è stato possibile estendere l'istrumentazione a moduli inizialmente non preventivati;

- Le dashboard Grafana sono state arricchite con pannelli aggiuntivi per la diagnosi a grana fine (latenza per *endpoint*, distribuzione degli errori per modulo, *top users* per consumo di risorse);

I requisiti funzionali, qualitativi e di vincolo descritti nel capitolo 3 sono stati soddisfatti nella loro totalità, compresi quelli desiderabili e quelli aggiuntivi proposti in corso d'opera. Le tabelle complete di tracciamento sono consultabili nell'appendice A.
Oltre al risultato architetturale, la telemetria introdotta ha permesso di individuare alcuni problemi reali del *framework* che prima rimanevano nascosti. In particolare sono emerse alcune *stored procedure* sensibilmente più lente del previsto su richieste sincrone, episodi di *pool starvation* sulle connessioni al database in concomitanza con il traffico di *long-polling*, e alcune chiamate API ridondanti originate dal *frontend* che ripetevano la stessa richiesta a ogni cambio di vista. L'identificazione di questi casi, prima invisibili nei canali di monitoraggio esistenti, ha consentito di migliorare le performance prima che producessero impatto sugli utenti finali.

8.1.1 Integrazione della telemetria in produzione

Per verificare che il sistema di osservabilità fosse in grado non solo di raccogliere segnali ma di guidare una diagnosi reale, è stata condotta un'analisi su sette giorni di un installazione reale del CRM aziendale.
Il sintomo più evidente emerso dall'aggregazione delle trace per *route* HTTP è stata la presenza di tempi di risposta nell'ordine delle decine di secondi su un sottoinsieme ristretto ma frequentato di *endpoint*. La Tabella 8.1 riporta i casi più significativi.

Tabella 8.1: Endpoint HTTP più lenti osservati in 7 giorni (durata $> 2\,s$)

Endpoint	Occorrenze	Massimo	p95	Avg
`GET api/Layout/layout`	535	47.7 s	40.6 s	13.2 s
`POST api/grid-core/page`	101	77.4 s	68.6 s	22.7 s
`POST api/calendar/activities`	39	150.3 s	148.7 s	22.9 s
`GET api/WidgetData/fullkvp`	120	68.3 s	38.5 s	10 s

Il caso più frequente, `GET api/Layout/layout`, invocato 535 volte con tempi superiori a 2 secondi nell'arco di una settimana, è anche il più rappresentativo della categoria di problemi riscontrati: un endpoint apparentemente semplice (il recupero della configurazione di un *layout* tramite chiave primaria) che si comporta come un'operazione pesante.

8.1.1.1 Analisi di alcuni problemi emersi e relative cause

La forza dell'architettura di osservabilità descritta nei capitoli precedenti, in particolare la possibilità di scendere dalla trace aggregata fino al singolo span SQL, ha permesso di isolare la causa di due dei problemi più impattanti.
Esaminando la trace di una singola chiamata a `api/Layout/layout` durata 47.6 secondi, l'unico span figlio di tipo SQL osservato è una query `SELECT * FROM`

`[component].Layouts WHERE id = @Id`, della durata di 36.8 secondi. Si tratta di una lettura per chiave primaria su una singola riga: in condizioni normali, un'operazione di questo tipo non dovrebbe richiedere più di qualche decina di millisecondi.
La telemetria non elimina i problemi, ma restringe lo spazio delle ipotesi. Senza la trace, il sintomo osservabile sarebbe stato solo "l'apertura della pagina è lenta"; con la trace, il problema è stato circoscritto a una query specifica su una tabella specifica, lasciando alla fase di analisi SQL, il compito di stabilire la causa. A seguito dell'analisi, è stato notato come la causa più plausibile fosse uno *spike* di richieste generate dalla mancanza di salvataggio in cache delle traduzioni già recuperate. Ciò provocava un accodamento di richieste superflue ad ogni caricamento di risorse testuali con traduzione allegata. Questa inefficienza portava ad un esaurimento del *pool* di connessioni.
Dopo l'intervento dedicato al risolvere il seguente problema, si è notato un incremento nelle prestazioni di caricamento delle varie pagine.
Il secondo caso riguarda l'endpoint `POST api/calendar/activities`, osservato con una durata massima di 150 secondi. Come nella maggioranza delle chiamate lente ai vari endpoint anche qui la trace ha mostrato quasi tutta la durata della richiesta concentrata in un singolo span SQL, in questo caso però relativo all'esecuzione di una *stored procedure* (139.6 secondi su 148.7 totali).
La latenza così elevata è sicuramente influenzata dalla *pool starvation* descritta precedentemente, ma la causa principale, in questo caso, è la composizione della *stored procedure* che presenta molte join tra tabelle, causando un rallentamento anche a livello di calcolo computazionale.

8.1.1.2 Impatto atteso dagli interventi correttivi

Alcuni interventi correttivi sono già stati applicati e validati durante lo stage: tra questi, la risoluzione delle chiamate ripetitive per la traduzione di testo, che la telemetria aveva resa visibile come fonte di overhead non necessarioLe correzioni apportate al codice hanno eliminato il collo di bottiglia, risolvendo la lentezza nel caricamento delle pagine con elevato contenuto testuale. Per la maggior parte dei problemi individuati nel caso di studio, tuttavia, non è stato possibile riportare un confronto prima/dopo misurato: lo stage si è concluso prima della pianificata scalatura verticale del server utilizzato dall'azienda per l'installazione analizzata, intervento necessario per validare in modo affidabile l'effetto delle correzioni proposte.
Va inoltre precisato che non è mai stato un obiettivo dello stage apportare direttamente modifiche al codice applicativo per migliorarne le prestazioni, per quanto l'occasione si sarebbe rivelata un'ottima fonte di riflessioni e conferme. Il valore di questa analisi sta piuttosto nell'aver dimostrato che l'infrastruttura di osservabilità progettata nei capitoli precedenti è in grado di guidare una diagnosi reale fino al livello della singola query SQL.

8.2 Conoscenze acquisite e analisi del lavoro svolto

Il tirocinio svolto ha permesso un arricchimento principalmente delle mie conoscenze, permettendo di apprendere ed affrontare le implicazioni dell'osservabilità in un *framework enterprise* reale e di comprendere come una pratica di lavoro AI-assistita

disciplinata possa rappresentare, se ben applicata, un passo importante per la produttività e per la qualità del software prodotto. Il punto più rilevante di questo tirocinio è stato certamente l'integrazione di un sistema di osservabilità in un *framework* che ne era sprovvisto, accompagnata dalla nascita di un'infrastruttura di *testing* a copertura della logica e dei dati del *framework* stesso.
L'attività è stata impegnativa e, nonostante il supporto del team aziendale, la parte progettuale e di effettivo sviluppo ha richiesto uno studio mirato di standard e linguaggi di query (OTel, PromQL, TraceQL, LogQL) non affrontati nel percorso accademico, oltre alla comprensione delle convenzioni interne del *framework Sia.Core*. Le scelte implementative e la loro realizzazione sono state attività intense ma sono state affrontate con successo, permettendomi di acquisire conoscenze e competenze in ambiti di osservabilità, *testing* automatico e *AI engineering* decisamente importanti per un percorso di Laurea Triennale come il mio.
Di massima, è possibile descrivere le principali conoscenze maturate nell'ambito progettuale:

- **standard OTel e modello dei segnali**: la comprensione del modello a *trace*, metriche, *log* ed *exemplar*, delle convenzioni semantiche e del protocollo di trasporto OTLP, ha permesso di progettare una telemetria coerente con un linguaggio comune adottato dall'intera comunità OTel, evitando soluzioni proprietarie e garantendo la portabilità dei segnali verso qualsiasi backend OTel-compatibile;
- **stack *Grafana LGTM* e linguaggi di query**: lo studio di *Loki*, *Tempo* e *Prometheus*, con i rispettivi linguaggi PromQL, TraceQL e LogQL, ha permesso di comprendere i *trade-off* di ciascun *backend* (cardinalità in Prometheus, *retention* dei segnali, indicizzazione in Loki) e di costruire dashboard navigabili che sfruttano la correlazione *cross*-segnale, elemento centrale di una vera osservabilità;
- **libreria *Serilog* e *sink* OTLP**: l'integrazione di *Serilog* con esportazione OTLP e la propagazione automatica di `TraceId` e `SpanId` hanno consentito di ottenere una correlazione *log-trace* senza modifiche nel codice applicativo, un risultato che dimostra come scelte architetturali centralizzate possano avere un impatto trasversale sull'intero *framework*;
- **infrastruttura di *testing***: lo studio di *xUnit*, *NSubstitute*, *FluentAssertions* per il *backend* e di *Vitest* e *Playwright* per il *frontend*, unito alla progettazione di Fixture riutilizzabili (`SiaWebApplicationFactory`) e di scenari *lifecycle data-driven* per CRUD e Grid, ha permesso di acquisire competenze concrete su un'attività, la scrittura di test su un *framework* esistente, spesso solo accennata nei corsi universitari;
- **applicazione di *design pattern* a un contesto reale**: l'uso del pattern `SiaXxxDiagnostics` per ogni modulo, della *factory* per la registrazione delle sorgenti nel *pipeline* e dell'*adapter* per integrare *Serilog* con OTel, ha permesso di osservare in concreto come pattern studiati in modo astratto durante il corso di Ingegneria del Software trovino una loro naturale applicazione in un *framework* condiviso;

- **metodologie di lavoro AI-assistite**: la creazione di agenti specializzati, il *prompt* e *skill engineering*, l'adozione del MCP per ridurre le "allucinazioni" lavorando su dati reali, sono competenze del tutto nuove rispetto al percorso accademico ed estremamente attuali in un mercato del lavoro che integra sempre più intensamente strumenti di intelligenza artificiale nei processi di sviluppo.

8.3 Scenari di applicabilità e sviluppi futuri

Il sistema di osservabilità realizzato e l'infrastruttura di *testing* introdotta rappresentano un punto di partenza importante per lo sviluppo ulteriore del *framework Sia.Core* e si prestano a essere estesi in diverse direzioni.
Sul piano operativo della telemetria, un primo sviluppo riguarda l'aspetto della ***retention* dei dati**. Ogni segnale ha caratteristiche di *storage* differenti: le *trace* su *Tempo* sono voluminose per singolo campione ma a bassa densità (sono campionate) e si prestano a una *retention* medio-breve (7–14 giorni), sufficiente per la diagnosi di incidenti recenti; le metriche su *Prometheus* sono aggregate e compatte, e la *retention* può estendersi a 90 giorni o più, permettendo il confronto di tendenze nel tempo; i *log* su *Loki* sono voluminosi ma indicizzati per *label*, con *retention* tipica di 30 giorni e possibilità di archiviazione a lungo termine in *storage* a basso costo. I tempi concreti di *retention* potranno essere configurati in funzione delle esigenze di indagine e del costo di *storage* accettabile per il singolo cliente.
L'analisi dei dati telemetrici raccolti consentirà di individuare eventuali nuovi colli di bottiglia e anomalie prestazionali emerse durante l'utilizzo del sistema in ambienti reali. In presenza di rallentamenti o comportamenti anomali, sarà quindi possibile **estendere progressivamente il livello di strumentazione**, introducendo ulteriori segnali di telemetria e aumentando il grado di dettaglio del tracciamento nelle componenti maggiormente critiche.
Questo approccio incrementale permetterà di approfondire specifiche porzioni della logica applicativa responsabili delle degradazioni prestazionali, facilitando l'identificazione delle cause alla radice e supportando interventi di ottimizzazione più mirati. Inoltre, la disponibilità di metriche, *trace* e log correlati consentirà di analizzare il comportamento del sistema da prospettive differenti, migliorando la capacità diagnostica e riducendo i tempi necessari all'individuazione dei problemi.
Un ulteriore sviluppo dell'infrastruttura di telemetria riguarda la definizione di soglie operative (*threshold*) sui principali indicatori di prestazione, con l'obiettivo di **introdurre un sistema di *alerting* automatico**.
Attraverso strumenti come *Grafana Alerting* e le metriche raccolte tramite *OpenTelemetry*, sarebbe possibile configurare notifiche automatiche verso i team di sviluppo e manutenzione in presenza di anomalie, quali picchi di latenza, aumento del tasso di errori o degradazione delle prestazioni di specifici moduli applicativi.
Un sistema di questo tipo consentirebbe di adottare un approccio maggiormente proattivo al monitoraggio, riducendo i tempi di individuazione dei problemi e permettendo interventi tempestivi prima che i rallentamenti abbiano un impatto significativo sugli utenti finali. Inoltre, la possibilità di definire soglie differenti in base al modulo o al contesto di utilizzo renderebbe il monitoraggio più flessibile e adattabile ai diversi scenari applicativi.

Sul piano della metodologia di sviluppo AI-assistita adottata durante lo stage, uno degli sviluppi futuri più promettenti riguarda l'evoluzione dell'agente dedicato alla telemetria.
L'obiettivo è estenderne le capacità oltre la semplice raccolta o propagazione dei segnali osservabili, introducendo funzionalità di **analisi automatica, interpretazione dei dati e generazione di report** sintetici destinati ai team di sviluppo.
In prospettiva, l'agente potrebbe essere istruito a correlare metriche, *trace* e *log* provenienti dai diversi moduli del framework, individuando automaticamente anomalie, regressioni prestazionali o pattern ricorrenti di errore. A partire da tali informazioni, sarebbe possibile produrre report riassuntivi leggibili anche senza una conoscenza approfondita degli strumenti di osservabilità.
Un approccio di questo tipo consentirebbe ai developer di ottenere una panoramica immediata dello stato del framework, riducendo la necessità di interrogare manualmente dashboard, query PromQL o sistemi di tracing.
L'agente assumerebbe quindi un ruolo più vicino a quello di un assistente operativo per l'osservabilità, capace non solo di raccogliere dati, ma anche di trasformarli in indicazioni utili e direttamente azionabili.
Nel lungo periodo, tale evoluzione potrebbe portare alla realizzazione di sistemi semi-autonomi di analisi, in grado di suggerire automaticamente nuove strumentazioni, individuare aree del codice che richiedono approfondimenti telemetrici o proporre interventi di ottimizzazione sulla base dei pattern osservati in produzione.

8.4 Valutazione personale

L'attività di stage si è rivelata molto utile per la crescita personale e professionale, permettendo di acquisire nuove conoscenze e competenze in ambiti di osservabilità, *testing* automatico e metodologie di lavoro AI-assistite, che sono stati approfonditi e applicati in modo concreto su un *framework enterprise* reale. L'ambito dell'osservabilità è stato studiato in modo approfondito, permettendo di esplorare in autonomia un campo con ripercussioni decisamente interessanti nel mondo dello sviluppo software, non visto né praticato in corsi universitari, e che si è rivelato un'esperienza assolutamente rilevante.
Spesso ambiti sconosciuti possono essere rilevanti da un punto di vista professionale, e questo è stato un esempio di come andare oltre i preconcetti e immergersi in un ambito ampiamente teorico, come la progettazione di una tassonomia di metriche con cardinalità controllata o l'uso degli *exemplar* per collegare metrica e *trace*, si riveli in realtà un'occasione di crescita e di apprendimento, richiedendo necessariamente di praticare a fondo uno studio mirato delle attività svolte. In questi casi, la passione per il conoscere guida e anticipa la necessità di apprendere, dando un'importante possibilità di conoscere e maturare, altrimenti non offerta in modo così vicino alle realtà aziendali rimanendo in ambito puramente accademico.
Particolarmente significativa è stata la possibilità di lavorare a stretto contatto con un *framework enterprise* reale, cimentandosi sia con la progettazione (scelte di cardinalità, *sampling*, correlazione *cross*-segnale) sia con la creazione di agenti AI specializzati e di *skill* procedurali. Questa seconda dimensione, in particolare, ha trasformato il mio modo di intendere il rapporto fra sviluppatore e strumenti di intelligenza artificiale: non più un assistente che risponde a richieste puntuali, ma

un *pair engineer* disciplinato dalle convenzioni del *framework*, in grado di propagare in modo coerente uno stesso pattern a decine di moduli con una qualità di output verificabile dallo sviluppatore. La collaborazione con il *team* aziendale e il confronto continuo sulle scelte architetturali sono stati parte integrante del percorso formativo, con un'attività di supporto costante che ha lasciato ampia libertà di organizzazione e di sviluppo autonomo, restando vicino alle mie necessità di studente.
Il progetto rappresenta per me una crescita che matura quanto compreso e visto in questi ultimi anni, permettendo di applicare in modo nuovo concetti per la maggior parte sconosciuti e con tecnologie attuali, calandomi in un contesto *enterprise* concreto, usando librerie e tecnologie di riferimento e sviluppando un pensiero critico nell'analisi delle scelte di osservabilità, affinando ulteriormente una visione d'insieme dei corsi fin qui frequentati nel mio percorso di laurea triennale, perfezionando il mio metodo di studio e il mio modo di affrontare un progetto di codifica con ripercussioni operative in produzione. Quanto studiato rappresenta un'importante partenza per il mio futuro, accademico e non, aggiornandomi su un ambito decisamente attuale, l'osservabilità dei sistemi distribuiti e l'*AI engineering*, di sicuro interesse per possibili sviluppi futuri.

Appendice A

Tracciamento completo dei requisiti

In questa appendice sono raccolte le tabelle complete dei requisiti (funzionali, qualitativi e di vincolo) con il relativo tracciamento sui casi d'uso. La struttura del codice identificativo dei requisiti è descritta nella sezione 3.3.

Tabella A.1: Tracciamento dei requisiti funzionali

Requisito	Descrizione	Use Case
RFO-1	Tracciamento del percorso completo di ogni richiesta HTTP in ingresso, incluse le operazioni di accesso al database	UC5, UC6
RFO-2	Le operazioni CRUD verso il database devono essere riconoscibili tramite attributi che espongono il metodo e la *repository* da cui è stata scatenata la richiesta	UC5, UC6
RFO-3	Esposizione di metriche operative per i moduli centrali del *framework* (autenticazione, componenti *Angular* come griglie e *form*, prestazioni hardware, accesso al database)	UC5, UC6
RFO-4	I *log* devono essere correlabili alle *trace* tramite `TraceId` e `SpanId`	UC6, UC7
RFO-5	Il sistema deve essere attivabile e disattivabile tramite configurazione, senza richiedere modifiche al codice	UC5
RFD-6	Configurazione indipendente del campionamento per ciascuna sorgente di *trace*	UC5
RFO-7	Visualizzazione di *trace* e metriche raggruppate in dashboard *Grafana* costruite sullo stack *LGTM*	UC6, UC9

RFO-8	Le dashboard di *Grafana* devono raccogliere i dati da più servizi e installazioni di clienti diversi, e permettere di filtrare per gli stessi	UC6
RFD-9	Il sistema deve arricchire automaticamente ogni *span* con l'identità dell'utente autenticato (`enduser.id`, `enduser.name`) senza richiedere modifiche ai singoli moduli applicativi	UC8
RFO-10	Le metriche di tipo *Histogram* devono esporre *exemplar* (puntatori al `TraceId` del campione) per abilitare la navigazione dalla metrica aggregata alla *trace* corrispondente	UC7
RFO-11	Devono essere prodotti *unit test* per la logica di business dei moduli core del *framework backend*, eseguibili in modo automatizzato	UC1, UC4
RFO-12	Devono essere prodotti *integration test* per la verifica dell'attendibilità dei dati presenti nel database, delle procedure SQL e delle prospettive di configurazione dei componenti	UC2, UC4
RFO-13	Devono essere prodotti *unit test* lato *frontend* per la verifica della logica applicativa dei componenti *Angular*	UC3, UC4
RFO-14	Devono essere prodotti test *end-to-end* lato *frontend* per verificare il corretto comportamento e la corretta visualizzazione dell'applicazione dal punto di vista dell'utente finale	UC3, UC4
RFO-15	La suite di test deve essere integrabile nel processo di *build* e fornire un report di copertura del codice	UC4
RFO-16	Devono essere realizzati agenti AI specializzati e *skill* procedurali per propagare in modo uniforme telemetria e test ai diversi moduli del *framework*	UC1, UC2, UC3, UC5, UC9

Tabella A.2: Tracciamento dei requisiti qualitativi

Requisito	**Descrizione**	**Fonte**
RQO-1	La telemetria non deve degradare sensibilmente le prestazioni in produzione	Aziendale
RQO-2	Le metriche devono essere progettate per non generare un numero eccessivo di serie temporali in Prometheus; i tag ad alta variabilità (es. identificativi utente o di record) non devono essere usati come attributi di metriche	Aziendale

RQO-3	Le dashboard Grafana devono essere versionabili insieme al codice del modulo che strumentano	Aziendale

Tabella A.3: Tracciamento dei requisiti di vincolo

Requisito	**Descrizione**	**Fonte**
RVD-1	I segnali prodotti devono essere compatibili con qualsiasi backend OTel-compatibile	Aziendale
RVO-2	Per lo sviluppo dei test di unità e integrazione del backend devono essere utilizzati dei pacchetti NuGet standard di .NET	Aziendale
RVO-3	I test di unità del frontend devono utilizzare Vitest come tecnologia di sviluppo	Aziendale
RVO-4	I test end-to-end del frontend devono utilizzare Playwright come tecnologia di sviluppo	Aziendale
RVO-5	Lo stack di osservabilità deve utilizzare OpenTelemetry come protocollo di raccolta dei segnali	Stagista
RVO-6	La piattaforma di visualizzazione deve essere Grafana con stack *LGTM*	Stagista
RVO-7	Le impostazioni devono essere modificabili per ambiente (sviluppo, *staging*, produzione) tramite *appsettings*	Aziendale

Appendice B

Diagrammi delle classi del sistema di telemetria

In questa appendice sono raccolti i diagrammi delle classi del sistema di telemetria descritti, in forma narrativa, nel capitolo 5. I diagrammi adottano uno stile UML semplificato: rettangoli con nome e membri salienti, frecce tratteggiate per le dipendenze d'uso, freccia con triangolo vuoto per la relazione di implementazione di interfaccia.

B.1 Pipeline condiviso in `BaseApplicationBuilder`

La figura B.1 mostra la pipeline OTel centralizzato: i tre provider `TracerProvider`, `MeterProvider` e `LoggerProvider` sono costruiti una sola volta da `BaseApplication Builder.ConfigureBuilder`.
Sul `TracerProvider` viene installato il `SiaParentBasedSampler` (con `SamplingRatio` globale e override per-source via `SamplingRatios`), seguito dal `BaggageActivityProcessor` che copia i valori di Baggage con prefisso `"enduser."` come tag su ogni *span* figlia; le route rumorose (es. `api/base-hub`) sono soppresse tramite `options.Filter` sull'instrumentation ASP.NET Core e tramite il middleware `SuppressOptionsActivityMiddleware`.
Il `LoggerProvider` corrisponde al *sink* OTLP di Serilog, con `LogsMinimumLevel` configurabile e `IncludedData` comprendente `TraceId`, `SpanId` e *message template* (testo del log).
Sul `MeterProvider` è attivo l'*exemplar filter* `TraceBased` e quattro configurazioni custom: `ExplicitBucketHistogramConfiguration` per estendere la coda dei bucket oltre il default da 10s.
Tutti i segnali condividono lo stesso `Resource` con gli attributi `service.name`, `sia.product_type` e `sia.customer_name`, ed esportano verso il *Collector* via `OtlpExporter` gRPC sull'endpoint `OpenTelemetry:OtlpEndpoint`.
Due middleware completano la pipeline a runtime: `SuppressOptionsActivityMiddleware` (soppressione preflight) e `EnrichActivityWithUserMiddleware` (arricchimento con `enduser.id` dalla claim `userId`); opzionalmente `HttpRequestBodyTagActionFilter` cattura il *body* della *request* HTTP (GET e POST) come tag `sia.request.body` sulla root span.

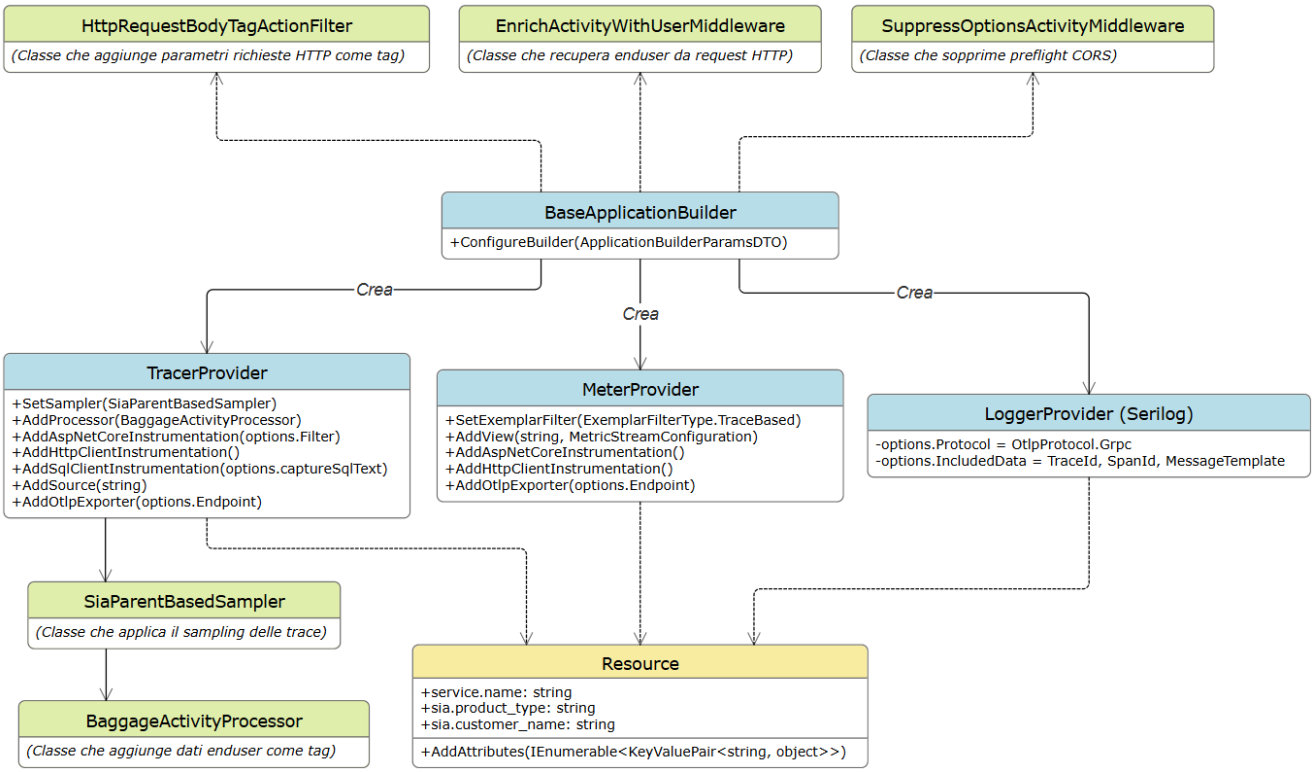

Figura B.1: Diagramma delle classi della Pipeline OpenTelemetry

B.2 Pattern Sia{Modulo}Diagnostics e Adapter

La figura B.2 illustra in forma generica la classe `Sia{Modulo}Diagnostics` che ogni modulo del framework espone. La classe è l'unico punto di istanziazione degli strumenti OTel del modulo (pattern *Static Factory*); inoltre, per i moduli che espongono un controller, ospita un `ControllerDiagnosticsAdapter` annidato che implementa il contratto condiviso `IControllerDiagnostics` (pattern *Adapter*). Il controller chiama il metodo condiviso `BaseApiController.RecordMetricError` passando il singleton `AsControllerDiagnostics`, senza conoscere l'implementazione concreta.

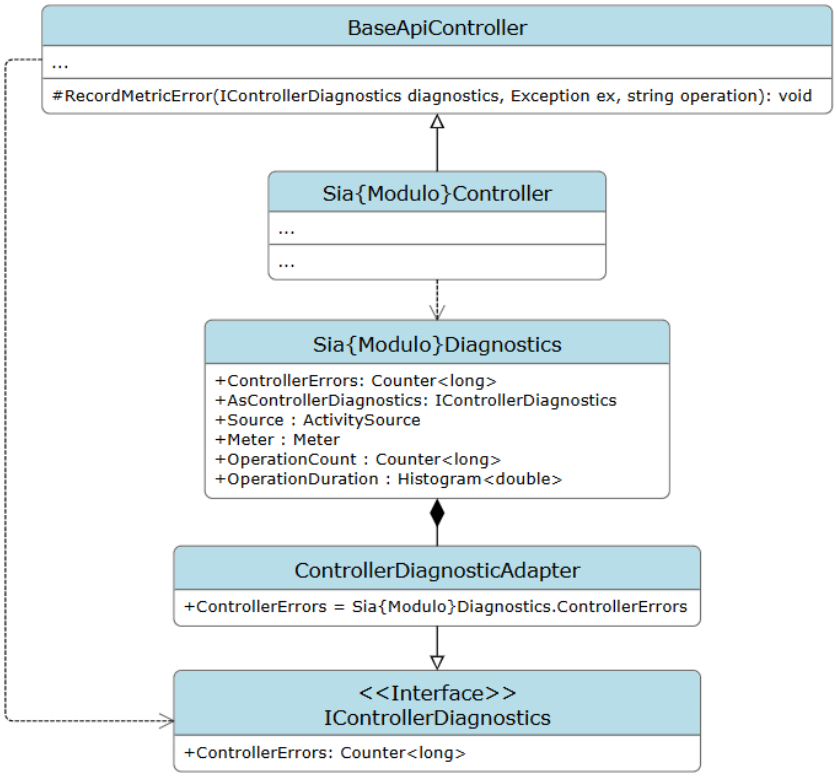

Figura B.2: Diagramma delle classi del pattern di diagnostica

B.3 Arricchimento del contesto utente

La figura B.3 mostra le collaborazioni che realizzano l'arricchimento automatico di ogni `Activity` con i *tag* `enduser.id` e `enduser.name`. Il `EnrichActivityWithUserMiddleware` legge gli attributi da `HttpContext.User` e pubblica le voci nel Baggage; il `BaggageActivityProcessor`, registrato come `ActivityProcessor` nel `TracerProvider`, copia ogni *entry* di Baggage con prefisso `"enduser."` come *tag* sull'`Activity` nel proprio *hook* `OnStart`. L'effetto è che qualsiasi `Activity` creata durante la richiesta, sia dalle classi `Sia{Modulo}Diagnostics`, sia dalle strumentazioni automatiche di ASP.NET Core e `SqlClient`, riceve i *tag* utente senza modifiche al codice del modulo.

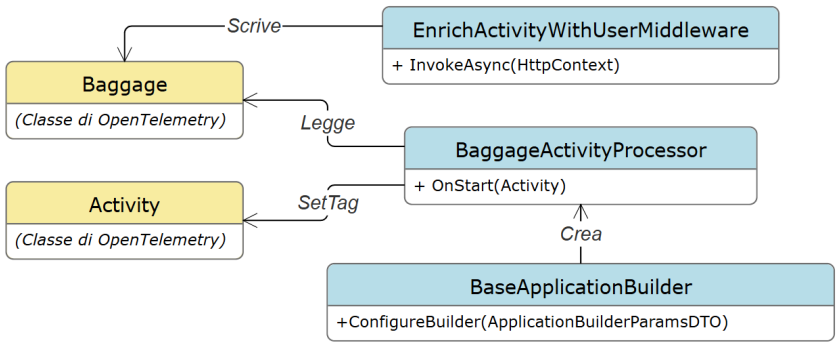

Figura B.3: Diagramma delle classi dell'arricchimento trace con dati utente

Appendice C

Listati di codice rappresentativi

In questa appendice sono raccolti i frammenti di codice più estesi citati in forma di rinvio nei capitoli del corpo principale. I frammenti sono pensati come riferimento canonico: ogni nuovo modulo del *framework* li adotta con minime variazioni di *naming*.

C.1 Pattern Adapter applicato alla diagnostica del modulo CRUD

Il listato C.1 mostra la classe statica `Sia{Modulo}Diagnostics` per il modulo CRUD, con il `ControllerDiagnosticsAdapter` annidato che riconcilia il `Counter` statico del modulo con il contratto comune `IControllerDiagnostics`. Il listato C.2 mostra il punto di chiamata, ridotto a una riga nel *controller* derivato.

Listing C.1: Pattern Adapter applicato alla diagnostica del modulo CRUD

```
public static class SiaDevextremeCrudDiagnostics
{
    public static readonly Meter Meter =
        new("Sia.Devextreme.Crud", "1.0.0");

    public static readonly Counter<long> ControllerErrors =
        Meter.CreateCounter<long>(
            "sia.devextreme.crud.controller.errors");

    public static readonly IControllerDiagnostics
        AsControllerDiagnostics = new
           ControllerDiagnosticsAdapter();

    private sealed class ControllerDiagnosticsAdapter
        : IControllerDiagnostics
    {
        public Counter<long> ControllerErrors =>
            SiaDevextremeCrudDiagnostics.ControllerErrors;
    }
}
```

Listing C.2: Invocazione di `RecordMetricError` dal controller derivato

```
catch (Exception ex)
{
    RecordMetricError(
        SiaDevextremeCrudDiagnostics.AsControllerDiagnostics,
        ex,
        nameof(GetData));
    throw;
}
```

Acronimi e abbreviazioni

API Application Program Interface. 7–9, 29, 33, 34, 37, 49, 52, 61, 77

CRM Customer Relationship Management. 1, 77

CRUD Create, Read, Update, Delete. 26, 38, 43, 44, 47, 63, 67, 74, 77

ERP Enterprise Resource Planning. 1, 77

LLM Large Language Model. 7, 12, 31, 46–51, 78

MCP Model Context Protocol. iii, 3–6, 9, 11–13, 15, 16, 25, 47, 49–51, 59, 64, 78

OTel OpenTelemetry. 5, 7, 9, 11, 13, 20, 27, 33, 35–37, 39, 41, 52, 54, 59, 60, 63, 69, 70, 72

OTLP OpenTelemetry Protocol. 9, 34, 35, 46, 47, 53, 63, 70, 78

PoC Proof of Concept. 28, 47, 48, 78

RFID Radio-Frequency Identification. 1, 78

SDK Software Development Kit. 46, 79

SPA Single Page Application. 29, 79

UML Unified Modeling Language. 12, 41, 70, 79

Glossario

API in informatica con il termine *Application Programming Interface API* (ing. interfaccia di programmazione di un'applicazione) si indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma. La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione. 76

Baggage in OpenTelemetry, meccanismo di propagazione di coppie chiave-valore lungo il contesto di esecuzione di una richiesta. A differenza degli attributi di *span*, le voci di *baggage* possono essere lette e scritte da qualsiasi punto del codice e sono ereditate automaticamente dalle operazioni figlie. 39, 54, 70, 73

CRM con *CRM, Customer Relationship Management* si indica un sistema o insieme di strumenti software utilizzati per gestire le relazioni e le interazioni con clienti attuali e potenziali. Un CRM consente di organizzare e analizzare informazioni relative ai clienti, supportare le attività di vendita, marketing e assistenza, e migliorare la qualità del servizio e la fidelizzazione. 61, 76

CRUD con *CRUD, Create, Read, Update, Delete* si indica l'insieme delle quattro operazioni fondamentali per la gestione dei dati in un sistema informatico. Rappresenta le azioni di creazione, lettura, aggiornamento ed eliminazione di entità all'interno di un database o di un'applicazione. Queste operazioni costituiscono la base per la maggior parte delle funzionalità di persistenza e manipolazione dei dati. 76

ERP con *ERP, Enterprise Resource Planning* si indica un sistema informativo aziendale integrato utilizzato per gestire e automatizzare i principali processi operativi di un'organizzazione, quali contabilità, gestione delle risorse umane, produzione, logistica e vendite. 76

Exemplar in OpenTelemetry, campione concreto allegato a un dato aggregato di metrica, che porta con sé il `TraceId` (e opzionalmente il `SpanId`) della richiesta che lo ha prodotto. Consente la navigazione diretta dalla metrica aggregata alla *trace* corrispondente. 8

Fixture nel contesto del testing software, una fixture è l'insieme di condizioni, dati, oggetti e configurazioni necessari per eseguire un test in modo controllato e

ripetibile. Una fixture prepara l'ambiente prima del test (setup) e, spesso, lo ripristina o pulisce dopo l'esecuzione. 63

gRPC Tecnologia per la realizzazione di API e comunicazioni tra servizi distribuiti basata su HTTP/2 e Protocol Buffers. Consente lo scambio efficiente di dati tra applicazioni con elevata velocità, ridotto consumo di banda e supporto nativo allo streaming bidirezionale. 35

LLM con *LLM, Large Language Model* si indica un modello di intelligenza artificiale basato su reti neurali di grandi dimensioni, addestrato su ampie quantità di dati testuali, in grado di comprendere e generare linguaggio naturale. 76

MCP *MCP, Model Context Protocol* è un protocollo che consente a un agente basato su modelli di intelligenza artificiale di interagire con sistemi e servizi esterni in modo strutturato, permettendo l'accesso a dati dinamici e contestuali. MCP abilita l'integrazione tra modelli linguistici e strumenti operativi, migliorando la capacità decisionale dell'agente e riducendo la dipendenza da informazioni statiche. 51, 76

OTLP protocollo binario e testuale, definito dallo standard OpenTelemetry, utilizzato per il trasporto di *trace*, metriche e *log* tra applicazioni, *Collector* e *backend* di *storage*. 76

PoC con *PoC, Proof of Concept* si indica un prototipo o implementazione preliminare sviluppata per verificare la fattibilità di una soluzione tecnica o di un'idea progettuale. Un PoC ha lo scopo di validare determinati aspetti, come prestazioni, integrazione o corretto funzionamento, prima di procedere con uno sviluppo completo. 76

RFID con *RFID, Radio-Frequency Identification* si indica una tecnologia che utilizza onde radio per identificare e tracciare oggetti, persone o animali attraverso dispositivi chiamati tag RFID. Questi tag contengono informazioni che possono essere lette a distanza da appositi lettori, senza necessità di contatto diretto o visibilità, rendendo la tecnologia particolarmente utile in ambiti come logistica, tracciamento e gestione degli inventari. 76

Sampler in OpenTelemetry, componente del pipeline di *trace* che decide, per ogni nuova operazione, se registrarla ed esportarla (*sampled*) oppure scartarla. La scelta può basarsi su criteri fissi (ratio probabilistico) o dinamici (es. presenza di errori, durata). 33, 36, 41, 47, 52, 53, 60

SCRUM metodologia agile per la gestione e lo sviluppo di progetti, basata su un approccio iterativo e incrementale. Il lavoro viene suddiviso in cicli temporali chiamati Sprint, al termine dei quali viene prodotto un incremento funzionante del sistema. Scrum definisce ruoli (come Scrum Master e Product Owner), eventi (Sprint, Daily Scrum, Sprint Review e Retrospective) e artefatti (Product Backlog e Sprint Backlog) per favorire collaborazione, adattabilità e miglioramento continuo del processo di sviluppo. 6

SDK con *SDK, Software Development Kit* si indica insieme di strumenti, librerie, documentazione e API messi a disposizione per facilitare lo sviluppo di applicazioni su una determinata piattaforma o tecnologia. Un SDK fornisce componenti riutilizzabili e funzionalità predefinite che permettono agli sviluppatori di integrare rapidamente servizi specifici, riducendo tempi e complessità di sviluppo. 76

SPA con *SPA, Single Page Application* si indica un'applicazione web costituita da una singola pagina HTML che viene aggiornata dinamicamente senza ricaricare completamente il contenuto. Le SPA migliorano l'esperienza utente rendendo la navigazione più fluida e veloce, grazie al caricamento asincrono dei dati e alla gestione lato client della logica applicativa. 76

UML in ingegneria del software *UML, Unified Modeling Language* (ing. linguaggio di modellazione unificato) è un linguaggio di modellazione e specifica basato sul paradigma object-oriented. L'*UML* svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa tale linguaggio per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico. 76

Bibliografia

Riferimenti bibliografici

Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, MA, USA: Addison-Wesley Professional, 1994. ISBN: 9780201633610 (cit. a p. 40).

Siti web consultati

Agile Alliance. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/iso/it/.

Anthropic, PBC. *Claude Code — Agentic Coding Tool*. URL: https://www.anthropic.com/claude-code.

— *Model Context Protocol — Specification*. URL: https://modelcontextprotocol.io/.

GitHub, Inc. *Spec-Kit — Spec-Driven Development Toolkit*. URL: https://github.github.com/spec-kit/.

Google LLC. *Angular — Official Documentation*. URL: https://angular.dev/overview.

Grafana Labs. *Grafana Documentation*. URL: https://grafana.com/docs/grafana/latest/.

— *Grafana Loki Documentation — Log Aggregation System*. URL: https://grafana.com/docs/loki/latest/.

— *Grafana Mimir Documentation — Scalable Long-Term Storage for Prometheus*. URL: https://grafana.com/docs/mimir/latest/.

— *Grafana Tempo Documentation — High-Scale Distributed Tracing Backend*. URL: https://grafana.com/docs/tempo/latest/.

Microsoft Corporation. *.NET — Free, Open-Source, Cross-Platform Developer Platform*. URL: https://dotnet.microsoft.com/it-it/.

— *C# Documentation*. URL: https://dotnet.microsoft.com/it-it/languages/csharp.

Microsoft Corporation. *TypeScript — JavaScript With Syntax For Types*. URL: https://www.typescriptlang.org/.

OpenTelemetry Authors. *OpenTelemetry Metrics Data Model — Exemplars*. URL: https://opentelemetry.io/docs/specs/otel/metrics/data-model/%5C#exemplars.

— *OpenTelemetry Specification*. URL: https://opentelemetry.io/docs/specs/otel/.

OpenTelemetry Authors e Cloud Native Computing Foundation. *OpenTelemetry — High-quality, Ubiquitous, and Portable Telemetry to Enable Effective Observability*. URL: https://opentelemetry.io/.

Serilog Contributors. *Serilog.Sinks.OpenTelemetry — A Serilog Sink Sending Log Events via OTLP*. URL: https://github.com/serilog/serilog-sinks-opentelemetry.