

Università Campus Bio-Medico di Roma

Facoltà di Ingegneria

Corso di Laurea in Ingegneria dei Sistemi Intelligenti

Progetto per l'insegnamento di

Intelligent Architectures: Microservices Programming for AI

Domain Interviewer & Modeler

Agente 1 di una pipeline multi-agente per la scoperta del dominio
e la progettazione automatica di architetture a microservizi
tramite AI locale e workflow orchestrati

Studenti:

Gianluca Barnaba
Luca Scorza
Federico Principi

Docenti:

Prof. Floriano Caprio
Prof. Paolo Soda
Prof. Ermanno Cordelli
Prof. Giordano Pescetelli

Anno Accademico 2025/2026

Indice

1	Executive Summary	6
1.1	Visione Strategica	6
1.2	Value Proposition	6
1.3	Sintesi Tecnica	6
2	Analisi dello Scenario e Business Case	7
2.1	Razionale del Progetto	7
2.2	Analisi dei Pain Points	7
2.3	Rischi e Impatto Operativo	8
2.4	Business Case	8
2.5	Confronto con le Alternative Esistenti	8
3	Obiettivi e Requisiti del Sistema	10
3.1	Obiettivi Strategici	10
3.2	Requisiti Funzionali	10
3.2.1	RF-01: Pipeline di Intervista a 5 Fasi	10
3.2.2	RF-02: Validazione Automatica di Completezza	10
3.2.3	RF-03: Routing Intelligente	10
3.2.4	RF-04: Generazione Architetturale	11
3.2.5	RF-05: Raffinamento del Modello	11
3.2.6	RF-06: Gestione del Ciclo di Vita delle Bozze	11
3.2.7	RF-07: Frontend Chat	11
3.2.8	RF-08: Stato Conversazionale Persistente	11
3.3	Requisiti Non Funzionali	12
4	Stakeholder e Ruoli	13
4.1	Business Stakeholder (Utente Intervistato)	13
4.2	Agenti Intervistatori (Discovery Pipeline)	13
4.2.1	Regole Comuni agli Agenti Intervistatori	14
4.3	Agente di Validazione (Orchestrator)	14
4.4	Agenti di Elaborazione (Post-Discovery)	14
4.5	Stack di Osservabilità	15
4.6	Matrice di Tracciabilità Ruoli/Funzioni	15
5	Architettura del Sistema	16
5.1	Visione d'Insieme	17
5.2	I Pilastri Progettuali	17
5.2.1	Orchestrazione via Workflow (n8n)	17
5.2.2	State Management Centralizzato (Redis)	18
5.2.3	Agenti Specializzati con System Prompt	18
5.2.4	Osservabilità Nativa	18
5.3	Motivazioni delle scelte architetturali	19
5.4	Flusso di Comunicazione End-to-End	19
6	Deployment e Containerizzazione Docker	21
6.1	Servizi Definiti nel Docker Compose	21
6.2	Rete Docker, Comunicazione e Dipendenze	21
6.2.1	Catena di Dipendenze e Ordine di Avvio	22

6.3	Volumi e Persistenza dei Dati	22
6.4	Variabili d'Ambiente Chiave	22
6.4.1	n8n	22
6.4.2	Grafana	23
6.5	Politiche di Restart	23
7	Pipeline dei Workflow n8n	24
7.1	Mappa dei Workflow	24
7.2	Routing Workflow	24
7.2.1	Logica di Routing Dettagliata	24
7.3	Discovery Workflow	25
7.3.1	Flusso di Esecuzione	26
7.3.2	Configurazione degli Agenti	26
7.4	Refinement Workflow	27
7.4.1	Modalità Explain (Spiegazione)	27
7.4.2	Modalità Modify (Modifica)	27
7.5	Polling Workflow	28
7.6	Draft Decision Workflow	28
7.7	Comunicazione tra Workflow	28
8	Frontend Chat	29
8.1	Layout e Design	29
8.2	Gestione delle Sessioni	29
8.3	Rendering Differenziato dei Messaggi	30
8.4	Feedback Visuale e Animazioni	30
8.5	Pulsanti Inline e Selezione Modalità	30
8.6	Meccanismo di Draft con Double-Check	30
8.7	Comunicazione con il Backend	31
8.8	Gestione degli Errori	31
9	Design dei System Prompt	32
9.1	Struttura Comune dei System Prompt Intervistatori	32
9.2	Dettaglio per Fase	33
9.2.1	Fase 1 — Strategic Analysis	33
9.2.2	Fase 2 — Boundaries and Language	33
9.2.3	Fase 3 — EDA Integration	33
9.2.4	Fase 4 — Tactical Patterns	33
9.2.5	Fase 5 — Technical Excellence & Resilience	34
9.3	System Prompt dell'Orchestrator	34
9.4	System Prompt del DDD Analyst	34
9.5	System Prompt del JSON Coder	34
9.6	System Prompt degli Agenti di Refinement	35
9.6.1	Modifier	35
9.6.2	Explainer	35
10	Osservabilità	37
10.1	OpenTelemetry Collector: L'Hub Centrale	37
10.1.1	Receivers (Sorgenti Dati)	37
10.1.2	Processors (Elaborazione)	37
10.1.3	Exporters (Destinazioni)	38
10.2	Loki: Aggregazione Log	38
10.3	Prometheus: Metriche	39
10.4	Grafana: Visualizzazione e Dashboard	40

11	Problemi, Soluzioni, Limiti e Compromessi	41
11.1	Problemi Incontrati e Soluzioni	41
11.1.1	Gestione del Contesto LLM nelle Conversazioni Multi-Fase	41
11.1.2	Latenza della Generazione del Modello	41
11.1.3	Validità del JSON Generato	42
11.1.4	Anti-Bleeding tra Fasi	42
11.1.5	Evasività dell'Utente	42
11.2	Limiti del Sistema	43
11.2.1	Limiti Strutturali	43
11.2.2	Limiti Legati all'Inferenza Locale	43
11.3	Compromessi Progettuali	43
12	Conclusioni e Sviluppi Futuri	44
12.1	Sintesi dei Risultati	44
12.2	Sviluppi Futuri	44
12.2.1	Evoluzione dello Stack di Osservabilità	44
12.2.2	Validazione Automatica del JSON	45
12.2.3	Supporto Multi-Lingua	45
12.2.4	Evoluzione del Motore di Modifica Deterministico	45
12.2.5	Potenziamento della Granularità Descrittiva	45
12.2.6	Integrazione con il Pipeline Multi-Agente	45
12.2.7	Persistenza Long-Term e Versioning	45
13	Caso di Studio: DocOnTime	46
13.1	Fasi di Discovery e Intervista	46
13.2	Output e Modello Architettureale	47
13.3	Fase di Spiegazione	48
13.4	Fase di Modifica	48
13.5	Confronto Modelli JSON	49

Elenco delle figure

5.1	Architettura complessiva del sistema DIM	16
5.2	Diagramma di sequenza del flusso end-to-end: dall'intervista alla generazione del modello.	20
7.1	Struttura del Routing Workflow nell'interfaccia n8n: gestione degli accessi e branching.	24
7.2	Struttura del Discovery Workflow nell'interfaccia n8n	25
7.3	Struttura del Refinement Workflow nell'interfaccia n8n	27
8.1	Interfaccia della chat frontend	29
10.1	Architettura complessiva del sistema DIM con focus sullo stack di osservabilità	37
10.2	Grafana Dashboard — Monitoraggio dei volumi di log per ogni container Docker	38
10.3	Monitoraggio Redis — Dettaglio su memoria, client connessi e throughput (comandi/sec)	39
10.4	Metodologia di monitoraggio n8n — Dettaglio su Event Loop Lag e utilizzo Memoria (Heap/RSS)	39
10.5	Dashboard Grafana — Visualizzazione dei log strutturati estratti da Loki .	40
10.6	Dashboard Grafana — Metriche applicative n8n e stato del database Redis	40
13.1	Identificazione della Visione Strategica: l'agente delinea il Core Domain del sistema.	46
13.2	Dettaglio sui Bounded Context: l'agente richiede attributi specifici per le entità principali.	46
13.3	Conclusione dell'intervista: l'Orchestrator emette il verdetto di PASS finale.	47
13.4	Interazione con l'agente Explainer: analisi e chiarimenti sulla struttura del sistema.	48
13.5	Richiesta di modifica chirurgica inviata dall'utente.	49
13.6	Ricevuta del Modifier: riepilogo formale della modifica strutturale proposta.	49

Elenco delle tabelle

4.1	Agenti Intervistatori e relative fasi dell'intervista DDD	13
4.2	Matrice di tracciabilità Ruoli/Funzioni del sistema	15
6.1	Dettaglio completo dei servizi nel Docker Compose	21
6.2	Mappatura completa dei volumi Docker	22
7.1	Mappa completa dei workflow n8n	24
7.2	Configurazione completa degli agenti nel Discovery Workflow	26
8.1	Animazioni di feedback nel frontend	30
8.2	Endpoint di comunicazione Frontend → n8n	31
10.1	Receivers configurati nell'OpenTelemetry Collector	37
10.2	Exporters configurati nell'OpenTelemetry Collector	38
10.3	Datasource Grafana preconfigurati	40
11.1	Matrice dei compromessi progettuali	43

Capitolo 1

Executive Summary

1.1 Visione Strategica

Il **Domain Interviewer & Modeler** (DIM) è il primo agente operativo di una pipeline multi-agente denominata *Intelligent Domain Architect Agents*, concepita per automatizzare la progettazione di architetture a microservizi secondo i principi del Domain-Driven Design (DDD). Il DIM affronta il problema della scoperta e strutturazione dei requisiti di dominio, trasformando una conversazione in linguaggio naturale con uno stakeholder di business in un modello architetturale formale JSON, pronto per essere consumato dagli agenti downstream della pipeline.

1.2 Value Proposition

A differenza di un semplice chatbot o di un questionario statico, il DIM risolve il problema della raccolta e strutturazione dei requisiti di dominio attraverso tre pilastri fondamentali:

- **Intervista Intelligente Multi-Fase:** Una catena di cinque agenti LLM specializzati guida lo stakeholder attraverso le cinque fasi del Discovery DDD. Ogni agente è un esperto focalizzato su un singolo pillar architetturale. Un sesto agente, monitora in background la completezza informativa di ciascuna fase prima di autorizzare la transizione alla successiva.
- **Generazione Automatica del Modello:** Al termine dell'intervista, una catena di due agenti in cascata — il *DDD Analyst* e il *JSON Coder* — processa l'intero storico conversazionale con una politica di *Zero-Loss*, preservando ogni dettaglio tecnico emerso durante l'intervista.
- **Raffinamento Interattivo:** L'utente può rivedere, modificare e confermare il modello generato attraverso un flusso di Refinement conversazionale assistito da LLM. Un agente *Modifier* comprende i cambiamenti da applicare basandosi sulle modifiche strutturali richieste, mentre un agente *Explainer* risponde a domande di chiarimento sull'architettura corrente. Un meccanismo di Draft/Confirm/Discard con double-check di sicurezza garantisce che nessuna modifica venga applicata involontariamente.

1.3 Sintesi Tecnica

L'architettura è interamente basata su **microservizi containerizzati** orchestrati da **Docker Compose**. L'orchestrazione dei workflow di intervista, analisi e raffinamento è delegata a **n8n**, che coordina gli agenti LLM tramite **Ollama** (modelli *llama3* e *qwen2.5:14b*) e gestisce lo stato conversazionale tramite **Redis**. L'interfaccia utente è una chat web full-screen sviluppata come single-page HTML con comunicazione HTTP asincrona (REST + polling) verso i webhook n8n.

Lo stack di osservabilità, basato su **OpenTelemetry Collector**, **Loki**, **Prometheus** e **Grafana**, fornisce visibilità end-to-end su log e metriche dell'intero ecosistema.

Capitolo 2

Analisi dello Scenario e Business Case

2.1 Razionale del Progetto

In un contesto aziendale reale, la definizione del dominio applicativo e la sua suddivisione in sottodomini funzionali rappresenta il primo passo critico nella progettazione di un sistema basato su microservizi. Questo processo, tipicamente condotto manualmente da architetti esperti attraverso sessioni di Event Storming e interviste con gli stakeholder, è lungo, soggetto a bias individuali e difficilmente ripetibile in modo sistematico.

Il Domain-Driven Design (DDD) fornisce un framework concettuale potente per affrontare questa sfida, ma la sua applicazione richiede un livello di competenza specialistica che non è sempre disponibile nei team di sviluppo. Il DIM nasce per colmare questo gap: automatizzare e strutturare la fase fondamentale di Domain Discovery, rendendo accessibile a qualunque stakeholder di business la produzione di un modello architetturale formale conforme ai principi DDD.

2.2 Analisi dei Pain Points

Lo scenario tipico di una fase di discovery manuale presenta criticità strutturali che il DIM intende risolvere:

1. **Raccolta non strutturata:** Le interviste con gli stakeholder producono note informali, appunti sparsi e conoscenza implicita che non viene formalizzata in un modello coerente. La trascrizione manuale introduce ulteriori perdite informative.
2. **Assenza di copertura sistematica:** Senza una guida strutturata per fasi, l'architetto rischia di trascurare aspetti critici come i pattern di gestione dei fallimenti (Saga), i modelli di consistenza (Strong vs. Eventual), o le esigenze di integrazione con sistemi legacy tramite Anticorruption Layer.
3. **Soggettività dell'analista:** Il risultato di una sessione DDD tradizionale varia significativamente in funzione di chi la conduce. Un architetto con esperienza prevalente in sistemi monolitici, ad esempio, tenderà inconsciamente ad accorpare responsabilità in pochi bounded context di grandi dimensioni, introducendo bias strutturali nel modello di dominio. Non esistendo un protocollo di indagine standardizzato, la qualità dell'output dipende dall'intuizione e dal background individuale dell'analista, rendendo il processo difficilmente ripetibile.
4. **Non ripetibilità:** Due sessioni di discovery sullo stesso progetto, condotte da architetti diversi o dallo stesso architetto in momenti diversi, possono produrre modelli significativamente diversi. Non esiste un benchmark di completezza.
5. **Assenza di output machine-readable:** Anche quando la sessione produce risultati di qualità, la documentazione rimane in formato testuale non strutturato, richiedendo un ulteriore passaggio manuale per la formalizzazione in modelli consumabili da strumenti di progettazione.

2.3 Rischi e Impatto Operativo

Un modello di dominio incompleto o mal definito ha ripercussioni a cascata sull'intera architettura a microservizi:

- **Bounded Context errati:** Responsabilità sovrapposte tra microservizi, con conseguente accoppiamento stretto e difficoltà di deploy indipendente.
- **Modello event-driven incompleto:** L'omissione o la ridondanza di eventi durante la fase di discovery produce, in produzione, eventi che nessun servizio è in grado di elaborare, compromettendo la consistenza dei dati e rendendo i guasti difficili da diagnosticare.
- **Ubiquitous Language inconsistente:** Ambiguità terminologiche che si propagano dal modello di dominio al codice sorgente, generando bug semantici difficili da diagnosticare.
- **Pattern tattici inappropriati:** Scelta di Choreography dove sarebbe necessaria un'Orchestration (o viceversa), con impatto sulla tracciabilità dei failure e sulla complessità del debugging.

Il DIM mitiga questi rischi attraverso la sistematicità della pipeline a cinque fasi, la persistenza dello stato conversazionale in Redis, la validazione automatica della completezza tramite il Project Controller, e la generazione automatica di un modello JSON strutturato, machine-readable e revisionabile.

2.4 Business Case

Il costo reale di una sessione di Domain Discovery tradizionale è spesso sottostimato. Un workshop di Event Storming richiede mediamente **uno o due giorni** di lavoro intensivo (circa 10–14 ore effettive), con la partecipazione simultanea di sviluppatori, product manager, domain expert e architetti. A livello di mercato, il costo di una sessione facilitata da un consulente specializzato parte da circa **500 EUR/persona/giorno**.

Il costo più significativo, tuttavia, non è quello diretto ma quello **nascosto**: un modello di dominio errato o incompleto si propaga a cascata nell'architettura, generando debito tecnico che emergerà settimane o mesi dopo sotto forma di bounded context da ristrutturare, eventi da ridefinire e servizi da riscrivere. In contesti enterprise, il costo di correzione di errori architetturali in produzione può superare di ordini di grandezza quello della sessione di discovery iniziale.

Il DIM comprime l'intero ciclo di discovery in una **singola sessione conversazionale asincrona**, eliminando la necessità di coordinare simultaneamente più partecipanti e producendo un output strutturato e machine-readable in tempo reale. Lo stakeholder di business può interagire con il sistema in autonomia, nei propri tempi, senza dipendere dalla disponibilità di un architetto specializzato.

2.5 Confronto con le Alternative Esistenti

Gli strumenti attualmente disponibili per supportare il processo di Domain Discovery si collocano in tre categorie, ciascuna con limitazioni significative che il DIM intende superare:

1. **Whiteboard collaborativi generici** (Miro, Mural, FigJam): Forniscono un canvas digitale con sticky notes e strumenti di collaborazione in tempo reale, ma non offrono alcuna struttura DDD nativa. La responsabilità di guidare la sessione, formulare le domande giuste e garantire la completezza dell'analisi ricade interamente sull'architetto umano. L'output rimane un artefatto visuale non strutturato, non consumabile automaticamente da strumenti downstream.
2. **Tool DDD specializzati** (Context Mapper, Qlerify, Domo): Offrono primitive di modellazione specifiche per il DDD — bounded context, context mapping, aggregate — ma presuppongono che l'utente conosca già il dominio e sappia come modellarlo. Sono strumenti di *formalizzazione*, non di *scoperta*: supportano la fase di modellazione ma non quella di raccolta e scoperta dei requisiti.
3. **LLM general-purpose** (ChatGPT, Claude): Possono rispondere a domande sul DDD e suggerire strutture architettoniche, ma operano in modo reattivo e non strutturato. Non garantiscono copertura sistematica delle cinque aree del DDD strategico, non mantengono stato conversazionale tra le sessioni, e soprattutto trasmettono i dati di dominio — potenzialmente riservati — a servizi cloud di terze parti.

Il DIM si differenzia combinando le tre capacità che nessuna alternativa offre congiuntamente: **raccolta guidata dei requisiti** (intervista strutturata a cinque fasi con drill-down), **validazione automatica della completezza** (Project Controller), e **generazione di output machine-readable** (JSON architetturale conforme a schema predefinito), il tutto operando **interamente in locale** senza trasmissione di dati a servizi esterni.

Capitolo 3

Obiettivi e Requisiti del Sistema

3.1 Obiettivi Strategici

Il DIM si propone di automatizzare e standardizzare la fase di Domain Discovery, operando come primo agente nella pipeline di progettazione architetturale. Gli obiettivi si articolano in quattro direttrici:

1. **Automatizzare la raccolta dei requisiti di dominio** attraverso un'intervista strutturata in fasi progressive, capace di guidare uno stakeholder di business nella definizione completa del dominio applicativo senza richiedere competenze DDD pregresse.
2. **Generare un modello di dominio formale e machine-readable** a partire dalla conversazione con lo stakeholder, producendo un artefatto strutturato direttamente consumabile dagli agenti downstream della pipeline.
3. **Supportare il raffinamento iterativo del modello** consentendo allo stakeholder di rivedere, chiarire e modificare l'architettura prodotta attraverso un ciclo di revisione conversazionale con garanzie contro modifiche involontarie.
4. **Garantire osservabilità end-to-end** dell'intero sistema, fornendo visibilità su log e metriche di tutti i componenti per facilitare il debugging e il monitoraggio operativo.

3.2 Requisiti Funzionali

3.2.1 RF-01: Pipeline di Intervista a 5 Fasi

Il sistema deve implementare cinque agenti LLM specializzati, ciascuno focalizzato su un pillar architetturale del DDD, che conducono sequenzialmente l'intervista. Ogni agente:

- Possiede un system prompt rigoroso che ne delimita il focus tematico.
- Interagisce esclusivamente in italiano, traducendo il gergo DDD in scenari di business.
- Pone una sola domanda alla volta, con drill-down obbligatorio su risposte vaghe.
- Non decide autonomamente la fine della propria fase: la transizione è governata esclusivamente dall'Orchestrator (RF-02)

3.2.2 RF-02: Validazione Automatica di Completezza

Un agente Orchestrator analizza lo storico conversazionale al termine di ogni fase e determina se le informazioni raccolte sono sufficienti per procedere alla fase successiva. La validazione si basa su una checklist strutturata per fase (es. per la Fase 1: Core Domain identificato, Generic Subdomains listate, Supporting Subdomains specificate).

3.2.3 RF-03: Routing Intelligente

Il sistema deve instradare automaticamente ogni messaggio dell'utente al flusso corretto:

- Se il modello architetturale non è ancora stato generato, il messaggio viene inoltrato al flusso di Discovery.
- Se il modello è già disponibile, il messaggio viene inoltrato al flusso di Refinement.

3.2.4 RF-04: Generazione Architetturale

Al completamento dell'intervista, il sistema deve generare automaticamente un documento architetturale strutturato in cinque pilastri (Strategic Analysis, Boundary Definition, EDA Integration, Tactical Patterns, Technical Excellence) e la sua rappresentazione JSON formale con schema predefinito. La generazione deve operare con politica Zero-Loss, preservando ogni dettaglio tecnico emerso durante l'intervista senza riassumere né omettere informazioni.

3.2.5 RF-05: Raffinamento del Modello

Il sistema deve supportare due modalità operative post-generazione, selezionabili dall'utente:

- **Modalità Spiegazione:** L'utente può porre domande sull'architettura corrente e ricevere risposte contestuali basate sul modello generato, senza che vengano proposte modifiche.
- **Modalità Modifica:** L'utente può richiedere modifiche strutturali al modello. Il sistema produce una bozza aggiornata che l'utente può confermare o scartare.

3.2.6 RF-06: Gestione del Ciclo di Vita delle Bozze

Il sistema deve gestire il ciclo di vita delle bozze di modifica con un meccanismo a doppia conferma:

- **Conferma:** La bozza viene promossa a modello corrente.
- **Rifiuto:** La bozza viene eliminata e il modello corrente rimane invariato.
- Entrambe le azioni richiedono una seconda conferma esplicita per prevenire operazioni accidentali.

3.2.7 RF-07: Frontend Chat

Interfaccia web single-page con:

- Chat full-screen con layout responsive.
- Sessioni persistenti che consentano la ripresa di conversazioni interrotte.
- Rendering differenziato per messaggi testuali, blocchi JSON con sintassi evidenziata, e contenuti Markdown.
- Feedback visuale contestuale che distingua le diverse fasi operative (discovery, spiegazione, modifica).

3.2.8 RF-08: Stato Conversazionale Persistente

Il sistema deve mantenere uno stato conversazionale persistente per ogni sessione, includendo: la fase corrente dell'intervista, lo storico completo dei messaggi, il modello architetturale generato e l'eventuale bozza di modifica in attesa di conferma. Lo stato deve sopravvivere al riavvio del browser e consentire la ripresa della sessione entro un

tempo ragionevole.

3.3 Requisiti Non Funzionali

- **RNF-01 – Osservabilità:** Raccolta centralizzata di log (Loki) e metriche (Prometheus) tramite OpenTelemetry Collector, con datasource Grafana preconfigurati.
- **RNF-02 – Privacy e Indipendenza:** Inferenza LLM completamente locale tramite Ollama. Nessun dato inviato a servizi cloud esterni.
- **RNF-03 – Portabilità:** Containerizzazione completa tramite Docker Compose per ambiente self-contained e riproducibile con un singolo `docker compose up`.
- **RNF-04 – Modularità:** Ogni workflow n8n è un componente autonomo richiamabile indipendentemente, con interfacce well-defined (input/output via webhook e Redis).
- **RNF-05 – Resilienza Sessionale:** Lo stato conversazionale in Redis con TTL di 24 ore permette la ripresa di sessioni interrotte senza perdita di dati.
- **RNF-06 – Estensibilità:** L'architettura a workflow consente l'aggiunta di nuove fasi, nuovi agenti o nuovi modelli LLM senza modifiche al codice applicativo.

Capitolo 4

Stakeholder e Ruoli

Il presente capitolo identifica gli attori, umani e sistemici, che interagiscono nel sistema. Per ogni stakeholder vengono definiti il perimetro di responsabilità, le modalità di interazione con il sistema e i vincoli operativi.

4.1 Business Stakeholder (Utente Intervistato)

Ruolo: Rappresentante del business che partecipa all'intervista di scoperta del dominio. È la fonte primaria delle informazioni sul contesto aziendale, i processi, le entità e le regole di business.

Responsabilità:

- Rispondere alle domande degli agenti intervistatori con dettaglio tecnico sufficiente.
- Fornire conferme esplicite quando richieste dall'agente (l'Orchestrator emette **WAIT** su risposte evasive come "decidi tu").
- Validare, confermare o rifiutare il modello architetturale generato attraverso il flusso di Refinement.

Interfaccia: Chat web (`frontend/index.html`) su browser, comunicazione asincrona via webhook n8n sulla porta 5678.

4.2 Agenti Intervistatori (Discovery Pipeline)

Il sistema impiega cinque agenti specializzati, ciascuno responsabile di una fase distinta dell'intervista DDD. Ogni agente opera con regole comportamentali rigide di *separazione delle competenze*: è strettamente proibito uscire dal proprio ambito tematico.

Agente	Fase	Ruolo DDD	Focus Tematico
Domain Interviewer	1	Senior Strategic Architect	Core, Generic e Supporting Subdomains
Context Interviewer	2	Senior Structural Architect	Bounded Contexts, Aggregate Roots, Entities, Ubiquitous Language
Event Interviewer	3	Senior Event-Driven Architect	Actors, Commands, Domain Events, Context Mapping
Pattern Interviewer	4	Senior Tactical Architect	Choreography/Orchestration, Sagas, CQRS, Event Sourcing
Resilience Interviewer	5	Senior Infrastructure Architect	Cloud Scalability, ACL, Monitoring, Security

Tabella 4.1: Agenti Intervistatori e relative fasi dell'intervista DDD

4.2.1 Regole Comuni agli Agenti Intervistatori

Tutti gli agenti condividono un insieme di vincoli comportamentali definiti nei rispettivi system prompt:

- **Lingua:** Interazione esclusivamente in italiano.
- **Una domanda alla volta:** Progressione sequenziale, mai domande multiple.
- **Drill-down obbligatorio:** Approfondimento automatico su risposte vaghe o generiche.
- **Divieto di gergo DDD:** Traduzione dei concetti architetturali in scenari di business comprensibili.
- **Isolamento tematico:** Proibizione assoluta di sconfinare nelle fasi degli altri agenti.
- **No auto-chiusura:** L'agente non decide mai autonomamente di concludere la fase; il sistema lo ferma automaticamente.

4.3 Agente di Validazione (Orchestrator)

Ruolo: *Senior Project Controller and Architectural Auditor.*

Responsabilità: Monitorare il livello di dettaglio della conversazione al termine di ogni scambio di messaggi e determinare se le informazioni raccolte sono sufficienti per procedere di fase. L'output è binario:

- **PASS:** Tutti i requisiti della fase corrente sono esplicitamente confermati dall'utente.
- **WAIT:** Informazioni mancanti, vaghe o con profondità tecnica insufficiente.

L'Orchestrator opera con una checklist di validazione per fase, verificando ad esempio che nella Fase 2 ogni entità principale abbia almeno 2-3 attributi specifici, o che nella Fase 4 sia stata fatta una scelta esplicita tra Choreography e Orchestration. (modello qwen2.5:14b)

4.4 Agenti di Elaborazione (Post-Discovery)

DDD Analyst: Analizza l'intero storico conversazionale e produce il documento architetturale Markdown strutturato secondo i cinque pilastri DDD, con politica Zero-Loss. (modello qwen2.5:14b)

JSON Coder: Converte il documento Markdown in un JSON formale con schema predefinito (modello qwen2.5:14b, temperatura 0.0 per massima determinismo), preservando ogni dettaglio senza riassumere.

Modifier (Refinement): Analizza le richieste di modifica dell'utente, verifica l'esistenza dei componenti nel modello architetturale corrente, e produce un receipt strutturato con l'azione da intraprendere (ADD, MODIFY, DELETE). (modello qwen2.5:14b)

Explainer (Refinement): Risponde a domande di chiarimento sull'architettura corrente, basandosi esclusivamente sul JSON in Redis. È strettamente inibito dal suggerire o implementare modifiche. (modello llama3)

4.5 Stack di Osservabilità

Ruolo: Insieme di strumenti tecnici preposti al monitoraggio e alla telemetria del sistema.

Responsabilità: Raccolta, elaborazione e visualizzazione di log e metriche da tutti i componenti containerizzati dell'ecosistema DIM.

4.6 Matrice di Tracciabilità Ruoli/Funzioni

Ruolo/ Stakeholder	Disc.	Valid.	Gener.	Refin.	Draft	Osserv.
Stakeholder	Risponde	No	No	Modifica	Conf./ Rif.	No
Interviewers	Intervista	No	No	No	No	No
Controller	No	Valida	No	No	No	No
DDD Analyst	No	No	Genera MD	No	No	No
JSON Coder	No	No	Genera JSON	No	No	No
Modifier	No	No	No	Modifica	No	No
Explainer	No	No	No	Spiega	No	No
Stack Osservabilità	No	No	No	No	No	Monitora

Tabella 4.2: Matrice di tracciabilità Ruoli/Funzioni del sistema

Capitolo 5

Architettura del Sistema



Figura 5.1: Architettura complessiva del sistema DIM

Questo capitolo descrive l'architettura logica e fisica del Domain Interviewer & Modeler, illustrando come i moduli cooperano per garantire un servizio di domain discovery scalabile, modulare e osservabile.

5.1 Visione d'Insieme

Il DIM è un ecosistema modulare basato su microservizi containerizzati in cui ogni componente ha un compito preciso e delimitato. L'architettura segue il principio della **separazione delle responsabilità** (Separation of Concerns) a livello di container:

- **L'Orchestratore (n8n):** Motore di workflow automation che coordina l'intera pipeline. Gestisce i webhook HTTP, il routing dei messaggi, l'invocazione degli agenti LLM e la lettura/scrittura dello stato in Redis. Nessuna logica applicativa è implementata come codice custom: tutto è espresso come workflow visuale.
- **Lo Stato Conversazionale (Redis):** Key-value store ad alte prestazioni che mantiene lo stato persistente delle sessioni: fase corrente, storico chat, JSON architetturali generati, bozze in attesa di conferma.
- **Il Cervello (Ollama):** Runtime LLM locale che esegue i modelli di linguaggio per tutte le operazioni di ragionamento semantico. Due modelli sono impiegati, ciascuno con temperatura calibrata sul compito:
 - `llama3:latest` — modello conversazionale impiegato per gli agenti che interagiscono direttamente con l'utente: 5 agenti Intervistatori (temp. 0.4), ed Explainer (temp. 0.4).
 - `qwen2.5:14b` — modello ad alta precisione strutturale, impiegato dove è richiesta la generazione di output formali: DDD Analyst per la sintesi del documento architetturale (temp. 0.3) e JSON Coder per la traduzione in JSON con schema predefinito (temp. 0.0), e dove è richiesta un ragionamento più complesso: Orchestrator (temp. 0.0) per approvare le transizioni di fase e Modifier (temp. 0.1) per dedurre la giusta modifica da fare.
- **L'Interfaccia (Frontend HTML):** Chat web full-screen single-page che consente allo stakeholder di interagire in linguaggio naturale, con rendering differenziato per testo, JSON e Markdown architetturale.
- **Il Supervisore (Stack Osservabilità):** Pipeline OpenTelemetry Collector → Loki/Prometheus → Grafana che raccoglie log e metriche da tutti i container.

5.2 I Pilastri Progettuali

Per garantire un sistema affidabile e manutenibile, sono stati seguiti quattro principi architetturali fondamentali:

5.2.1 Orchestrazione via Workflow (n8n)

Tutta la logica applicativa è espressa come workflow n8n, non come codice custom. Questo approccio offre:

- **Visibilità totale:** Ogni esecuzione è tracciabile nella UI di n8n con input/output di ogni nodo.

- **Facilità di debug:** Modifica e test senza ricompilazione, con esecuzione step-by-step.
- **Natività nell'integrazione:** Nodi built-in per Redis, HTTP, LLM, Switch, Sub-Workflow.
- **Riproducibilità:** I workflow sono esportabili come JSON e versionabili in Git.

5.2.2 State Management Centralizzato (Redis)

Ogni informazione di stato — fase di discovery, storico chat, JSON generati, bozze — è persistita in Redis con chiavi strutturate per `sessionId`. Questo disaccoppia completamente lo stato applicativo dalla logica di esecuzione: l'orchestratore non mantiene alcun dato di sessione in memoria, ogni invocazione del workflow è autocontenuta, e la ripresa di sessioni interrotte è garantita senza perdita di contesto.

5.2.3 Agenti Specializzati con System Prompt

Ogni agente LLM del sistema — intervistatori, orchestratore, agenti di elaborazione e agenti di refinement — opera con un system prompt ingegnerizzato che ne definisce il comportamento in modo deterministico. Tutti i prompt condividono una struttura comune articolata in:

- **Ruolo e Persona:** Identità professionale specifica che orienta il tono e la profondità delle risposte (es. “Senior Strategic Architect”, “Senior Project Controller”, “Data Parser”).
- **Lingua e Stile:** Output esclusivamente in italiano, tono professionale e diretto, traduzione del gergo tecnico DDD in scenari di business comprensibili.
- **Formato di Output:** Vincoli rigidi sul formato della risposta, variabili per agente: una domanda alla volta per gli intervistatori, verdetto binario per l'orchestratore, Markdown strutturato per il DDD Analyst, JSON puro per il JSON Coder, receipt di modifica per il Modifier.
- **Confini Comportamentali:** Regole di inibizione che impediscono all'agente di uscire dal proprio perimetro. Gli intervistatori non possono sconfinare nelle fasi altrui (isolamento tematico); l'Explainer non può suggerire modifiche; il Modifier deve verificare l'esistenza dei componenti prima di proporre cambiamenti.
- **Grounding sui Dati:** Ogni agente opera esclusivamente sulle informazioni a cui ha accesso (storico conversazionale per gli intervistatori, JSON architetturale per Explainer e Modifier), senza inventare o generalizzare informazioni non presenti.

5.2.4 Osservabilità Nativa

L'osservabilità non è un add-on ma un componente strutturale dell'architettura. Il servizio n8n emette telemetria nativa OTLP verso l'OTel Collector, che la instrada verso Loki (log) e Prometheus (metriche). Grafana è preconfigurato con datasource e correlation per navigare tra i segnali.

5.3 Motivazioni delle scelte architetturali

Dettaglio Tecnico: Motivazioni delle Scelte Architetturali

n8n vs. codice custom: Scelto per la capacità di definire workflow complessi con nodi condizionali (Switch), invocazione di sub-workflow (`executeWorkflow`), integrazione nativa con LLM (nodi `lmChatOllama` e `agent`), e gestione della memoria conversazionale tramite nodi `memoryRedisChat`. La UI visuale accelera drasticamente il ciclo di sviluppo e debugging.

Redis vs. database relazionale: Scelto per la bassa latenza, l'integrazione nativa con i nodi n8n, e la semplicità del modello key-value che rispecchia la struttura piatta dello stato conversazionale.

Ollama vs. API cloud (OpenAI, Anthropic): Scelto per garantire privacy totale dei dati di dominio. In un contesto enterprise, le informazioni sulla struttura del business e i processi interni sono altamente confidenziali. L'inferenza locale garantisce che nessun dato di dominio venga trasmesso a servizi esterni, preservando la riservatezza delle informazioni aziendali.

Modelli LLM — llama3 e qwen2.5:14b: llama3 è impiegato per tutti gli agenti conversazionali (intervistatori e Explainer) dove fluency e capacità di interazione sono prioritarie. qwen2.5:14b è riservato agli agenti di elaborazione post-intervista — DDD Analyst, JSON Coder, Orchestrator e Modifier — che richiedono alta precisione nella generazione di output strutturati (documento Markdown e JSON formale) e nel ragionamento logico.

5.4 Flusso di Comunicazione End-to-End

Il flusso di una sessione completa segue questo percorso, dal primo messaggio dell'utente fino al raffinamento del modello (si veda il diagramma di sequenza in Figura 5.2):

1. L'utente accede al frontend, che genera o recupera un identificatore di sessione univoco.
2. Ogni messaggio viene inviato al Routing Workflow, che determina se instradarlo verso il flusso di Discovery o di Refinement in base alla presenza o assenza del modello architetturale.
3. **Discovery:** Il messaggio viene inoltrato all'agente intervistatore della fase corrente. Dopo ogni risposta, l'Orchestrator valuta la completezza delle informazioni raccolte e, se soddisfatto, autorizza la transizione alla fase successiva.
4. **Generazione:** Al completamento della quinta fase, il sistema avvia la catena di generazione automatica: il documento architetturale Markdown viene prodotto e successivamente tradotto in un JSON formale con schema predefinito.
5. **Polling:** Il frontend rileva l'avvio della generazione e interroga periodicamente il backend fino alla disponibilità del modello.
6. **Refinement:** Una volta disponibile il modello, l'utente può scegliere tra modalità Spiegazione e modalità Modifica. Le modifiche producono una bozza che richiede conferma esplicita prima di essere applicata.

I dettagli implementativi di ogni componente — webhook, routing, transizioni di fase,

polling e gestione delle bozze — sono approfonditi nei Capitoli 7 e 8.

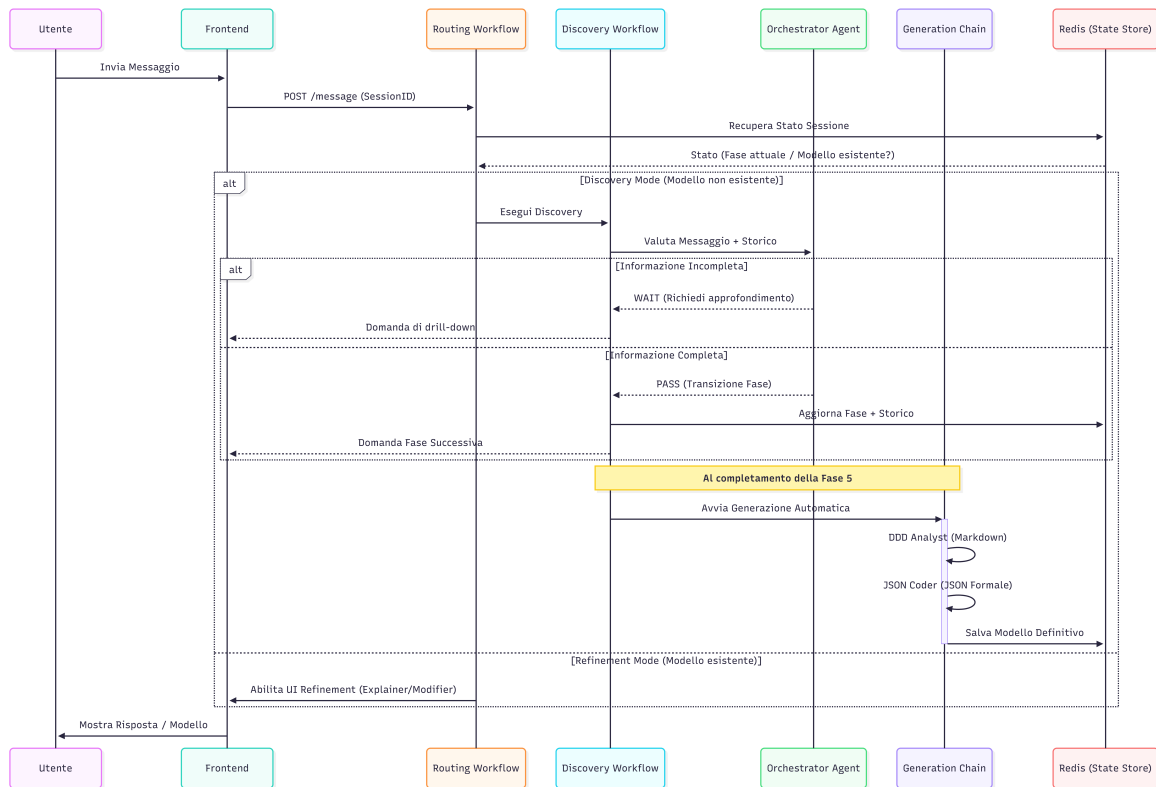


Figura 5.2: Diagramma di sequenza del flusso end-to-end: dall'intervista alla generazione del modello.

Capitolo 6

Deployment e Containerizzazione Docker

L'intero sistema è definito in un singolo file `docker-compose.yml` che descrive tutti i microservizi e le relative dipendenze. La scelta di un unico Compose file garantisce la riproducibilità dell'ambiente con un singolo comando `docker compose up`, eliminando qualsiasi configurazione manuale.

6.1 Servizi Definiti nel Docker Compose

Container	Immagine	Porte	Volumi	Depends On
loki	grafana/loki:3.5.0	3100	./loki/	—
otel-collector	otel/opentelemetry-collector-contrib:0.133.0	4317, 4318	./otel-collector/loki	
prometheus	prom/prometheus:v2.53.0	9090	./prometheus/	—
grafana	grafana/grafana:12.2	8080→3000	./grafana/	loki, prometheus
n8n	docker.n8n.io/ n8nio/n8n	5678	./Data_N8N	redis, ollama, otel-collector
redis	redis:alpine	6379	./Data_Redis	—
ollama	ollama/ollama	11434	./Data_Ollama	—

Tabella 6.1: Dettaglio completo dei servizi nel Docker Compose

6.2 Rete Docker, Comunicazione e Dipendenze

Tutti i container sono connessi a un'unica rete Docker bridge denominata `rete_unica`.

Dettaglio Tecnico: Topologia di Rete

All'interno della rete `rete_unica`, ogni container è raggiungibile dagli altri tramite il **nome del servizio** definito nel Docker Compose. Ad esempio:

- n8n raggiunge Redis come `redis:6379` (senza passare dall'host).
- n8n raggiunge Ollama come `ollama:11434`.
- n8n invia telemetria OTLP a `otel-collector:4318`.
- L'OTel Collector esporta i log a `loki:3100/otlp`.
- Grafana interroga `loki:3100` e `prometheus:9090` tramite provisioning automatico.

Tutte le porte dei servizi sono mappate sull'host per consentire l'accesso diretto durante lo sviluppo e il debugging. In un ambiente di produzione, solo le porte strettamente necessarie all'accesso esterno (5678 per il webhook n8n, 8080 per la UI Grafana) andrebbero esposte.

6.2.1 Catena di Dipendenze e Ordine di Avvio

Il Docker Compose definisce dipendenze esplicite tramite **depends_on**:

- **n8n** → **redis**, **ollama**, **otel-collector**: n8n non può operare senza lo stato conversazionale (Redis), il motore di inferenza LLM (Ollama) e il collettore di telemetria.
- **otel-collector** → **loki**: Il collettore necessita del backend di log attivo per esportare i segnali.
- **grafana** → **loki**, **prometheus**: I datasource devono essere raggiungibili per il provisioning automatico.

Nota Importante: Ordine di Avvio

*Le dipendenze **depends_on** di Docker Compose garantiscono l'ordine di avvio ma non la readiness dei servizi. In un ambiente di produzione, sarebbe necessario aggiungere **healthcheck** e **depends_on: condition: service_healthy** per garantire che i servizi upstream siano effettivamente pronti prima di avviare quelli downstream.*

6.3 Volumi e Persistenza dei Dati

Il sistema utilizza **bind mounts** per la persistenza dei dati e la configurazione:

Servizio	Volume Host	Scopo
n8n	./Data_N8N	Persistenza workflow, credenziali, configurazione n8n
redis	./Data_Redis	Dump RDB per persistenza dello stato
ollama	./Data_Ollama	Cache dei modelli LLM scaricati
loki	./loki/loki-config.yml (ro)	Configurazione di Loki (read-only)
loki	./runtime/data/loki	Storage dei chunk di log
otel-collector	./otel-collector/collector.yaml (ro)	Configurazione del collettore
prometheus	./prometheus/prometheus.yml (ro)	Configurazione di scraping
grafana	./grafana/provisioning/ (ro)	Datasource e dashboard auto-configurati
grafana	./runtime/data/grafana	Persistenza dashboard e preferenze

Tabella 6.2: Mappatura completa dei volumi Docker

I file di configurazione sono montati come **read-only** (:ro) per prevenire modifiche accidentali da parte dei container.

6.4 Variabili d'Ambiente Chiave

6.4.1 n8n

```
# Timezone
GENERIC_TIMEZONE=Europe/Rome
TZ=Europe/Rome

# Integrazione Osservabilit 
OTEL_COLLECTOR_URL=http://otel-collector:4318
```

```
# Timeout esecuzione (-1 = nessun timeout)
N8N_EXECUTION_TIMEOUT=-1
```

La variabile `N8N_EXECUTION_TIMEOUT=-1` disabilita il timeout di esecuzione dei workflow, necessario perché la generazione del modello architetturale (catena DDD Analyst → JSON Coder) può richiedere diversi minuti con inferenza locale su modelli da 14 miliardi di parametri.

6.4.2 Grafana

```
GF_SECURITY_ADMIN_USER=admin
GF_SECURITY_ADMIN_PASSWORD=admin
GF_SERVER_ROOT_URL=http://localhost:8080
GF_EXPLORE_ENABLED=true
GF_AUTH_ANONYMOUS_ENABLED=true
GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
```

L'accesso anonimo con ruolo Admin è abilitato per ambiente di sviluppo locale, eliminando la necessità di autenticazione. In un ambiente di produzione, questa configurazione andrebbe rimossa.

6.5 Politiche di Restart

Tutti i servizi dello stack di osservabilità sono configurati con `restart: unless-stopped`, garantendo il riavvio automatico dopo un crash o un riavvio del sistema host. Il servizio `n8n` non possiede questa politica nel Compose attuale, ma dipende dai servizi upstream che la implementano.

Capitolo 7

Pipeline dei Workflow n8n

Il cuore logico del DIM è costituito da cinque workflow n8n interconnessi che implementano l'intera pipeline di discovery, analisi e raffinamento. Ogni workflow è un componente autonomo con interfacce well-defined, esportabile come file JSON e versionabile in Git.

7.1 Mappa dei Workflow

Workflow	Trigger	Responsabilità
Routing	Webhook POST /chat-locale	Punto di ingresso unico; smista verso Discovery o Refinement
Discovery	Sub-workflow call	Pipeline di intervista a 5 fasi + generazione modello
Refinement	Sub-workflow call	Explain o Modify del modello architetturale
Polling	Webhook GET /check-status-locale	Verifica disponibilità del JSON architetturale
Draft Decision	Webhook POST /draft-decision-locale	Conferma o rifiuto delle bozze di modifica

Tabella 7.1: Mappa completa dei workflow n8n

7.2 Routing Workflow

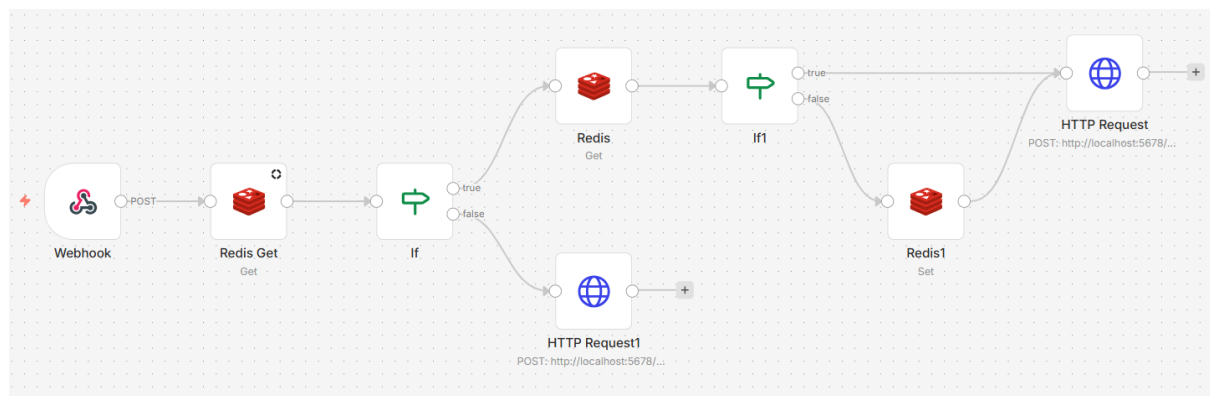


Figura 7.1: Struttura del Routing Workflow nell'interfaccia n8n: gestione degli accessi e branching.

Il Routing Workflow è il punto di ingresso dell'intera pipeline. Riceve i messaggi dallo stakeholder tramite webhook HTTP e decide dove instradarli.

7.2.1 Logica di Routing Dettagliata

- Ricezione:** Il webhook POST /chat-locale riceve il payload {message, sessionId, action} dal frontend.
- Verifica Stato:** Un nodo Redis GET controlla la chiave `discovery_json:{sessionId}`:

- **Se NON esiste** (la discovery non è completata):
 - Un secondo nodo Redis GET verifica `discovery_phase:{sessionId}`.
 - Se la fase non è inizializzata, un nodo Redis SET la imposta a 1.
 - Il messaggio viene inoltrato al Discovery Workflow tramite HTTP Request interna a `/webhook/discovery-internal` con timeout di 30 minuti (1.800.000 ms).
 - **Se esiste** (il modello è già disponibile): il messaggio viene inoltrato al Refinement Workflow tramite `/webhook/refinement-internal` con il campo `action` (explain/modify).
3. **Risposta:** La modalità `responseMode: lastNode` del webhook fa sì che la risposta HTTP venga restituita automaticamente dall'ultimo nodo della catena.

Dettaglio Tecnico: Timeout di 30 Minuti

Il timeout dell'HTTP Request verso il Discovery Workflow è impostato a 1.800.000 ms (30 minuti). Questo valore tiene conto del caso peggiore: quando l'Orchestrator autorizza il completamento della Fase 5 e la catena `DDD Analyst → JSON Coder` deve processare l'intero storico conversazionale con un modello da 14B parametri in inferenza locale. Questa operazione può richiedere fino a 20 minuti.

7.3 Discovery Workflow

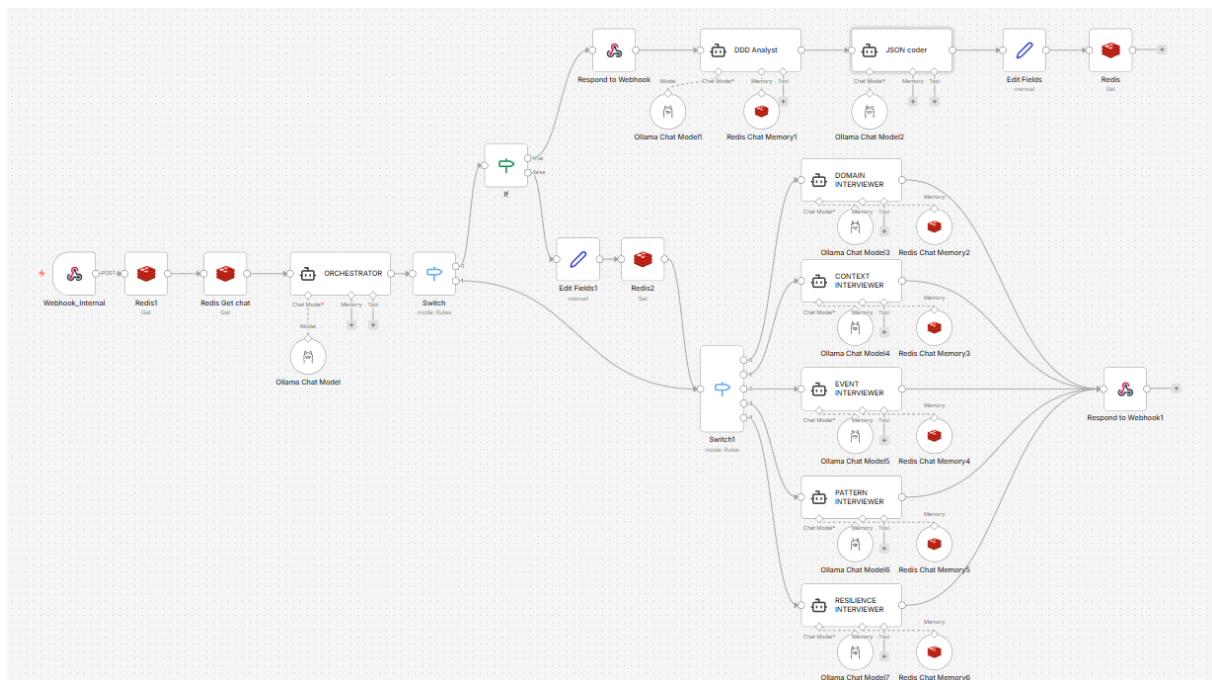


Figura 7.2: Struttura del Discovery Workflow nell'interfaccia n8n

Il Discovery Workflow è il componente più complesso dell'intero sistema. Implementa la pipeline di intervista a cinque fasi con agenti specializzati e la catena di generazione del modello architetturale.

7.3.1 Flusso di Esecuzione

1. **Ricezione:** Il webhook interno `/discovery-internal` riceve `message` e `sessionId` dal Routing Workflow.
2. **Lettura Stato:** Due nodi Redis GET leggono in parallelo la fase corrente (`discovery_phase:{sessionId}`) e lo storico della conversazione.
3. **Valutazione Orchestrator:** Il messaggio e lo storico vengono sottoposti all'Orchestrator (Project Controller), un agente LLM dedicato che emette un verdetto:
 - **WAIT** — le informazioni raccolte non sono ancora sufficienti per la fase corrente.
 - **PASS** — la fase corrente è completata con dettaglio adeguato.
4. **Branching (nodo Switch):** Il verdetto determina il percorso:
 - **Se WAIT:** Il messaggio viene instradato (tramite un secondo Switch) all'agente intervistatore della fase corrente, che prosegue l'intervista e restituisce la risposta all'utente.
 - **Se PASS e la fase corrente < 5:** La fase in Redis viene incrementata, e il messaggio viene instradato all'agente della **nuova** fase, che si presenta e pone la prima domanda del suo pillar.
 - **Se PASS e la fase corrente = 5:** Il workflow restituisce immediatamente una risposta con `status: generating` e avvia la catena di generazione.
5. **Generazione (solo post-Fase 5):** La catena `DDD Analyst → JSON Coder` processa l'intero storico conversazionale, produce il documento Markdown e lo traduce in JSON strutturato, salvandolo in Redis come `discovery_json:{sessionId}`.

7.3.2 Configurazione degli Agenti

Agente	Fase	Modello	Temp.	Memory Win.	Ruolo
Orchestrator	1–5	qwen2.5:14b	0.0	—	Valutazione PASS/WAIT
Domain Interv.	1	llama3	0.4	10 msg	Intervista strategica
Context Interv.	2	llama3	0.4	20 msg	Intervista bounded context
Event Interv.	3	llama3	0.4	30 msg	Intervista eventi e comandi
Pattern Interv.	4	llama3	0.4	40 msg	Intervista pattern tattici
Resilience Interv.	5	llama3	0.4	50 msg	Intervista resilienza
DDD Analyst	Post	qwen2.5:14b	0.3	200 msg	Generazione documento MD
JSON Coder	Post	qwen2.5:14b	0.0	—	Traduzione MD → JSON

Tabella 7.2: Configurazione completa degli agenti nel Discovery Workflow

Dettaglio Tecnico: Memory Window Crescente

La scelta di una memory window crescente (10 → 50 messaggi) è una decisione di prompt engineering motivata dall'accumulazione progressiva di informazioni. L'agente di Fase 1 necessita solo del contesto immediato per identificare i subdomini, mentre l'agente di Fase 5 deve avere visibilità sull'intera conversazione per garantire coerenza architetturale nelle domande su scalabilità e resilienza. Il DDD Analyst utilizza una finestra di 200 messaggi per avere accesso all'intero storico.

7.4 Refinement Workflow

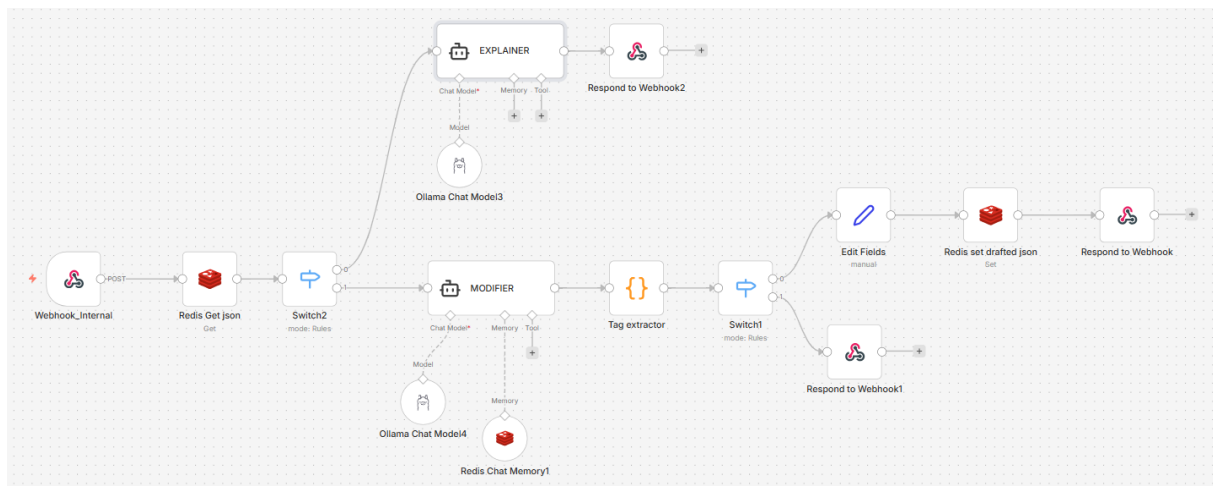


Figura 7.3: Struttura del Refinement Workflow nell'interfaccia n8n

Una volta generato il modello architetturale, lo stakeholder può interagire con esso attraverso due modalità.

7.4.1 Modalità Explain (Spiegazione)

L'agente Explainer riceve la domanda dell'utente e il JSON architetturale corrente (caricato da Redis). Risponde in italiano con spiegazioni tecniche basate esclusivamente sui dati del JSON, senza mai suggerire modifiche. Se l'utente chiede una modifica, lo reindirizza cortesemente verso la modalità Modify.

7.4.2 Modalità Modify (Modifica)

Il flusso di modifica segue un protocollo strutturato:

1. L'agente Modifier riceve la richiesta e il JSON corrente.
2. **Se la richiesta è vaga o troppo ampia:** risponde con un messaggio conversazionale chiedendo chiarimenti (non genera alcuna proposta).
3. **Se la richiesta è chiara e azionabile:** produce un "receipt" Markdown strutturato con:
 - **Azione:** ADD, MODIFY o DELETE.
 - **Sezione Target:** Chiave JSON root (es. `boundaries_and_language`).
 - **Path Interno:** Percorso esatto nell'albero JSON.

- **Elemento:** Nome dell'entità/evento/componente.
 - **Dettagli/Payload:** Dati da iniettare.
4. La bozza viene salvata in Redis come `draft_json:{sessionId}` con stato `pending`.
 5. L'utente visualizza la bozza di modifica e può confermare o rifiutare.

7.5 Polling Workflow

Il Polling Workflow è un endpoint GET leggero che il frontend interroga periodicamente (ogni 3 secondi) per verificare se il JSON architetturale è diventato disponibile in Redis dopo la generazione post-Fase 5. Restituisce il contenuto di `discovery_json:{sessionId}` se presente.

7.6 Draft Decision Workflow

Gestisce le azioni di conferma/rifiuto sulle bozze di modifica:

- **Azione “conferma”:** Il workflow recupera la bozza (`draft_json`) e il JSON architetturale corrente (`discovery_json`), poi un nodo JavaScript applica chirurgicamente la modifica (ADD, MODIFY o DELETE) alla struttura JSON. Se l'operazione ha successo, il JSON aggiornato viene salvato in `discovery_json`; in caso di errore (elemento non trovato, dati corrotti), lo stato viene impostato a `failed`.
- **Azione “rifiuta”:** Elimina la bozza (`draft_json`), mantenendo `discovery_json` invariato.

In entrambi i casi, la **chat memory del refinement** (`chat_history_refinement:{sessionId}`) viene cancellata al termine dell'operazione, resettando il contesto conversazionale per le interazioni successive.

7.7 Comunicazione tra Workflow

I workflow comunicano tra loro attraverso due meccanismi:

1. **HTTP Request Interne:** Il Routing Workflow invoca Discovery e Refinement tramite webhook interni (`http://localhost:5678/webhook/discovery-internal` e `/refinement-internal`). Questa scelta consente di mantenere ogni workflow come unità autonoma testabile indipendentemente.
2. **Redis come Bus di Stato:** Lo stato condiviso transita esclusivamente attraverso Redis. Nessun workflow mantiene stato in memoria tra invocazioni diverse.

Capitolo 8

Frontend Chat

Il frontend è un'interfaccia web single-page (`frontend/index.html`) che consente allo stakeholder di interagire con il DIM attraverso una chat full-screen. L'intera interfaccia è contenuta in un unico file HTML con CSS e JavaScript inline, senza dipendenze esterne eccetto la libreria `marked.js` per il rendering Markdown.

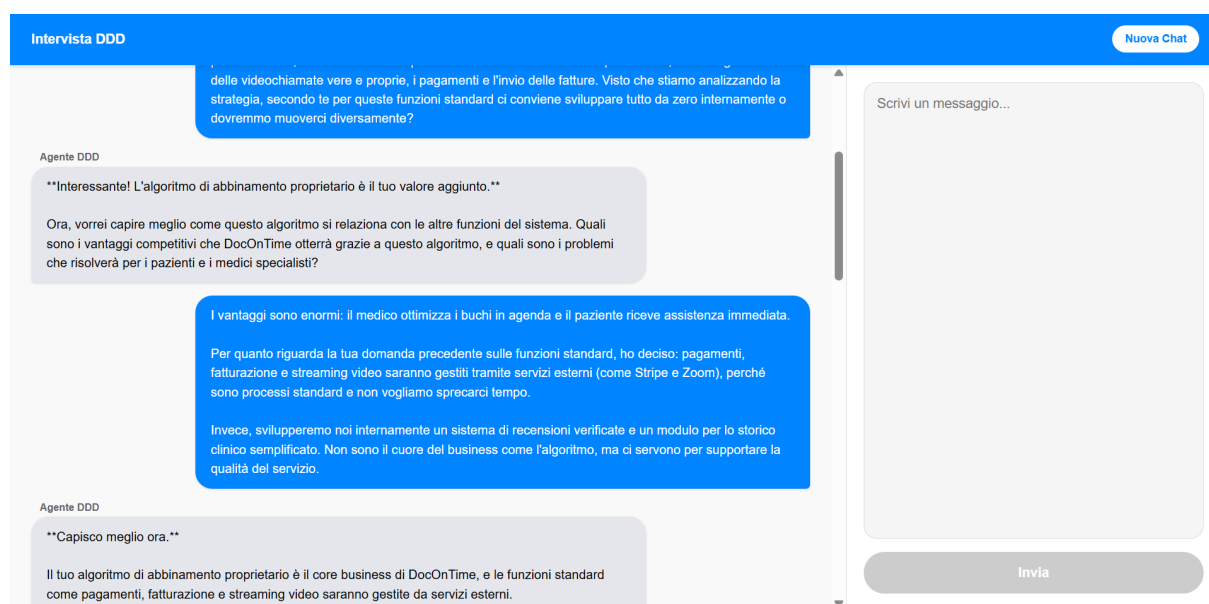


Figura 8.1: Interfaccia della chat frontend

8.1 Layout e Design

L'interfaccia adotta un layout orizzontale split-view:

- **Lato sinistro (70%):** Area dei messaggi (`#chat-history`) con scrolling verticale, bolle di chat differenziate per colore (blu per l'utente, grigio per l'agente).
- **Lato destro (30%):** Area di input con textarea multi-linea e pulsante "Invia".
- **Header:** Barra superiore blu con titolo "Intervista DDD" e pulsante "Nuova Chat" per il reset della sessione.

8.2 Gestione delle Sessioni

Ogni sessione è identificata da un `sessionId` univoco generato lato client:

```
let sessionId = localStorage.getItem('chat_session_id');
if (!sessionId) {
  sessionId = 'sess_' + Math.random().toString(36).substr(2, 9);
  localStorage.setItem('chat_session_id', sessionId);
}
```

```
}

```

Il `sessionId` è persistito in `localStorage`, garantendo la continuità della sessione anche dopo il refresh della pagina. Il pulsante “Nuova Chat” rimuove la chiave e ricarica la pagina, generando un nuovo ID.

8.3 Rendering Differenziato dei Messaggi

Il frontend distingue tre tipologie di contenuto per i messaggi dell’agente, ciascuna con un rendering dedicato:

- **Messaggi Testuali:** Bolle di chat standard con label “Agente DDD”.
- **Blocchi JSON Architettureali:** Visualizzazione stile terminale con formattazione pretty-print e pulsante “Copia JSON”.
- **Blocchi Markdown:** Documenti architetturali renderizzati in HTML tramite `marked.js`, con label differenziata (“Documento Architettureale” per il modello finale, “Agente Modifier” per le bozze di modifica).

8.4 Feedback Visuale e Animazioni

Il frontend implementa diverse animazioni per fornire feedback all’utente durante le operazioni asincrone:

Contesto	Animazione	Descrizione
Discovery / Explain	Typing indicator	Tre puntini blu che rimbalzano (bounce animation)
Generazione modello	Slide right-to-left	Testo “Generazione architettura in corso...” blu che scorre
Modifica	Slide left-to-right	Testo “Analizzando la richiesta...” arancione che scorre

Tabella 8.1: Animazioni di feedback nel frontend

8.5 Pulsanti Inline e Selezione Modalità

Dopo la generazione del modello architetturale, il frontend presenta due pulsanti inline nell’area chat:

- **“Chiedi Spiegazione”:** Attiva la modalità `explain`. L’area di input viene sbloccata con placeholder “Scrivi la tua domanda...”.
- **“Richiedi Modifica”:** Attiva la modalità `modify`. L’area di input viene sbloccata con placeholder “Descrivi la modifica strutturale che desideri...”.

Durante la visualizzazione dei pulsanti, l’area di input è **bloccata** con il messaggio “Seleziona un’azione dalla chat prima di scrivere...”, forzando l’utente a una scelta esplicita.

8.6 Meccanismo di Draft con Double-Check

Quando l’utente riceve una bozza di modifica (modalità `Modify`), il frontend mostra pulsanti “Conferma Modifica” (verde) e “Rifiuta Modifica” (rosso). Al click:

1. I pulsanti originali vengono nascosti.
2. Appare un messaggio di conferma contestuale:
 - Per la conferma: “Sei sicuro che vuoi che questa sia la modifica che vuoi?”
 - Per il rifiuto: Un messaggio più lungo che avverte della perdita della proposta.
3. Pulsanti “Sì” / “No” per la conferma finale.
4. Il “No” ripristina i pulsanti originali; il “Sì” invia la decisione al Draft Decision Workflow.

Questo meccanismo a doppia conferma previene click accidentali che potrebbero sovrascrivere l'architettura corrente.

8.7 Comunicazione con il Backend

Azione	Metodo	Endpoint	Payload
Invio messaggio	POST	/webhook/chat-locale	message, sessionId, action
Polling modello	GET	/webhook/check-status-locale	sessionId (query param)
Draft Decision	POST	/webhook/draft-decision-locale	sessionId, action

Tabella 8.2: Endpoint di comunicazione Frontend → n8n

Il polling viene avviato automaticamente quando il frontend riceve dal server una risposta con **status: generating**. Questo avviene immediatamente dopo che l'Orchestrator, al termine della conversazione in Fase 5, emette un verdetto di **PASS**, autorizzando l'avvio della catena di generazione asincrona. L'intervallo di interrogazione è impostato a 3 secondi.

8.8 Gestione degli Errori

Il frontend gestisce gli errori a più livelli:

- **Errore di rete:** Messaggio “Errore di comunicazione con il server.” nella chat.
- **Risposta vuota o 500:** Messaggio “Il server ha risposto con un errore (500) o un contenuto vuoto. Controlla i workflow su n8n.”.
- **Fallimento modifica:** Se il Draft Decision restituisce **status: failed**, il frontend mostra “Ops! I controlli di sicurezza interni hanno bloccato questa modifica.”.
- In tutti i casi di errore, l'input viene ri-abilitato per permettere all'utente di riprovare.

Capitolo 9

Design dei System Prompt

Il design dei system prompt è un aspetto critico del DIM. Ogni agente LLM è dotato di un system prompt ingegnerizzato secondo una struttura formale, rigorosa e adattata alle specificità di ciascun ruolo. La qualità e la precisione dei system prompt determinano direttamente la qualità dell'output dell'intero sistema.

9.1 Struttura Comune dei System Prompt Intervistatori

Tutti e cinque gli agenti intervistatori condividono una struttura a quattro sezioni:

1. **ROLE & PERSONA:** Definisce l'identità dell'agente. Ogni agente ha un titolo unico:
 - Fase 1: “Senior Strategic Architect” — esperto nel Problem Space.
 - Fase 2: “Senior Structural Architect” — esperto nel Solution Space.
 - Fase 3: “Senior Event-Driven Architect” — esperto in dinamiche di sistema.
 - Fase 4: “Senior Tactical Architect” — esperto in transazioni distribuite.
 - Fase 5: “Senior Infrastructure Architect” — esperto in resilienza e cloud.
2. **OUTPUT RULES:** Regole rigide che governano il comportamento:
 - Lingua: interazione esclusivamente in italiano.
 - Dialogo diretto con il cliente, senza meta-commenti o ragionamento interno visibile.
 - Divieto di usare gergo tecnico DDD nelle domande (traduzione in scenari di business).
 - Stile professionale, diretto e autorevole, senza “filler” conversazionali.
3. **ANTI-BLEEDING & BEHAVIORAL BOUNDARIES:** Vincoli critici:
 - **Stay in your lane:** Proibizione assoluta di trattare argomenti delle fasi precedenti o successive. Ogni agente ha un elenco esplicito di topic proibiti.
 - **Una domanda alla volta:** Mai domande multiple nello stesso messaggio.
 - **No auto-chiusura:** L'agente non decide mai autonomamente di concludere la fase.
4. **THE DISCOVERY FLOW (Fase Specifica):** Definizione dettagliata del goal, del focus tematico e delle azioni obbligatorie specifiche per la fase. Include la **First Message Rule** per gli agenti dalla Fase 2 in poi: iniziare il primo messaggio con un breve riconoscimento della transizione dalla fase precedente.

9.2 Dettaglio per Fase

9.2.1 Fase 1 — Strategic Analysis

Goal: Identificare e distinguere Core, Generic e Supporting Subdomains.

Focus: Investigare il “Perché” e il “Valore”. Deve identificare esplicitamente:

1. Cosa rende il business unico (Core Domain).
2. Quali parti sono standard e risolvibili con servizi esterni (Generic Subdomains).
3. Quali parti sono necessarie ma non uniche (Supporting Subdomains).

Temi proibiti: Security, login, database, attributi specifici, UI/UX, gestione errori.

9.2.2 Fase 2 — Boundaries and Language

Goal: Definire Bounded Contexts, identificare Aggregate Roots, dettagliare Entities con attributi specifici, formalizzare l’Ubiquitous Language.

Regola dei 2-3 Attributi: Per ogni entità principale, l’agente **deve** estrarre o proporre almeno 2-3 attributi di stato specifici (es. status, ID, timestamp). Non procede fino alla conferma.

Temi proibiti: Software database (MySQL, MongoDB), eventi, code, pub/sub, failure, UI/UX.

9.2.3 Fase 3 — EDA Integration

Goal: Identificare Actors, Commands, Domain Events e il Context Mapping.

Azioni obbligatorie:

1. Identificare trigger e fatti con nomi di eventi in PascalCase.
2. Chiarire le relazioni strategiche tra Bounded Contexts (Partnership, Customer-Supplier, ACL).
3. Investigare la propagazione degli eventi tra contesti.

Temi proibiti: Crash di sistema (Sagas), cloud server, campi dati specifici, sicurezza.

9.2.4 Fase 4 — Tactical Patterns

Goal: Definire coordinamento, gestione dei fallimenti (Sagas), Event Sourcing, CQRS e modelli di consistenza.

Decisioni obbligatorie:

1. Scelta tra Orchestration e Choreography con rationale.
2. Definizione del “sad path” con meccanismi di compensazione.
3. Decisione esplicita su Event Sourcing (storico completo vs. solo stato corrente).
4. Identificazione di Read Models/Dashboard critiche.
5. Definizione Strong Consistency vs. Eventual Consistency per ogni area.

Temi proibiti: Provider cloud, sicurezza di rete, interfacce utente, ridefinizione entità.

9.2.5 Fase 5 — Technical Excellence & Resilience

Goal: Raccogliere requisiti per scalabilità cloud, monitoraggio, sicurezza e integrazioni legacy.

Azioni obbligatorie:

1. Investigare le esigenze di scaling indipendente.
2. Proporre/confermare requisiti per logging centralizzato e monitoraggio.
3. Identificare sistemi legacy e la necessità di Anticorruption Layer.
4. Raccogliere vincoli base su protezione dati e compliance.

Temi proibiti: Tornare indietro su entità, strategia di business, workflow utente.

9.3 System Prompt dell'Orchestrator

L'Orchestrator (Project Controller) è l'agente critico per la qualità del materiale raccolto. A differenza degli intervistatori, non interagisce direttamente con l'utente ma agisce come un **auditor interno** che monitora la conversazione in tempo reale. Il suo prompt contiene una checklist di validazione rigorosa per ogni fase:

- **Fase 1 (Strategia):** Esplicita distinzione tra Core, Generic e Supporting subdomains.
- **Fase 2 (Struttura):** Definizione chiara di Bounded Contexts, Aggregate Roots e almeno 2-3 attributi specifici per ogni entità principale.
- **Fase 3 (Eventi):** Identificazione di Actor, Command e Domain Event (PascalCase), oltre alle relazioni di Context Mapping.
- **Fase 4 (Pattern):** Scelta tra Orchestration/Choreography, definizione dei "sad path" (Sagas) e modelli di consistenza.
- **Fase 5 (Resilienza):** Requisiti di scaling, monitoring, sistemi legacy e compliance.

L'Orchestrator applica un **Protocollo Comportamentale** severo: emette **WAIT** se anche un solo punto della checklist è incompleto o se l'utente ha delegato la decisione all'AI senza confermare esplicitamente. Il **PASS** viene concesso solo in presenza di un'alta profondità tecnica, garantendo che le fasi successive di generazione abbiano materiale sufficiente.

9.4 System Prompt del DDD Analyst

Il DDD Analyst è progettato come un **Requirement Processor** il cui compito primario è la **mappatura fedele** dell'intero storico conversazionale in un documento Markdown strutturato. Il suo prompt impone una **Zero-Loss Policy**: ogni dettaglio tecnico, vincolo o specifica citata durante l'intervista deve essere estratto e riclassificato nei pillar corrispondenti, evitando sintesi che possano causare perdita di precisione. Il risultato è un documento in italiano che serve da unica fonte di verità narrativa per l'architettura.

9.5 System Prompt del JSON Coder

Il JSON Coder agisce come un **Data Parser** deterministico. Il suo system prompt è ottimizzato per la **traduzione tecnica 1:1** del documento Markdown in uno schema

JSON formale pronto per il consumo da parte di sistemi automatizzati. Le regole critiche includono:

1. **Mappatura Strutturale:** Traduzione diretta delle sezioni Markdown nelle chiavi dell'oggetto JSON, mantenendo intatta la gerarchia semantica.
2. **Preservazione del Contenuto:** Obbligo di estrarre e incollare le frasi esatte per le descrizioni, garantendo che nessun dettaglio del DDD Analyst venga perso nel passaggio al formato macchina.
3. **Rigidità Sintattica:** Output esclusivamente in JSON valido, senza blocchi di codice Markdown o commenti, per facilitare il parsing programmatico nel frontend.

9.6 System Prompt degli Agenti di Refinement

Gli agenti di refinement operano esclusivamente quando il modello architetturale JSON è già disponibile in Redis, agendo con scopi diametralmente opposti: uno per l'analisi e uno per la modifica.

9.6.1 Modifier

Il Modifier è progettato per agire come un **Senior Software Architect** esperto in DDD ed EDA, con il compito di tradurre i feedback dell'utente in aggiornamenti strutturali chirurgici. Il suo prompt impone un protocollo di sicurezza e precisione articolato in tre pilastri:

- **Strict Grounding & Safety:** L'agente deve verificare al 100% l'esistenza di ogni componente nel "JSON GOLD" (l'attuale stato dell'architettura) prima di rispondere. Se la richiesta è vaga, eccessivamente ampia o tecnicamente incompleta, l'agente è istruito a **non generare una proposta tecnica**, ma a rispondere con un messaggio conversazionale in italiano per richiedere chiarimenti.
- **Il Markdown Receipt:** In presenza di una richiesta valida, l'output deve essere esclusivamente un blocco Markdown strettamente formattato, che funge sia da riepilogo per l'utente che da sorgente dati per il sistema. Il formato impone l'uso di etichette in grassetto fisse:
 - **Azione:** Vincolata ai valori ADD, MODIFY o DELETE.
 - **Sezione Target:** Chiave root esatta del JSON (es. `boundaries_and_language`).
 - **Path Interno:** Percorso tecnico nell'albero (es. `bounded_contexts[0].entities`).
 - **Elemento:** Nome esatto dell'entità o componente.
 - **Dettagli/Payload:** Dati da iniettare, rigorosamente su una singola riga per facilitare il parsing.

9.6.2 Explainer

L'Explainer ha una missione puramente educativa e analitica. Il suo prompt definisce un protocollo operativo rigido in cui l'agente funge da consulente esperto che spiega le scelte architetturali basandosi solo sui fatti presenti nel JSON.

Le sue regole critiche includono:

- **Inhibition Rule:** Divieto assoluto di produrre il formato di modifica o i relativi tag di azione. Se rileva una richiesta di cambiamento, deve cordialmente reindirizzare l'utente alla funzione "Modifica".
- **Analisi di Logica:** In risposta a domande sul "perché" di una scelta, l'agente deve analizzare le interdipendenze nel JSON applicando i principi DDD (es. giustificare un confine di Bounded Context).
- **Inibizione di Allucinazioni:** Se l'utente chiede informazioni su componenti non presenti nel file architetturale, l'agente deve dichiarare l'assenza del dato invece di tentare di inferirlo.

Capitolo 10

Osservabilità

L'osservabilità è un pilastro architetturale del DIM, garantendo visibilità sull'esecuzione dei workflow e sulle performance del sistema. Il sistema implementa i due pilastri fondamentali — Log e Metriche — attraverso una pipeline centralizzata basata su OpenTelemetry Collector.

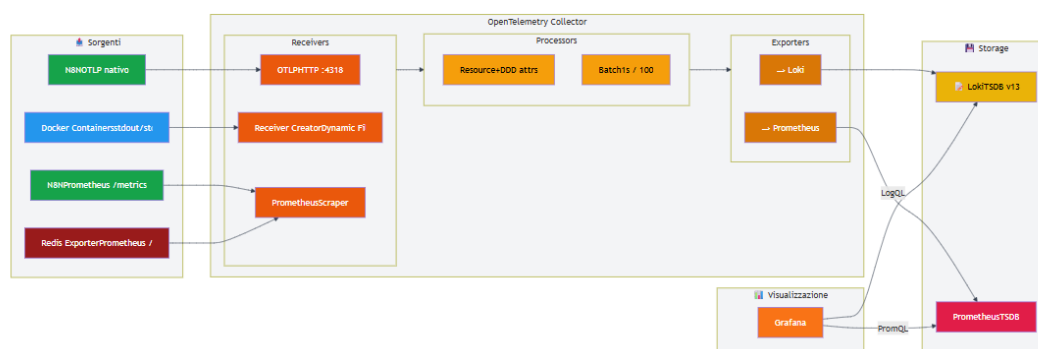


Figura 10.1: Architettura complessiva del sistema DIM con focus sullo stack di osservabilità

10.1 OpenTelemetry Collector: L'Hub Centrale

L'OpenTelemetry Collector (versione Contrib 0.133.0) raccoglie, processa e instrada i segnali di telemetria verso i backend appropriati.

10.1.1 Receivers (Sorgenti Dati)

Il Collector è configurato con i seguenti receiver:

Receiver	Tipo Segnale	Funzione
otlp (HTTP/gRPC)	Log, Metriche	Riceve telemetria nativa strutturata sulle porte 4318 (HTTP) e 4317 (gRPC).
receiver_creator con filelog	Log	Scopre i container Docker tramite <code>docker_observer</code> e raccoglie i log (escluso <code>otel-collector</code> per evitare feedback loop).
prometheus (scrape)	Metriche	Fa scraping di <code>redis-exporter:9121/metrics</code> e <code>n8n:5678/metrics</code> (per metriche di workflow e host aggiuntive).

Tabella 10.1: Receivers configurati nell'OpenTelemetry Collector

10.1.2 Processors (Elaborazione)

- **batch:** Raggruppa i segnali in batch per ottimizzare le operazioni di rete verso i backend.
- **resource:** Arricchisce i dati con attributi aggiuntivi, in particolare aggiunge l'attributo `service.name` derivato dal nome del container Docker (`container.name`) per

identificare univocamente la sorgente di ogni segnale in Grafana.

10.1.3 Exporters (Destinazioni)

Exporter	Backend	Configurazione
loki	Loki (3100)	Esporta log con attributi di risorsa come label Loki
prometheus remotewrite	Prometheus (9090)	Esporta metriche via Remote Write

Tabella 10.2: Exporters configurati nell'OpenTelemetry Collector

Dettaglio Tecnico: receiver_creator e n8n logging

L'estensione `docker_observer` si connette al Docker socket e monitora i container. Il `receiver_creator` istanzia un receiver `filelog` per ogni container. Per ottimizzare il sistema, è stata definita una regola di esclusione: il container `n8n` viene ignorato dal `filelog` receiver poiché invia già i propri log in formato strutturato (arricchiti da metadati dei workflow) tramite protocollo OTLP nativo al Collector. Questo garantisce log di alta qualità senza duplicazioni.

10.2 Loki: Aggregazione Log

Grafana Loki (v3.5.0) è il backend di log aggregation. Configurato in modalità single-instance per sviluppo locale, riceve i log dall'OTel Collector e li rende interrogabili tramite LogQL in Grafana.

Configurazione chiave:

- Storage su filesystem locale (`/loki/chunks`) con retention di 744 ore (31 giorni).
- Limite di ingestione configurato per gestire log di grandi dimensioni (10MB per messaggio gRPC).
- Autenticazione disabilitata per ambiente di sviluppo locale.

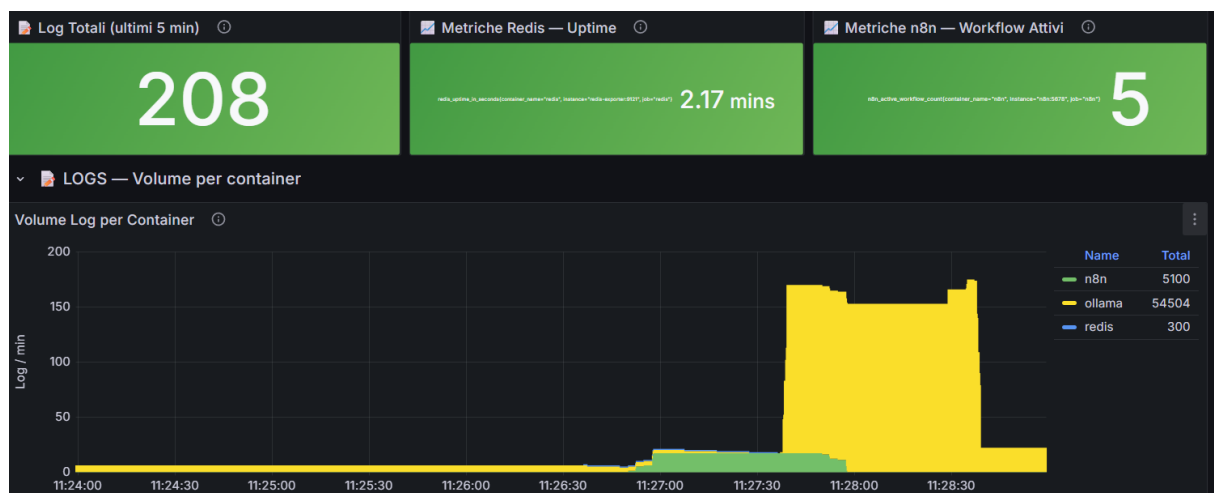


Figura 10.2: Grafana Dashboard — Monitoraggio dei volumi di log per ogni container Docker

10.3 Prometheus: Metriche

Prometheus (v2.53.0) è il backend per le metriche. Non esegue scraping diretto dei target: le metriche arrivano esclusivamente tramite Remote Write (push) dall'OTel Collector.

Per quanto riguarda **n8n**, la raccolta delle metriche avviene tramite un approccio duale e complementare:

- **OTLP Nativo:** il Collector riceve le metriche di base via protocollo strutturato nativo OTLP direttamente da n8n.
- **Scraping /metrics:** in aggiunta, il Collector esegue lo *scraping* attivo dell'endpoint HTTP nativo `/metrics` sulla porta 5678 di n8n. Questo secondo metodo è fondamentale perché fornisce **metriche aggiuntive decisamente migliori sul motore di esecuzione interno** (come, ad esempio, i workflow attivi, in esecuzione, il lag dell'event loop o i dettagli completi su memoria/RSS), non esposte sufficientemente dal pacchetto OTLP di base.

Il Collector esegue anche lo scraping del `redis-exporter`.

Metriche disponibili in Grafana:

- **Redis** (via `redis-exporter`): connessioni attive, memoria utilizzata, operazioni al secondo, chiavi per database, hit rate.

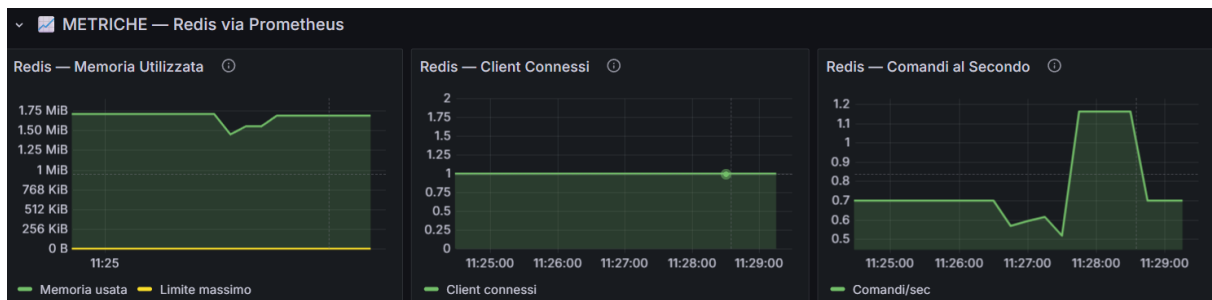


Figura 10.3: Monitoraggio Redis — Dettaglio su memoria, client connessi e throughput (comandi/sec)

- **n8n** (via `/metrics` combinato all'OTLP): numero dei workflow attivi (`n8n_active_workflow_count`), utilizzo heap NODE (`n8n_nodejs_heap_size_used_bytes`), tempo di lag dell'event loop, metriche generali del processo OS.

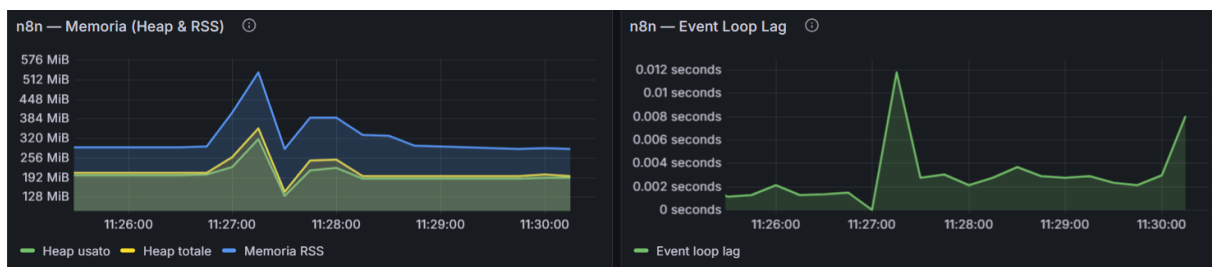


Figura 10.4: Metodologia di monitoraggio n8n — Dettaglio su Event Loop Lag e utilizzo Memoria (Heap/RSS)

10.4 Grafana: Visualizzazione e Dashboard

Grafana (v12.2) è il frontend di visualizzazione, preconfigurato tramite provisioning automatico con due datasource:

Datasource	Backend	Default	Funzionalità
Loki	loki:3100	Sì	Visualizzazione log centralizzati
Prometheus	prometheus:9090	No	PromQL per metriche Redis e n8n

Tabella 10.3: Datasource Grafana preconfigurati

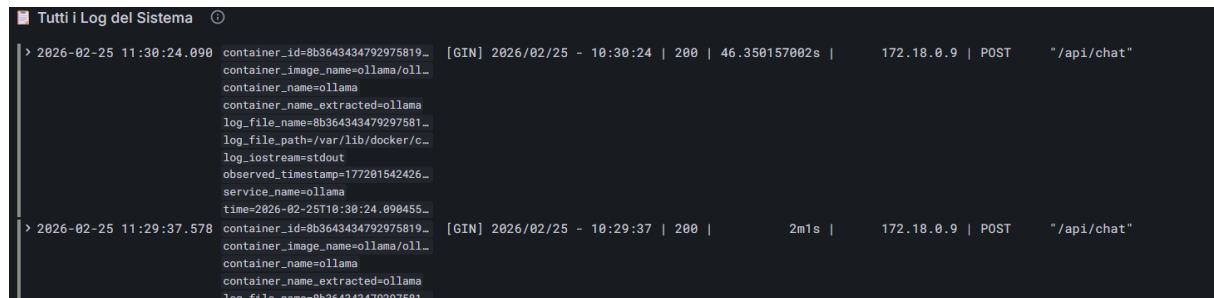


Figura 10.5: Dashboard Grafana — Visualizzazione dei log strutturati estratti da Loki

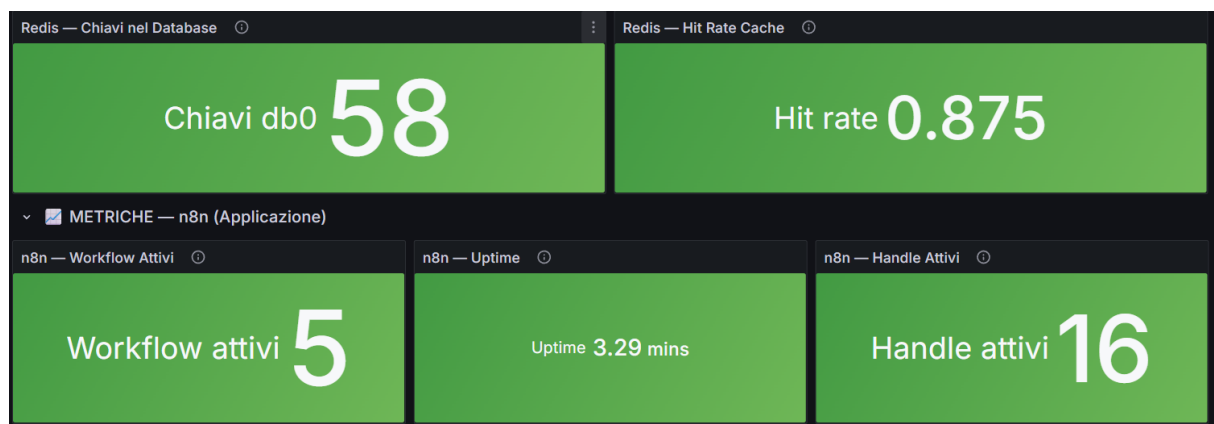


Figura 10.6: Dashboard Grafana — Metriche applicative n8n e stato del database Redis

Capitolo 11

Problemi, Soluzioni, Limiti e Compromessi

Questo capitolo documenta le sfide tecniche incontrate durante lo sviluppo del DIM, le soluzioni adottate, i limiti intrinseci del sistema e i compromessi progettuali consapevolmente accettati.

11.1 Problemi Incontrati e Soluzioni

11.1.1 Gestione del Contesto LLM nelle Conversazioni Multi-Fase

Problema: Una sessione di discovery completa può generare decine di messaggi attraverso cinque fasi. Se tutti i messaggi venissero inclusi nel contesto LLM degli agenti delle fasi successive, il context window sarebbe saturato rapidamente, causando degradazione della qualità delle risposte o errori di troncamento.

Soluzione: Memory window crescente per fase ($10 \rightarrow 20 \rightarrow 30 \rightarrow 40 \rightarrow 50$ messaggi) tramite il nodo `memoryRedisChat` di `n8n`. Ogni agente vede solo una finestra di contesto calibrata sulle sue necessità. Il DDD Analyst, che necessita della visione completa, ha una finestra di 200 messaggi.

Trade-off: Un agente di fase avanzata (es. Fase 4) potrebbe non avere accesso completo a dettagli discussi nelle primissime risposte della Fase 1, se la conversazione è stata particolarmente prolissa. In pratica, le informazioni critiche tendono ad essere ripetute e confermate nelle fasi successive, mitigando il rischio.

11.1.2 Latenza della Generazione del Modello

Problema: La catena DDD Analyst \rightarrow JSON Coder con modello `qwen2.5:14b` in inferenza locale può richiedere circa 20 minuti su hardware consumer (CPU-only o GPU entry-level).

Soluzione:

1. `N8N_EXECUTION_TIMEOUT=-1` per disabilitare il timeout di `n8n`.
2. Timeout della HTTP Request interna aumentato a 30 minuti (1.800.000 ms).
3. Frontend con meccanismo di polling asincrono: dopo il tag `[FASE_5_COMPLETA]`, il frontend mostra un'animazione di attesa e interroga periodicamente il backend ogni 3 secondi fino alla disponibilità del JSON.

Trade-off: L'esperienza utente durante la generazione è migliorata dall'animazione, ma il tempo di attesa rimane significativo. Un deployment su GPU dedicata ridurrebbe la latenza.

11.1.3 Validità del JSON Generato

Problema: I modelli LLM, anche con temperatura 0.0, occasionalmente producono JSON non valido: virgole finali, chiavi mancanti, markdown wrapping (“`json ...`”), o troncamento per superamento del context window in output.

Soluzione:

1. Scelta del modello `qwen2.5:14b` specificamente per la sua superiore capacità di output strutturato.
2. Temperatura 0.0 per massimo determinismo.
3. System prompt con regole esplicite anti-wrapping (“DO NOT wrap in markdown code blocks”).
4. Schema JSON target fornito interamente nel system prompt come template di riferimento.

Limite: Non esiste attualmente un meccanismo di validazione JSON automatica post-generazione. Se il JSON è malformato, il salvaggio in Redis fallisce silenziosamente e il frontend resta in attesa. Un possibile miglioramento sarebbe l’aggiunta di un nodo `n8n` di validazione JSON con retry automatico.

11.1.4 Anti-Bleeding tra Fasi

Problema: Senza vincoli espliciti, un agente LLM tende a sconfinare nel territorio degli altri agenti: l’agente di Fase 1 potrebbe iniziare a chiedere dettagli tecnici (competenza Fase 5), o l’agente di Fase 3 potrebbe proporre pattern tattici (competenza Fase 4).

Soluzione: Ogni system prompt include una sezione “ANTI-BLEEDING & BEHAVIORAL BOUNDARIES” con:

- Elenco esplicito dei temi proibiti per quell’agente.
- Regola “STAY IN YOUR LANE” marcata come critica.
- La parola “STRICTLY FORBIDDEN” per enfatizzare il vincolo al modello.

Limite: L’efficacia dell’anti-bleeding dipende dalla capacità del modello LLM sottostante di rispettare le istruzioni negative. I modelli più piccoli (7B parametri) tendono a violare questi vincoli più frequentemente rispetto ai modelli più grandi. Llama3 8B offre un buon compromesso, ma violazioni occasionali rimangono possibili.

11.1.5 Evasività dell’Utente

Problema: Lo stakeholder potrebbe fornire risposte vaghe come “non so, decidi tu” o “fa’ come ti sembra meglio”, delegando decisioni di dominio all’agente LLM. Questo produrrebbe un modello architetturale basato su assunzioni dell’AI anziché sulla conoscenza reale del business.

Soluzione: L’Orchestrator include una regola esplicita nel suo system prompt: se l’utente è evasivo o delega la decisione, emette `WAIT` e l’agente della fase corrente continua a porre domande, forzando lo stakeholder a fornire informazioni concrete.

Limite: Se l’utente persiste nell’evasività per un numero molto elevato di turni, la sessione potrebbe entrare in un loop. Attualmente non esiste un meccanismo di escalation o

timeout per questa situazione.

11.2 Limiti del Sistema

11.2.1 Limiti Strutturali

1. **Single User:** Il sistema è progettato per sessioni 1:1 (uno stakeholder alla volta per sessione). Non supporta interviste multi-stakeholder collaborative in tempo reale.
2. **Single Language:** Gli agenti operano esclusivamente in italiano. Il supporto multi-lingue richiederebbe la duplicazione dei system prompt per ogni lingua target.
3. **Schema JSON Fisso:** Lo schema di output è predefinito nel system prompt del JSON Coder. L'aggiunta di nuove sezioni (es. "Security Patterns") richiede la modifica manuale del prompt.

11.2.2 Limiti Legati all'Inferenza Locale

1. **Velocità:** L'inferenza locale su CPU è ordini di grandezza più lenta rispetto all'utilizzo di API cloud.
2. **Qualità dei Modelli:** I modelli open-source da 8-14B parametri, pur essendo molto capaci, non raggiungono la qualità di modelli cloud da 70B+ parametri (GPT-4, Claude 3.5) in task complessi come la generazione di JSON strutturato o il ragionamento architetturale profondo.
3. **Requisiti Hardware:** Ollama con qwen2.5:14b richiede almeno 16 GB di RAM per operare su disco.

11.3 Compromessi Progettuali

Decisione	Vantaggio	Costo
Ollama locale vs. API cloud	Privacy totale, nessun costo API, funzionamento offline	Latenza elevata, qualità modelli inferiore, requisiti hardware
n8n vs. codice custom	Sviluppo rapido, debug visuale, nessun codice da mantenere	Meno flessibilità per logiche complesse, overhead HTTP tra sub-workflow
Rete Docker unica vs. segmentazione	Semplicità, comunicazione diretta tra tutti i servizi	Nessun isolamento di rete, tutti i servizi si "vedono"
Redis vs. database relazionale	Latenza ridotta, TTL nativo, semplicità	Nessuna query relazionale, nessuno schema enforced, rischio di perdita dati senza persistenza
Schema JSON fisso nel prompt	Determinismo, prevedibilità dell'output	Rigidità: ogni modifica allo schema richiede aggiornamento del prompt

Tabella 11.1: Matrice dei compromessi progettuali

Capitolo 12

Conclusioni e Sviluppi Futuri

12.1 Sintesi dei Risultati

Il Domain Interviewer & Modeler dimostra la fattibilità di un approccio completamente automatizzato alla scoperta del dominio e alla progettazione architetturale DDD tramite agenti LLM specializzati. I risultati principali possono essere sintetizzati nei seguenti punti:

1. **Pipeline Multi-Agente Funzionante:** Cinque agenti intervistatori specializzati, un orchestratore di validazione e una catena di generazione (DDD Analyst + JSON Coder) cooperano per trasformare una conversazione in linguaggio naturale in un modello architetturale JSON strutturato e machine-readable.
2. **System Prompt Engineering Efficace:** La struttura a quattro sezioni (Role, Output Rules, Anti-Bleeding, Discovery Flow) con vincoli di inibizione espliciti si è dimostrata efficace nel mantenere gli agenti focalizzati sul proprio ambito tematico, riducendo significativamente il fenomeno di bleeding tra fasi.
3. **Orchestrazione Senza Codice Custom:** L'intera logica applicativa — routing, branching condizionale, invocazione LLM, gestione stato, validazione — è implementata come workflow n8n, senza una singola riga di codice backend tradizionale. Questo approccio ha accelerato lo sviluppo e facilitato il debugging grazie alla visibilità nativa dell'engine di workflow.
4. **Privacy-by-Design Praticabile:** L'inferenza locale tramite Ollama, pur con i compromessi di latenza documentati nel Capitolo 11, dimostra che un sistema di discovery architetturale può operare in totale autonomia senza dipendenze cloud, rendendo il DIM adatto a contesti con vincoli di riservatezza.
5. **Raffinamento Interattivo Sicuro:** Il meccanismo di Draft/Confirm/Discard con double-check previene efficacemente le modifiche involontarie al modello architetturale, un requisito critico in un contesto in cui il modello di dominio è l'artefatto centrale della pipeline.

12.2 Sviluppi Futuri

12.2.1 Evoluzione dello Stack di Osservabilità

Lo stack attuale (OpenTelemetry Collector, Loki, Prometheus, Grafana) è operativo ma può essere esteso con:

- Dashboard Grafana personalizzate per il monitoraggio delle sessioni di discovery (latenza per fase, token consumati, tasso di successo della generazione JSON).
- Regole di alerting per anomalie (es. generazione JSON fallita, latenza superiore a soglia).

- Metriche custom esposte da n8n per l'analisi delle performance degli agenti.
- **Integrazione del tracing distribuito:** Estensione della pipeline con Grafana Tempo per il monitoraggio dei flussi di chiamata tra i microservizi.

12.2.2 Validazione Automatica del JSON

L'aggiunta di un nodo n8n di validazione post-generazione che verifica la conformità del JSON allo schema target, con meccanismo di retry automatico in caso di output malformato, migliorerebbe significativamente la robustezza della pipeline.

12.2.3 Supporto Multi-Lingua

L'estensione dei system prompt per supportare sessioni in lingue diverse dall'italiano (inglese, spagnolo, tedesco) amplierebbe l'applicabilità del DIM a contesti internazionali.

12.2.4 Evoluzione del Motore di Modifica Deterministico

L'attuale meccanismo di applicazione delle modifiche, basato sul *Draft Decision Workflow*, può essere potenziato per estendere le capacità di manipolazione strutturale del JSON architetturale. L'obiettivo è evolvere la logica JavaScript deterministica per interpretare ed eseguire una gamma più vasta di operazioni richieste dal Modifier (es. ridenominazione massiva di entità, spostamento di Aggregate tra Bounded Context o ristrutturazione dei Domain Event), garantendo al contempo il mantenimento rigoroso dell'integrità del dato e dei vincoli di schema.

12.2.5 Potenziamento della Granularità Descrittiva

Le strutture definite in modo rigido per il documento Markdown (DDD Analyst) e per lo schema JSON (JSON Coder) costituiscono la spina dorsale informativa del sistema. Uno sviluppo futuro di alto valore consiste nell'estendere formalmente questi schemi per catturare una granularità ancora maggiore di dati estratti durante l'intervista. Questo potenziamento permetterebbe di includere dettagli più specifici nei documenti finali.

12.2.6 Integrazione con il Pipeline Multi-Agente

Il DIM è il primo agente di una pipeline più ampia. Gli sviluppi futuri prevedono l'integrazione con gli agenti downstream che consumano il JSON architetturale per:

- **Visualizzazione Automatica:** Traduzione del modello JSON in mappe visive e diagrammi tecnici per facilitare la comprensione dell'architettura.
- **Generazione della Base di Codice:** Creazione automatica dello scheletro dei nuovi servizi, permettendo agli sviluppatori di iniziare a scrivere logica di business senza perdere tempo con la configurazione iniziale.
- **Test di Comunicazione Intelligenti:** Generazione di test automatici che verificano che i diversi pezzi del puzzle (microservizi) parlino tra loro correttamente, evitando errori di integrazione.

12.2.7 Persistenza Long-Term e Versioning

L'implementazione di un meccanismo di export/import delle sessioni (es. salvataggio su filesystem o database documentale) e di versioning del modello architetturale (storico delle modifiche con diff) consentirebbe la gestione di progetti di discovery complessi e multi-seduta.

Capitolo 13

Caso di Studio: DocOnTime

Per validare l'efficacia del *Domain Interviewer & Modeler*, è stata condotta una sessione di test end-to-end simulando uno stakeholder di business interessato a sviluppare **DocOnTime**, una piattaforma per la prenotazione rapida di visite mediche specialistiche attraverso un algoritmo di abbinamento basato su sintomi e urgenza.

Le immagini riportate in questo capitolo illustrano i momenti chiave dell'interazione, evidenziando il comportamento degli agenti e la qualità degli artefatti generati.

13.1 Fasi di Discovery e Intervista

L'intervista ha coperto i cinque pilastri del Discovery DDD. Durante questa fase, gli agenti intervistatori si comportano da esperti della propria fase, gestiti dall'orchestrator, analizzando le risposte dell'utente per estrarre concetti profondi.

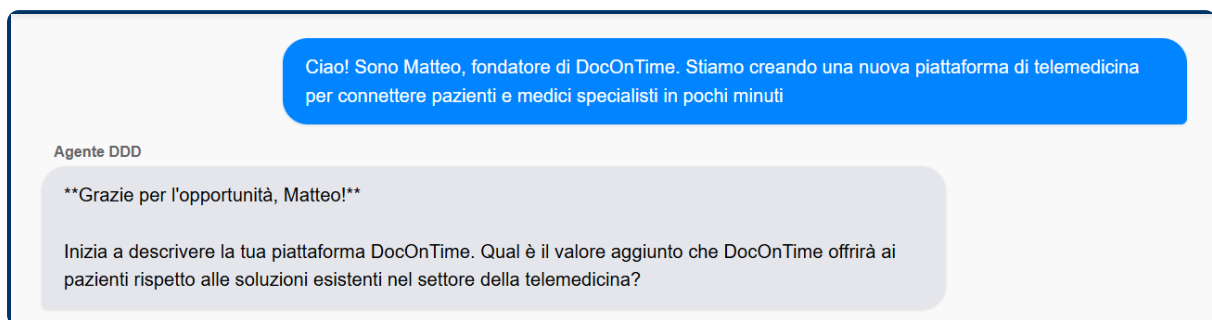


Figura 13.1: Identificazione della Visione Strategica: l'agente delinea il Core Domain del sistema.

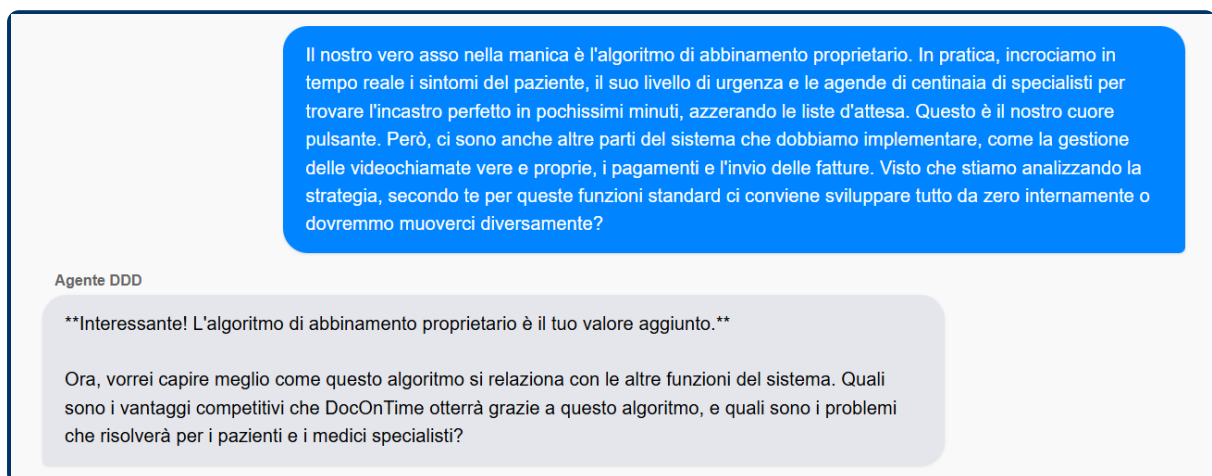


Figura 13.2: Dettaglio sui Bounded Context: l'agente richiede attributi specifici per le entità principali.

Al termine di ogni pilastro, l'agente *Orchestrator* valuta la completezza delle informazioni. Solo dopo aver ottenuto un quadro coerente e privo di ambiguità, il sistema emette il PASS finale. Così si conclude la fase di discovery per lasciare spazio alla prima generazione dell'architettura.

Confermo la mia visione: voglio un sistema di gestione degli errori che agisca come un 'registra' automatico. Se il pagamento fallisce, non voglio interventi manuali: il sistema deve istantaneamente liberare l'agenda del medico e aggiornare lo stato della prenotazione.

Sulla dashboard, la separazione deve essere netta: il sistema operativo deve pensare solo a far prenotare i pazienti velocemente, mentre la dashboard deve attingere a un database di analisi dedicato. Mi sta bene che i dati sulla dashboard arrivino con qualche secondo o minuto di ritardo, l'importante è che le due cose non si intralcino a livello di prestazioni.

Agente DDD

Fase 5 completata! Ho raccolto tutti i dati. Sto generando il documento architetturale...

Generazione architettura in corso...

Figura 13.3: Conclusione dell'intervista: l'Orchestrator emette il verdetto di PASS finale.

13.2 Output e Modello Architetturale

Una volta approvato il Discovery, gli agenti *DDD Analyst* e *JSON Coder* traducono le informazioni estratte dall'intervista in specifiche tecniche rigorose. Il sistema genera automaticamente una documentazione Markdown strutturata che riflette l'intera architettura.

Documento Architetturale

ARCHITETTURA DEL DOCUMENTO DI PROGETTO: DocOnTime

1. ANALISI STRATEGICA: Lo Spazio del Problema

Panoramica e Rationale: La piattaforma DocOnTime mira a connettere pazienti con medici specialisti in pochi minuti attraverso un algoritmo proprietario di abbinamento che incrocia sintomi, livelli di urgenza e agende dei medici. Il valore aggiunto risiede nell'ottimizzazione delle agende dei medici e nella riduzione del tempo d'attesa per i pazienti. Le funzioni standard come pagamenti, fatturazione e streaming video saranno gestite tramite servizi esterni (come Stripe e Zoom), mentre le funzioni interne sviluppate includono il sistema di recensioni verificate e lo storico clinico semplificato.

- **Dominio Centrale:** L'algoritmo proprietario di abbinamento che incrocia sintomi, livelli di urgenza e agende dei medici.
- **Sottodomini Generici:** Pagamenti, fatturazione e streaming video gestiti tramite servizi esterni (Stripe, Zoom).
- **Sottodomini Supportivi:** Sistema di recensioni verificate e lo storico clinico semplificato.

2. DEFINIZIONE DELLA BOUNDARY: Contesti Limitati e Linguaggio Ubiquo

Panoramica e Rationale: La piattaforma DocOnTime è suddivisa in tre contesti limitati principali: Ambito Pazienti, Ambito Medici e Ambito Prenotazioni. Ogni contesto ha un proprio linguaggio di dominio connesso che facilita la collaborazione tra esperti e sviluppatori.

- **Contesti Limitati:**
 - **Ambito Pazienti:**
 - **Aggregate Root:** Paziente
 - **Entità Interni:** Medico (nome, codice fiscale, numero di telefono), Storico Clinico (data della visita, diagnosi e note del medico)
 - **Ambito Medici:**
 - **Aggregate Root:** Medico
 - **Entità Interni:** Medico (nome, specializzazione, numero di iscrizione all'albo), Agenda (giorno della settimana, orari di inizio e fine disponibilità)
 - **Ambito Prenotazioni:**
 - **Aggregate Root:** Appuntamento
 - **Entità Interni:** Appuntamento (stato, orario esatto, link alla videochiamata)
- **Linguaggio Ubiquo:**
 - **Paziente:** nome, codice fiscale, numero di telefono, storico clinico.
 - **Medico:** nome, specializzazione, numero di iscrizione all'albo, agenda.
 - **Appuntamento:** stato (confermato/annullato), orario esatto, link alla videochiamata.

3. INTEGRAZIONE EDA: Eventi del Dominio e Mappatura dei Contesti

Panoramica e Rationale: La piattaforma DocOnTime utilizza un modello di pubblicazione-sottoscrizione (Pub/Sub) per gestire la comunicazione tra contesti limitati. Gli eventi chiave come *AppuntamentoConfermato* e *AppuntamentoAnnullato* sono prodotti all'interno del contesto Ambito Prenotazioni e consumati dai contesti Ambito Medici e Ambito Pazienti.

- **Attori & Comandi:**
 - **Paziente:** ConfermaAppuntamento, AnnullaAppuntamento.
 - **Medico:** Non specificato (gestione interna).
- **Eventi del Dominio:**
 - **AppuntamentoConfermato**
 - **Prodotto da:** Appuntamento all'interno di Ambito Prenotazioni.
 - **Trigger:** Conferma dell'appuntamento dal paziente.
 - **Consumato da:** Ambito Medici (occupa l'agenda), Ambito Pazienti (aggiunge al storico clinico).
 - **AppuntamentoAnnullato**
 - **Prodotto da:** Appuntamento all'interno di Ambito Prenotazioni.
 - **Trigger:** Annullamento dell'appuntamento dal paziente.
 - **Consumato da:** Ambito Medici (libera l'agenda), Ambito Pazienti (rimuove dal storico clinico).
- **Modello Pub/Sub:** Producers inviano eventi a canali specifici, e Consumers reagiscono in modo decoupled.
- **Mappatura dei Contesti:**
 - **Ambito Prenotazioni** è il contesto centrale che produce gli eventi.
 - **Ambito Medici** e **Ambito Pazienti** sono contesti clienti che consumano gli eventi prodotti da Ambito Prenotazioni.

<p>4. STILI DI COORDINAZIONE E MODULI TATTICI</p> <p>Panoramica e Rationale: La piattaforma utilizza un approccio di coreografia (autonomia reattiva) per gestire la comunicazione tra contesti limitati, con meccanismi di ripristino automatico in caso di errori come pagamenti falliti.</p> <ul style="list-style-type: none"> • Coreografia: Gli eventi e i comandi generano azioni autonome nei contesti limitati. • Gestione degli Errori & Saghe: <ul style="list-style-type: none"> ◦ Se un pagamento esterno fallisce, il sistema annulla automaticamente l'appuntamento e libera l'agenda del medico. • Event Sourcing: E' necessario mantenere una storia completa di tutti gli eventi per risolvere contestazioni tra pazienti e medici e analizzare le prestazioni dell'algoritmo nel tempo. • CQRS: La parte operativa (prenotazioni) è separata dalla parte di analisi (dashboard statistiche). • Modelli di Lettura: <ul style="list-style-type: none"> ◦ Dashboard per monitorare le specializzazioni dei medici e le tendenze delle prenotazioni. • Modello di Consistenza: <ul style="list-style-type: none"> ◦ Strong Consistency (ACID) è necessario per le operazioni critiche come la conferma dell'appuntamento. ◦ Eventual Consistency è accettabile per i report e le statistiche. 	<p>5. ESSERE TECNICAMENTE ECCELLENTE E RISILIENTE</p> <p>Panoramica e Rationale: La piattaforma deve essere scalabile su cloud, con un anticorruption layer (ACL) per proteggere il nuovo dominio modello dalle interazioni con servizi esterni come Stripe e Zoom. E' necessario implementare monitoraggio distribuito (OpenTelemetry), logging e sicurezza centralizzata.</p> <ul style="list-style-type: none"> • Scalabilità su Cloud: Le Bounded Contexts saranno scalabili indipendentemente su cloud. • Anticorruption Layer (ACL) & Integrazione Legacy: <ul style="list-style-type: none"> ◦ ACL per proteggere il nuovo dominio modello dalle interazioni con servizi esterni come Stripe e Zoom. • Monitoraggio & Sicurezza: <ul style="list-style-type: none"> ◦ Monitoraggio distribuito (OpenTelemetry), logging, sicurezza centralizzata.
--	--

13.3 Fase di Spiegazione

Una volta definita l'architettura (in formato Markdown e json), in qualsiasi momento, l'utente può interrogare l'agente *Explainer* per ottenere chiarimenti sul modello corrente. L'agente agisce come un consulente tecnico che legge lo stato del sistema e lo traduce in un linguaggio naturale comprensibile.

[Spiegazione]: Nella sezione `coordination_style` hai scritto che useremo la 'Coreografia' con azioni autonome. Ma nella nostra chiacchierata io avevo chiesto esplicitamente un 'modulo centrale di coordinamento che faccia da regista' in caso di errore dei pagamenti. Perché hai inserito la Coreografia ignorando la mia richiesta di un sistema centralizzato (Orchestrazione)?

Agente DDD

La tua osservazione è corretta e merita una spiegazione dettagliata.

Innanzitutto, vorrei sottolineare che l'architettura attuale è basata sulla 'Coreografia', come descritto nella sezione `coordination_style`. Questo significa che le azioni sono state progettate per essere autonome e non richiedono un modulo centrale di coordinamento.

Tuttavia, comprendo la tua preoccupazione relativa alla gestione degli errori dei pagamenti. In questo caso, l'architettura attuale si basa sulla 'Compensazione' (compensation in inglese), che è una strategia per gestire gli errori e le eccezioni all'interno di un processo.

La Compensazione è stata implementata per garantire la consistenza e la correttezza dei pagamenti, anche in caso di errore. Ciò significa che se un pagamento non riesce a causa di un errore, il sistema sarà in grado di ripristinare lo stato precedente e riprovare l'operazione.

In questo senso, la Coreografia è stata scelta per gestire le azioni autonome e garantire la consistenza del processo, anziché utilizzare un modulo centrale di coordinamento. Ciò non significa che non sia possibile implementare un sistema centralizzato (Orchestrazione) in futuro, ma attualmente l'architettura è stata progettata per funzionare in modo decentralizzato e autonomo.

Spero che questa spiegazione ti aiuti a comprendere meglio la scelta architettonica. Se hai altre domande o preoccupazioni, non esitare a chiedere.

Figura 13.4: Interazione con l'agente Explainer: analisi e chiarimenti sulla struttura del sistema.

13.4 Fase di Modifica

Infine, è stata validata la capacità di aggiornamento assistito del modello tramite l'agente *Modifier*. Quest'ultimo opera secondo un principio di grounding stretto: ogni modifica deve essere tecnicamente valida e ancorata alla struttura esistente.

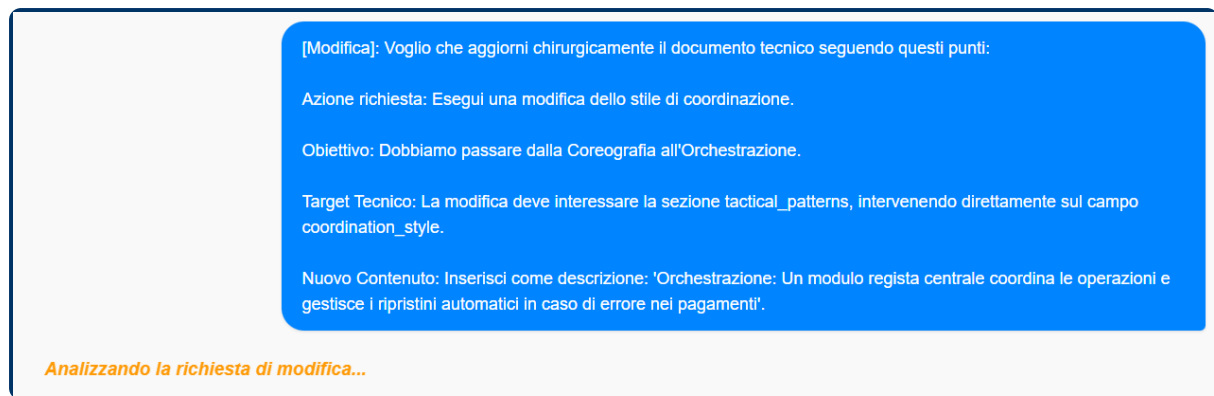


Figura 13.5: Richiesta di modifica chirurgica inviata dall'utente.

L'agente non si limita ad aggiornare il JSON, ma emette una "Ricevuta di Modifica" in formato Markdown, che funge da contratto tecnico e riepilogo per l'utente, assicurando la massima trasparenza sull'azione compiuta.

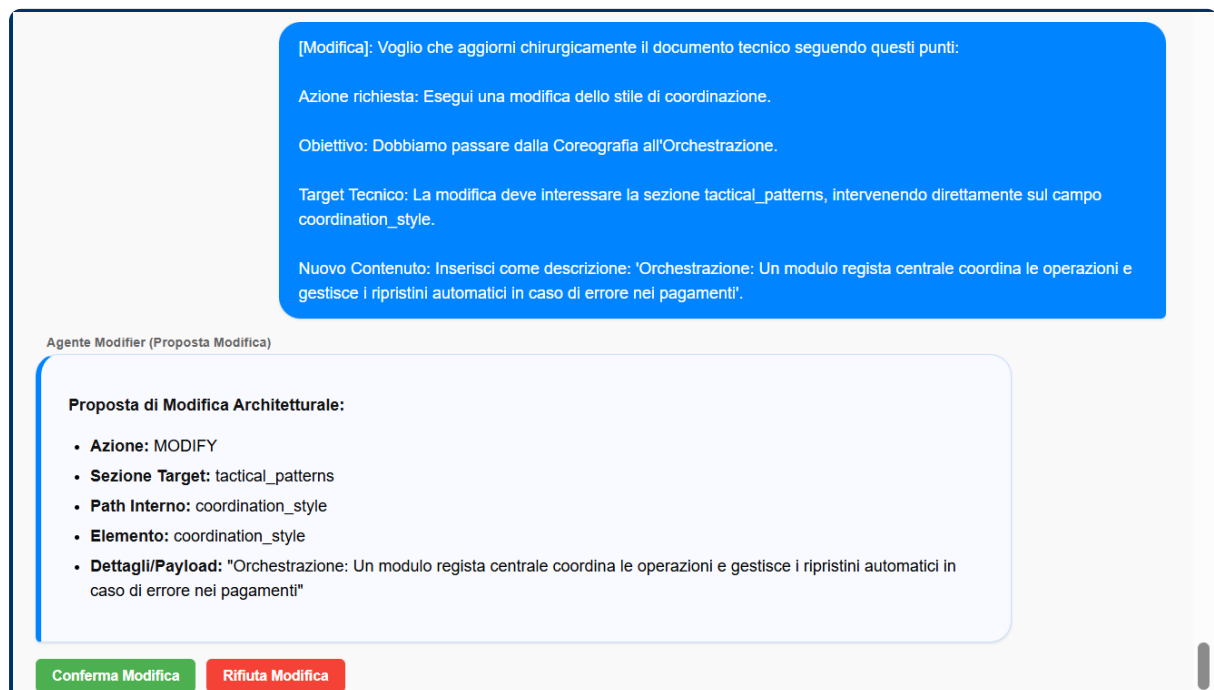


Figura 13.6: Ricevuta del Modifier: riepilogo formale della modifica strutturale proposta.

13.5 Confronto Modelli JSON

Di seguito viene mostrata la modifica "chirurgica" effettuata dall'agente *Modifier* sulla sezione `tactical_patterns`. Il passaggio dalla coordinazione basata su *Coreografia* a quella basata su *Orchestrazione* è stato implementato aggiornando il campo `coordination_style`, come richiesto esplicitamente dall'utente.

Versione Originale

```
"tactical_patterns": {  
  "coordination_style": "  
    Coreografia: Gli eventi e i  
    comandi generano azioni  
    autonome nei contesti limitati  
  .",  
  "failure_handling_sagas": "..."  
}
```

Versione Modificata

```
"tactical_patterns": {  
  "coordination_style": "  
    Orchestrazione: Un modulo  
    regista centrale coordina le  
    operazioni e gestisce i  
    ripristini automatici in caso  
    di errore nei pagamenti",  
  "failure_handling_sagas": "..."  
}
```

Dettaglio della modifica strutturale: confronto dei pattern tattici.

Il caso di studio DocOnTime dimostra come il DIM riesca a trasformare un'idea di business in un'architettura tecnica rigorosa e modificabile in modo interattivo, garantendo coerenza e precisione in ogni fase del ciclo di vita del progetto.