Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea Triennale in Informatica per il Management

Progettazione e sviluppo di un Indexer semantico per il Web of Things in architetture blockchain-oriented (ZONIA)

Relatore:
Prof.
Federico Montori

Presentata da:
Koudja Gochui Franks Toscani

Correlatore:

Dott.

Lorenzo Gigli

Sessione Marzo 2026

Anno Accademico 2025/2026

Abstract

Nel contesto Internet of Things e Web of Things (WoT), la disponibilità di risorse eterogenee rende cruciale individuare rapidamente i produttori di dati compatibili con una richiesta applicativa. In scenari blockchain-oriented, questa esigenza si combina con la necessità di coordinare richieste pubblicate on-chain e di rendere i risultati accessibili a componenti off-chain (ad esempio oracoli), mantenendo interoperabilità e tracciabilità.

La presente tesi descrive la progettazione e lo sviluppo di un indexer semantico per il WoT, con riferimento all'architettura ZONIA. L'obiettivo del lavoro è realizzare un componente software in grado di ascoltare eventi on-chain, selezionare Thing Descriptions (TD) compatibili con i vincoli della richiesta (nel prototipo basati su metadati semantici quali `@type`), registrare i risultati e renderli disponibili tramite un'interfaccia REST minimale interrogabile dagli oracoli a partire dal `requestId`. Il contributo principale consiste nel tradurre l'idea di un indexer per la discovery WoT in un prototipo end-to-end integrato con smart contract, misurabile e riproducibile, che gestisce metadati e matching semantico e materializza la catena evento on-chain $\rightarrow$ selezione delle TD $\rightarrow$ disponibilità via API.

L'implementazione è stata realizzata in ambiente Node.js, con integrazione di smart contract tramite ethers.js e persistenza PostgreSQL. Per garantire interoperabilità tra componenti e processi distinti, il sistema adotta una strategia a doppio livello: cache in memoria per accesso rapido locale e persistenza su database per condivisione e fallback.

La validazione sperimentale è stata condotta in ambiente locale Hardhat su dataset sintetico di 10.000 TD, valutando scenari di carico a 1, 10 e 30 richieste/minuto. I risultati mostrano stabilità funzionale (assenza di errori API nei campioni osservati), latenza API contenuta (nell'ordine delle centinaia di millisecondi) e un comportamento di saturazione del throughput end-to-end nella pipeline complessiva. Tali evidenze confermano la fattibilità dell'approccio e indicano i principali punti di intervento per miglioramenti di scalabilità.

Indice

Abstract **i**

1 Introduzione **1**
1.1 Contesto e motivazione 1
1.2 Problema affrontato 2
1.3 Obiettivi e contributi 2
1.4 Struttura della tesi 3

2 Stato dell'arte **4**
2.1 Web of Things e Thing Description 4
2.2 IoT e Blockchain 4
2.3 Integrazione WoT–Blockchain: esigenze e criticità 5
2.4 Gap e motivazione del lavoro 6
2.5 Blockchain e oracoli per l'accesso a dati off-chain 7
2.5.1 Oracle 7
2.6 Architettura ZONIA (Zero-trust Oracle Network for IoT Applications) e ruolo dell'Indexer 8
2.6.1 Attori e componenti principali 9
2.6.2 Ruolo dell'Indexer nel flusso ZONIA 9
2.6.3 Workflow di alto livello: dalla richiesta ai produttori 10
2.6.4 Perimetro della tesi rispetto a ZONIA 10

3 Progettazione **11**
3.1 Obiettivi di progetto 11
3.2 Requisiti funzionali e non funzionali 11
3.2.1 Requisiti funzionali (RF) 12
3.2.2 Requisiti non funzionali (RNF) 12
3.3 Architettura logica dell'Indexer 13
3.3.1 Architettura logica proposta 15
3.3.2 Scelte di progetto rilevanti 23

4 Implementazione **25**
4.1 Stack tecnologico e struttura del progetto 25
4.2 Caricamento e persistenza delle Thing Descriptions 26
4.3 Matching semantico e persistenza dei match 27
4.4 Integrazione con blockchain: submit richiesta e listener eventi 29
4.5 API REST: recupero dei risultati per requestId 30
4.6 Script di test e benchmark 31
4.6.1 Benchmark end-to-end dell'Indexer: funzionamento e procedura 32
4.6.2 Benchmark API latency: funzionamento e procedura 36

5 Valutazione esperimentale e Risultati ottenuti **40**
5.1 Obiettivo della valutazione 40
5.2 Metriche 40
5.3 Generazione del dataset sintetico e motivazioni della scelta 42
5.4 Risultati osservati 44
5.4.1 Latenza API 44
5.4.2 Latenza end-to-end (E2E) 49
5.4.3 Throughput end-to-end e saturazione 50
5.4.4 Matches per request e match-rate 51
5.5 Discussione 52
5.6 Limiti sperimentali e minacce alla validità 54

Conclusioni **56**
Sviluppi futuri 57

Bibliografia **58**

Ringraziamenti **60**

Elenco delle figure

2.1 Panoramica ZONIA. 8

3.1 Architettura dell'Indexer: ingestione TD, ascolto eventi on-chain, matching semantico, persistenza dei match e pubblicazione via `GET /data/:requestId`. 14
3.2 Sequenza principale delle interazioni dell'Indexer nel contesto ZONIA (ingestion TD, evento on-chain, matching, persistenza e retrieval via API). 22

5.1 Scenario 1_rpm – Round 1: latenza per oracolo finto (10 chiamate concorrenti all'end-point). 45
5.2 Scenario 10_rpm – Round 2: latenza per oracolo finto (10 chiamate concorrenti all'end-point). 46
5.3 Scenario 30_rpm – Round 3: latenza per oracolo finto (10 chiamate concorrenti all'end-point sullo stesso requestId). 47
5.4 Sintesi della latenza API nei tre round (10 chiamate concorrenti all'end-point sullo stesso requestId). 48
5.5 Confronto tra target nominale e throughput E2E osservato. 51

Elenco delle tabelle

2.1 Confronto qualitativo tra directory WoT e Indexer blockchain-oriented. 6
2.2 Confronto qualitativo tra approcci di discovery e componente Indexer nel contesto WoT–Blockchain. 10

3.1 Requisiti funzionali dell'Indexer. 12
3.2 Requisiti non funzionali dell'Indexer. 12

4.1 Distribuzione dei valori di `@type` nel dataset sintetico (10.000 TD) generato da `seedLargeTds.js`. 27
4.2 Endpoint REST esposto dall'Indexer. 30

5.1 Latenza per oracolo finto – Scenario 10_rpm, Round 1; (requestId: `0xae2926aa1e0b35301e104a6bc50d7276690a985d4b6d7f56bf503c61b1fa83cd`) 45
5.2 Latenza per oracolo finto – Scenario 10_rpm, Round 2; (requestId: `0xae2926aa1e0b35301e104a6bc50d7276690a985d4b6d7f56bf503c61b1fa83cd`) 46
5.3 Latenza per oracolo finto – Scenario 30_rpm, Round 3 47
5.4 Latenza end-to-end (E2E) per scenario di carico. 49
5.5 Throughput end-to-end per scenario di carico. 50
5.6 Matches per request e match-rate nei tre scenari di carico. 52

Capitolo 1

Introduzione

1.1 Contesto e motivazione

Il paradigma del *Web of Things* (WoT)[3] mira a rendere dispositivi e servizi eterogenei interoperabili tramite descrizioni standardizzate, consentendo discovery, integrazione e orchestrazione su larga scala. Nel modello WoT, la *Thing Description* (TD)[5] rappresenta il contratto descrittivo di un *Thing*[1], includendo metadati, affordance (proprietà, azioni, eventi) e informazioni utili all'interazione.

Parallelamente, le applicazioni *blockchain-oriented* e le moderne applicazioni decentralizzate (dApp)[7] necessitano spesso di accedere a dati esterni (off-chain) per eseguire logiche on-chain. Tuttavia, gli smart contract[2] non possono interrogare direttamente risorse esterne in modo nativo: tale vincolo richiede meccanismi di integrazione che garantiscano tracciabilità, verificabilità e resistenza a manipolazioni. In questo scenario, gli *oracoli* fungono da ponte tra blockchain e mondo esterno.

L'integrazione tra WoT e blockchain pone quindi un'esigenza specifica: identificare in modo affidabile e scalabile quali dispositivi (o servizi) siano compatibili con una richiesta, descritta tramite vincoli semantici e tecnici, e rendere disponibile tale informazione in una pipeline che coinvolge componenti on-chain e off-chain.

[1] Nel Web of Things, un *Thing* è un dispositivo o servizio connesso che espone funzionalità accessibili via rete (ad esempio proprietà, azioni ed eventi). Per *accesso alle risorse* si intendono le informazioni operative (endpoint, protocollo, metodi e vincoli di sicurezza) necessarie per interagire con tali funzionalità.

[2] Uno *smart contract* è un programma distribuito eseguito su una blockchain, che applica automaticamente regole definite nel codice e produce risultati verificabili dalla rete.

1.2 Problema affrontato

Il problema centrale affrontato in questa tesi è la realizzazione di un *indexer semantico* in grado di: (i) acquisire e gestire Thing Descriptions provenienti da sorgenti WoT, (ii) valutare la compatibilità semantica tra una richiesta e le TD indicizzate, (iii) interagire con smart contract per ricevere richieste e candidarsi come fornitore, (iv) esporre i risultati tramite API per supportare componenti esterni (ad esempio oracoli o servizi di integrazione).

Il lavoro è svolto con riferimento all'architettura *ZONIA*[1], nella quale l'indexer ricopre un ruolo cruciale di discovery/matching e mediazione tra richieste on-chain e risorse WoT descrivibili tramite TD.

1.3 Obiettivi e contributi

L'obiettivo generale è progettare e implementare un indexer aderente ai vincoli architetturali di ZONIA, con focus sulla gestione di *metadati semantici* (Thing Descriptions) e con esclusione di componenti geospaziali, non considerate per ragioni di perimetro di tesi.

I principali contributi del lavoro sono:

- progettazione di un flusso end-to-end per l'acquisizione, memorizzazione e matching di Thing Descriptions in un contesto blockchain-oriented;
- realizzazione di un componente off-chain capace di ascoltare eventi on-chain, effettuare matching e candidarsi on-chain come fornitore;
- definizione di una strategia di persistenza (cache in memoria e/o database PostgreSQL) per garantire interoperabilità tra processi distinti e accesso efficiente ai dati;
- implementazione di API REST per la consultazione dei risultati e la gestione operativa dell'indexer;
- valutazione sperimentale in ambiente locale (Hardhat)[12] su dataset sintetico di 10.000 TD, con analisi di latenza e throughput in scenari a carico crescente.

1.4 Struttura della tesi

Il documento è organizzato come segue. Il Chapter 2 introduce lo stato dell'arte relativo a WoT, Thing Description, oracoli e architetture blockchain-oriented di integrazione. Il Chapter 3 descrive la progettazione dell'indexer secondo l'architettura ZONIA, con requisiti, modello dati e flussi operativi. Il Chapter 4 dettaglia le scelte implementative e i moduli principali del sistema. Il Chapter 5 presenta la valutazione sperimentale e discute i risultati ottenuti, incluse le minacce alla validità. Infine, il Section 5.6 conclude il lavoro e propone sviluppi futuri.

Capitolo 2

Stato dell'arte

2.1 Web of Things e Thing Description

Il Web of Things (WoT)[4] propone un modello di interoperabilità per dispositivi e servizi eterogenei, basato su descrizioni standard e su pattern di interazione uniformi. In tale contesto, la Thing Description (TD)[5] rappresenta un artefatto descrittivo che specifica metadati, le funzionalità esposte (proprietà, azioni, eventi) e dettagli utili per l'accesso alle risorse del Thing.

Dal punto di vista sistemistico, la TD consente di separare *descrizione* e *implementazione*, favorendo discovery e composizione. In particolare, una TD ben strutturata rende possibile: (i) identificare capacità e interfacce esposte; (ii) selezionare Things compatibili rispetto a vincoli funzionali; (iii) integrare automaticamente risorse WoT in applicazioni e workflow.

2.2 IoT e Blockchain

La crescita dell'Internet of Things (IoT) ha incrementato in modo significativo la quantità e la varietà di dati disponibili, generati da dispositivi eterogenei e distribuiti. Nonostante ciò, molte piattaforme IoT continuano a basarsi su architetture centralizzate e su ecosistemi proprietari, con conseguenti limiti di interoperabilità e difficoltà nel garantire trasparenza e tracciabilità del dato [1].

In questo scenario, la blockchain è spesso proposta come tecnologia abilitante per mitigare parte di tali criticità, grazie a proprietà come decentralizzazione, immutabilità e auditabilità: eventi e risultati possono essere registrati in modo verificabile e condiviso tra i partecipanti. Tuttavia, la blockchain non risolve da sola il problema dell'accesso ai dati del mondo reale: per ragioni di sicurezza e determinismo, gli smart contract non possono interrogare direttamente fonti esterne (off-chain).

Ne consegue che l'integrazione IoT–blockchain richiede architetture ibride, in cui il coordinamento avviene on-chain mentre l'acquisizione e l'elaborazione dei dati rimangono off-chain. In particolare:

- la blockchain gestisce regole, coordinamento e incentivi;
- componenti off-chain si occupano di acquisire i dati e renderli disponibili ai contratti intelligenti.

Tali componenti sono comunemente denominati *oracle*[2] e hanno il compito di collegare il mondo off-chain alla blockchain, fornendo agli smart contract dati esterni in forma verificabile secondo il modello adottato . In ambito WoT, oltre al problema dell'accesso al dato, emerge anche quello della selezione della sorgente adeguata tra molte risorse disponibili, come discusso nella sezione seguente.

2.3 Integrazione WoT–Blockchain: esigenze e criticità

La disponibilità delle Thing Descriptions (TD) nel WoT permette di descrivere risorse e capability in modo standardizzato. Tuttavia, quando il numero di risorse cresce, diventa necessario supportare in modo sistematico la *discovery* e la selezione di Things compatibili con una richiesta, evitando approcci basati su semplici keyword o su ricerca manuale.

L'integrazione tra WoT e blockchain introduce criticità specifiche:

- **eterogeneità delle risorse**: dispositivi con modelli e capability differenti, descritti da TD non uniformi per complessità e contenuto;
- **selezione affidabile della sorgente**: identificare Things compatibili rispetto a richieste e vincoli semantici, riducendo ambiguità e falsi positivi;
- **scalabilità**: gestire cataloghi numerosi di TD e richieste concorrenti senza degradare eccessivamente le prestazioni;
- **separazione on-chain/off-chain**: coordinare richieste pubblicate on-chain con processi off-chain (discovery, matching, retrieval) mantenendo tracciabilità del flusso.

Queste esigenze motivano l'introduzione di componenti off-chain specializzati nella gestione dei metadati (TD) e nel matching, mantenendo on-chain il coordinamento e la verificabilità. In tale direzione si colloca l'Indexer previsto in ZONIA, oggetto dell'implementazione descritta in questa tesi.

2.4 Gap e motivazione del lavoro

Nel Web of Things, la *discovery* delle risorse consiste nell'individuazione di Things disponibili e nel reperimento delle relative Thing Descriptions. Il W3C[1] definisce meccanismi e pattern per la discovery, includendo l'uso di directory e registri in cui le TD possono essere pubblicate e recuperate [4]. In questo approccio, la directory svolge un ruolo centrale nel rendere interrogabili i metadati delle risorse WoT, abilitando scenari in cui consumer e applicazioni trovano Things compatibili a partire da criteri di ricerca.

Oltre alle specifiche W3C, sono state proposte soluzioni dedicate a rendere scalabili tali directory. Un esempio è ZION[6], una directory WoT progettata per supportare la memorizzazione e la ricerca di TD su larga scala. Rispetto a una discovery puramente testuale o basata su metadati semplici, le directory WoT permettono un accesso strutturato alle TD e possono essere ottimizzate per carichi elevati e grandi cataloghi.

Tuttavia, in un contesto blockchain-oriented emergono requisiti aggiuntivi: le richieste e il coordinamento avvengono on-chain, mentre la discovery e l'accesso ai metadati restano off-chain. Di conseguenza, una directory WoT generica non copre nativamente aspetti quali l'associazione tra una richiesta on-chain e l'insieme di TD compatibili, né i meccanismi di candidatura/segnalazione di disponibilità previsti da architetture come ZONIA. Questo gap motiva l'introduzione di un componente specializzato (Indexer) che colleghi discovery semantica e workflow on-chain/off-chain.

Tabella 2.1: Confronto qualitativo tra directory WoT e Indexer blockchain-oriented.

Soluzione	**Focus**	**Limiti nel contesto blockchain-oriented**
W3C WoT Discovery / TD Directory [4]	Pattern e meccanismi di discovery per risorse WoT	Non include logiche di coordinamento on-chain o associazione requestId → risultati
ZION [6]	Directory WoT scalabile per memorizzazione/-ricerca di TD	Non modellata per workflow on-chain/off-chain e candidatura legata a richieste on-chain
Indexer ZONIA / questa tesi [1]	Discovery/matching legati a richieste on-chain e retrieval per `requestId`	Prototipo semantic-only; la pipeline end-to-end può saturare in funzione del carico

[1] Il *World Wide Web Consortium* (W3C) è un'organizzazione internazionale che sviluppa standard tecnici per il Web, come specifiche e raccomandazioni adottate in modo interoperabile.

2.5 Blockchain e oracoli per l'accesso a dati off-chain

Gli smart contract operano in un ambiente deterministico e isolato, nel quale l'accesso diretto a risorse esterne (off-chain) non è nativamente disponibile. Per colmare tale limitazione, vengono adottati meccanismi noti come *oracoli*[8, 9], che acquisiscono dati dal mondo esterno e li rendono disponibili on-chain con garanzie variabili di affidabilità, integrità e tracciabilità.

In architetture blockchain-oriented, la pipeline tipica include: (i) emissione di richieste on-chain; (ii) raccolta ed elaborazione off-chain; (iii) pubblicazione o attestazione del risultato on-chain; (iv) consumo on-chain o via servizi esterni. In presenza di risorse IoT/WoT, diventa cruciale selezionare correttamente la sorgente dei dati, evitando ambiguità e riducendo il rischio di manipolazioni o di risultati non verificabili.

2.5.1 Oracle

Gli oracoli fungono da ponte tra blockchain e mondo esterno, acquisendo dati off-chain e rendendoli disponibili on-chain. In ambito IoT, il cosiddetto *oracle problem*[2] è accentuato da eterogeneità delle sorgenti, qualità variabile dei dati e rischio di centralizzazione dei provider oracle.

La letteratura ha proposto soluzioni basate su Trusted Execution Environments (TEE), utili sotto il profilo della protezione dell'esecuzione ma potenzialmente limitanti in termini di apertura e scalabilità dell'ecosistema [1]. In alternativa, i modelli zero-trust puntano su selezione decentralizzata, reputazione e incentivi economici per ridurre la dipendenza da fiducia implicita in singoli attori.

Nel dominio IoT, il problema si amplifica per almeno tre motivi:

- elevata eterogeneità dei dispositivi e delle interfacce;
- affidabilità variabile delle sorgenti dati;
- rischio di manipolazione, collusione o centralizzazione del servizio oracle.

Diversi approcci in letteratura usano Trusted Execution Environments (TEE) per aumentare la sicurezza. Pur efficaci su alcuni fronti, tali soluzioni possono introdurre barriere all'ingresso (hardware specifico, complessità operativa) e favorire concentrazione su pochi attori/sorgenti.

[2]Problema per cui gli smart contract non possono ottenere dati dal mondo esterno in modo nativo e devono affidarsi a un *oracle*, introducendo un punto di fiducia.

Un orientamento alternativo è quello *zero-trust*, in cui:

- nessun nodo è considerato affidabile a priori;
- selezione, reputazione e incentivi economici governano la qualità del risultato;
- il risultato finale deriva da più sorgenti e da un processo di consenso/aggregazione.

2.6 Architettura ZONIA (Zero-trust Oracle Network for IoT Applications) e ruolo dell'Indexer

ZONIA (Zero-trust Oracle Network for IoT Applications) [1] è un'architettura *oracle-oriented* per applicazioni blockchain in ambito IoT/WoT, progettata per coordinare la risoluzione di richieste on-chain e l'accesso a dati off-chain attraverso un insieme di componenti specializzati. L'idea di fondo è separare chiaramente il coordinamento e l'incentivazione (gestiti on-chain) dalle attività di discovery, selezione delle sorgenti e retrieval dei dati (gestite off-chain), riducendo dipendenze da entità centralizzate e adottando un paradigma *zero-trust*.

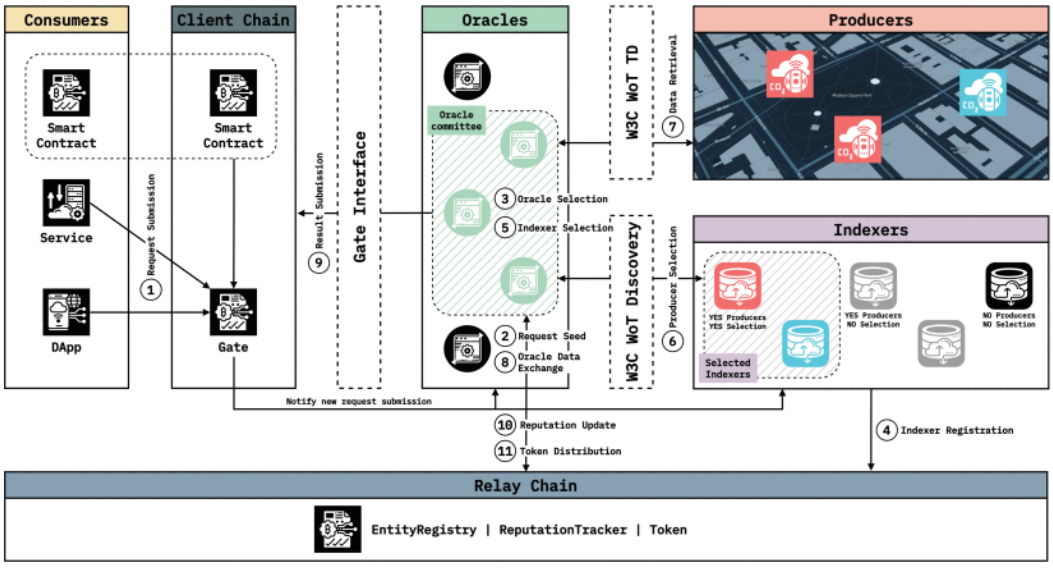

Figura 2.1: Panoramica ZONIA.

2.6.1 Attori e componenti principali

ZONIA coinvolge diversi attori e moduli cooperanti:

- **Consumer**: il soggetto che sottomette una richiesta on-chain per ottenere dati (es. uno smart contract o un'applicazione).
- **Producer**: la sorgente di dati (sensore/dispositivo/servizio WoT) descritta tramite Thing Description.
- **Indexer**: mantiene metadati sui Producer (TD), supporta la *discovery* e restituisce agli oracoli l'elenco di TD compatibili con una richiesta.
- **Oracle**: raccoglie dati dai Producer selezionati e contribuisce alla risoluzione della richiesta (con logiche di aggregazione/validazione previste dall'architettura).
- **Smart contract di coordinamento**: gestiscono sottomissione delle richieste e interazioni on-chain tra i partecipanti (registrazioni, selezione e incentivi secondo il workflow del sistema).

In questo schema, l'Indexer non decide il valore "vero" del dato (truth inference[3]) né partecipa al consenso finale: il suo ruolo è abilitare la fase di *discovery* dei produttori e rendere recuperabili i risultati del matching in modo deterministico a partire dall'identificativo della richiesta.

2.6.2 Ruolo dell'Indexer nel flusso ZONIA

All'interno di ZONIA, l'Indexer svolge quattro funzioni fondamentali:

- **Acquisizione e memorizzazione delle Thing Descriptions**: costruisce e mantiene un catalogo di TD dei Producer noti.
- **Valutazione di compatibilità semantica**: dato un vincolo espresso in una richiesta, seleziona le TD compatibili (discovery basata su metadati).
- **Supporto al flusso on-chain**: quando sono presenti match, l'Indexer può segnalare la propria disponibilità per contribuire alla risoluzione della richiesta.
- **Esposizione dei risultati via API**: permette agli Oracle di recuperare le TD già selezionate per una specifica richiesta, usando il relativo `requestId`.

[3]In ZONIA, la *truth inference* è la fase in cui i valori raccolti da più oracle vengono confrontati e aggregati per stimare il risultato più attendibile (ad esempio in base a coerenza tra risposte e criteri di affidabilità), prima della finalizzazione on-chain.

2.6.3 Workflow di alto livello: dalla richiesta ai produttori

Il ciclo tipico può essere descritto come segue:

1. **Sottomissione della richiesta**: un Consumer pubblica on-chain una richiesta contenente i parametri necessari (ad esempio il vincolo semantico).

2. **Notifica agli Indexer**: gli Indexer osservano l'arrivo della richiesta (evento on-chain) e avviano la fase di matching sulle TD disponibili.

3. **Matching e registrazione risultati**: per ogni richiesta, l'Indexer identifica le TD compatibili e registra l'associazione con il `requestId`, così che il risultato sia recuperabile dagli Oracle.

4. **Interrogazione da parte degli Oracle**: gli Oracle richiedono agli Indexer le TD associate a quel `requestId` e utilizzano tali informazioni per contattare i Producer e proseguire nelle fasi successive del processo.

2.6.4 Perimetro della tesi rispetto a ZONIA

L'architettura completa di ZONIA prevede anche funzionalità avanzate (ad esempio supporto a vincoli geospaziali e criteri più ricchi di selezione). In questa tesi, l'implementazione dell'Indexer è limitata alla sola dimensione semantica delle TD, concentrandosi su: (i) gestione e persistenza dei metadati WoT, (ii) matching semantico (nel prototipo basato su `@type`), (iii) integrazione con il flusso on-chain/off-chain, (iv) disponibilità dei risultati tramite API minimale interrogabile con `requestId`. La componente geospaziale e meccanismi avanzati di ranking/selezione sono considerati estensioni future.

Tabella 2.2: Confronto qualitativo tra approcci di discovery e componente Indexer nel contesto WoT–Blockchain.

Approccio	**Obiettivo**	**Limite nel contesto WoT–Blockchain**
Discovery testuale / metadati semplici	Trovare risorse per keyword o attributi base	Compatibilità semantica debole; difficile selezione affidabile della sorgente
Registry WoT generico	Memorizzare e recuperare TD	Non integra nativamente flussi on-chain/off-chain e meccanismi di candidatura
Indexer ZONIA (questa tesi)	Matching semantico + integrazione con richieste on-chain	Fuori scope: geospaziale; scalabilità end-to-end influenzata dalla pipeline on-chain/off-chain

Capitolo 3

Progettazione

3.1 Obiettivi di progetto

In questo capitolo vengono descritti i criteri progettuali e le scelte architetturali adottate per realizzare l'indexer semantico nel contesto ZONIA. La progettazione è guidata da requisiti funzionali e non funzionali che derivano dal flusso end-to-end (richiesta on-chain, matching off-chain, candidatura on-chain, esposizione risultati).

3.2 Requisiti funzionali e non funzionali

I requisiti sono organizzati in requisiti funzionali (RF), relativi alle funzionalità offerte dal sistema, e requisiti non funzionali (RNF), relativi a qualità, vincoli e proprietà di esecuzione.

3.2.1 Requisiti funzionali (RF)

Tabella 3.1: Requisiti funzionali dell'Indexer.

Codice	Descrizione
RF1	Acquisire Thing Descriptions (TD) da un dataset locale in formato JSON e validarne la struttura minima prima della persistenza.
RF2	Memorizzare e indicizzare le TD per consentire query e retrieval efficiente dei metadati.
RF3	Ricevere richieste generate on-chain (eventi emessi da smart contract) e trasformarle in vincoli di matching.
RF4	Valutare la compatibilità tra richiesta e TD disponibili tramite matching semantico/strutturale.
RF5	Candidarsi on-chain come fornitore quando viene individuata una o più TD compatibili, includendo identificativi e metadati necessari.
RF6	Esporre un'API REST minimale per consentire agli Oracle di recuperare le TD associate a una richiesta a partire dal relativo `requestId`.

3.2.2 Requisiti non funzionali (RNF)

Tabella 3.2: Requisiti non funzionali dell'Indexer.

Codice	Descrizione
RNF1	Interoperabilità cross-process: i dati devono essere accessibili a componenti distinti (indexer, oracle, servizi) tramite persistenza condivisa.
RNF2	Efficienza: il sistema deve garantire tempi di risposta contenuti per le API e per la fase di matching in scenari di carico controllato.
RNF3	Robustezza: gestione di input malformati e failure parziali (es. richiesta on-chain non valida, TD incompleta) senza compromettere il servizio.
RNF4	Tracciabilità: logging degli eventi principali (ingestion, matching, candidatura) per supportare debug e valutazione sperimentale.
RNF5	Modularità: separazione tra ingestione TD, matching e interazione con smart contract per favorire estendibilità.

3.3 Architettura logica dell'Indexer

Nel sistema ZONIA, l'Indexer è il componente incaricato di gestire e rendere interrogabili i metadati relativi ai produttori di dati IoT. Esso si colloca logicamente tra i *Producer*, che pubblicano le proprie Thing Descriptions (TD), e gli *Oracle*, che devono individuare in modo affidabile quali produttori siano idonei a soddisfare una richiesta generata on-chain. In questo senso, l'Indexer svolge un ruolo di *discovery* e *selezione* basata su metadati, fungendo da punto di raccordo tra la disponibilità delle risorse WoT e il processo di risoluzione delle richieste in un contesto blockchain-oriented.

In particolare, l'Indexer assolve quattro funzioni principali:

- **Memorizzazione dei metadati semantici** dei produttori, rappresentati tramite Thing Descriptions, così da mantenere un catalogo locale strutturato delle risorse disponibili.
- **Ricerca e filtraggio semantico** dei produttori compatibili con una richiesta, valutando la compatibilità in base ai vincoli espressi nella richiesta e alle informazioni descrittive contenute nelle TD.
- **Registrazione della disponibilità** quando rileva che una richiesta on-chain può essere soddisfatta tramite le TD presenti nel proprio archivio, contribuendo al coordinamento del flusso di risoluzione senza prendere decisioni finali.
- **Restituzione agli Oracle** dell'insieme di TD precedentemente selezionate per una specifica richiesta (identificata dal relativo `requestId`), in modo che gli Oracle possano proseguire con le fasi successive del processo.

Di conseguenza, l'Indexer non partecipa né alla *truth inference* né al consenso finale sulla risposta, ma rimane un componente essenziale per rendere la selezione dei produttori più efficiente, tracciabile e compatibile con l'interazione on-chain/off-chain tipica dell'architettura ZONIA.

Nel presente lavoro, l'implementazione dell'Indexer è intenzionalmente limitata alla sola dimensione semantica delle Thing Descriptions, escludendo la componente geospaziale prevista nell'architettura completa di ZONIA. Questa scelta deriva da considerazioni di perimetro e tempi di sviluppo e permette di concentrare l'attenzione sugli aspetti fondamentali del prototipo: gestione dei metadati WoT, filtraggio basato su vincoli semantici (ad esempio sul campo `@type`) e integrazione con il flusso di risoluzione delle richieste. La componente geospaziale e altri meccanismi avanzati (ad esempio ranking e criteri di selezione multi-vincolo) vengono quindi considerati come possibili estensioni future, senza impattare la correttezza del workflow principale implementato in questa tesi.

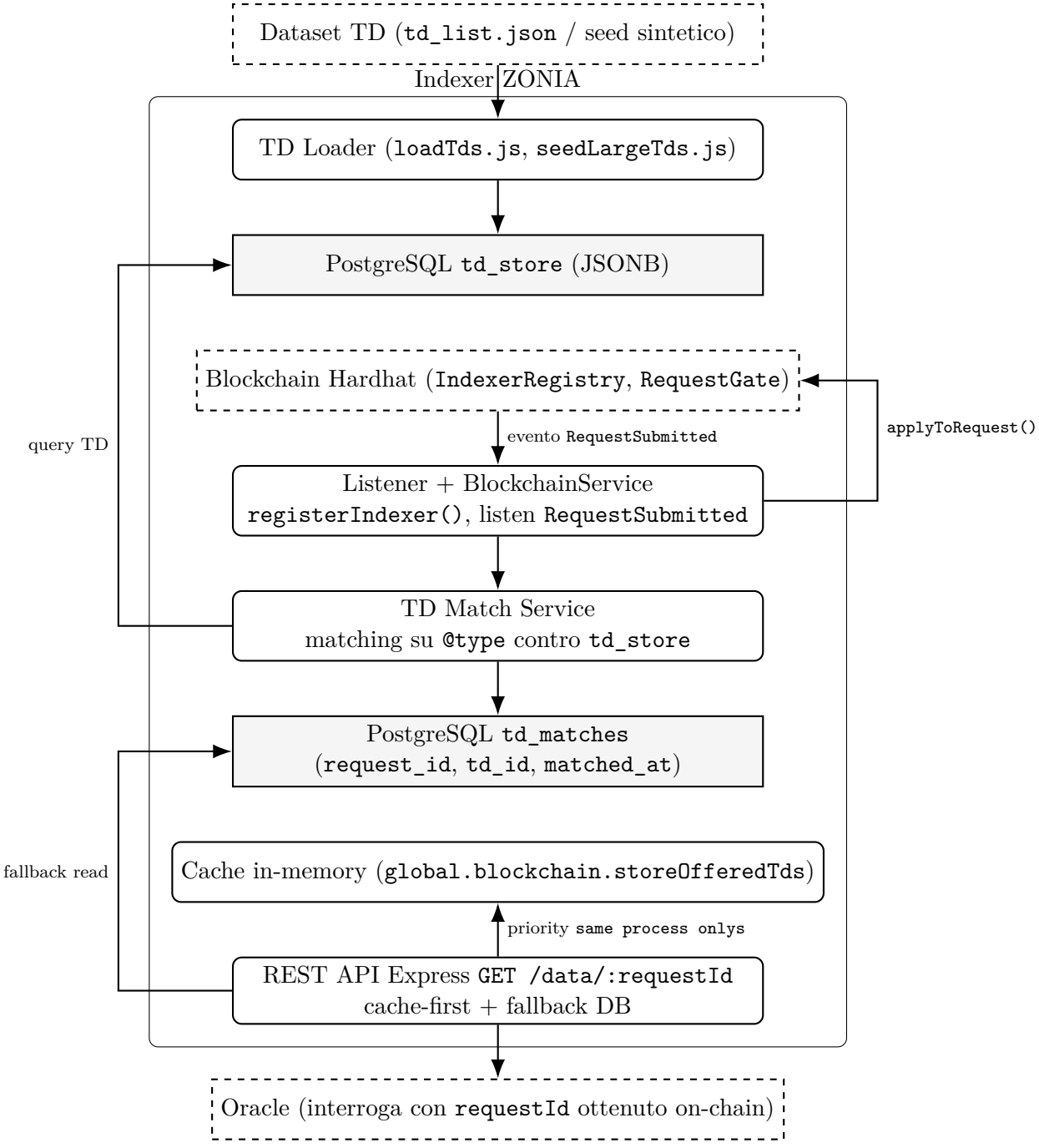

Figura 3.1: Architettura dell'Indexer: ingestione TD, ascolto eventi on-chain, matching semantico, persistenza dei match e pubblicazione via `GET /data/:requestId`.

3.3.1 Architettura logica proposta

Dal punto di vista logico, l'Indexer è suddiviso in moduli cooperanti, ciascuno con una responsabilità ben definita. Tale separazione favorisce manutenibilità, estendibilità e chiarezza architetturale. In questa sezione descriviamo ogni modulo con particolare attenzione ai dettagli implementativi e ai flussi dati sottostanti.

(1) Modulo di persistenza dei TD. Gestisce salvataggio e recupero delle Thing Descriptions nel database PostgreSQL. Ogni TD è memorizzata in formato `jsonb`, preservando la struttura originale e consentendo interrogazioni sui campi rilevanti (in particolare `@type` e altri metadati). Le responsabilità includono inserimento, aggiornamento, recupero e supporto alle query semantiche. **Responsabilità principali:**

- Inserimento di nuove TD con validazione sintattica
- Ricerca per tipo (`@type`) utilizzando operatori JSONB PostgreSQL
- Recupero dei dati completi (JSON) partendo da un array di ID
- Gestione coerente degli errori e logging per tracciabilità

Le query PostgreSQL sfruttano l'operatore `@>` (contains) e `->>` (extract text) per supportare sia il caso in cui `@type` sia una stringa singola sia un array di tipi:

Listing 3.1: Query JSONB per ricerca per tipo

```
SELECT id FROM td_store
WHERE (td->>'@type' = $1
    OR td->'@type' @> $2::jsonb)
```

Questo approccio consente di trovare TD indipendentemente dal formato del campo `@type`, migliorando la robustezza della ricerca semantica.

(2) Modulo di caricamento e validazione delle Thing Descriptions. Questo modulo acquisisce le TD da file JSON (come nel prototipo) o da altre fonti. Si occupa di lettura, verifica sintattica e preparazione per la persistenza. In evoluzioni future può essere integrato con strumenti WoT dedicati (ad esempio librerie conformi allo standard WoT) per una gestione più automatizzata delle TD. **Workflow di caricamento:**

Listing 3.2: Script di caricamento TD da file JSON

```
async function loadTds() {
    try {
        console.log("Caricamento␣TD␣da␣file...");
```

```
4
5         const filePath = config.files.tdList;
6         const raw = await fs.readFile(filePath, "utf-8");
7         const tdArray = JSON.parse(raw);
8
9         if (!Array.isArray(tdArray)) {
10            throw new Error("Il␣file␣TD␣deve␣contenere␣un␣array␣JSON");
11        }
12
13        let inserted = 0;
14        for (const td of tdArray) {
15            await tdRepository.insertTd(td);
16            console.log(`Caricato: ${td.title || td.id}`);
17            inserted++;
18        }
19
20        console.log(`${inserted} TD caricate con successo`);
21        process.exit(0);
22    } catch (error) {
23        console.error("Errore:", error.message);
24        process.exit(1);
25    }
26 }
```

Il modulo valida che ogni TD contenga il campo obbligatorio `@type` prima dell'inserimento. In una versione più sofisticata, potrebbero essere applicate validazioni contro uno schema WoT standard.

(3) Modulo di query semantica. Il servizio di matching semantico implementa il nucleo della logica di ricerca. Dato un tipo richiesto, il servizio:

1. Interroga il repository per trovare tutti gli ID delle TD del tipo compatibile
2. Se non trova risultati, ritorna un array vuoto
3. Altrimenti recupera i dati completi (JSON) dalle TD candidate
4. Ritorna l'array di TD complete al modulo richiedente

Listing 3.3: TdMatchService: ricerca di TD compatibili

```
1 class TdMatchService {
2     async findCompatibleTds(requiredType) {
3         try {
```

```
4             // Step 1: Trova gli ID delle TD del tipo richiesto
5             const candidateTdIds = await tdRepository.findTdIdsByType(
                  requiredType);
6
7             if (candidateTdIds.length === 0) {
8                 console.log(`Nessun TD compatibile per tipo: ${
                      requiredType}`);
9                 return [];
10            }
11
12            // Step 2: Recupera i dati completi delle TD
13            const tds = await tdRepository.findTdsByIds(candidateTdIds)
                  ;
14
15            console.log(`Trovati ${tds.length} TD compatibili`);
16            return tds;
17        } catch (error) {
18            console.error(`Errore ricerca TD:`, error.message);
19            return [];
20        }
21    }
22 }
23
24 module.exports = new TdMatchService();
```

Il servizio è implementato come *singleton*, garantendo una singola istanza durante l'intera lifecycle dell'applicazione.

(4) Modulo di gestione delle richieste. Questo modulo è il cuore del flusso di ZONIA. Reagisce agli eventi `RequestSubmitted` emessi dal contratto `RequestGate` e orchestra l'intero flusso:

1. Intercetta l'evento RequestSubmitted contenente un `requestId` e una query (tipo richiesto)
2. Invoca il servizio di query semantica per trovare TD compatibili
3. Se vengono trovati match, registra l'offerta on-chain tramite `applyToRequest()`
4. Memorizza i risultati sia in cache locale (stesso processo) che nel database (accesso cross-process)

La chiave architettonica è che *i risultati vengono calcolati al momento dell'evento*, non on-demand alla chiamata dell'Oracle. Questo approccio "pre-calculation" riduce la latenza della risposta API e migliora i tempi di risposta.

Listing 3.4: Listener: handler per RequestSubmitted

```
const onRequestHandler = async (requestId, requiredType) => {
    console.log(`Nuova richiesta arrivata: ${requestId}`);

    // Step 1: Ricerca TD compatibili
    const tds = await tdMatchService.findCompatibleTds(requiredType);

    if (tds.length > 0) {
        // Step 2: Registra l'offerta on-chain
        await blockchain.applyToRequest(requestId);
        console.log(`Offerta registrata on-chain per: ${requestId}`);

        // Step 3: Layer 1 - Memorizza in cache (same process)
        blockchain.storeOfferedTds(requestId, tds);

        // Step 4: Layer 2 - Persisti su DB (cross-process)
        try {
            const savedCount = await tdMatchesRepository.insertMatches(
                requestId,
                tds
            );
            console.log(`Persistiti ${savedCount} TD in td_matches`);
        } catch (error) {
            console.error(`Errore salvataggio td_matches:`, error.
                message);
        }
    }
};
```

(5) Modulo di memorizzazione dei match request–TD. Il repository dei match persiste l'associazione tra `requestId` e gli ID delle TD selezionate. Implementa un archivio di match ad accesso rapido, permettendo a qualsiasi processo (incluso il process Express del server API) di recuperare i risultati in modo coerente. La struttura della tabella `td_matches` è semplice ma efficace:

Listing 3.5: Schema della tabella td_matches

```
CREATE TABLE td_matches (
    request_id UUID NOT NULL,
    td_id INTEGER NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(request_id, td_id)
);

CREATE INDEX idx_request_id ON td_matches(request_id);
```

Il vincolo `UNIQUE(request_id, td_id)` impedisce duplicati, garantendo idempotenza. L'indice su `request_id` accelera le query di retrieval.

(6) Modulo API REST (Express Controller). L'endpoint `GET /data/:requestId` fornisce l'interfaccia off-chain con cui gli Oracle recuperano i risultati *pre-calcolati* dal listener. Il controller adotta una strategia **cache-first** a due livelli:

- **Layer 1 – cache in-memory**: se i TD sono stati appena memorizzati nello stesso processo (listener e API co-locati cioè che girano sullo stesso processo), il retrieval avviene senza accesso al database;
- **Layer 2 – fallback su database**: in caso di cache miss (ad esempio API e listener su processi distinti), il controller interroga PostgreSQL tramite `td_matches` per ottenere i risultati in modo deterministico.

Listing 3.6: Endpoint `GET /data/:requestId`: cache-first con fallback DB (estratto)

```
async function getTdData(req, res, next) {
  const { requestId } = req.params;
  let source = "cache";

  // Layer 1: cache in-memory (stesso processo)
  let tds = global.blockchain?.getOfferedTds(requestId) || [];

  // Layer 2: fallback DB (processi separati)
  if (tds.length === 0) {
    tds = await tdMatchesRepository.findTdsByRequestId(requestId);
    source = "db";
  }

  res.set("x-data-source", source);
  res.json({ requestId, source, data: tds });
```

```
16 }
```

Per supportare analisi sperimentali e debugging, l'header `x-data-source` indica al client se la risposta è stata servita da cache oppure tramite fallback su database. Nel codice completo, il controller include inoltre gestione di errori e deduplicazione dei risultati prima della risposta.

(7) Modulo di registrazione on-chain (`BlockchainService`). Il modulo `BlockchainService` incapsula le interazioni dell'Indexer con gli smart contract tramite ethers.js.Ha due responsabilità principali:

- **Registrazione one-time:** registra l'Indexer nel registro on-chain per stabilirne l'identità.
- **Registrazione one-per-request:** quando il matching produce almeno una TD compatibile, segnala on-chain la disponibilità dell'Indexer per lo specifico `requestId`.

Inizializzazione e setup. All'avvio, il servizio configura provider[1] e wallet[2] e istanzia i contratti a partire da ABI e indirizzi.(semplificato in Listing 3.7).

Listing 3.7: `BlockchainService`: setup di provider, wallet e contratti (estratto)

```
1  this.provider = new ethers.JsonRpcProvider(config.rpcUrl);
2  this.wallet = new ethers.Wallet(config.privateKey, this.provider);
3
4  this.indexerRegistry = new ethers.Contract(
5    config.indexerRegistryAddress, indexerRegistryAbi, this.wallet
6  );
7  this.requestGate = new ethers.Contract(
8    config.requestGateAddress, requestGateAbi, this.wallet
9  );
10
11 this.didToTds = new Map(); // cache in-memory dei TD offerti
```

Registrazione one-time. Il listener esegue una sola volta la registrazione dell'Indexer nel contratto `IndexerRegistry`. L'operazione è resa idempotente: se l'identità è già registrata (es. riavvio), il servizio continua senza interrompere il flusso.

[1]Un *provider* è il componente che permette all'applicazione di comunicare con la blockchain (leggere stato, inviare chiamate e transazioni) tramite un nodo, ad esempio via JSON-RPC.

[2]Un *wallet* contiene/usa una chiave privata per firmare le transazioni: in pratica identifica l'attore on-chain e gli consente di eseguire operazioni che richiedono autorizzazione.

Listing 3.8: Registrazione on-chain one-time (estratto)

```
async registerIndexer() {
  const tx = await this.indexerRegistry.register(this.config.
      indexerDid);
  await tx.wait();
}
```

Ascolto richieste e candidatura per-richiesta. Il servizio ascolta l'evento `RequestSubmitted` emesso da `RequestGate`, legge i dettagli della richiesta on-chain (in particolare `query`) e invoca l'handler applicativo. Se l'handler produce match, `applyToRequest` segnala la disponibilità dell'Indexer per quel `requestId`.

Listing 3.9: Evento `RequestSubmitted` e candidatura on-chain (estratto)

```
this.requestGate.on("RequestSubmitted", async (requestId) => {
  const request = await this.requestGate.getRequest(requestId);
  const requiredType = request.query;

  const tds = await onRequestHandler(requestId, requiredType);
  if (tds.length > 0) {
    const tx = await this.requestGate.applyToRequest(this.config.
        indexerDid, requestId);
    await tx.wait();
  }
});
```

Cache locale dei risultati e fallback su database. Per velocizzare il retrieval, il servizio mantiene in memoria i TD offerti per richiesta. Questa ottimizzazione è efficace in scenari single-process; in caso di assenza in cache o in deployment multi-process, la sorgente affidabile per il retrieval resta la persistenza su PostgreSQL (tabella `td_matches`), utilizzata dal layer API tramite fallback.

Listing 3.10: Memorizzazione e retrieval in cache (estratto)

```
storeOfferedTds(requestId, tds) {
  const did = this.config.indexerDid;
  const list = this.didToTds.get(did) || [];
  list.push({ requestId, tds });
  this.didToTds.set(did, list);
}

getOfferedTds(requestId) {
  const did = this.config.indexerDid;
  const list = this.didToTds.get(did) || [];
```

```
11    const entry = list.find((e) => e.requestId === requestId);
12    return entry ? entry.tds : [];
13  }
```

Nota operativa. Il servizio include inoltre gestione degli errori e disconnessione controllata dei listener durante lo shutdown, per evitare perdite di risorse (memory leak) e garantire robustezza in esecuzione prolungata.

Integrazione tra moduli: il flusso end-to-end

I sette moduli descritti cooperano secondo il seguente flusso:

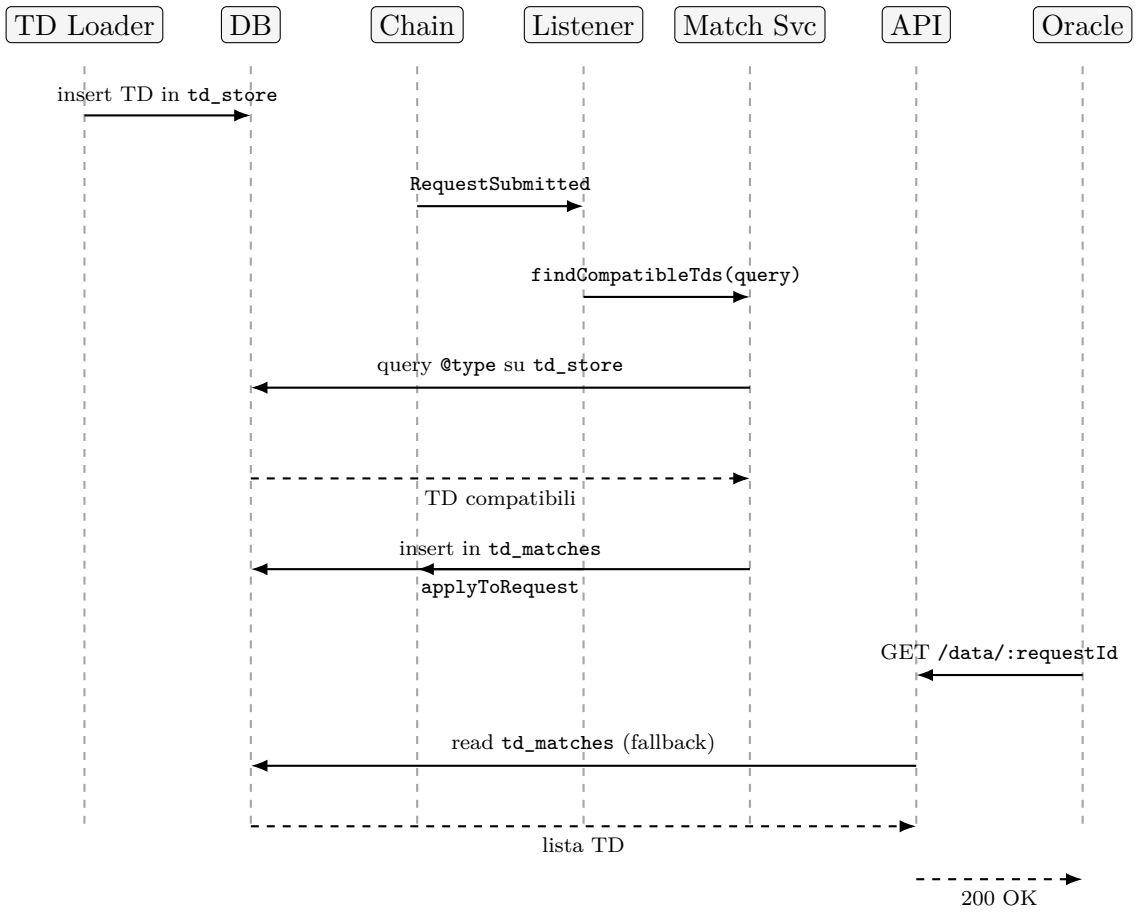

Figura 3.2: Sequenza principale delle interazioni dell'Indexer nel contesto ZONIA (ingestion TD, evento on-chain, matching, persistenza e retrieval via API).

Questa architettura modulare garantisce separazione delle responsabilità, facilita il testing, e permette l'evoluzione indipendente di ciascun componente.

3.3.2 Scelte di progetto rilevanti

La progettazione del prototipo è stata guidata da un criterio pragmatico: minimizzare la latenza percepita dagli Oracle, mantenere il flusso semplice da verificare sperimentalmente e garantire evoluzione futura senza riscritture radicali. In questa prospettiva, sono state adottate due scelte architetturali principali: (i) **pre-computazione dei risultati** al momento dell'evento on-chain, e (ii) **API minimale** centrata su un solo endpoint.

Pre-computazione event-driven vs calcolo on-demand Nel nostro caso d'uso, la richiesta nasce on-chain e contiene già il criterio semantico necessario al matching. Anticipare il calcolo quando arriva l'evento consente di:

- ridurre la latenza della fase di retrieval (l'Oracle trova risultati già pronti);
- disaccoppiare il costo computazionale dal momento della chiamata HTTP;
- ottenere tempi di risposta più stabili sotto carico.

L'alternativa on-demand avrebbe semplificato la pipeline iniziale, ma con svantaggi significativi: maggiore variabilità dei tempi API, picchi di CPU/DB durante le richieste concorrenti e rischio di degradazione proprio nella fase più sensibile (interazione Oracle–API).

API minimale vs API general-purpose L'endpoint unico `GET /data/:requestId` è coerente con il flusso ZONIA: l'Oracle possiede già il `requestId` dalla componente on-chain, quindi non sono necessari endpoint di discovery, ricerca libera o navigazione complessa. Questa scelta:

- riduce superficie di attacco e complessità applicativa;
- rende più semplice osservabilità, test e benchmark;
- concentra l'ottimizzazione sul path critico realmente usato.

Un'API più ampia (CRUD completo, query avanzate, filtri multipli lato client) sarebbe più flessibile, ma avrebbe introdotto complessità non necessaria rispetto agli obiettivi sperimentali della tesi.

Motivazione dello stack tecnologico adottato Lo stack scelto (*Node.js + Express + PostgreSQL JSONB + ethers.js/Hardhat*) è stato preferito ad alternative diffuse perché allinea bene requisiti funzionali e vincoli sperimentali:

- **Node.js/Express** (vs framework più pesanti come Spring o stack multi-linguaggio): modello I/O non bloccante adatto a listener eventi + API concorrente, con implementazione rapida e uniforme in JavaScript.
- **ethers.js** (vs integrazioni eterogenee o bridge esterni): integrazione diretta e matura con contratti EVM[3], riducendo overhead di interoperabilità tra processi e linguaggi.
- **PostgreSQL con JSONB** (vs NoSQL puro): combina flessibilità sul documento TD con garanzie ACID[4], vincoli relazionali e query efficienti sui metadati semantici.
- **Hardhat locale** (vs testnet remota): ambiente riproducibile, costi nulli di gas reale, cicli rapidi di test e benchmark controllabili.

Perché non altre opzioni come scelta primaria Soluzioni come MongoDB, Redis-only o API GraphQL non sono state adottate come asse principale perché, nel perimetro della tesi, avrebbero privilegiato flessibilità generale rispetto a requisiti più critici: tracciabilità forte request–match, riproducibilità degli esperimenti e semplicità del percorso operativo Oracle. In particolare, Redis[5] rimane un candidato utile come cache evolutiva, ma non sostituisce la persistenza strutturata necessaria per analisi e validazione sperimentale.

In sintesi, le scelte effettuate privilegiano **coerenza con il flusso on-chain**, **prestazioni sul percorso critico** e **riproducibilità sperimentale**, risultando più adatte agli obiettivi della tesi rispetto ad alternative più generiche ma meno focalizzate sul caso d'uso.

[3]Per *contratti EVM* si intendono smart contract eseguiti dalla *Ethereum Virtual Machine* (EVM), cioè l'ambiente di esecuzione compatibile con Ethereum e con molte blockchain "EVM-compatible", che supporta bytecode e standard come Solidity.

[4]Le garanzie *ACID* indicano proprietà di affidabilità delle transazioni: *Atomicità* (tutto o niente), *Consistenza* (rispetta i vincoli), *Isolamento* (transazioni concorrenti non si interferiscono) e *Durabilità* (una volta confermato, il dato resta persistente).

[5]Redis è un database *in-memory* (a chiave-valore) molto usato come cache: conserva i dati in RAM per accessi rapidissimi e può applicare politiche di scadenza (TTL), riducendo il carico sul database principale.

Capitolo 4

Implementazione

4.1 Stack tecnologico e struttura del progetto

L'implementazione dell'indexer è stata realizzata in ambiente Node.js. Il servizio espone API REST tramite Express, interagisce con smart contract in ambiente Ethereum locale (Hardhat) tramite ethers.js e utilizza PostgreSQL come persistenza, memorizzando le Thing Descriptions in formato `jsonb`.

La struttura del codice è organizzata per responsabilità, separando: (i) gestione e persistenza delle TD; (ii) motore di matching; (iii) integrazione blockchain (listener eventi e invio transazioni); (iv) layer API. **Requisiti coperti: RF1–RF6** (Tabella 3.1).

Listing 4.1: Struttura principale del progetto

```
src/
  app.js
  server.js
  api/
    controllers/dataController.js
    routes/dataRoutes.js
    middlewares/errorHandler.js
  config/index.js
  core/
    services/tdMatchService.js
    repositories/tdRepository.js
    repositories/tdMatchesRepository.js
  infrastructure/
    blockchain.js
    db.js
  scripts/
    initDb.js
```

```
    loadTds.js
    seedLargeTds.js
    listener.js
    benchmarkIndexer.js
    benchmarkApiLatency.js
contracts/
  IndexerRegistry.abi.json
  RequestGate.abi.json
tds/
  td_list.json
```

4.2 Caricamento e persistenza delle Thing Descriptions

Requisiti coperti: RF1 e **RF2** (Tabella 3.1).

Le Thing Descriptions possono essere acquisite da un dataset locale in formato JSON (file `td_list.json` o percorso equivalente configurabile) e inserite in `td_store`. Per la validazione sperimentale su larga scala è stato utilizzato un dataset sintetico di 10.000 TD generato tramite lo script `seedLargeTds.js` e successivamente persistito nel database. In fase di ingestione, le TD vengono normalizzate e memorizzate nella tabella `td_store`, preservando il documento completo in formato `jsonb`. Questa scelta mantiene la TD come sorgente unica di verità e consente sia la consultazione via API sia l'estrazione dei campi necessari al matching. Il percorso del file è gestito tramite configurazione locale dell'ambiente di esecuzione, in modo da non vincolare il sistema a una specifica struttura di directory. In particolare, il controllo che il file contenga un array JSON e la persistenza in `td_store` soddisfano rispettivamente la validazione minima e la memorizzazione richiesta da **RF1** e **RF2**.

Listing 4.2: Configurazione del path dataset TD

```
files: {
  tdList: process.env.TD_LIST_FILE
    ? path.resolve(process.env.TD_LIST_FILE)
    : path.join(__dirname, "..", "..", "tds", "td_list.json"),
},
```

Listing 4.3: Caricamento dataset e inserimento TD in PostgreSQL

```
const filePath = config.files.tdList;
const raw = await fs.readFile(filePath, "utf-8");
const tdArray = JSON.parse(raw);
```

```
4
5  if (!Array.isArray(tdArray)) {
6    throw new Error("Il␣file␣TD␣deve␣contenere␣un␣array␣JSON");
7  }
8
9  for (const td of tdArray) {
10   await tdRepository.insertTd(td);
11 }
```

Listing 4.4: Inserimento della TD in `td_store`

```
1 const { rows } = await client.query(
2   "INSERT␣INTO␣td_store␣(td)␣VALUES␣($1)␣RETURNING␣id",
3   [td]
4 );
```

Per esperimenti di scalabilità è disponibile anche uno script di seeding sintetico (`seedLargeTds.js`) con generazione batch e cardinalità configurabile (default 10.000 TD). In tale modalità, la distribuzione dei tipi `@type` è deterministica e pesata: `Sensor` (40%), `Actuator` (20%) e `Gateway`/`Meter`/`Camera`/`Thermostat` (10% ciascuno). Questa distribuzione è rilevante per l'interpretazione delle metriche di *match-rate* e *matches per request*, poiché il matching del prototipo confronta `query` con `@type`.

Tabella 4.1: Distribuzione dei valori di `@type` nel dataset sintetico (10.000 TD) generato da `seedLargeTds.js`.

Tipo (`@type`)	**Conteggio**	**Percentuale**
Sensor	4000	40.0%
Actuator	2000	20.0%
Gateway	1000	10.0%
Meter	1000	10.0%
Camera	1000	10.0%
Thermostat	1000	10.0%

4.3 Matching semantico e persistenza dei match

Requisiti coperti: RF4 (e supporto a **RF2** per query/retrieval) (Tabella 3.1).
Il matching confronta i vincoli della richiesta con le Thing Descriptions disponibili. In questa implementazione, la compatibilità viene valutata principalmente su metadati e struttura della TD (ad esempio *types* e affordance), producendo un insieme di TD compatibili. Gli esiti vengono registrati nella tabella `td_matches` associando

l'identificativo della richiesta (`request_id`) agli identificativi delle TD compatibili (`td_id`) e al timestamp di matching.
Questa componente realizza il requisito **RF4** traducendo il vincolo della richiesta in una selezione di TD compatibili e registrando gli esiti in `td_matches` per il retrieval successivo.

Listing 4.5: Matching per tipo nel service layer

```
async findCompatibleTds(requiredType) {
  const candidateTdIds = await tdRepository.findTdIdsByType(
      requiredType);

  if (candidateTdIds.length === 0) {
    return [];
  }

  const tds = await tdRepository.findTdsByIds(candidateTdIds);
  return tds;
}
```

Listing 4.6: Query `jsonb` per filtro su `@type`

```
const { rows } = await client.query(
  `SELECT id FROM td_store
   WHERE (td->>'@type' = $1
   OR td->'@type' @> $2::jsonb)`,
  [type, JSON.stringify([type])]
);
```

Gli esiti vengono salvati in `td_matches` con vincolo di unicità sulla coppia `(request_id, td_id)`, così da rendere l'inserimento idempotente.

Listing 4.7: Schema e vincolo di unicità in `td_matches`

```
CREATE TABLE IF NOT EXISTS td_matches (
  id SERIAL PRIMARY KEY,
  request_id TEXT NOT NULL,
  td_id UUID NOT NULL,
  matched_at TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE UNIQUE INDEX IF NOT EXISTS td_matches_request_td_uniq
ON td_matches (request_id, td_id);
```

Listing 4.8: Inserimento idempotente dei match

```
const { rowCount } = await client.query(
  `INSERT INTO td_matches (request_id, td_id, matched_at)
   SELECT $1, unnest($2::uuid[]), NOW()
   ON CONFLICT (request_id, td_id) DO NOTHING`,
  [requestId, tdIds]
);
```

4.4 Integrazione con blockchain: submit richiesta e listener eventi

Requisiti coperti: RF3 e **RF5** (Tabella 3.1).
La generazione del carico è orchestrata dal benchmark, che invoca lo script Hardhat di submit richiesta per ciascuna fase. Il listener intercetta l'evento `RequestSubmitted`, esegue il matching e persiste i risultati.
In particolare, l'intercettazione dell'evento `RequestSubmitted` soddisfa **RF3**, mentre la chiamata `applyToRequest` implementa la candidatura/registrazione di disponibilità prevista da **RF5**.

Listing 4.9: Differenziazione reale dei tre scenari nel benchmark (estratto)

```
const phases = [
  { name: "1_rpm", command: options.command1 },
  { name: "10_rpm", command: options.command10 },
  { name: "30_rpm", command: options.command30 },
];

async function runPhase(options, phaseName, command) {
  const targetRpm = phaseName === "1_rpm" ? 1 : phaseName === "10_rpm"
      ? 10 : 30;
  const intervalMs = Math.floor(60000 / targetRpm);

  const phaseEndMs = Date.now() + options.phaseSeconds * 1000;
  let nextRunAtMs = Date.now();

  while (Date.now() < phaseEndMs) {
    if (Date.now() < nextRunAtMs) {
      await sleep(Math.min(200, nextRunAtMs - Date.now()));
      continue;
    }

```

```
20      // stesso comando, ma pianificato con intervalMs diverso
21      await runSingleCommand(command, options.protocolPath);
22      nextRunAtMs += intervalMs;
23    }
24  }
```

Nota. I comandi di submit possono essere identici tra le fasi; la differenza tra `1_rpm`, `10_rpm` e `30_rpm` è introdotta dal benchmark tramite `targetRpm` e dall'intervallo di scheduling `intervalMs = 60000/targetRpm`.

Listing 4.10: Listener evento + matching + persistenza

```
1  const onRequestHandler = async (requestId, requiredType) => {
2    const tds = await tdMatchService.findCompatibleTds(requiredType);
3
4    if (tds.length > 0) {
5      await blockchain.applyToRequest(requestId); // offerta on-chain
6      blockchain.storeOfferedTds(requestId, tds); // cache in-memory
7      await tdMatchesRepository.insertMatches(requestId, tds); //
           persistenza DB
8    }
9  };
10
11 blockchain.listenToRequests(onRequestHandler);
```

4.5 API REST: recupero dei risultati per requestId

Requisiti coperti: RF6 (Tabella 3.1).
L'endpoint `GET /data/:requestId` restituisce i risultati con lookup in memoria e fallback su DB. A scopo sperimentale, l'header `x-data-source` segnala la sorgente effettiva (`cache`/`db`).

Tabella 4.2: Endpoint REST esposto dall'Indexer.

Metodo	Path	Parametro	Risposta
GET	`/data/:requestId`	`requestId`	Lista di TD

Listing 4.11: Route e controller dell'endpoint `GET /data/:requestId`

```
router.get("/:requestId", dataController.getTdData);

// controller (estratto)
let source = "cache";
let tds = global.blockchain?.getOfferedTds(requestId) || [];

if (tds.length === 0) {
  tds = await tdMatchesRepository.findTdsByRequestId(requestId);
  source = "db";
}

// NB: uniqueTds rappresenta l'array dopo deduplica/normalizzazione.
res.set("x-data-source", source);
res.json({ requestId, count: uniqueTds.length, source, data: uniqueTds
    });
```

L'endpoint `GET /data/:requestId` soddisfa **RF6** fornendo agli Oracle l'accesso ai risultati del matching associati a una richiesta, con lookup in cache e fallback su `td_matches`.

L'API: espone un unico endpoint per ottenere le Thing Descriptions associate a una specifica richiesta. Questa scelta riflette il perimetro del lavoro e i requisiti del flusso ZONIA, nel quale gli oracoli interrogano l'indexer disponendo già dell'identificativo della richiesta (`requestId`) ottenuto dalla componente on-chain. L'endpoint recupera i risultati prioritariamente dalla cache in memoria e, in caso di assenza, effettua un fallback su PostgreSQL tramite la tabella `td_matches`.

4.6 Script di test e benchmark

Per supportare la validazione del prototipo e garantire riproducibilità sperimentale, sono stati sviluppati due strumenti principali: (i) `benchmarkIndexer.js`, che misura il comportamento end-to-end dell'Indexer lungo l'intera pipeline (submit richiesta → matching → persistenza → disponibilità via API), e (ii) `benchmarkApiLatency.js`, che isola la sola latenza dell'endpoint `GET /data/:requestId` sotto concorrenza controllata.

Workload e scenari

Il workload è generato tramite lo script Hardhat `simulateRequest.js`, invocato dal benchmark principale: ogni esecuzione sottomette una richiesta on-chain

(`submitRequest`) con un valore `query` che rappresenta il vincolo semantico utilizzato nel matching. Nel prototipo, il matching confronta `query` con il campo `@type` delle TD memorizzate nel database; di conseguenza, le metriche *matches per request* e *match-rate* dipendono dalla distribuzione dei tipi nel dataset e dalla sequenza (casuale) delle richieste generate in ciascuna fase.

4.6.1 Benchmark end-to-end dell'Indexer: funzionamento e procedura

Lo script `benchmarkIndexer.js` implementa il benchmark principale del prototipo e caratterizza il comportamento dell'Indexer lungo la pipeline operativa: generazione di richieste (submit on-chain tramite script Hardhat), acquisizione dei `requestId`, osservazione dei match persistiti su database, campionamento dell'API `GET /data/:requestId` e produzione di report aggregati per fase. A differenza del benchmark *API-only*, questo strumento misura il sistema nel suo complesso (nel setup sperimentale adottato), permettendo di confrontare tre regimi di carico nominale: `1_rpm`, `10_rpm`, `30_rpm`.

Parametri e modalità di esecuzione. Lo script è parametrico e permette di eseguire l'intera campagna o una singola fase (`-only-phase`). I parametri principali controllano: path del progetto protocol[1] (`-protocol-path`), durata della fase (`-phase-seconds`), dimensione campione per API latency (`-sample-size`), e directory di output (`-output-dir`). È inoltre disponibile `-reset-between-phases` per azzerare `td_matches` tra fasi quando si desidera un isolamento più netto.

Fasi di benchmark e selezione scenari. La campagna è organizzata in fasi distinte, una per ogni regime di carico. Questo facilita confronti diretti perché ogni fase produce un blocco coerente di misure e un riepilogo separato.
Ogni run del benchmark esegue le fasi in modo sequenziale: per default vengono eseguiti `1_rpm`, `10_rpm` e `30_rpm` in quest'ordine; in alternativa, è possibile eseguire una singola fase tramite `-only-phase`.

[1] Nel contesto del progetto, la cartella `protocol` indica la directory che contiene lo stack "on-chain" usato nel benchmark (progetto Hardhat con script di simulazione e artefatti dei contratti). Il parametro `protocolPath` serve a lanciare i comandi Hardhat nel working directory corretto.

Listing 4.12: Definizione delle fasi e selezione (estratto)

```
const phases = [
  { name: "1_rpm", command: options.command1 },
  { name: "10_rpm", command: options.command10 },
  { name: "30_rpm", command: options.command30 },
];

const selectedPhases = options.onlyPhase
  ? phases.filter((p) => p.name === options.onlyPhase)
  : phases;
```

Generazione del carico: rate control. All'interno di ciascuna fase, lo script converte il target nominale (rpm) in un intervallo temporale tra invii e schedula l'esecuzione del comando di submit in modo regolare fino al termine della finestra temporale. Questo evita burst[2] casuali e rende confrontabili le fasi.

Listing 4.13: Rate control per fase (estratto)

```
const targetRpm = phaseName === "1_rpm" ? 1 : phaseName === "10_rpm" ?
    10 : 30;
const intervalMs = Math.floor(60000 / targetRpm);

const phaseEndMs = startedAt.getTime() + options.phaseSeconds * 1000;
let nextRunAtMs = startedAt.getTime();

while (Date.now() < phaseEndMs) {
  if (Date.now() < nextRunAtMs) {
    await sleep(Math.min(200, nextRunAtMs - Date.now()));
    continue;
  }

  const singleResult = await runSingleCommand(command, options.
      protocolPath);
  // ...
  nextRunAtMs += intervalMs;
}
```

Tracciamento delle submission e parsing dei log. Per ogni invio, lo script registra un timestamp locale e ricava dai log il `requestId`. Inoltre, estrae la `query`

[2]Con *burst* si intende un picco improvviso di richieste inviate in un intervallo molto breve; la schedulazione regolare evita carichi "a scatti" e rende i test più confrontabili.

stampata dallo script Hardhat, utile per analisi diagnostiche (distribuzione dei tipi richiesti) e per interpretare la variabilità del match-rate.

Listing 4.14: Tracciamento submission e parsing di `requestId`/`query` (estratto)

```
const commandStartedAtMs = Date.now();
const singleResult = await runSingleCommand(command, options.
    protocolPath);

const output = `${singleResult.stdoutText}\n${singleResult.stderrText
    }`;
const requestId = extractRequestId(output);
const query = extractQuery(output);

submissions.push({
  submittedAtMs: commandStartedAtMs,
  requestId,
  query,
  commandSucceeded: !singleResult.startupError && singleResult.code
     === 0,
});
```

Raccolta evidenze dal database (`td_matches`). A fine fase, lo script interroga `td_matches` nella finestra temporale della fase per ottenere: conteggio richieste distinte con match, numero totale di righe match e, per ciascun `requestId`, la cardinalità dei match e il primo timestamp di match. In questo modo le misure sono ancorate a evidenze persistite (e non solo a log runtime).

Listing 4.15: Query DB per statistiche di fase (estratto)

```
const statsQuery = `
  SELECT COUNT(DISTINCT request_id)::int AS request_count,
         COUNT(*)::int AS td_match_count
  FROM td_matches
  WHERE matched_at >= $1 AND matched_at <= $2
`;

const perRequestQuery = `
  SELECT request_id,
         COUNT(*)::int AS match_count,
         MIN(matched_at) AS first_matched_at
  FROM td_matches
  WHERE matched_at >= $1 AND matched_at <= $2
```

```
14   GROUP BY request_id
15 `;
```

Campionamento API a fine fase. Per includere il comportamento del retrieval HTTP nel contesto della fase, lo script seleziona un campione di `requestId` recenti (ordinati per `first_matched_at` decrescente) e invoca l'endpoint `GET /data/:requestId`. Le chiamate vengono classificate tramite `x-data-source` per distinguere cache e fallback DB.

Listing 4.16: Selezione campione e invocazione `sampleApiLatency` (estratto)

```
1 const sampledRequestIds = perRequestRows
2   .slice()
3   .sort((a, b) => new Date(b.first_matched_at) - new Date(a.
       first_matched_at))
4   .map((row) => row.request_id);
5
6 const apiMetrics = await sampleApiLatency(options.apiBaseUrl,
     sampledRequestIds, options.sampleSize);
```

Costruzione delle metriche end-to-end. La componente end-to-end viene ricostruita accoppiando, per ogni `requestId`, il timestamp di submission registrato dallo script e il timestamp del primo match persistito su database (`first_matched_at`). Questa scelta rende la misura indipendente dalla presenza di logging interno all'Indexer e sfrutta la persistenza come riferimento temporale.

Listing 4.17: Accoppiamento submission e primo match (estratto)

```
1 const firstMatchedAtMs = new Date(row.first_matched_at).getTime();
2 const e2eMs = firstMatchedAtMs - submittedAtMs;
3 if (e2eMs >= 0) e2eLatencies.push(e2eMs);
```

Report per fase e report complessivo. Per ogni fase lo script produce una struttura risultato che include: parametri di esecuzione, contatori di submit (tentativi/successi/fallimenti), statistiche da DB, sintesi API campionata, sintesi E2E, qualità del matching e distribuzione delle `query`. Al termine della campagna, tali risultati vengono salvati in JSON/CSV e viene aggiornato uno storico cumulativo (`benchmark-history.csv`) per facilitare confronti tra run.

Listing 4.18: Esportazione report JSON/CSV e storico cumulativo (estratto)

```
1 await fs.writeFile(jsonPath, JSON.stringify(report, null, 2));
2 await fs.writeFile(csvPath, toCsv(results));
```

```
3
4 if (historyExists) {
5   await fs.appendFile(historyCsvPath, `${bodyOnly}\n`);
6 } else {
7   await fs.writeFile(historyCsvPath, historyBlock);
8 }
```

Nota sul perimetro. Il benchmark end-to-end descrive il comportamento del prototipo nel setup locale adottato e consente confronti coerenti tra regimi di carico. L'interpretazione dei risultati deve tenere conto dei limiti discussi nella sezione di validità (ambiente locale, run limitate, variabilità del workload), ma la struttura a fasi e la persistenza automatica dei report garantiscono tracciabilità e riproducibilità delle misure.

4.6.2 Benchmark API latency: funzionamento e procedura

Il benchmark *API latency* è stato progettato per caratterizzare il comportamento dell'endpoint `GET /data/:requestId` in condizioni controllate, focalizzandosi sul solo layer di retrieval (HTTP + cache/DB) e riducendo l'influenza della pipeline end-to-end (evento on-chain, matching e persistenza). In pratica, lo script simula più Oracle concorrenti che interrogano l'Indexer per ottenere i risultati associati a un determinato `requestId`.

Input e parametri di controllo. Lo script `benchmarkApiLatency.js` accetta parametri che permettono di controllare la pressione concorrente e la ripetizione del test:

- `apiBaseUrl`: URL base dell'API dell'Indexer (default `http://localhost:3000`);
- `requestId`: identificativo della richiesta da interrogare (opzionale);
- `oracles`: numero di Oracle finti concorrenti per round (default 10);
- `rounds`: numero di round consecutivi (default 1);
- `roundDelayMs`: pausa opzionale tra round (default 0);
- `timeoutMs`: timeout per singola richiesta HTTP (default 15000 ms);
- `outputDir`: directory di output dei report (default `reports/`).

Questa parametrizzazione consente di ripetere la stessa prova variando una sola variabile alla volta (ad esempio il numero di Oracle concorrenti), mantenendo costanti le altre condizioni.

Selezione del `requestId`. Il benchmark può operare in due modalità:

- **requestId esplicito:** l'utente fornisce il `requestId` da testare;
- **requestId automatico:** se non viene specificato, lo script recupera l'ultimo `requestId` disponibile nella tabella `td_matches`, ordinando per `matched_at` decrescente. In questo modo si evita di testare identificativi obsoleti e si garantisce che il `requestId` sia associato a risultati effettivamente persistiti.

Listing 4.19: Selezione automatica dell'ultimo `requestId` da `td_matches` (estratto)

```
const result = await pool.query(‘
  SELECT request_id
  FROM td_matches
  ORDER BY matched_at DESC
  LIMIT 1
‘);
const requestId = result.rows[0]?.request_id;
```

Modello di concorrenza a round. L'esecuzione è organizzata in round. In ciascun round, lo script crea O task asincroni (uno per Oracle finto) e li avvia in parallelo con `Promise.all`. Ogni Oracle invia una chiamata `GET` sullo stesso `requestId`; quindi, ogni round produce O chiamate totali, e ciascun Oracle effettua una chiamata per round.

Listing 4.20: Esecuzione concorrente di un round con `Promise.all` (estratto)

```
const tasks = [];
for (let oracleId = 1; oracleId <= options.oracles; oracleId += 1) {
  tasks.push(runOracleRequest(options.apiBaseUrl, requestId, options.
      timeoutMs, oracleId, round));
}
const roundRows = await Promise.all(tasks);
```

Misura per chiamata e classificazione cache/DB. Per ogni chiamata lo script registra un record contenente: numero di round, identificativo dell'oracolo finto, esito, status HTTP, latenza e sorgente dati. La sorgente viene estratta prioritariamente dall'header `x-data-source` (impostato dall'API dell'Indexer) e, in alternativa, dal

campo `source` nel body della risposta. Questo consente di distinguere le risposte servite dal percorso veloce (cache) da quelle servite tramite fallback su database.

Listing 4.21: Misura latenza e classificazione `cache/db` (estratto)

```
const startedAt = Date.now();
const response = await axios.get(`${apiBaseUrl}/data/${requestId}`, {
    timeout: timeoutMs });
const latencyMs = Date.now() - startedAt;

const source = String(
  response.headers?.["x-data-source"] || response.data?.source || "
     unknown"
).toLowerCase();
```

Aggregazione per round e per run. Al termine di ogni round, lo script aggrega i record del round e stampa una sintesi a console (successi/errori, latenza media/-massima, cache hit-rate). Terminati tutti i round, esegue una sintesi complessiva sull'intera run. Questa struttura (round → sintesi round → sintesi run) è utile anche per osservare effetti transitori come warm-up (ad esempio differenze tra primo e secondo round).

Report e riproducibilità. Lo script salva automaticamente i risultati in `outputDir` in due formati:

- **JSON**: configurazione di esecuzione, riepilogo aggregato e lista completa dei record per chiamata;
- **CSV**: tabella con una riga per chiamata (round, oracleId, ok, status, latencyMs, source, error).

I file sono nominati con timestamp (`api-latency-<timestamp>.json/csv`), rendendo ogni esecuzione identificabile e confrontabile.

Listing 4.22: Esportazione report JSON/CSV con timestamp (estratto)

```
const stamp = new Date().toISOString().replace(/[:.]/g, "-");
const jsonPath = path.join(outputDir, `api-latency-${stamp}.json`);
const csvPath = path.join(outputDir, `api-latency-${stamp}.csv`);

await fs.writeFile(jsonPath, JSON.stringify(report, null, 2));
await fs.writeFile(csvPath, toCsvRows(allRows));
```

Nota sul perimetro. Il benchmark descrive il comportamento dell'API dell'Indexer (serving via cache/DB) sotto concorrenza controllata. Non include le fasi on-chain né il tempo di matching, che sono invece considerate nella valutazione end-to-end.

Capitolo 5

Valutazione esperimentale e Risultati ottenuti

5.1 Obiettivo della valutazione

La valutazione sperimentale ha lo scopo di verificare la fattibilità dell'approccio e di quantificare il comportamento dell'indexer in scenari controllati, con particolare attenzione a: (i) stabilità dell'API REST; (ii) latenza di recupero dei risultati; (iii) limiti di throughput nella pipeline end-to-end che coinvolge richieste e interazioni on-chain.

5.2 Metriche

Le metriche sono calcolate automaticamente dallo script di benchmark e si definiscono come segue.

- **Latenza API.** Misura la latenza delle chiamate riuscite `GET /data/{requestId}`. Per ciascuna richiesta r:

$$L_{\text{API}}(r) = t_{\text{response}}(r) - t_{\text{request}}(r). \tag{5.1}$$

  I risultati sono riassunti tramite valore medio e massimo nel campione osservato:

$$\overline{L}_{\text{API}} = \frac{1}{N}\sum_{i=1}^{N} L_i, \qquad L_{\text{API}}^{\max} = \max_i L_i. \tag{5.2}$$

- **Latenza end-to-end (E2E):** misura l'intervallo tra la sottomissione della richiesta (timestamp applicativo registrato dal benchmark) e l'istante in cui il primo match relativo a quella richiesta risulta persistito in `td_matches`

(`first_matched_at`). Per ciascuna richiesta completata r:

$$L_{\mathrm{E2E}}(r) = t_{\mathrm{first_matched}}(r) - t_{\mathrm{submitted}}(r), \tag{5.3}$$

dove $t_{\mathrm{submitted}}(r)$ è il timestamp di sottomissione (lato benchmark) e $t_{\mathrm{first_matched}}(r)$ è il timestamp del primo match persistito in `td_matches`. Anche in questo caso i risultati sono riassunti tramite `avg` e `max` nel campione:

$$\overline{L}_{\mathrm{E2E}} = \frac{1}{N}\sum_{i=1}^{N} L_i, \qquad L_{\mathrm{E2E}}^{\max} = \max_i L_i. \tag{5.4}$$

- **Throughput end-to-end e saturazione.** Misura quante richieste vengono completate per minuto nell'intero workflow (submit richiesta → matching → persistenza). Sia $N_{\mathrm{completed}}$ il numero di richieste completate nella fase e T_{fase} la durata della fase in secondi:

$$\mathrm{Throughput}_{\mathrm{E2E}}\ (\mathrm{req/min}) = \frac{N_{\mathrm{completed}}}{T_{\mathrm{fase}}\ (\mathrm{s})} \cdot 60. \tag{5.5}$$

  La *saturazione* è identificata come la soglia di carico oltre la quale il throughput non cresce proporzionalmente al target nominale.

- **Matches per request e match-rate.** Descrive la "selettività" del matching misurando (i) il numero di TD compatibili per richiesta e (ii) la quota di richieste che producono almeno un match.

  Indicando con m_r il numero di match associati alla richiesta r, si definiscono:

$$\overline{m} = \frac{1}{R}\sum_{r=1}^{R} m_r, \tag{5.6}$$

$$m^{\min} = \min_r m_r, \qquad m^{\max} = \max_r m_r, \tag{5.7}$$

  dove R è il numero di richieste osservate nella fase.

  Sia inoltre $R_{\geq 1} = \#\{r \mid m_r \geq 1\}$ il numero di richieste con almeno un match e sia D un denominatore che rappresenta il numero di richieste osservate nella fase (ad esempio $D = \max(N_{\mathrm{successfulSubmissions}}, R)$). Il match-rate è definito come:

$$\mathrm{MatchRate}(\%) = \frac{R_{\geq 1}}{D} \cdot 100. \tag{5.8}$$

5.3 Generazione del dataset sintetico e motivazioni della scelta

Le Thing Descriptions possono essere acquisite da un dataset locale in formato JSON (ad esempio `td_list.json`) e inserite in `td_store`. Tuttavia, nel contesto della validazione sperimentale della presente tesi, le 10.000 TD utilizzate nel benchmark non provengono da un file esterno: vengono generate sinteticamente dallo script `seedLargeTds.js` e caricate a batch nel database. Questo approccio consente di costruire un dataset controllato, riproducibile e facilmente parametrizzabile, riducendo l'overhead operativo nella ripetizione degli esperimenti.

Listing 5.1: Generazione sintetica delle 10.000 TD nello script di seed (estratto)

```
function parseArgs(args) {
  const parsed = {
    count: 10000, // default: 10.000 TD
    batchSize: 500,
    truncate: true,
  };
  // ...
  return parsed;
}

function buildTd(index) {
  const idNumber = index.toString().padStart(5, "0");
  const type = pickType(index);

  return {
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "@type": type,
    title: `${type} Device ${idNumber}`,
    id: `urn:dev:zonia:${type.toLowerCase()}:${idNumber}`,
    description: `Synthetic ${type} generated for benchmark dataset`,
  };
}

for (let index = start; index < endExclusive; index += 1) {
  const td = buildTd(index + 1);
  batch.push(td);
}

await insertBatch(client, batch); // inserimento in td_store
```

In particolare, lo script imposta un valore di default pari a `count=10000` e costruisce programmaticamente ogni TD (campi `@context`, `@type`, `title`, `id`, `properties`, ecc.). Il valore del tipo `@type` (`Sensor`, `Actuator`, `Gateway`, `Meter`, `Camera`, `Thermostat`) è assegnato secondo una distribuzione pesata e deterministica; successivamente le TD vengono inserite in batch nella tabella `td_store`.

Motivazioni della scelta rispetto a un file JSON "gonfiato". L'adozione di uno script di seeding è preferibile rispetto al mantenimento di un singolo file JSON statico di grandi dimensioni (ad esempio un `td_list.json` contenente 10.000 TD) per i seguenti motivi:

- **Riproducibilità.** Lo stesso dataset può essere ricreato da zero con un comando, riducendo il rischio di modifiche manuali non tracciate o di incoerenze nel contenuto.
- **Manutenibilità.** Un file JSON molto grande è difficile da revisionare, lento da aprire e poco adatto alla gestione in repository; lo script, invece, rimane compatto e leggibile.
- **Flessibilità sperimentale.** Parametri come `-count` e `-batch-size` permettono di variare facilmente la cardinalità del dataset (ad esempio 1k, 10k, 50k) senza rigenerare e versionare nuovi file.
- **Controllo della distribuzione dei tipi.** Lo script definisce in modo esplicito e deterministico il mix dei tipi (`Sensor`, `Actuator`, ecc.), facilitando la documentazione sperimentale e l'interpretazione di metriche come match-rate e matches per request.
- **Efficienza operativa.** L'inserimento batch nel database è più efficiente del parsing ripetuto di un JSON molto grande, soprattutto quando gli esperimenti devono essere eseguiti più volte.

In sintesi, la generazione sintetica tramite `seedLargeTds.js` fornisce un compromesso efficace tra controllo sperimentale, ripetibilità e semplicità operativa, rendendo più robusto il processo di valutazione del prototipo. In particolare, poiché nel prototipo il matching confronta `query` con `@type`, la distribuzione dei tipi nel dataset determina direttamente la probabilità di ottenere almeno un match (*match-rate*) e la numerosità dei risultati (*matches per request*); rendere tale distribuzione esplicita e controllabile è quindi essenziale per interpretare correttamente i risultati sperimentali.

5.4 Risultati osservati

5.4.1 Latenza API

Per ciascuno scenario di carico (`1_rpm`, `10_rpm`, `30_rpm`), è stata prima eseguita la simulazione del workload con lo script principale di benchmark, così da generare richieste reali e popolare `td_matches`. Successivamente, è stato selezionato un `requestId` valido associato allo scenario e utilizzato come input del test API-only.

Setup sperimentale. Per ogni `requestId` selezionato, il benchmark API ha simulato `10` oracoli concorrenti (client HTTP fittizi) che interrogano simultaneamente l'endpoint. Ogni chiamata produce una misura puntuale contenente: identificativo dell'oracolo, esito (`OK/FAIL`), codice HTTP, sorgente della risposta (`cache/db`) e latenza in millisecondi.

Criteri di affidabilità. Per evitare misure non rappresentative, i test API sono stati eseguiti immediatamente dopo la generazione dello scenario corrispondente, utilizzando `requestId` effettivamente presenti in `td_matches`. In questo modo si è ridotto il rischio di falsi negativi dovuti a identificativi obsoleti o non più disponibili. L'intero processo è stato reso ripetibile tramite comandi scriptati e report versionati, così da garantire tracciabilità e riproducibilità sperimentale.

Esempio per-oracolo (concorrenza su uno stesso requestId). Per evidenziare la variabilità tra richieste parallele, si riporta un esempio di vista *per-oracolo*, in cui 10 consumer interrogano simultaneamente lo stesso `requestId`. La dispersione delle latenze riflette la normale variabilità del percorso di esecuzione (scheduling runtime, concorrenza lato server e tempi di accesso al livello dati).

Tabella 5.1: Latenza per oracolo finto – Scenario 10_rpm, Round 1; (requestId: `0xae2926aa1e0b35301e104a6bc50d7276690a985d4b6d7f56bf503c61b1fa83cd`)

Oracle ID	Esito	HTTP Status	Source	Latency (ms)
1	OK	200	db	532
2	OK	200	db	558
3	OK	200	db	411
4	OK	200	db	593
5	OK	200	db	419
6	OK	200	db	466
7	OK	200	db	565
8	OK	200	db	509
9	OK	200	db	576
10	OK	200	db	475

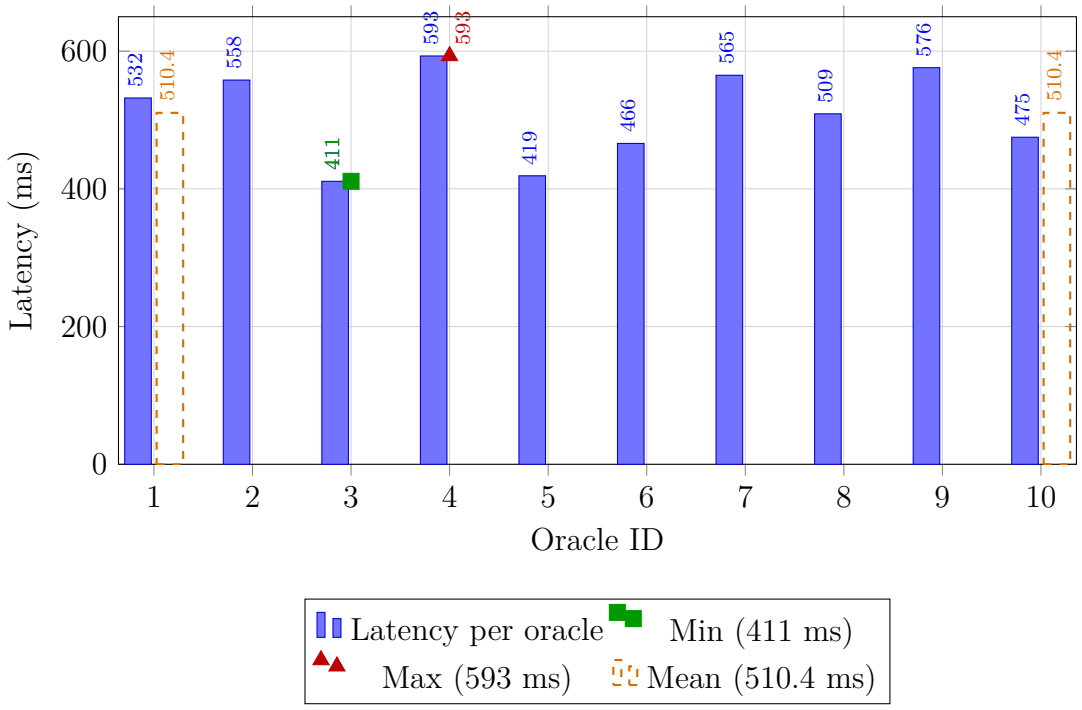

Figura 5.1: Scenario 1_rpm – Round 1: latenza per oracolo finto (10 chiamate concorrenti all'end-point).

Tabella 5.2: Latenza per oracolo finto – Scenario 10_rpm, Round 2; (requestId: `0xae2926aa1e0b35301e104a6bc50d7276690a985d4b6d7f56bf503c61b1fa83cd`)

Oracle ID	Esito	HTTP Status	Source	Latency (ms)
1	OK	200	db	45
2	OK	200	db	33
3	OK	200	db	66
4	OK	200	db	54
5	OK	200	db	79
6	OK	200	db	91
7	OK	200	db	106
8	OK	200	db	120
9	OK	200	db	142
10	OK	200	db	151

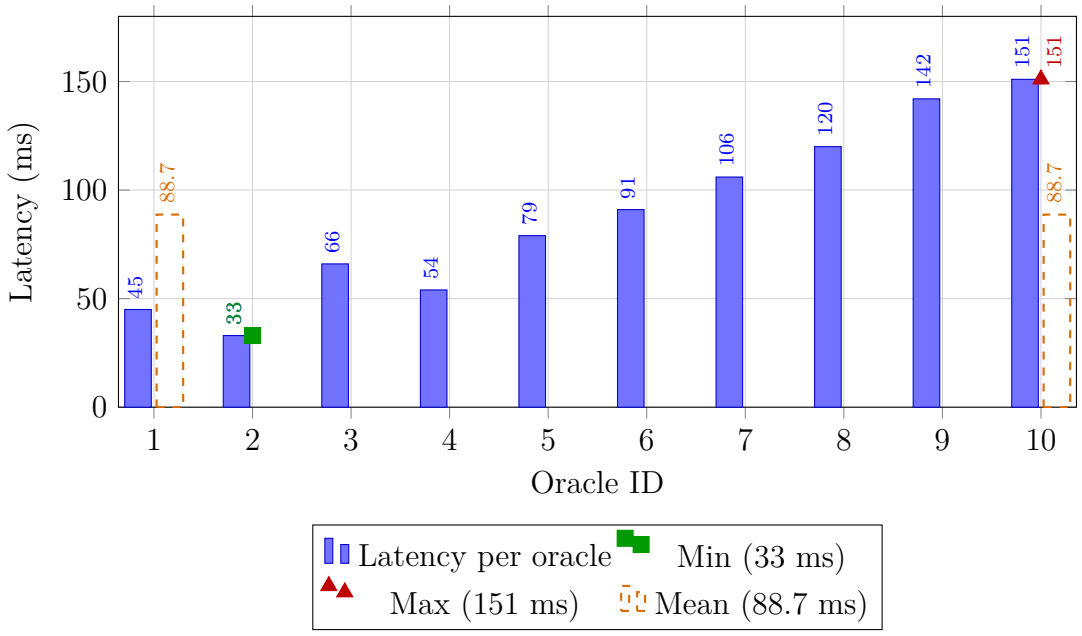

Figura 5.2: Scenario 10_rpm – Round 2: latenza per oracolo finto (10 chiamate concorrenti all'end-point).

Tabella 5.3: Latenza per oracolo finto – Scenario 30_rpm, Round 3

Oracle ID	Esito	HTTP Status	Source	Latency (ms)
1	OK	200	db	33
2	OK	200	db	42
3	OK	200	db	51
4	OK	200	db	62
5	OK	200	db	78
6	OK	200	db	92
7	OK	200	db	128
8	OK	200	db	138
9	OK	200	db	119
10	OK	200	db	181

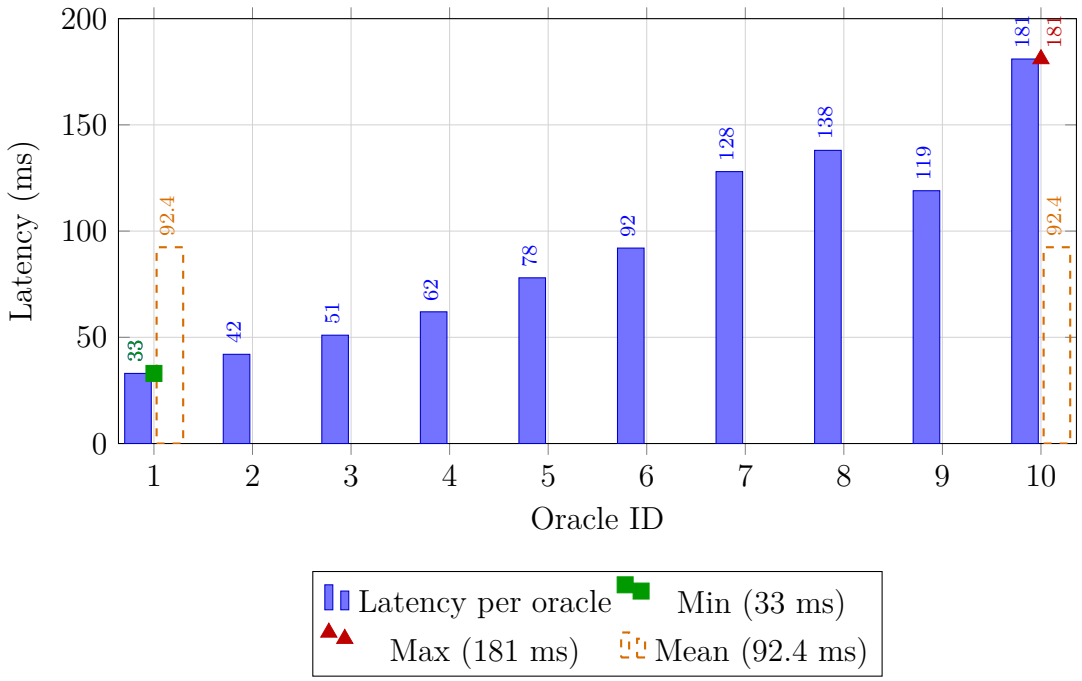

Figura 5.3: Scenario 30_rpm – Round 3: latenza per oracolo finto (10 chiamate concorrenti all'end-point sullo stesso requestId).

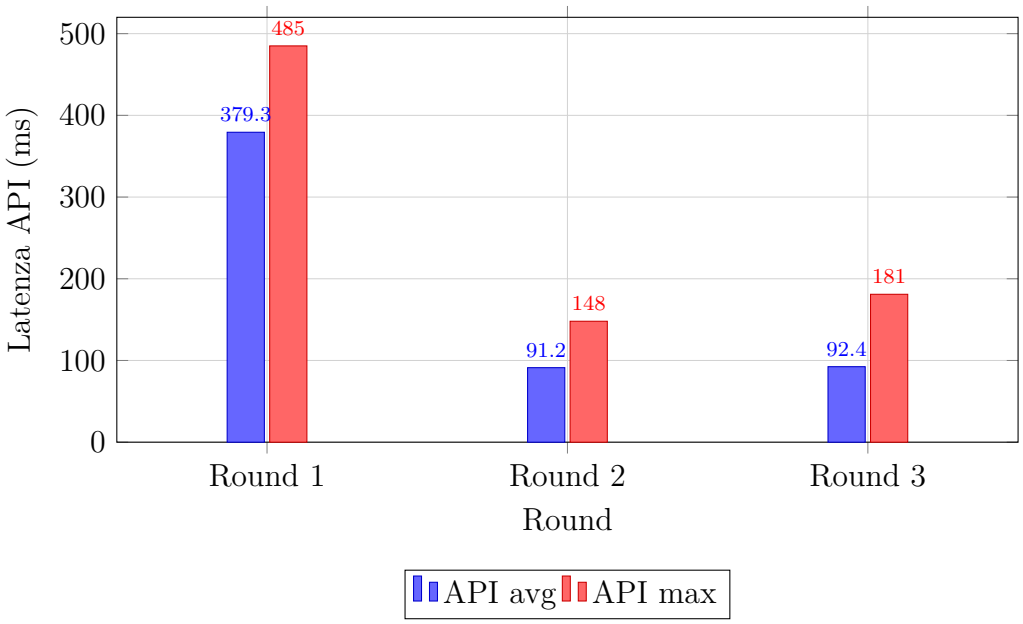

Figura 5.4: Sintesi della latenza API nei tre round (10 chiamate concorrenti all'endpoint sullo stesso requestId).

Interpretazione della Figura 5.4. La Figura 5.4 evidenzia un comportamento in due fasi. Nel **Round 1** la latenza risulta significativamente più alta (avg 379.3 ms, max 485 ms), mentre nei **Round 2** e **Round 3** si osserva una riduzione netta e una successiva stabilizzazione (avg $\approx$ 91–92 ms, max 148–181 ms).

La differenza tra il primo round e i successivi è coerente con un *effetto di warm-up* dell'intero percorso di serving: inizializzazione del runtime Node.js, stabilizzazione delle connessioni (HTTP e database), caching di risorse interne e ottimizzazione del percorso di esecuzione lato server. In altre parole, il Round 1 include costi di avvio e assestamento che non rappresentano pienamente il regime di funzionamento a sistema "caldo".

Il confronto tra Round 2 e Round 3 mostra invece prestazioni sostanzialmente stabili sul valore medio, con un lieve aumento del massimo nel Round 3 (181 ms vs 148 ms). Questo indica che, pur in condizioni stabili, permane una *variabilità residua* dovuta a concorrenza e scheduling (es. gestione parallela delle richieste, contesa su risorse, fluttuazioni nel tempo di accesso al livello dati). Nel complesso, l'assenza di errori e la stabilità dei valori medi nei round successivi suggeriscono che l'endpoint mantiene un comportamento robusto anche sotto interrogazioni concorrenti nello scenario 30_rpm.

5.4.2 Latenza end-to-end (E2E)

I risultati sono riportati tramite media e massimo osservati.

Tabella 5.4: Latenza end-to-end (E2E) per scenario di carico.

Scenario	Completate (n)	Sottomesse (n)	E2E avg (ms)	E2E max (ms)
1_rpm	1	1	9643.0	9643
10_rpm	6	8	9448.0	9873
30_rpm	6	8	9146.5	9698

Osservazioni. Come mostrato nella Table 5.4, la latenza E2E rimane nell'ordine di circa 9–10 secondi in tutte le fasi. Il fatto che E2E sia di un ordine di grandezza superiore alla latenza API indica che il costo dominante non è nel layer REST, bensì nella pipeline complessiva di gestione della richiesta e persistenza del primo match nel setup sperimentale adottato.

Perché `attese > sottomesse > completate`. Negli scenari `10_rpm` e `30_rpm` si osserva che il numero di richieste completate è inferiore a quello delle richieste sottomesse, a sua volta inferiore al numero teoricamente atteso. Tale comportamento è coerente con la natura *time-based* del benchmark: il numero atteso è calcolato su base nominale (es. 10 o 30 richieste/min), mentre la sottomissione reale dipende dal tempo effettivo di esecuzione del comando di invio e dal pacing del loop. Quando il costo di una singola iterazione (spawn processo, esecuzione script Hardhat, latenza blockchain e conferma transazione) è non trascurabile rispetto all'intervallo target, il generatore non riesce a mantenere perfettamente il rate nominale e produce meno invii del previsto.

Dalla sottomissione al completamento E2E. La metrica E2E richiede inoltre che la richiesta percorra l'intera pipeline: evento on-chain, intercettazione listener, matching, persistenza in `td_matches`. Per questo motivo una parte delle richieste sottomesse può non risultare "completata" entro la finestra temporale della fase. In prossimità della saturazione, infatti, il sistema accumula ritardo (*backlog*) e alcune richieste terminano oltre il limite della finestra di osservazione.

Interpretazione. La catena `attese > sottomesse > completate` non indica necessariamente errore funzionale, ma segnala un limite di capacità sotto quel carico e in quella durata di test. In altre parole, il rate nominale rappresenta un obiettivo di input, mentre il throughput E2E rappresenta la capacità realmente sostenibile

del sistema. Questa differenza è precisamente l'informazione utile per identificare il punto di saturazione.

In breve Negli scenari ad alto carico il benchmark è limitato dal costo reale di invio (processo + transazione + conferma), quindi le richieste sottomesse risultano inferiori al target nominale. A valle, solo una parte delle richieste sottomesse completa l'intera pipeline E2E entro la finestra temporale del test. Ne segue la relazione `attese > sottomesse > completate`, che rappresenta un effetto di saturazione e backlog, non un'anomalia della misura.

5.4.3 Throughput end-to-end e saturazione

La saturazione è identificata come la condizione in cui, aumentando il carico nominale (target req/min), il throughput osservato non cresce in modo proporzionale e/o diminuisce la percentuale di richieste effettivamente completate end-to-end.

Osservazioni La Table 5.5 riassume i risultati nei tre scenari. Nel caso `1_rpm` il sistema mantiene il target nominale (1 richiesta completata su 1 sottomessa), ottenendo 1.00 req/min. A `10_rpm` il throughput osservato sale a 6.97 req/min, ma con una lieve perdita di completamento (7 richieste completate su 8 sottomesse, 87.5%). Nel caso `30_rpm`, pur aumentando il target, il throughput scende a 5.37 req/min e la percentuale di completamento cala ulteriormente (6 completate su 9 sottomesse, 66.7%).

Interpretazioni Questo andamento evidenzia un regime di saturazione: oltre una certa soglia, incrementare il rate di sottomissione non produce un aumento proporzionale del numero di richieste completate end-to-end, ma introduce backlog e riduce il tasso di completamento complessivo.

Tabella 5.5: Throughput end-to-end per scenario di carico.

Scenario	Target (req/min)	Durata (s)	Sottomesse (n)	Completate (n)	Throughput (req/min)	Completion (%)
1_rpm	1	60.00	1	1	1.00	100.0
10_rpm	10	60.28	8	7	6.97	87.5
30_rpm	30	67.06	9	6	5.37	66.7

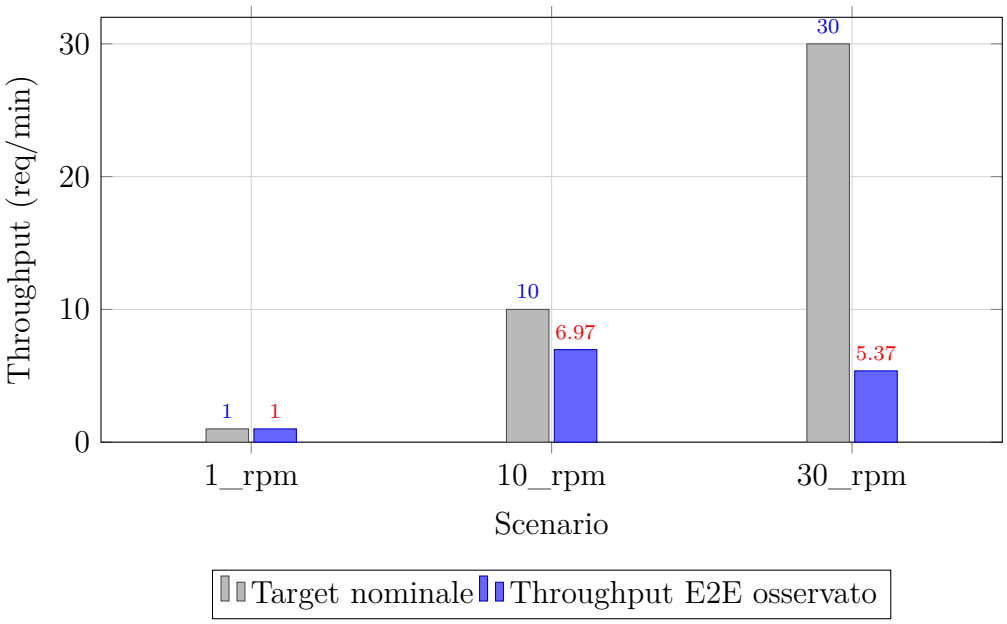

Figura 5.5: Confronto tra target nominale e throughput E2E osservato.

Interpretazione della Figura 5.5. Il sistema soddisfa il target nel caso `1_rpm`, mentre ai carichi maggiori emerge una divergenza crescente tra throughput nominale e throughput E2E realmente ottenuto. In particolare, passando da `10_rpm` a `30_rpm` il throughput osservato non aumenta in modo proporzionale, ma tende a stabilizzarsi su valori inferiori. Questo comportamento è indicativo di saturazione della pipeline end-to-end nel setup sperimentale considerato.

5.4.4 Matches per request e match-rate

Per descrivere la *selettività* del matching, si misura (i) il numero di Thing Descriptions compatibili restituite per ciascuna richiesta (statistiche min, max e media) e (ii) la quota di richieste che producono almeno un match, definita come:

$$\texttt{matchRate} = \frac{\#\{\texttt{requests con matches} > 0\}}{N} \cdot 100.$$

Osservazioni. La Table 5.6 mostra che il numero medio di match per richiesta aumenta passando da 1_rpm a 30_rpm, mentre l'intervallo osservato rimane 1000–4000. Il match-rate risulta elevato ma varia tra le fasi: tale variazione è coerente con la generazione casuale della `query` nello script di workload e con la distribuzione dei tipi presenti nel dataset, che può rendere alcune richieste più o meno "facili" da soddisfare in ciascuna finestra temporale.

Tabella 5.6: Matches per request e match-rate nei tre scenari di carico.

Scenario	Matches avg	Matches min	Matches max	Match-rate (%)
1_rpm	1000.00	1000	1000	100.00
10_rpm	1428.57	1000	4000	87.50
30_rpm	1666.67	1000	4000	66.67

Interpretazione della variazione del match-rate. La riduzione del match-rate passando da `1_rpm` a `30_rpm` è spiegabile alla luce del metodo di generazione del workload e della composizione del dataset. Nel benchmark, il campo `query` è selezionato casualmente da un insieme finito di tipi, mentre il matching nel prototipo verifica la compatibilità confrontando `query` con `@type`. Di conseguenza, la probabilità che una richiesta produca almeno un match dipende direttamente sia dalla distribuzione dei tipi nel dataset (Tabella 4.1) sia dalla sequenza casuale di tipi estratti durante la finestra temporale della fase. Inoltre, nel setup adottato la lista dei tipi generati dal workload include anche `Location`, che non è presente nel dataset sintetico: le richieste di tipo `Location` generano necessariamente zero match e contribuiscono alla riduzione del match-rate osservata. Infine, in condizioni di carico più elevato la saturazione end-to-end può ridurre il numero di richieste che completano la pipeline entro la finestra considerata, influenzando il conteggio dei match effettivamente osservabili nella fase.

5.5 Discussione

I risultati sperimentali mostrano un quadro coerente tra le diverse viste adottate (API-only con 10 oracoli concorrenti e benchmark end-to-end). Da un lato, le misure *API-only* indicano che l'endpoint `GET /data/:requestId` mantiene stabilità funzionale sotto concorrenza (assenza di errori nel campione) e tempi di risposta contenuti rispetto alla scala temporale complessiva del sistema. La differenza tra Round 1 e Round 2/3 evidenzia inoltre un chiaro effetto di warm-up: una volta "a regime", la latenza media si stabilizza e la variabilità residua resta limitata.

Dall'altro lato, le metriche end-to-end evidenziano che il tempo dominante e i limiti di capacità non sono determinati dal layer HTTP, bensì dalla pipeline complessiva che porta dalla sottomissione della richiesta alla persistenza del primo match in `td_matches`. In particolare, la latenza E2E rimane nell'ordine di 9–10 secondi, mentre il throughput E2E mostra un comportamento di saturazione: all'aumentare del carico nominale oltre una certa soglia, la capacità di completare richieste end-to-end non cresce in modo proporzionale e tende a stabilizzarsi o a ridursi.

Un ulteriore elemento interpretativo riguarda la qualità del matching. Le statistiche *matches per request* e il *match-rate* risultano complessivamente elevati, ma variano tra le fasi. Tale variazione è attesa, poiché lo script di workload seleziona casualmente la `query` (tipi richiesti) e l'esito dipende dalla distribuzione dei tipi presenti nel dataset di TD. Ne consegue che, a parità di infrastruttura, alcune finestre temporali possono produrre richieste più o meno "facili" da soddisfare.

Nel perimetro di questa tesi, gli interventi più efficaci per migliorare la scalabilità riguardano quindi l'ottimizzazione della pipeline end-to-end (ad esempio riduzione delle operazioni critiche, batching e strategie di coordinamento), più che la sola persistenza dei metadati o l'endpoint REST, che nello scenario considerato risulta adeguatamente performante e robusto.

Confronto con ZONIA

Il lavoro presentato in questa tesi si colloca nel solco dell'architettura ZONIA [1], che propone un sistema oracle-oriented e zero-trust per applicazioni blockchain in ambito IoT/WoT, distinguendo chiaramente tra coordinamento on-chain e componenti off-chain specializzati (tra cui l'Indexer). Rispetto alla visione completa di ZONIA, l'implementazione qui sviluppata è focalizzata sul solo Indexer e, per ragioni di perimetro, limita il matching alla dimensione semantica delle Thing Descriptions (senza componente geospaziale e senza meccanismi avanzati di ranking).

Dal punto di vista funzionale, i risultati sperimentali confermano la fattibilità del flusso previsto: una richiesta pubblicata on-chain viene intercettata off-chain, processata tramite matching su metadati e resa disponibile agli Oracle tramite un endpoint REST interrogabile a partire dal `requestId`. In linea con l'impostazione di ZONIA, emerge inoltre che il layer API non rappresenta il collo di bottiglia principale (latenze contenute nei campioni osservati), mentre i limiti più rilevanti si manifestano nella pipeline end-to-end che include la gestione delle richieste on-chain e la sincronizzazione con la persistenza.

In termini di implicazioni, questo risultato è coerente con l'obiettivo di ZONIA di mantenere on-chain il coordinamento e off-chain le operazioni più intensive (discovery/matching), ma evidenzia che la scalabilità complessiva dipende fortemente dall'implementazione pratica della pipeline e dall'overhead introdotto dai passaggi di submit/processing. Pertanto, le estensioni più naturali del prototipo includono l'introduzione di criteri di selezione più ricchi (oltre `@type`), l'eventuale supporto geospaziale previsto dall'architettura e strategie per migliorare il throughput end-to-end (ad esempio ottimizzazioni nella gestione delle richieste e nel coordinamento con la componente on-chain).

5.6 Limiti sperimentali e minacce alla validità

Nonostante i risultati ottenuti siano coerenti con gli obiettivi del prototipo, l'interpretazione deve considerare alcuni limiti metodologici.

In primo luogo, le prove sono state condotte in ambiente locale controllato (Hardhat localhost), con tempi di mining e condizioni di rete non rappresentativi di testnet o reti pubbliche. Tale configurazione riduce la variabilità esterna ma può sottostimare latenze, congestione e modalità di errore tipiche di ambienti distribuiti reali.

In secondo luogo, il dataset utilizzato, pur ampio (10.000 TD), è sintetico e caratterizzato da distribuzioni più regolari rispetto a cataloghi eterogenei reali. Di conseguenza, le prestazioni e la selettività del matching potrebbero differire in presenza di metadati più complessi, di TD non omogenee o di distribuzioni fortemente sbilanciate.

Un ulteriore elemento riguarda la generazione del carico e la misurazione end-to-end. Le metriche E2E e il throughput osservato descrivono il comportamento complessivo nel setup adottato, ma non isolano completamente il contributo dei singoli sottosistemi (ad esempio tempi legati a sottomissione/elaborazione della richiesta e persistenza del match), che possono mascherare eventuali miglioramenti nel solo layer API o nel solo matching.

Inoltre, Gli scenari di benchmark (`1_rpm`, `10_rpm`, `30_rpm`) sono stati eseguiti in un numero limitato di run nel setup adottato; di conseguenza, le metriche riportate non includono intervalli di confidenza. I valori presentati (medie/massimi o percentili, a seconda della metrica) sono quindi da interpretare come evidenze empiriche nel contesto sperimentale considerato. Una caratterizzazione statistica più robusta richiederebbe esecuzioni ripetute per ciascuno scenario, la stima della varianza tra run indipendenti e la costruzione di intervalli di confidenza (ad esempio al 95%) per confronti quantitativi più solidi.

Infine, l'analisi della latenza API e delle altre metriche è basata su un campione finito di richieste e su un numero limitato di round per scenario. Sebbene le evidenze risultino stabili e indicative, una caratterizzazione statistica più robusta richiederebbe ripetizioni multiple per scenario, intervalli di confidenza e analisi della varianza tra run indipendenti.

Per chiarezza, le principali minacce alla validità possono essere riassunte come segue:

- **Ambiente di esecuzione**: setup locale Hardhat non comparabile a reti pubbliche (latenza, congestione, gas, comportamento dei nodi).
- **Dati in ingresso**: dataset sintetico, potenzialmente diverso da TD reali per eterogeneità, complessità e distribuzione dei tipi.

- **Misurazione end-to-end**: throughput e latenza influenzati dall'intera pipeline, che può mascherare i contributi del solo matching o del layer API.
- **Campionamento e replicabilità**: numero finito di richieste e round per fase; sono necessari più run e analisi statistica per generalizzare.

Nel complesso, i risultati restano utili per una valutazione comparativa interna del prototipo e per identificare i colli di bottiglia principali, ma devono essere letti come evidenze preliminari in vista di una validazione estesa in ambienti più realistici.

Conclusioni

Questo lavoro ha presentato la progettazione e l'implementazione di un indexer semantico per il Web of Things nel contesto blockchain-oriented dell'architettura ZONIA. L'indexer acquisisce Thing Descriptions, esegue il matching rispetto alle richieste generate on-chain e rende disponibili i risultati tramite un endpoint REST interrogabile dagli oracoli a partire dall'identificativo della richiesta.

I contributi principali possono essere sintetizzati nei seguenti punti:

- progettazione di un'architettura modulare che separa persistenza delle TD, matching e integrazione con la componente blockchain;
- implementazione della persistenza su PostgreSQL con memorizzazione delle TD in formato `jsonb` (`td_store`) e registrazione degli esiti di matching su `td_matches`;
- integrazione con ambiente Ethereum locale (Hardhat) tramite ethers.js per la generazione delle richieste e l'ascolto degli eventi;
- esposizione di un'API volutamente minimale (`GET /data/:requestId`), coerente con il caso d'uso in cui gli oracoli conoscono già il `requestId` da componente on-chain, con retrieval da cache e fallback su database;
- valutazione sperimentale su dataset sintetico di 10.000 TD e scenari di carico a 1, 10 e 30 richieste/minuto, includendo test API-only con concorrenza controllata e misure end-to-end.

Sul piano sperimentale, i risultati confermano che il layer REST mantiene stabilità funzionale sotto concorrenza e tempi di risposta contenuti nel setup adottato, mentre le metriche end-to-end evidenziano un comportamento di saturazione della pipeline complessiva all'aumentare del carico. In altri termini, i limiti osservati non sono imputabili principalmente al serving HTTP, ma alla catena end-to-end che collega la generazione della richiesta, la sua gestione e la persistenza dei match.

Sviluppi futuri

Le principali direzioni di evoluzione includono:

- **Scalabilità della pipeline end-to-end**: riduzione delle operazioni critiche e strategie di coordinamento (es. batching o riduzione dei passaggi necessari) per aumentare il throughput in presenza di carichi elevati;
- **Ranking e selezione**: criteri di ordinamento tra più TD compatibili e introduzione di metriche di qualità/affidabilità, utili quando il matching produce insiemi ampi di risultati;
- **Ottimizzazione della persistenza**: indici mirati su `td_matches` e, ove necessario, ottimizzazioni sulle query `jsonb` per scalare a cataloghi più grandi;
- **Ingestion da sorgenti WoT reali**: acquisizione automatizzata delle TD da endpoint/dispositivi e gestione degli aggiornamenti nel tempo;
- **Estensioni di perimetro**: supporto a vincoli geospaziali e scenari multi-indexer con meccanismi di replica e tolleranza ai guasti.

In conclusione, la tesi fornisce un'implementazione concreta e una valutazione riproducibile di un indexer semantico nel contesto ZONIA, evidenziando sia la fattibilità dell'approccio sia i punti su cui intervenire per un impiego più scalabile e realistico.

Bibliografia

[1] L. Gigli, I. Zyrianoff, F. Montori, L. Sciullo, C. Kamienski, M. Di Felice, *ZONIA: A Zero-Trust Oracle System for Blockchain IoT Applications*, IEEE Xplore, document ID 11087691. Disponibile: `https://ieeexplore.ieee.org/abstract/document/11087691`,

[2] Ethereum.org, *Oracles*, Disponibile: `https://ethereum.org/developers/docs/oracles/`,

[3] W3C Web of Things Working Group, *Web of Things (WoT) Architecture 1.1*, W3C Recommendation, 2023. Disponibile: `https://www.w3.org/TR/wot-architecture11/`

[4] W3C Web of Things Working Group, *Web of Things (WoT) Discovery*, W3C Working Draft. Disponibile: `https://www.w3.org/TR/wot-discovery/`,

[5] W3C Web of Things Working Group, *Web of Things (WoT) Thing Description 1.1*, W3C Recommendation, 2023. Disponibile: `https://www.w3.org/TR/wot-thing-description11/`

[6] C. Aguzzi, L. Gigli, I. Zyrianoff, L. Roffia, *ZION: A Scalable W3C Web of Things Directory*, in *Proceedings of the 2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, pp. 1–6, 2024. Disponibile: `https://doi.org/10.1109/CCNC51664.2024.10454685`

[7] Ethereum.org, *Decentralized applications (dapps)*, Disponibile: `https://ethereum.org/developers/docs/dapps/`, ultimo accesso: 11/03/2026.

[8] A. Pasdar, Y.-C. Lee, Z. Dong, *Connect API with Blockchain: A Survey on Blockchain Oracle Implementation*, ACM Computing Surveys, vol. 55, no. 10, pp. 1–39, 2023. doi: `10.1145/3567582`. Disponibile: `https://doi.org/10.1145/3567582`

[9] S. Ellis, A. Juels, S. Nazarov, *Chainlink: A Decentralized Oracle Network*, Whitepaper, 2017. Disponibile: `https://chain.link/whitepaper`

[10] G. Wood, *Ethereum: A Secure Decentralised Generalised Transaction Ledger*, Ethereum Yellow Paper, 2014–. Disponibile: `https://ethereum.github.io/yellowpaper/paper.pdf`

[11] ethers.js contributors, *ethers.js Documentation (v6)*, 2024–2026. Disponibile: `https://docs.ethers.org/v6/`

[12] Nomic Foundation, *Hardhat Documentation*, 2024–2026. Disponibile: `https://hardhat.org/docs`

[13] The PostgreSQL Global Development Group, *PostgreSQL Documentation*, 2024–2026. Disponibile: `https://www.postgresql.org/docs/`

[14] OpenJS Foundation, *Node.js Documentation (API Reference)*, disponibile online: `https://nodejs.org/docs/latest/api/`,

[15] Ecma International, *ECMAScript® 2025 Language Specification (ECMA-262)*, Disponibile: `https://tc39.es/ecma262/2025/`.

[16] J. Park, H. Kim, G. Kim, and J. Ryou, *Smart Contract Data Feed Framework for Privacy-Preserving Oracle System on Blockchain*, Computers, vol. 10, no. 1, p. 7, 2021. Disponibile: `https://doi.org/10.3390/computers10010007`

Ringraziamenti

Desidero esprimere la mia sincera gratitudine al mio relatore, Dott. *Federico Montori*, per i suggerimenti e il supporto forniti durante lo sviluppo della tesi.

Ringrazio inoltre il correlatore, Dott. *Lorenzo Gigli*, per la disponibilità, i consigli e l'attenzione con cui ha seguito questo lavoro, guidandomi nella trasformazione di un'idea progettuale in un risultato concreto.

Infine, ringrazio i colleghi e le persone incontrate durante il percorso universitario per il confronto e l'aiuto reciproco, che hanno contribuito alla mia crescita personale e professionale.