DIPARTIMENTO DI INGEGNERIA ELETTRICA, ELETTRONICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN: INGEGNERIA INFORMATICA

INTERNET OF THINGS
BASED SMART SYSTEMS

Urban Monitoring Multi-Agent System

Documentazione

STUDENTI:
STEFANO CARAMAGNO
FEDERICO CALABRESE

DOCENTI:
PROF. VINCENZO CATANIA
PROF. MAURIZIO PALESI

ANNO ACCADEMICO 2025-2026

SOMMARIO

1. INTRODUZIONE

1.1. OBIETTIVI DEL SISTEMA

L'obiettivo principale del sistema è la realizzazione di una piattaforma per il **monitoraggio urbano multi-quartiere** capace di:

- raccogliere in tempo reale eventi provenienti da sensori distribuiti sul territorio (o da simulatori);
- individuare situazioni *critiche* in termini di traffico, inquinamento e, in prospettiva, altri fenomeni urbani;
- supportare un **processo decisionale automatico** che includa azioni locali, meccanismi di escalation e coordinamento tra quartieri;
- fornire al personale operativo una **vista unificata e tracciabile** degli eventi e delle azioni intraprese.

In particolare, il sistema persegue i seguenti obiettivi specifici:

- **Integrazione di IoT e sistemi multi-agente**
  Combinare una infrastruttura di comunicazione *publish/subscribe* basata su MQTT con un **sistema multi-agente** (MAS) che modella quartieri urbani e un coordinatore centrale, ciascuno con una prospettiva e responsabilità specifiche.
- **Automazione delle decisioni operative**
  Implementare logiche automatiche che, a partire dal flusso di eventi, consentano di classificare la severità delle situazioni, decidere se attivare un'escalation e definire azioni di mitigazione o coordinamento.
- **Uso di un LLM a supporto del decision making**
  Integrare un **Large Language Model (LLM)** come componente di supporto alle decisioni, sia per la valutazione della criticità di un singolo evento, sia per la **pianificazione di azioni coordinate** tra quartieri sulla base di uno stato urbano sintetico.
- **Tracciabilità completa del ciclo di vita degli eventi**
  Registrare in modo strutturato gli eventi urbani e le azioni intraprese dal MAS, così da consentire audit, analisi successive e ricostruzione puntuale delle motivazioni alla base delle decisioni.
- **Osservabilità e fruibilità per l'operatore umano**
  Mettere a disposizione un **cruscotto web** che consenta all'operatore di consultare eventi e azioni, filtrare per criteri rilevanti (quartiere, severità, tipo di sensore) e comprendere, tramite metadati e motivazioni, il comportamento complessivo del sistema.

1.2. DESCRIZIONE DEL DOMINIO DI MONITORAGGIO URBANO

Il dominio applicativo considerato è quello di una **città suddivisa in quartieri** (o distretti), ognuno caratterizzato da una propria dinamica di traffico, qualità dell'aria e potenziali ulteriori fenomeni di interesse per la gestione urbana.

In questo scenario:

- ogni **quartiere** è dotato di una serie di sensori (fisici o simulati) che generano eventi relativi allo stato dell'ambiente urbano;
- gli eventi vengono trasmessi in maniera continuativa tramite **messaggistica IoT** basata su MQTT, utilizzando topic strutturati che codificano quartiere e tipologia di sensore;
- i dati raccolti rappresentano misure puntuali, ad esempio:
    - *livello di traffico veicolare* in un determinato segmento stradale
    - *concentrazione di inquinanti* (ad esempio particolato o NO_2) in una determinata area.

Il sistema modella due classi principali di attori:

- l'**operatore della città**, che monitora la situazione urbana attraverso una dashboard e, se necessario, interviene sulla base delle indicazioni del sistema;
- un insieme di **agenti software**, che incarnano la logica decisionale:
    - *agenti di quartiere* (DistrictMonitoringAgent), responsabili della visione locale e della gestione degli eventi associati al proprio distretto;
    - un *agente coordinatore di città* (CityCoordinatorAgent), responsabile della visione globale e del coordinamento tra quartieri.

La gestione degli eventi va oltre il semplice logging:

- per eventi **non critici**, il sistema si occupa della loro registrazione persistente e della messa a disposizione per la consultazione e l'analisi storica;
- per eventi **potenzialmente critici** (ad esempio picchi di traffico o di inquinamento), il sistema:
    - valuta la severità della situazione e la necessità di un'escalation,
    - può coinvolgere un **LLM** per affinare la valutazione della criticità e proporre strategie di intervento,
    - può pianificare **azioni coordinate tra quartieri**, come deviazioni del traffico da un'area congestionata verso distretti limitrofi meno carichi o l'attivazione di allerte su più zone in caso di peggioramento della qualità dell'aria.

Il sistema si comporta quindi come una **cabina di regia digitale**, in cui sensori, agenti software e operatori umani vengono messi in relazione, consentendo di passare dalla *rilevazione* di un evento alla *decisione* e all'eventuale *attuazione* delle azioni corrispondenti in modo strutturato e tracciabile.

1.3. REQUISITI FUNZIONALI

I requisiti funzionali descrivono il comportamento atteso del sistema e i servizi offerti ai diversi attori coinvolti.

Ingestione e gestione degli eventi sensoristici

- Il sistema deve ricevere eventi generati da sensori (o simulatori) tramite protocollo **MQTT**, utilizzando topic che codifichino quartiere e tipo di sensore.
- Ogni evento deve includere almeno: *quartiere*, *tipo di sensore*, *valore misurato*, *unità di misura*, *severità stimata*, *timestamp* e *topic di origine*.
- Gli eventi ricevuti devono essere **normalizzati** in un modello dati interno uniforme per la successiva elaborazione da parte del MAS e del backend.

Monitoraggio locale tramite agenti di quartiere

- Per ciascun quartiere deve essere presente un **agente di quartiere** incaricato di processare gli eventi associati al proprio distretto.
- L'agente di quartiere deve **valutare la severità** degli eventi, distinguendo tra situazioni ordinarie e situazioni potenzialmente critiche.
- Tutti gli eventi, indipendentemente dal livello di criticità, devono essere **registrati persistentemente** in un database.
- L'agente di quartiere deve mantenere un **contesto locale** (ad esempio una finestra temporale di eventi recenti) per supportare decisioni più informate.

Escalation e coordinamento a livello di città

- Quando un evento viene classificato come potenzialmente critico, l'agente di quartiere deve valutare la necessità di un'**escalation** verso il coordinatore di città.
- In caso di escalation, l'agente di quartiere deve inviare al **CityCoordinatorAgent** un messaggio strutturato che contenga il riassunto dell'evento critico, la motivazione dell'escalation e le informazioni contestuali rilevanti.
- Il coordinatore deve mantenere una **visione globale** dello stato della città, aggregando indicatori sintetici per ciascun quartiere.
- Il coordinatore deve generare un **piano di coordinamento multi-quartiere**, identificando i distretti coinvolti, le azioni suggerite per ciascuno di essi e le relative motivazioni.

Uso del LLM nel ciclo decisionale

- Il sistema deve integrare un **LLM** tramite un gateway applicativo esposto via API REST.
- L'agente di quartiere deve poter richiedere al LLM una **valutazione della necessità di escalation** per un evento, tenendo conto del contesto locale recente.
- Il coordinatore deve poter richiedere al LLM un **piano di coordinamento** tra quartieri, basato su una rappresentazione sintetica dello stato urbano.
- Le interazioni con il LLM devono usare **formati JSON strutturati** per input e output, così da consentire validazione, parsing controllato e gestione degli errori.

Persistenza e consultazione di eventi e azioni

- Il sistema deve **persistire** tutti gli eventi in una tabella dedicata, includendo metadati e timestamp di registrazione.
- Il sistema deve **registrare** le azioni di coordinamento e le decisioni del MAS in una tabella specifica, includendo quartiere sorgente, quartiere target, tipo di azione, motivazione e snapshot dell'evento che ha originato l'azione.
- Deve essere possibile **interrogare** il database per filtrare eventi e azioni secondo criteri come quartiere, tipo di sensore, severità, tipologia di azione e intervallo temporale.

Interfaccia web per l'operatore

- Il sistema deve fornire un'**interfaccia web** accessibile via browser.
- La dashboard deve mostrare un elenco degli eventi recenti, un elenco delle azioni intraprese e informazioni sintetiche sullo stato dei quartieri.
- L'operatore deve poter **filtrare e ordinare** le informazioni in base a criteri rilevanti (quartiere, severità, tipo di sensore, tempo).
- La dashboard deve consentire di **correlare eventi e azioni**, facilitando la comprensione del comportamento del sistema e delle decisioni prese.

1.4. REQUISITI NON FUNZIONALI

I requisiti non funzionali definiscono le caratteristiche qualitative che il sistema deve garantire, indipendentemente dalle specifiche funzionalità implementate.

Modularità e separazione delle responsabilità

- L'architettura deve essere **modulare**, con componenti separati per simulazione dei sensori, broker MQTT, sistema multi-agente, backend web e gateway LLM.
- Ogni componente deve avere **responsabilità chiaramente definite**, riducendo le dipendenze incrociate e facilitando la sostituzione o l'evoluzione indipendente dei moduli.

Estensibilità

- Il sistema deve poter essere **esteso** con nuovi tipi di sensori, nuovi quartieri e nuove tipologie di azioni senza modificare la struttura globale dell'architettura.
- L'introduzione di nuovi attributi negli eventi o nelle azioni deve richiedere modifiche limitate ai modelli dati e alle API, preservando quanto più possibile la compatibilità con i componenti esistenti.

Robustezza e tolleranza ai guasti

- Il sistema deve essere in grado di **continuare a funzionare** anche in presenza di malfunzionamenti del LLM o del relativo gateway, adottando **politiche di fallback** deterministiche.

- In caso di errore nella comunicazione con il LLM (timeout, risposta non valida, eccezioni), il MAS deve registrare l'anomalia nei log e applicare regole interne conservative per decidere escalation e piani di coordinamento.
- Il malfunzionamento di un singolo componente non deve compromettere la capacità del sistema di ricevere e registrare eventi o di consultare lo storico già persistito.

Osservabilità e tracciabilità

- Il sistema deve produrre **log significativi** per le principali operazioni: ingestione di eventi, decisioni degli agenti, chiamate al LLM, errori e fallback.
- La combinazione di log applicativi e dati persistiti deve permettere di **ricostruire a posteriori** quali eventi hanno portato a determinate azioni e quali decisioni sono state influenzate dal LLM o dai meccanismi di fallback.
- Tramite la dashboard l'operatore deve poter osservare una vista aggregata dello stato recente del sistema.

Portabilità e facilità di deployment

- Il sistema deve essere eseguibile in modo **riproducibile** tramite Docker Compose, riducendo al minimo la configurazione manuale.
- Le dipendenze esterne, in particolare il modello LLM, devono essere configurabili tramite variabili di ambiente, rendendo la piattaforma portabile su host differenti.

Manutenibilità

- Il codice deve essere organizzato in **moduli coerenti**, con separazione chiara tra configurazione, logica applicativa, accesso ai dati e integrazione con servizi esterni.
- La presenza di diagrammi architetturali, di sequenza e di schema dati deve facilitare la comprensione del sistema e il lavoro di manutenzione da parte di altri sviluppatori.

Sicurezza di base

- La superficie esposta dal sistema deve essere limitata agli endpoint strettamente necessari (API interne e dashboard), riducendo potenziali vettori di attacco.
- Le comunicazioni tra i micro-servizi devono essere concepite in modo da poter essere protette (ad esempio tramite reti interne e protocolli sicuri) in eventuali deployment futuri, mantenendo l'architettura **compatibile con l'introduzione di meccanismi di sicurezza più avanzati**.

2. ANALISI DEL PROBLEMA E DEI REQUISITI

2.1. SCENARIO URBANO DI RIFERIMENTO E TIPOLOGIE DI EVENTI

Lo scenario di riferimento è quello di una **città suddivisa in quartieri** (o distretti), caratterizzati da dinamiche eterogenee di traffico, qualità dell'aria e, più in generale, di pressione sulle infrastrutture urbane. Ogni quartiere costituisce un'unità logica di osservazione e intervento, pur rimanendo interdipendente rispetto agli altri distretti.

In questo contesto, il sistema deve essere in grado di gestire in modo omogeneo diverse **tipologie di eventi urbani**, originati da sensori fisici o simulatori. A titolo esemplificativo, si considerano:

- eventi di **traffico**:
    - flussi veicolari su determinati assi stradali;
    - situazioni di congestionamento o rallentamento significativo;
    - anomalie locali (incidenti, blocchi temporanei, deviazioni impreviste);
- eventi di **inquinamento atmosferico**:
    - misure puntuali di concentrazione di inquinanti (es. particolato, NO_2);
    - superamento di soglie di attenzione o di allarme;
    - variazioni rapide che suggeriscono fenomeni emergenti su scala di quartiere o inter-quartiere.

Ogni evento viene concettualmente descritto da un insieme di attributi fondamentali: *quartiere di riferimento*, *tipo di sensore*, *valore misurato*, *unità di misura*, *severità associata*, *timestamp*, oltre a metadati di origine (ad esempio il *topic* MQTT da cui è stato ricevuto).

La città viene osservata come un **sistema dinamico**, nel quale:

- eventi isolati a bassa severità rappresentano il **rumore di fondo** del funzionamento ordinario;
- sequenze di eventi di severità crescente o aggregata possono indicare **situazioni critiche locali**;
- pattern distribuiti di eventi in quartieri diversi possono suggerire la presenza di **fenomeni globali**, che richiedono azioni coordinate (ad esempio la ridistribuzione dei flussi di traffico o l'attivazione di piani congiunti di mitigazione dell'inquinamento).

L'analisi del problema richiede quindi di interpretare gli eventi non solo come misure puntuali, ma come **segnali informativi** inseriti in un contesto spaziale (quartieri) e temporale (evoluzione nel tempo), con potenziali effetti a catena tra più distretti.

2.2. ATTORI E RUOLI PRINCIPALI

Il sistema coinvolge diversi attori, sia **umani** sia **software**, che interagiscono con il dominio urbano secondo ruoli complementari. A livello concettuale, si distinguono:

- l'**operatore della città**, responsabile del monitoraggio e dell'eventuale intervento umano;
- i **sensori** (fisici o simulati), che costituiscono la sorgente primaria di informazioni sullo stato del territorio;
- il **sistema multi-agente**, composto da agenti di quartiere e da un coordinatore di città, che realizza la logica di interpretazione, escalation e coordinamento.

Questi attori condividono un unico obiettivo di alto livello: trasformare un flusso di dati grezzi provenienti dai sensori in **decisioni operative coerenti, tracciabili e comprensibili**.

2.2.1. OPERATORE DELLA CITTA'

L'**operatore della città** rappresenta il punto di contatto tra il sistema automatizzato e l'organizzazione che gestisce la rete urbana. Il suo ruolo principale consiste nel:

- monitorare la situazione in tempo quasi reale tramite una **dashboard web**, che espone:
    - eventi recenti,
    - azioni intraprese dal sistema,
    - indicazioni sintetiche sulla severità e sulla distribuzione spaziale dei fenomeni;
- utilizzare le informazioni fornite dal sistema per:
    - validare o integrare le decisioni automatiche,
    - adottare ulteriori misure manuali (ad esempio contattare altre unità operative, attivare protocolli esterni, aggiornare piani di emergenza);
- verificare ex-post la **coerenza delle decisioni** prese dagli agenti e l'allineamento con le linee guida operative dell'ente gestore.

Dal punto di vista dei requisiti informativi, l'operatore necessita di:

- una vista **aggregata e filtrabile** sugli eventi (per quartiere, severità, tipologia);
- una vista sulle **azioni di coordinamento** applicate ai vari quartieri, con motivazioni e riferimenti all'evento sorgente;
- la possibilità di ricostruire **catene causa-effetto** tra eventi e azioni, anche a posteriori.

2.2.2. SENSORI FISICI E SIMULATORI DI SENSORI

I **sensori** costituiscono le sorgenti primarie di informazione e possono esistere in due forme:

- sensori **fisici**, distribuiti sul territorio e connessi alla rete;
- **simulatori di sensori**, utilizzati per:
    - testare il sistema in assenza di infrastrutture hardware estese;
    - riprodurre scenari critici e casi di studio;
    - esercitare le funzioni di escalation e coordinamento in situazioni controllate.

Indipendentemente dalla loro natura, i sensori sono responsabili di:

- effettuare misure periodiche o event-driven (ad esempio al superamento di soglie predefinite);
- inviare gli eventi tramite **canali di comunicazione IoT** standardizzati, in questo caso MQTT, seguendo una struttura di topic che codifica almeno:
    - il contesto geografico (*quartiere*),
    - la tipologia di grandezza monitorata (*traffic*, *pollution*, …).

I sensori non implementano logica decisionale: operano come **produttori di dati**, delegando a livelli superiori (MAS, LLM, operatori umani) l'interpretazione degli eventi e la conseguente azione.

2.2.3. AGENTI DI QUARTIERE E COORDINATORE DI CITTA'

Il **sistema multi-agente** rappresenta il nucleo logico del sistema e si articola in due classi di agenti:

- **agenti di quartiere** (DistrictMonitoringAgent);
- **agente coordinatore di città** (CityCoordinatorAgent).

Gli **agenti di quartiere** incarnano la visione locale e sono responsabili di:

- ricevere gli eventi relativi al proprio distretto;
- mantenerne una **storia recente** per costruire il contesto locale (trend e pattern temporali);
- valutare la severità degli eventi, distinguendo tra:
    - situazioni ordinarie gestibili localmente,
    - situazioni potenzialmente critiche che possono richiedere escalation;
- registrare gli eventi nel sistema di persistenza;
- innescare, ove necessario, un **processo di escalation** verso il coordinatore, arricchendo la richiesta con informazioni contestuali e motivazioni.

L’**agente coordinatore di città** assume una prospettiva globale ed è responsabile di:

- aggregare lo stato dei diversi quartieri in una **vista sintetica della città**;
- valutare, alla luce di questa vista, gli impatti sistemici degli eventi critici;
- costruire e aggiornare un **piano di coordinamento multi-quartiere**, che identifichi:
    - i distretti coinvolti,
    - le azioni da applicare in ciascuno di essi,
    - le dipendenze e le possibili ricadute tra le aree urbane;
- comunicare agli agenti di quartiere le **azioni da eseguire** sotto forma di comandi di coordinamento.

Gli agenti scambiano messaggi strutturati, che modellano la **cooperazione** tra distretti e la progressione dal livello locale (quartiere) al livello globale (città) e viceversa.

2.3. REQUISITI DI MONITORAGGIO, NOTIFICA, ATTUAZIONE ED ESCLATION

L’analisi del problema evidenzia che il sistema deve coprire l’intero ciclo operativo che va dalla **rilevazione di un evento** alla **attuazione di azioni** e, se necessario, all’**escalation** verso livelli superiori di decisione.

Dal punto di vista del **monitoraggio**, sono richiesti:

- capacità di ingestione continua di eventi da più sorgenti eterogenee (sensori e simulatori);
- normalizzazione degli eventi in un modello comune, indipendente dalle specificità dei singoli sensori;
- possibilità di definire e aggiornare **soglie di attenzione** e criteri di severità per le diverse tipologie di fenomeno (traffico, inquinamento, ecc.).

Per quanto riguarda la **notifica**, il sistema deve:

- rendere disponibili agli agenti e all’operatore informazioni tempestive su eventi critici o in via di diventare tali;
- propagare al coordinatore le richieste di escalation in modo affidabile e strutturato;
- mettere a disposizione della dashboard informazioni facilmente interpretabili, in termini sia di **stato corrente** sia di **storico recente**.

Sul piano dell’**attuazione**, il problema richiede che:

- le decisioni prese dagli agenti (locali o globali) si traducano in **azioni esplicite**:
    - azioni puramente informative (notifiche, log, aggiornamenti di stato),
    - azioni operative (comandi logici che, in un contesto esteso, potrebbero essere collegati ad attuatori reali);

- le azioni siano **tracciate** con il relativo contesto:
    - evento sorgente,
    - quartiere sorgente e target,
    - motivazione, inclusa l'eventuale influenza del LLM.

Infine, relativamente alla **escalation**, il sistema deve:

- permettere agli agenti di quartiere di riconoscere automaticamente condizioni in cui la sola gestione locale risulta insufficiente;
- supportare una valutazione della criticità che può includere:
    - logiche deterministiche interne,
    - il contributo di un componente di **decision support** basato su LLM;
- prevedere meccanismi per evitare escalation superflue, ad esempio attraverso una normalizzazione della severità che tenga conto del contesto complessivo.

2.4. REQUISITI DI MONITORAGGIO, NOTIFICA, ATTUAZIONE ED ESCLATION

Il dominio urbano è caratterizzato da **interdipendenze significative** tra quartieri: un intervento su un distretto può avere impatti positivi o negativi sui distretti confinanti o funzionalmente collegati (ad esempio tramite assi viari principali o corridoi di ventilazione).

I requisiti di cooperazione tra quartieri possono essere così articolati:

- necessità di una **visione globale** dello stato cittadino:
    - aggregazione di indicatori per quartiere,
    - classificazione dello stato di ciascun distretto (es. normale, sotto pressione, critico);
- capacità di elaborare **piani di coordinamento** che:
    - definiscano azioni coerenti su più quartieri,
    - minimizzino gli effetti collaterali indesiderati (es. spostamento del traffico da un'area critica a un'altra già congestionata);
- gestione di **priorità** e **trade-off**:
    - valutazione dell'impatto di un'azione su diversi quartieri,
    - selezione di strategie che bilancino gli interessi locali con quelli globali.

Dal punto di vista del sistema multi-agente, questo si traduce nella necessità di:

- disporre di un **protocollo di comunicazione interno** che consenta agli agenti di quartiere di:
    - segnalare situazioni locali di criticità,
    - ricevere comandi di coordinamento dal livello centrale;

- definire un ruolo chiaro per il **coordinatore**, che agisca come:
    - punto di raccolta delle escalation,
    - motore di calcolo dei piani multi-quartiere,
    - sorgente di comandi verso i distretti coinvolti.

La cooperazione tra quartieri è quindi un requisito strutturale: non si tratta di una semplice aggregazione di logiche locali, ma di un vero e proprio **comportamento collettivo** guidato da regole e piani condivisi.

2.5. REQUISITI DI OSSERVABILITA', TRACCIABILITA' E PERSISTENZA

Un sistema di monitoraggio urbano che coinvolge componenti autonomi e un supporto decisionale basato su LLM deve essere progettato per garantire **osservabilità**, **tracciabilità** e **persistenza** in modo sistematico.

Sul versante dell'**osservabilità**:

- è necessario che l'evoluzione del sistema sia visibile sia a livello tecnico sia a livello operativo;
- log e metriche devono permettere di comprendere:
    - quali eventi sono stati ricevuti,
    - quali decisioni sono state prese dagli agenti,
    - quali chiamate sono state effettuate verso il LLM e con quali esiti.

Per quanto riguarda la **tracciabilità**:

- deve essere possibile correlare ciascuna **azione** a:
    - uno o più **eventi** sorgenti,
    - le decisioni intermedie del MAS e, ove rilevante, del LLM;
- le strutture dati devono includere riferimenti espliciti (come snapshot di eventi e motivazioni testuali o codificate) che permettano di ricostruire la **catena decisionale**;
- le scelte dell'LLM devono essere rese ispezionabili almeno attraverso:
    - la conservazione di output strutturati,
    - l'indicazione sintetica delle motivazioni o delle raccomandazioni fornite.

Infine, la **persistenza** svolge un ruolo centrale:

- gli eventi devono essere conservati in una base dati che supporti:
    - query per quartiere, tipo di sensore, severità, intervallo temporale;
    - analisi successive, ad esempio per la validazione dei modelli decisionali o per studi statistici;

- le azioni devono essere archiviate con sufficiente dettaglio da consentire:
    - audit operativi,
    - analisi dell’efficacia delle strategie applicate,
    - confronto tra comportamenti con e senza il contributo del LLM.

Questi requisiti impongono una modellazione dei dati e dei flussi di logging che non sia meramente funzionale alla visualizzazione in tempo reale, ma che permetta anche un utilizzo del sistema come **fonte affidabile di evidenza storica**, a supporto di decisioni future e di revisioni delle politiche di gestione urbana.

3. VISIONE D'INSIEME DELL'ARCHITETTURA

3.1. PANORAMICA GENERALE DEL SISTEMA E DEI MICRO-SERVIZI

L'architettura è organizzata come un insieme di **micro-servizi containerizzati** che cooperano all'interno di un unico ambiente orchestrato (rete Docker). Ciascun servizio ha responsabilità ben definite ed è isolato dal punto di vista del deployment, pur partecipando a un flusso applicativo end-to-end coerente.

I componenti principali sono:

- **Simulated Sensors Service (`sims`)**
  Servizio che ospita i *simulatori di sensori* (es. traffico e inquinamento).
  - Genera eventi in formato JSON.
  - Pubblica messaggi su topic MQTT strutturati del tipo `city/{district}/{sensor_type}`.
- **MQTT Broker (`mqtt-broker`)**
  Implementato con **Eclipse Mosquitto**, gestisce la messaggistica *publish/subscribe*:
  - riceve i messaggi pubblicati dai simulatori;
  - distribuisce i messaggi ai subscriber registrati (in particolare il MAS).
- **MAS Core (`mas-core`)**
  Servizio che incapsula il **sistema multi-agente** e le logiche di orchestrazione interna:
  - un *listener MQTT* che si sottoscrive ai topic degli eventi;
  - un *router* che normalizza e distribuisce gli eventi agli agenti di quartiere;
  - uno o più **DistrictMonitoringAgent** (uno per quartiere);
  - un **CityCoordinatorAgent** con visione globale;
  - integrazione con:
    - il backend web (via HTTP/REST) per la persistenza di eventi e azioni;
    - il gateway LLM (via HTTP/REST) per il supporto decisionale.
- **Web Backend (`web-backend`)**
  Servizio che espone:
  - un'API REST per la **persistenza** di eventi (`/api/events`) e azioni (`/api/actions`);
  - una **dashboard web** (template HTML/Jinja2) per l'operatore, accessibile via browser (es. `/dashboard`, `/events`, `/actions`);
  - un database **SQLite** per memorizzare in modo strutturato eventi e azioni del sistema.

- **LLM Gateway (`llm-gateway`)**
  Servizio dedicato all'integrazione con il **Large Language Model**:
  - espone endpoint REST (es. `/llm/decide_escalation`, `/llm/plan_coordination`);
  - riceve richieste dal MAS (agenti di quartiere e coordinatore);
  - traduce tali richieste in chiamate verso il motore LLM esterno (es. Ollama);
  - valida, normalizza e restituisce risposte JSON in un formato controllato.
- **LLM Engine (host esterno, es. `ollama`)**
  Sistema esterno all'ambiente Docker che esegue il modello linguistico:
  - riceve richieste dal gateway LLM;
  - elabora prompt e contesti JSON;
  - restituisce risposte testuali e strutturate, poi ulteriormente processate dal gateway.

Dal punto di vista dell'utente umano, l'unico punto di accesso diretto è la **dashboard web** del backend; tutti gli altri servizi operano in background per garantire il flusso dati sensori → MAS → LLM → persistenza → visualizzazione.

3.2. SCELTE TECNOLOGICHE PRINCIPALI

La scelta delle tecnologie non è meramente implementativa, ma risponde a requisiti di **modularità**, **osservabilità**, **integrazione con il mondo IoT** e **sperimentazione controllata** di componenti LLM all'interno di un sistema distribuito.

3.2.1. MQTT PER LA MESSAGGISTICA IoT

Per il trasporto degli eventi generati dai sensori è stato adottato **MQTT**, protocollo di messaggistica *publish/subscribe* ampiamente utilizzato in ambito IoT.

Le caratteristiche rilevanti nel contesto del sistema sono:

- **decoupling tra produttori e consumatori**:
  i simulatori pubblicano su topic logici (`city/{district}/{sensor_type}`) senza dover conoscere gli endpoint dei consumatori; il MAS si limita a sottoscrivere i topic di interesse;
- **leggerezza e semplicità**:
  MQTT introduce un overhead minimo, risultando adatto anche a scenari in cui l'infrastruttura fosse estesa a sensori fisici con risorse limitate;
- **modellazione naturale del dominio**:
  la gerarchia dei topic consente di rappresentare in modo diretto:
  - la dimensione geografica (*quartiere*),
  - la dimensione semantica (*tipo di sensore*).

Queste proprietà rendono MQTT una scelta coerente con un sistema che deve poter scalare sia in termini di numero di sensori, sia in termini di flessibilità degli schemi di sottoscrizione.

3.2.2. PYTHON PER MAS, SIMULATORI E SERVIZI WEB

Il linguaggio **Python** è stato scelto come tecnologia unificata per:

- la logica del **sistema multi-agente** (MAS core);
- i **simulatori** di sensori;
- i servizi web (backend e gateway LLM).

Le motivazioni principali sono:

- **rapidità di sviluppo** e chiarezza espressiva, particolarmente utili in contesti dove si vogliono sperimentare rapidamente diverse strategie di decisione e cooperazione tra agenti;
- ampia disponibilità di **librerie per integrazione di servizi**, gestione di HTTP, MQTT, serializzazione JSON, connettività verso database e servizi di terze parti (come gli engine LLM);
- uniformità dello stack applicativo: avere MAS, simulatori, backend e gateway scritti nello stesso linguaggio semplifica notevolmente manutenzione, debugging e test.

L'adozione di Python come linguaggio comune permette inoltre di modellare gli agenti come entità relativamente leggere (thread/processi) integrate in un runtime uniforme, senza introdurre complicazioni derivanti dall'uso di linguaggi eterogenei.

3.2.3. FastAPI PER BACKEND REST E GATEWAY LLM

Per l'esposizione delle API REST è stato adottato **FastAPI**, utilizzato sia nel **backend web** sia nel **LLM Gateway**.

I benefici principali sono:

- **performance e asincronia**:
  FastAPI si basa su ASGI e supporta nativamente la programmazione asincrona, risultando adatto a gestire carichi di I/O intensivi (chiamate HTTP, accessi a DB, invocazioni al LLM) con risorse contenute;
- **integrazione con Pydantic**:
  la definizione di *modelli di richiesta e risposta tipizzati* (Pydantic) è particolarmente importante nel caso del gateway LLM, dove:
  - gli input verso il modello devono essere strutturati e validati;
  - gli output devono essere normalizzati e resi robusti a possibili risposte non conformi;
- **documentazione automatica delle API**:
  la generazione automatica di documentazione (OpenAPI/Swagger) facilita l'esplorazione degli endpoint interni (es. `/api/events`, `/api/actions`,

`/llm/decide_escalation`, `/llm/plan_coordination`), agevolando lo sviluppo e il test.

L'impiego di FastAPI in entrambi i servizi REST riduce le differenze architetturali tra backend e gateway e rende più semplice la gestione del ciclo di vita delle richieste.

3.2.4. SQLite PER LA PERSISTENZA DEI DATI

La persistenza è affidata a **SQLite**, integrata nel backend web tramite un ORM (ad esempio SQLAlchemy).

Le ragioni della scelta sono:

- **leggerezza e assenza di dipendenze server-side**:
  SQLite è un database embedded basato su file, ideale per scenari in cui si desidera:
  - minimizzare la complessità dell'infrastruttura,
  - avere un singolo artefatto dati facilmente esportabile e versionabile;
- **sufficienza rispetto ai volumi previsti**:
  il sistema, nella sua configurazione attuale, non ha requisiti di scalabilità orizzontale estremi; le funzionalità di SQLite coprono pienamente la necessità di:
  - inserire eventi e azioni,
  - eseguire query filtrate per quartiere, severità, tipologia, intervallo temporale;
- **integrazione naturale con strumenti di analisi offline**:
  il file di database può essere facilmente ispezionato, copiato e utilizzato per analisi successive.

L'adozione di SQLite non preclude la possibilità di migrare in futuro verso DBMS più sofisticati (es. PostgreSQL) qualora i requisiti di scalabilità o multi-utenza lo richiedano.

3.2.5. DOCKER COMPOSE PER L'ORCHESTRAZIONE DEI SERVIZI

L'intero sistema è orchestrato tramite **Docker Compose**, che definisce:

- i container corrispondenti ai micro-servizi (`sims`, `mqtt-broker`, `mas-core`, `web-backend`, `llm-gateway`);
- la rete virtuale che consente loro di comunicare tra loro tramite hostname logici;
- le dipendenze di avvio (ad esempio, l'MQTT broker deve essere disponibile prima che il MAS attivi le sottoscrizioni);
- le variabili d'ambiente necessarie alla configurazione (es. URL dell'engine LLM, path del database, porte esposte).

L'utilizzo di Docker Compose garantisce:

- **riproducibilità dell'ambiente di esecuzione**;
- semplificazione delle procedure di avvio e spegnimento del sistema;

- isolamento dei servizi rispetto alla macchina host, con configurazioni controllo del networking interno.

3.3. ARCHITETTURA LOGICA E INTERAZIONI TRA COMPONENTI

Dal punto di vista logico, l'architettura può essere vista come una **pipeline di elaborazione eventi** arricchita da un layer decisionale basato su MAS e LLM.

I principali flussi di interazione sono:

- **Flusso eventi sensori → MAS**
  - I simulatori (servizio `sims`) generano eventi e li pubblicano su topic MQTT: `city/{district}/{sensor_type}`.
  - Il MAS (`mas-core`) si sottoscrive ai topic di interesse (`city/+/+`).
  - Il listener MQTT riceve i messaggi, li passa al *router* che:
    - effettua il parsing del payload,
    - costruisce un oggetto evento interno,
    - instrada l'evento all'agente di quartiere corrispondente.
- **Elaborazione locale negli agenti di quartiere**
  - Ciascun **DistrictMonitoringAgent** aggiorna il proprio contesto locale (es. buffer di eventi recenti).
  - L'agente classifica l'evento (non critico / potenzialmente critico).
  - In ogni caso:
    - effettua una **invocazione HTTP** verso il backend web (`/api/events`) per registrare l'evento;
  - se l'evento risulta potenzialmente critico:
    - valuta la necessità di coinvolgere il LLM per affinare la decisione di escalation.
- **Interazione MAS ↔ LLM Gateway ↔ LLM Engine**
  - L'agente di quartiere può invocare il gateway LLM (`/llm/decide_escalation`) fornendo:
    - evento corrente,
    - contesto locale;
  - il gateway costruisce il prompt e il JSON di input, inoltra la richiesta al **LLM Engine** (es. Ollama) e:
    - riceve una risposta JSON con:
      - severità normalizzata,
      - indicazione di escalation,

        - eventuali motivazioni;
      - valida e restituisce al MAS una risposta strutturata.
  - In modo analogo, il **CityCoordinatorAgent** può invocare il gateway (/llm/plan_coordination) per ottenere un **piano di coordinamento** a partire da una rappresentazione sintetica dello stato urbano.
- **Interazione tra agenti e coordinatore**
  - In caso di evento critico, l'agente di quartiere invia al **coordinatore** un messaggio di `ESCALATION_REQUEST`, contenente:
    - descrizione dell'evento,
    - motivazione (LLM o fallback rule),
    - eventuale contesto.
  - Il coordinatore aggiorna la propria vista globale e:
    - elabora, eventualmente con supporto LLM, un piano multi-quartiere,
    - invia ai singoli agenti di quartiere messaggi di `COORDINATION_COMMAND` che specificano:
      - azioni da applicare,
      - quartieri coinvolti,
      - motivazioni.
- **Persistenza e visualizzazione**
  - Gli agenti di quartiere, una volta applicate le azioni (locali o coordinate), invocano il backend web (`/api/actions`) per registrare le decisioni prese.
  - Il backend salva gli eventi nella tabella `events` e le azioni nella tabella `actions` del database SQLite.
  - L'operatore accede alla **dashboard web**, che:
    - interroga il database,
    - presenta una vista combinata di eventi e azioni,
    - consente filtraggio e navigazione mirata.

Questa architettura logica separa nettamente:

- **strato di sensing e trasporto** (simulatori + MQTT),
- **strato di decisione** (MAS + LLM),
- **strato di persistenza e presentazione** (backend + DB + dashboard), favorendo chiarezza e manutenibilità.

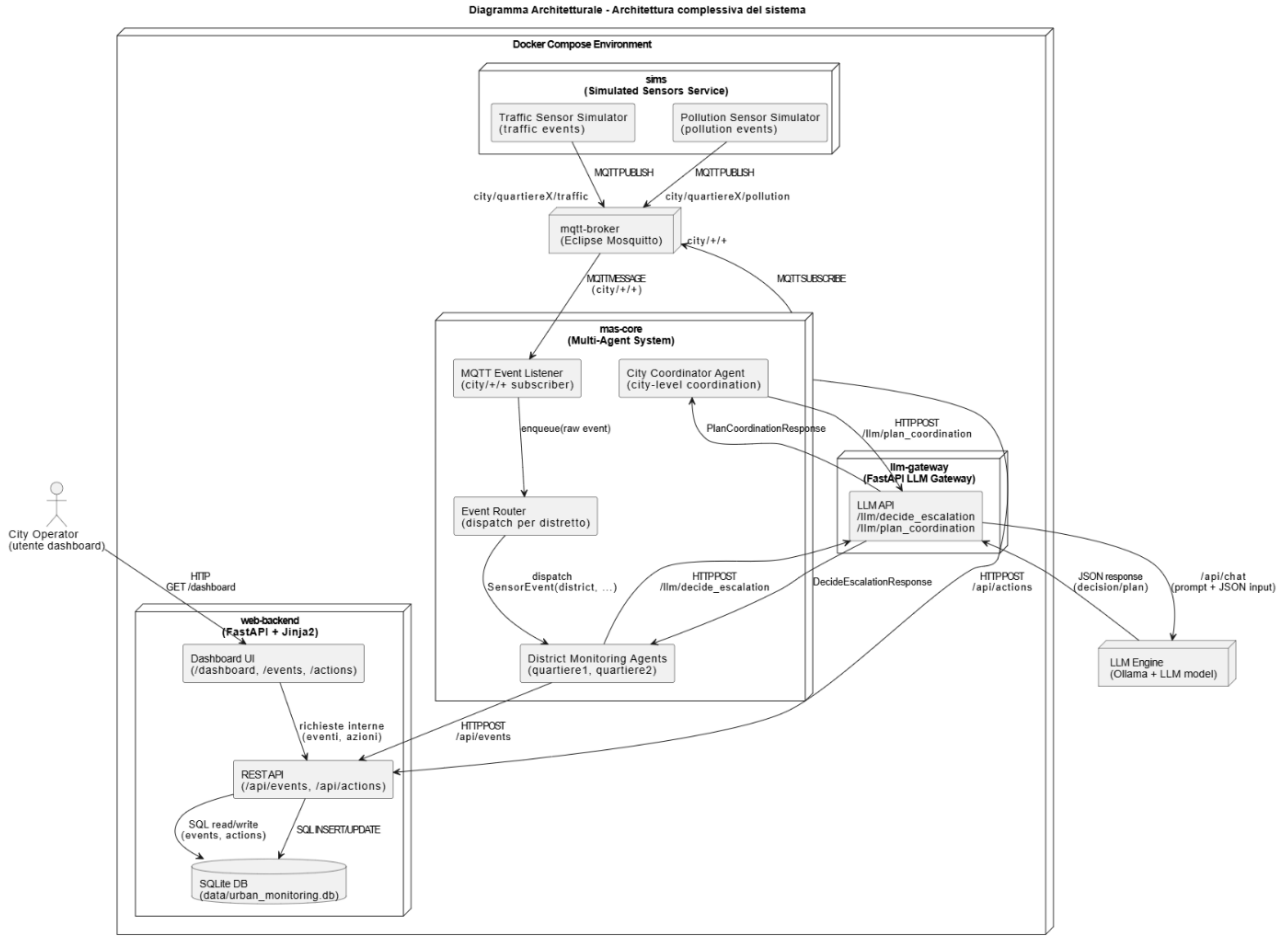
3.4. DIAGRAMMA ARCHITETTURALE – ARCHITETTURA COMPLESSIVA DEL SISTEMA

L'architettura complessiva è sintetizzata in un **diagramma architetturale UML** che rappresenta:

- i micro-servizi principali come nodi distinti:
  - `sims` (simulated sensors service),
  - `mqtt-broker` (Eclipse Mosquitto),
  - `mas-core` (Multi-Agent System),
  - `web-backend` (FastAPI + Jinja2 + SQLite),
  - `llm-gateway` (FastAPI LLM Gateway),
  - `LLM Engine` (Ollama o servizio equivalente esterno);
- il ruolo dell'**operatore umano** come attore esterno che interagisce esclusivamente con il `web-backend` via HTTP/HTTPS;
- i canali di comunicazione tra servizi:
  - **MQTT** tra sims e mas-core (topic city/+/+);
  - **HTTP/REST** tra:
    - `mas-core` e `web-backend` (`/api/events`, `/api/actions`);
    - `mas-core` e `llm-gateway` (`/llm/decide_escalation`, `/llm/plan_coordination`);
    - llm-gateway e LLM Engine (API del motore LLM);
  - **accesso diretto al DB** limitato al solo `web-backend`, che funge da *owner* del modello persistente;
- la rete Docker come contesto esecutivo condiviso per i micro-servizi interni.

Il diagramma mette in evidenza:

- la **centralità del MAS** come fulcro del decision making;
- la **separazione di responsabilità** tra raccolta eventi, ragionamento, persistenza e interfaccia utente;
- il fatto che il **LLM** sia trattato come un *servizio esterno*, protetto da uno strato di adattamento e validazione (LLM Gateway).



3.5. MOTIVAZIONI DELLE SCELTE ARCHITETTURALI

Le scelte architetturali adottate sono guidate da alcuni principi chiave: **separazione dei ruoli**, **riduzione dell'accoppiamento**, **facilità di evoluzione** e **robustezza rispetto a componenti non deterministici** (come il LLM).

Gli elementi centrali possono essere sintetizzati come segue:

- **Micro-servizi e confini chiari**
  La suddivisione in servizi distinti (simulatori, broker, MAS, backend, gateway LLM) consente:
  - di isolare problemi e malfunzionamenti,
  - di evolvere ciascun componente in modo indipendente,
  - di sostituire tecnologie (ad esempio passare da SQLite a un DBMS diverso) senza impattare l'intero sistema.
- **MQTT come backbone per gli eventi**
  L'uso di MQTT evita un accoppiamento diretto sensori → MAS, permettendo:
  - di aggiungere o rimuovere consumer senza modificare i produttori;
  - di introdurre in futuro ulteriori servizi di analisi real-time che si sottoscrivano agli stessi topic.

- **MAS come layer di orchestrazione intelligente**
  La logica multi-agente offre un modello naturale per:
  - rappresentare quartieri e città come entità dotate di *agency* e *responsabilità proprie*;
  - modellare la cooperazione e la negoziazione tra parti locali e globali;
  - integrare in modo controllato un LLM come *consulente* del processo decisionale.
- **Gateway LLM come strato di protezione**
  Inserire un **gateway dedicato** tra MAS e LLM:
  - protegge il resto del sistema da risposte potenzialmente non strutturate o incoerenti del modello;
  - permette di centralizzare:
    - la definizione dei prompt,
    - la validazione degli output,
    - la gestione degli errori e delle politiche di fallback;
  - rende possibile il **cambio di modello o di provider** senza alterare il codice degli agenti.
- **Backend e DB come unica fonte di verità persistente**
  Delegare la persistenza al solo `web-backend`:
  - evita che più servizi modifichino il database in modo non coordinato;
  - consente di concentrare le logiche di validazione e mapping dei dati in un unico strato;
  - offre un punto centrale per l'esposizione dei dati verso l'operatore, assicurando coerenza tra ciò che viene salvato e ciò che viene mostrato.
- **Containerizzazione e orchestration locale**
  L'utilizzo di Docker Compose:
  - assicura che l'intero sistema possa essere eseguito in ambienti diversi con comportamento prevedibile;
  - rende agevole la distribuzione verso altri contesti (test, staging, demo interna), grazie alla definizione dichiarativa delle dipendenze e delle reti.

Queste scelte congiunte mirano a realizzare un sistema che **coniughi sperimentazione avanzata (integrazione LLM, MAS, IoT)** con una struttura architetturale ordinata, documentata e facilmente estendibile.

4. MODELLAZIONE DEI DATI

4.1. REQUISITI INFORMATIVI: EVENTI, AZIONI E TRACCIABILITA'

La modellazione dei dati è guidata da tre esigenze principali:

- rappresentare in modo **strutturato e coerente** gli eventi urbani generati dai sensori;
- formalizzare le **azioni** intraprese dal sistema multi-agente, incluse le motivazioni e il contesto;
- garantire un livello elevato di **tracciabilità** lungo l'intero ciclo di vita dell'informazione, dalla rilevazione all'azione.

Sul piano informativo, gli **eventi urbani** devono conservare:

- il **contesto spaziale**, attraverso il riferimento al *quartiere* in cui è stato rilevato il fenomeno;
- il **contesto semantico**, tramite il *tipo di sensore* (es. traffico, inquinamento);
- il **valore misurato** e la relativa *unità di misura*, necessari sia per l'analisi in tempo reale sia per eventuali elaborazioni successive;
- una **stima di severità**, espressa in forma simbolica (es. `low`, `medium`, `high`), che sintetizza l'impatto potenziale dell'evento;
- un **timestamp** coerente con la sequenza temporale degli eventi;
- metadati tecnici quali il *topic* MQTT di origine, utili per diagnosi e verifiche.

Le **azioni** rappresentano l'output del processo decisionale del sistema e devono catturare:

- la **provenienza** dell'azione (quartiere sorgente o agente che la innesca);
- il **bersaglio** dell'azione (quartiere target, che può coincidere o meno con il sorgente);
- la **tipologia di intervento** (es. ridistribuzione del traffico, incremento del livello di allerta);
- la **motivazione**, che può derivare direttamente dalle regole del MAS oppure dal contributo del LLM;
- un **snapshot dell'evento** che ha generato l'azione, per preservare un vincolo logico anche in assenza di una foreign key esplicita.

Per la **tracciabilità**, è necessario che:

- ogni evento sia persistito e consultabile ex post;
- ogni azione sia collegabile al proprio contesto tramite lo *snapshot* dell'evento e i metadati associati;
- sia possibile correlare eventi e azioni attraverso query sul database, in modo da ricostruire percorsi decisionali e verificare la coerenza del sistema.

La combinazione di queste esigenze porta a una modellazione centrata su due entità principali — **Event** e **Action** — con relazioni concettuali ben definite e campi progettati per supportare query operative e analisi successive.

4.2. MODELLAZIONE LOGICA DELLE ENTITA'

La modellazione logica nel database SQLite è realizzata tramite ORM (SQLAlchemy) e fa emergere due tabelle centrali:

- `events`, che rappresenta il flusso degli eventi urbani;
- `actions`, che rappresenta le decisioni del sistema multi-agente.

Le due entità sono progettate per essere **minimali ma espressive**, evitando ridondanze non necessarie e mantenendo allo stesso tempo la capacità di esprimere relazioni causali tra eventi e azioni.

4.2.1. ENTITA' `Event` – RAPPRESENTAZIONE DEGLI EVENTI URBANI

L'entità `Event` costituisce il modello canonico per gli eventi provenienti dai sensori. A livello logico, la tabella `events` presenta i seguenti attributi principali:

- `id`
  Identificatore univoco dell'evento (`INTEGER`, chiave primaria).
  Viene utilizzato come riferimento interno dal backend, ad esempio per ordinare gli eventi e per identificarli senza ambiguità.
- `district`
  Rappresenta il *quartiere* a cui l'evento si riferisce (`VARCHAR`, indicizzato).
  È tipicamente una stringa simbolica (es. `"quartiere1"`, `"quartiere2"`) che consente:
  - il filtraggio per area geografica nella dashboard;
  - l'aggregazione e l'analisi per distretto da parte del MAS.
- `sensor_type`
  Indica il *tipo di sensore* che ha generato l'evento (`VARCHAR`, indicizzato).
  Valori tipici includono, ad esempio, `"traffic"` e `"pollution"`.
  Questo campo fornisce al sistema:
  - la possibilità di applicare logiche diverse a seconda del tipo di fenomeno;
  - la capacità di analizzare pattern separati per ogni categoria di sensore.
- `value`
  Il *valore misurato* dal sensore (`REAL / FLOAT`).
  È un campo numerico utilizzato per:
  - confronti con soglie,
  - calcolo di indici sintetici,
  - eventuali analisi statistiche.

- `unit`
  L'*unità di misura* del valore (`VARCHAR`).
  Consente di interpretare correttamente il significato di `value` (es. `"vehicles/h"`, `"µg/m³"`), mantenendo la possibilità di gestire sensori eterogenei.
- `severity`
  Una **stima simbolica della severità** (`VARCHAR`, indicizzato).
  Valori esemplificativi: `"low"`, `"medium"`, `"high"`.
  Questo campo consente agli agenti di:
  - ragionare a un livello di astrazione più alto rispetto al solo valore numerico;
  - attivare logiche di escalation o di semplice logging in base a soglie semantiche.
- `timestamp`
  Rappresentazione testuale del momento in cui l'evento è stato generato (`TEXT / STRING`).
  Può contenere un timestamp ISO 8601 o un tempo epoch serializzato.
  È concettualmente distinto da `created_at`, in quanto:
  - `timestamp` rappresenta il *tempo di rilevazione* lato sensore;
  - `created_at` rappresenta il *tempo di inserimento* nel sistema di persistenza.
- `topic`
  Il *topic MQTT* di origine (`VARCHAR`).
  Questo campo mantiene un legame diretto con il livello di trasporto, es. `city/quartiere1/traffic`.
  È utile per:
  - diagnostica (verifica della corretta configurazione dei sensori);
  - analisi sul comportamento dei publisher MQTT.
- `created_at`
  Timestamp di **inserimento nel database** (DATETIME).
  Viene impostato automaticamente al momento della creazione dell'evento dal backend.
  È utilizzato per:
  - ordinare gli eventi in base alla loro comparsa nel sistema,
  - effettuare query per intervallo temporale lato persistente (ad esempio ultimi N minuti/ore).

Questa struttura consente di separare nettamente:

- il **contenuto semantico** dell'evento (district, sensor_type, value, unit, severity, timestamp);
- i **metadati tecnici** (topic, created_at);

rendendo possibile un'analisi sia tecnica sia di dominio sui dati raccolti.

4.2.2. ENTITA' `Action` – RAPPRESENTAZIONE DELLE AZIONI DI COORDINAMENTO

L'entità `Action` modella le **decisioni operative** prese dal sistema multi-agente, sia a livello locale sia a livello di coordinamento cittadino. La tabella `actions` è strutturata nei seguenti attributi principali:

- `id`
  Identificatore univoco dell'azione (`INTEGER`, chiave primaria).
  È utilizzato dal backend per identificare e presentare le azioni in modo ordinato e univoco.
- `source_district`
  Quartiere *sorgente* dell'azione (`VARCHAR`, indicizzato).
  Rappresenta il distretto dal quale ha avuto origine la decisione o il messaggio che ha portato all'azione, tipicamente:
  - il quartiere in cui si è verificato l'evento critico,
  - oppure il quartiere che ha richiesto escalation.
- `target_district`
  Quartiere *target* dell'azione (`VARCHAR`, indicizzato).
  È il distretto sul quale l'azione viene applicata.
  Può coincidere con `source_district` (azione locale) o differirne (azione di coordinamento multi-quartiere).
- `action_type`
  Categoria dell'azione intrapresa (`VARCHAR`, indicizzato).
  Alcuni esempi possibili:
  - `"REDIRECT_TRAFFIC"`,
  - `"INCREASE_ALERT_LEVEL"`,
  - `"LIMIT_ACCESS"`, ecc.

  Questo campo offre:
  - una classificazione sintetica delle strategie adottate,
  - la possibilità di filtrare e aggregare le azioni per tipologia.
- `reason`
  Motivazione sintetica che giustifica l'azione (`VARCHAR`).
  Può contenere:
  - riferimenti al contributo del LLM (es. `"llm: high pollution, suggest reroute"`),
  - riferimenti a regole di fallback deterministiche (es. `"fallback_rule"`),
  - altre descrizioni brevi del criterio decisionale.

  Questo attributo è fondamentale per la **trasparenza** del comportamento del sistema.

- `event_snapshot`
  Snapshot testuale dell'evento che ha generato l'azione (`TEXT`).
  Tipicamente contiene una serializzazione JSON dell'oggetto evento (o di una sua parte significativa).
  Assolve a diverse funzioni:
    - preservare il contesto originario anche in assenza di una foreign key diretta verso `events`;
    - consentire audit e ricostruzioni ex post senza dipendere dall'integrità referenziale del database;
    - permettere analisi indipendenti su eventi e azioni anche se la struttura della tabella `events` evolvesse nel tempo.
- `created_at`
  Timestamp di **registrazione dell'azione** (`DATETIME`).
  Viene impostato automaticamente al momento in cui il backend riceve la richiesta di persistenza dell'azione.
  Garantisce la possibilità di:
    - ricostruire la sequenza temporale delle decisioni,
    - interrogare il sistema per intervalli temporali mirati.

La combinazione di `source_district`, `target_district`, `action_type`, `reason ed event_snapshot` rende l'entità `Action` particolarmente adatta a descrivere la **componente decisionale** del sistema e a fornire una base solida per analisi successive sulle politiche di coordinamento adottate.

4.3. RELAZIONI CONCETTUALI TRA EVENTI E AZIONI

Dal punto di vista concettuale, la relazione tra Event e Action è di tipo **uno-a-molti**:

- un singolo **evento critico** può generare:
    - una o più azioni localizzate nel quartiere d'origine;
    - una o più azioni distribuite su quartieri diversi, se è attivato un piano di coordinamento multi-quartiere;
- una singola **azione** è generalmente riconducibile a **un evento sorgente** principale, anche se, nei casi più complessi, può essere influenzata da un contesto composto da più eventi.

Nel modello dati adottato, tale relazione non è espressa tramite una *foreign key* tradizionale (ad esempio un campo `event_id` nella tabella `actions`), ma viene realizzata attraverso il campo `event_snapshot`:

- lo snapshot cattura una **copia congelata** delle informazioni rilevanti sull'evento al momento della decisione;

- l'assenza di un vincolo referenziale rigido verso `events` consente:
    - di mantenere l'azione semanticamente consistente anche qualora l'evento originale venisse rimosso o il suo schema modificato;
    - di trattare `actions` come registro decisionale *auto-consistente*, dotato di tutte le informazioni necessarie per l'analisi ex post.

Dal punto di vista del **MAS**, la relazione è resa esplicita a livello applicativo:

- gli agenti mantengono un riferimento all'evento corrente in memoria;
- al momento della creazione di un'azione, serializzano l'evento (o una sua vista) in `event_snapshot`;
- il backend utilizza questa informazione per preservare il legame semantico tra evento e azione.

Questa scelta predilige una **tracciabilità logica flessibile** rispetto a una rigidità schematica, in linea con l'obiettivo di poter evolvere il modello dei dati senza compromettere la leggibilità storica delle decisioni.

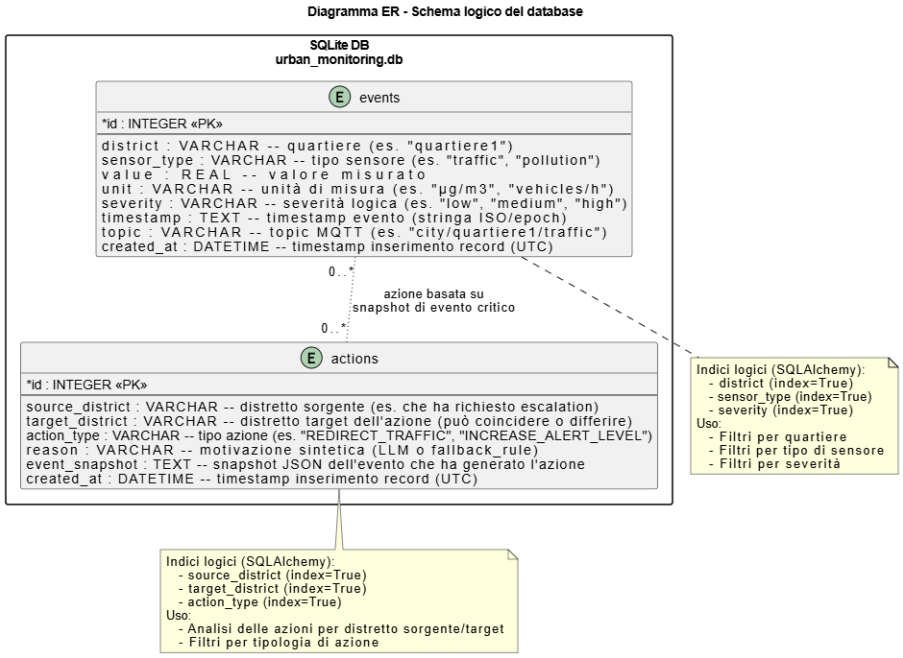
4.4. DIAGRAMMA ER – SCHEMA LOGICO DEL DATABASE

Lo schema logico del database è sintetizzato in un **diagramma ER** che evidenzia:

- le due entità principali:
    - `events`, con attributi relativi a quartiere, tipo di sensore, valore, unità, severità, timestamp di rilevazione, topic e timestamp di creazione;
    - `actions`, con attributi relativi a distretto sorgente, distretto target, tipo di azione, motivazione, snapshot dell'evento e timestamp di creazione;
- la relazione concettuale tra `events` e `actions`, rappresentata come una **associazione molti-a-molti concettuale**, ma implementata a livello fisico tramite:
    - uno *0..* per `events` verso `actions` (un evento può non generare azioni, oppure generarne molte);
    - uno *0..* per `actions` verso `events` (un'azione può essere legata concettualmente a un evento, ma ne conserva una copia in `event_snapshot` anziché un riferimento diretto).

Il diagramma ER mette anche in evidenza:

- le **chiavi primarie** (`id` per entrambe le tabelle);
- la presenza di **attributi indicizzati** (`district`, `sensor_type`, `severity` per `events`; `source_district`, `target_district`, `action_type` per `actions`);
- la distinta natura dei due timestamp (`timestamp` vs `created_at`) nell'entità `Event`.



Questa rappresentazione grafica supporta lo sviluppo e la manutenzione del sistema, fornendo una vista compatta ma precisa delle strutture dati centrali e delle loro interrelazioni concettuali.

4.5. SCELTE RELATIVE A INDICI, QUERY PRINCIPALI E GESTIONE DEI DATI

Le scelte sugli **indici** e sulle modalità di accesso ai dati sono guidate dall'analisi dei casi d'uso:

- nel caso di `events`, l'accesso tipico avviene per:
  - **quartiere** (`district`): ad esempio per visualizzare gli eventi di una determinata area nella dashboard;
  - **tipo di sensore** (`sensor_type`): per analisi mirate su traffico o inquinamento;
  - **severità** (`severity`): per filtrare rapidamente gli eventi critici o potenzialmente critici;
  - **recenza** (`created_at`): per mostrare gli eventi più recenti o limitare l'intervallo temporale analizzato.
- nel caso di `actions`, i pattern di accesso più comuni includono:
  - il filtraggio per **quartiere sorgente** (`source_district`) o **target** (`target_district`), utile per capire:
    - quali distretti stanno generando più escalation;
    - su quali distretti si concentrano le azioni di coordinamento;
  - il filtraggio per **tipo di azione** (`action_type`), per studiare la distribuzione delle strategie adottate;

    - l'ordinamento per `created_at`, per analizzare la sequenza temporale delle decisioni.

Gli attributi **indicizzati** riflettono esattamente queste esigenze:

- su `events`: `district`, `sensor_type`, `severity` sono marcati come `index=True` a livello ORM;
- su `actions`: `source_district`, `target_district`, `action_type` sono anch'essi indicizzati.

Tali indici:

- ottimizzano le query eseguite dal backend per popolare la dashboard;
- supportano eventuali analisi più complesse (ad esempio, estrazione degli ultimi N eventi critici per quartiere, o delle azioni più frequenti per una certa tipologia).

Per quanto riguarda la **gestione dei dati** nel tempo:

- `created_at` su entrambe le tabelle rende possibile implementare in futuro:
    - politiche di **retention** (pulizia di dati molto vecchi, eventualmente previa archiviazione);
    - estrazioni mirate su finestre temporali definite;
- l'uso di `event_snapshot` come campo testuale/ad alta capacità (`TEXT`) consente di introdurre nuove chiavi o attributi negli eventi:
    - senza richiedere modifiche immediate alla struttura di `actions`;
    - mantenendo comunque una rappresentazione fedele del contesto decisionale all'epoca dell'azione.

La combinazione di una modellazione essenziale, di indici mirati e di un uso consapevole dei campi testuali ad alta capacità garantisce un equilibrio tra **prestazioni operative**, **flessibilità evolutiva** e **rigore nella tracciabilità** delle decisioni.

5. PROGETTAZIONE DEL SISTEMA MULTI-AGENTE

5.1. RUOLO DEL SISTEMA MULTI-AGENTE NEL MONITORAGGIO URBANO

Il sistema multi-agente costituisce il **nucleo decisionale** dell'intera piattaforma. La sua funzione principale è trasformare un flusso di eventi grezzi, provenienti dai sensori distribuiti nei quartieri, in **decisioni operative strutturate**, potenzialmente coordinate su scala cittadina.

I compiti fondamentali del MAS sono:

- **interpretazione locale degli eventi**
  Ogni quartiere è rappresentato da un agente dedicato che:
  - riceve eventi relativi al proprio distretto;
  - valuta la severità delle condizioni locali;
  - decide se l'evento possa essere gestito come fenomeno ordinario o se richieda una *escalation*.
- **gestione dell'escalation e visione globale**
  Quando la sola prospettiva locale non è sufficiente, l'agente di quartiere delega al **coordinatore di città**:
  - la valutazione dell'impatto sistemico dell'evento;
  - la generazione di un **piano di coordinamento multi-quartiere**.
- **cooperazione tra distretti**
  Il coordinatore, operando su una vista aggregata dello stato urbano, emette **comandi di coordinamento** verso gli agenti di quartiere, che li traducono in azioni locali.
  Questa dinamica realizza un comportamento collettivo in cui:
  - le decisioni locali tengono conto del contesto globale;
  - le decisioni globali si concretizzano in azioni coerenti nei singoli quartieri.
- **integrazione con il supporto LLM**
  Il MAS funge da *mediatore* tra il modello linguistico e il dominio urbano:
  - predispone il contesto strutturato da fornire al LLM (eventi, stato dei quartieri);
  - interpreta le risposte del LLM in termini di *decisioni operative* (escalation sì/no, piani di coordinamento);
  - applica politiche di fallback qualora il LLM non sia disponibile o restituisca output non validi.

Il risultato è un sistema che unisce la **reattività locale** degli agenti di quartiere con la **coerenza globale** garantita dal coordinatore, mantenendo al contempo un livello elevato di **controllo e tracciabilità** sulle decisioni prese.

5.2. TIPOLOGIE DI AGENTI E RESPONSABILITÀ

La progettazione del MAS si basa su una chiara **separazione dei ruoli** tra:

- agenti con *visione locale* (uno per quartiere);
- un agente con *visione globale* (coordinatore di città).

Questa distinzione consente di modulare la complessità decisionale:

- delegando la gestione del **rumore di fondo** e di molte situazioni ordinarie ai soli agenti di quartiere;
- riservando al coordinatore la gestione di scenari critici e di **interazioni tra distretti**.

5.2.1. DistrictMonitoringAgent (AGENTI DI QUARTIERE)

I **DistrictMonitoringAgent** è l'astrazione software del *quartiere intelligente*. Ogni quartiere monitorato è associato a una istanza dedicata di questo agente, inizializzata con:

- l'identificativo del distretto (es. `"quartiere1"`, `"quartiere2"`);
- parametri di configurazione specifici (soglie, policy locali, ecc.);
- riferimenti ai servizi esterni di cui ha bisogno:
    - backend web per la persistenza degli eventi e delle azioni;
    - gateway LLM per la valutazione delle escalation;
    - riferimento al coordinatore per l'invio delle richieste di escalation.

Le responsabilità principali dell'agente di quartiere sono:

- **ricezione e gestione degli eventi locali**
  L'agente riceve, attraverso il router interno, gli eventi estratti dai topic MQTT relativi al proprio distretto. Per ciascun evento:
    - aggiorna il proprio **contesto locale** (tipicamente una finestra temporale di eventi recenti);
    - valuta una severità preliminare, basata su regole deterministiche o su soglie di dominio;
    - decide se l'evento è *non critico* o *potenzialmente critico*.
- **persistenza degli eventi**
  Ogni evento, critico o non critico, viene trasformato in un oggetto `Event` e inviato, tramite chiamata HTTP al backend (`/api/events`), per essere registrato nel database.
  Questo garantisce che il **flusso informativo** sia completamente tracciabile, indipendentemente dall'esito della valutazione di criticità.
- **valutazione dell'escalation**
  Quando l'evento supera determinate soglie di severità, l'agente può:
    - decidere in modo puramente deterministico se procedere con l'escalation;

- oppure coinvolgere il **LLM** (via gateway) per una decisione più raffinata, inviando:
  - l'evento corrente,
  - un estratto del contesto locale,
  - eventuali metadati rilevanti.

  L'esito della valutazione viene normalizzato in termini di:
  - decisione di *escalate / not escalate*,
  - severità normalizzata,
  - motivazione testuale o simbolica.

- **costruzione e invio delle richieste di escalation**
  Se l'esito è un'escalation, l'agente costruisce un **messaggio interno** di tipo `ESCALATION_REQUEST` contenente:
  - il riassunto dell'evento critico,
  - la motivazione (inclusa l'eventuale contribuzione del LLM),
  - informazioni ausiliarie (es. indicatore di confidenza, ove previsto).

  Il messaggio è inviato al **CityCoordinatorAgent** tramite il meccanismo di messaging interno al MAS.

- **reazione ai comandi di coordinamento**
  L'agente riceve dal coordinatore messaggi di tipo `COORDINATION_COMMAND` che specificano:
  - il tipo di azione da eseguire,
  - il legame con l'evento critico che ha generato il piano,
  - la motivazione del comando.

  L'agente:
  - applica la logica locale corrispondente (ad esempio aggiornamento di stati interni, eventuale logica di attuazione);
  - registra l'azione nel backend (`/api/actions`), includendo uno snapshot dell'evento e i metadati di decisione.

Dal punto di vista implementativo, il DistrictMonitoringAgent è progettato come **unità autonoma** con un proprio *loop di elaborazione* che consuma messaggi da una coda interna, mantenendo separata la gestione degli eventi (input) dalla ricezione dei comandi di coordinamento.

5.2.2. CityCoordinatorAgent (COORDINATORE DI CITTÀ)

Il **CityCoordinatorAgent** rappresenta la *prospettiva globale* della città. A differenza degli agenti di quartiere, non è legato a un singolo distretto, ma gestisce una **mappa di stati** dei diversi quartieri.

Le sue responsabilità principali sono:

- **gestione delle richieste di escalation**
  Il coordinatore riceve messaggi `ESCALATION_REQUEST` dagli agenti di quartiere. Ogni richiesta contiene:
  - l'evento critico che ha innescato l'escalation;
  - la motivazione proposta dall'agente locale (LLM o fallback);
  - eventuali indicatori sintetici (es. severità normalizzata).

  Il coordinatore:
  - integra l'informazione nell'**immagine globale** dello stato urbano;
  - decide se sia necessario un *coordinamento multi-quartiere*.
- **mantenimento dello stato globale**
  Per ciascun quartiere, il coordinatore mantiene uno **stato aggregato**, che può includere:
  - contatori di eventi recenti per tipologia;
  - valori medi o massimi su finestre temporali;
  - indicatori di criticità sintetici.

  Questo stato funge da input per:
  - decisioni basate su regole deterministiche,
  - generazione di contesti da fornire al LLM.
- **pianificazione del coordinamento multi-quartiere**
  Quando una richiesta di escalation è considerata rilevante a livello cittadino, il coordinatore:
  - costruisce un input strutturato che rappresenta:
    - l'evento critico sorgente,
    - lo stato sintetico dei quartieri,
  - decide se:
    - applicare un piano deterministico di fallback,
    - oppure invocare il **LLM** tramite il gateway per proporre un **piano di coordinamento**.

Il risultato è un insieme di *entries* (piano) ciascuna delle quali specifica:

  - quartiere target,
  - tipo di azione da applicare,
  - motivazione.

- **distribuzione dei comandi di coordinamento**
  Per ogni entry del piano, il coordinatore invia un messaggio `COORDINATION_COMMAND` all'agente di quartiere interessato.
  Il comando contiene:

  - i parametri dell'azione (tipo, gravità, eventuali parametri operativi),
  - il riferimento all'evento critico,
  - la motivazione.

- **interazione con il backend per la registrazione delle azioni**
  Il coordinatore può delegare la **persistenza delle azioni**:

  - direttamente, inviando richieste al backend;
  - oppure indirettamente, demandando agli agenti di quartiere la creazione delle azioni dopo l'applicazione locale dei comandi.

  In entrambi i casi, l'obiettivo è garantire che ogni fase del piano di coordinamento sia **registrata e tracciabile**.

Anche il CityCoordinatorAgent è implementato con un *loop di elaborazione* che consuma messaggi da una coda interna, separando esplicitamente la ricezione delle richieste di escalation dalla generazione e distribuzione dei comandi di coordinamento.

5.3. MODELLAZIONE DEI MESSAGGI E DELLE CODE INTERNE

La cooperazione tra agenti è basata su un **meccanismo di messaging interno** che separa nettamente:

- gli **eventi esterni**, provenienti dai sensori via MQTT;
- i **messaggi interni**, scambiati tra agenti nel MAS.

La modellazione dei messaggi interni fa uso di una struttura logica uniforme, che può essere rappresentata come:

- `msg_type`
  Una stringa simbolica che identifica il tipo di messaggio, ad esempio:

  - `"ESCALATION_REQUEST"`,
  - `"COORDINATION_COMMAND"`.

  Questo campo guida lo *smistamento* e la logica di handling all'interno degli agenti.

- `source`
  Identificativo dell'agente sorgente (es. `"quartiere2"`, `"city_coordinator"`).
  Serve sia per scopi di tracciabilità, sia per permettere eventuali risposte o feedback.
- `target`
  Identificativo dell'agente destinatario (es. `"city_coordinator"`, `"quartiere1"`).
  Viene utilizzato dal layer di routing interno per inserire il messaggio nella coda corretta.
- `payload`
  Struttura dati (tipicamente un dizionario serializzabile in JSON) che contiene il contenuto applicativo del messaggio, ad esempio:
  - per `ESCALATION_REQUEST`:
    - evento critico,
    - motivazione dello scoring,
    - contesto locale sintetico;
  - per `COORDINATION_COMMAND`:
    - tipo di azione da eseguire,
    - quartiere target,
    - riferimento all'evento sorgente,
    - motivazione.

Ogni agente dispone di una **coda di input** (ad esempio una `queue.Queue` in Python) nella quale vengono inseriti:

- i messaggi interni destinati a quell'agente;
- eventuali "eventi" già normalizzati e convertiti in messaggi interni (nel caso degli agenti di quartiere).

Gli agenti sono implementati con un loop del tipo:

- estrazione del messaggio dalla coda (bloccante o con timeout);
- dispatch della logica in base a `msg_type`;
- eventuale emissione di nuovi messaggi verso altri agenti.

Questo approccio consente di:

- disaccoppiare il **momento della ricezione** dal **momento dell'elaborazione**, rendendo il sistema più robusto a picchi di traffico;
- isolare la logica di *routing dei messaggi* in un layer distinto, mantenendo il codice degli agenti focalizzato sul comportamento applicativo.

5.4. PROTOCOLLI COOPERATIVI TRA AGENTI

La cooperazione tra agenti è formalizzata tramite **protocolli interni**, definiti in termini di sequenze di messaggi e ruoli degli agenti coinvolti. I due protocolli principali sono:

- il **protocollo di escalation** (quartiere → città);
- il **protocollo di coordinamento** (città → quartieri).

Nel **protocollo di escalation**:

1. L'agente di quartiere riceve un evento localmente critico.
2. Valuta la severità e, se necessario, valuta l'escalation (eventualmente con supporto LLM).
3. Costruisce un messaggio `ESCALATION_REQUEST` con:
    - evento critico,
    - motivazione,
    - informazioni contestuali.
4. Invia il messaggio al CityCoordinatorAgent, che lo inserisce nella propria coda.

Nel **protocollo di coordinamento**:

1. Il coordinatore elabora l'`ESCALATION_REQUEST` e aggiorna lo stato globale.
2. Determina se sia necessario un piano multi-quartiere (via regole o LLM).
3. Genera un insieme di comandi di coordinamento, uno per ciascun quartiere coinvolto.
4. Per ogni comando:
    - costruisce un messaggio `COORDINATION_COMMAND`;
    - lo indirizza all'agente di quartiere target.
5. Ogni agente di quartiere riceve il comando, lo applica e registra le azioni risultanti.

Il **diagramma di sequenza** dedicato al protocollo interno MAS rappresenta visivamente queste interazioni, mettendo in luce:

- la direzione dei messaggi,
- il contenuto concettuale degli scambi,
- i punti in cui le decisioni vengono prese e tradotte in azioni.

Questi protocolli sono progettati in modo da essere **estensibili**: nuovi tipi di messaggi possono essere introdotti (ad esempio per negoziazione, conferma o rollback di azioni), mantenendo la stessa struttura di base e la semantica del `msg_type`.

5.5. DIAGRAMMA DI SEQUENZA – PROTOCOLLO INTERNO TRA AGENTI (SOLO MAS)

Il **diagramma di sequenza del protocollo interno tra agenti** formalizza la dinamica di cooperazione all'interno del MAS, isolando la componente di *coordinamento logico* dalla presenza di servizi esterni (MQTT, backend, LLM).

Gli elementi essenziali mostrati nel diagramma sono:

- i partecipanti:
  - uno specifico **DistrictMonitoringAgent** sorgente (es. `quartiere2`);
  - il **CityCoordinatorAgent**;
  - eventuali **DistrictMonitoringAgent** target (es. `quartiere1`, `quartiere3`);
- la struttura astratta dei messaggi `ESCALATION_REQUEST` e `COORDINATION_COMMAND`;
- il ciclo di vita di un evento critico all'interno del MAS:
  - *evento locale* → *decisione di escalation* → `ESCALATION_REQUEST` → *piano globale* → `COORDINATION_COMMAND` → *azioni locali*.

Il diagramma evidenzia che:

- gli agenti operano su **code di messaggi**, evitando chiamate sincrone bloccanti;
- il coordinatore agisce come *hub logico* che riceve richieste e invia comandi;
- la struttura del payload consente di veicolare tutte le informazioni necessarie senza introdurre dipendenze rigide tra le implementazioni interne degli agenti.

Questo livello di descrizione fornisce una vista rigorosa del **protocollo cooperativo** senza entrare nei dettagli di implementazione dei servizi esterni, che sono trattati separatamente.

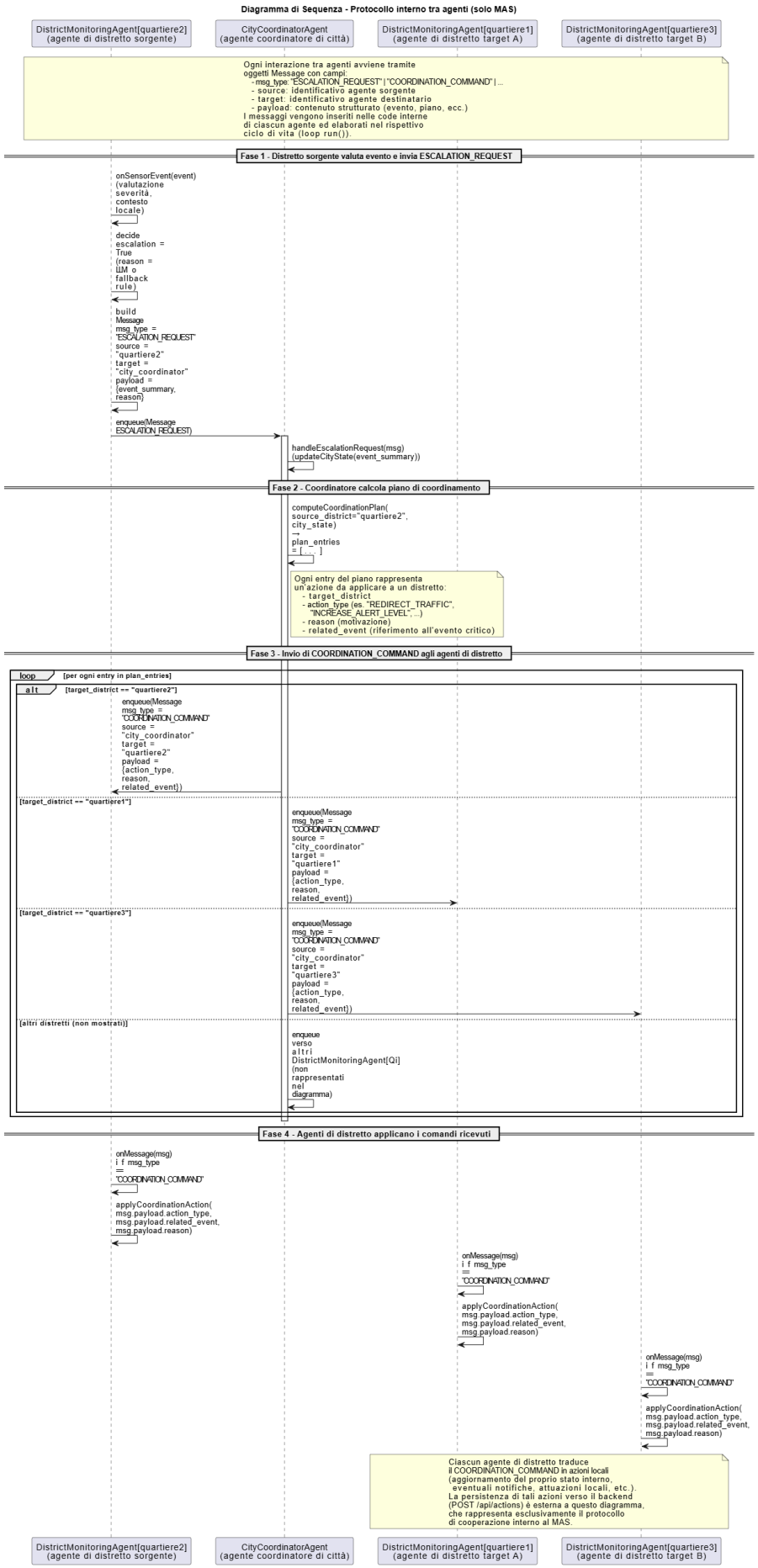


Diagramma di Sequenza - Protocollo interno tra agenti (solo MAS)
DistrictMonitoringAgent[quartiere2] (agente di distretto sorgente)
CityCoordinatorAgent (agente coordinatore di città)
DistrictMonitoringAgent[quartiere1] (agente di distretto target A)
DistrictMonitoringAgent[quartiere3] (agente di distretto target B)
Ogni interazione tra agenti avviene tramite oggetti Message con campi:
- msg_type: "ESCALATION_REQUEST" | "COORDINATION_COMMAND" | ...
- source: identificativo agente sorgente
- target: identificativo agente destinatario
- payload: contenuto strutturato (evento, piano, ecc.)
I messaggi vengono inseriti nelle code interne di ciascun agente ed elaborati nel rispettivo ciclo di vita (loop run()).
Fase 1 - Distretto sorgente valuta evento e invia ESCALATION_REQUEST
onSensorEvent(event) (valutazione severità, contesto locale)
decide escalation = True (reason = LLM o fallback rule)
build Message msg_type = "ESCALATION_REQUEST" source = "quartiere2" target = "city_coordinator" payload = {event_summary, reason}
enqueue(Message ESCALATION_REQUEST)
handleEscalationRequest(msg) (updateCityState(event_summary))
Fase 2 - Coordinatore calcola piano di coordinamento
computeCoordinationPlan(source_district="quartiere2", city_state) → plan_entries = [...]
Ogni entry del piano rappresenta un'azione da applicare a un distretto:
- target_district
- action_type (es. "REDIRECT_TRAFFIC", "INCREASE_ALERT_LEVEL", ...)
- reason (motivazione)
- related_event (riferimento all'evento critico)
Fase 3 - Invio di COORDINATION_COMMAND agli agenti di distretto
loop [per ogni entry in plan_entries]
alt [target_district == "quartiere2"]
enqueue(Message msg_type = "COORDINATION_COMMAND" source = "city_coordinator" target = "quartiere2" payload = {action_type, reason, related_event})
[target_district == "quartiere1"]
enqueue(Message msg_type = "COORDINATION_COMMAND" source = "city_coordinator" target = "quartiere1" payload = {action_type, reason, related_event})
[target_district == "quartiere3"]
enqueue(Message msg_type = "COORDINATION_COMMAND" source = "city_coordinator" target = "quartiere3" payload = {action_type, reason, related_event})
[altri distretti (non mostrati)]
enqueue verso altri DistrictMonitoringAgent[Qi] (non rappresentati nel diagramma)
Fase 4 - Agenti di distretto applicano i comandi ricevuti
onMessage(msg) if msg_type == "COORDINATION_COMMAND"
applyCoordinationAction(msg.payload.action_type, msg.payload.related_event, msg.payload.reason)
onMessage(msg) if msg_type == "COORDINATION_COMMAND"
applyCoordinationAction(msg.payload.action_type, msg.payload.related_event, msg.payload.reason)
onMessage(msg) if msg_type == "COORDINATION_COMMAND"
applyCoordinationAction(msg.payload.action_type, msg.payload.related_event, msg.payload.reason)
Ciascun agente di distretto traduce il COORDINATION_COMMAND in azioni locali (aggiornamento del proprio stato interno, eventuali notifiche, attuazioni locali, etc.).
La persistenza di tali azioni verso il backend (POST /api/actions) è esterna a questo diagramma, che rappresenta esclusivamente il protocollo di cooperazione interno al MAS.
DistrictMonitoringAgent[quartiere2] (agente di distretto sorgente)
CityCoordinatorAgent (agente coordinatore di città)
DistrictMonitoringAgent[quartiere1] (agente di distretto target A)
DistrictMonitoringAgent[quartiere3] (agente di distretto target B)

5.6. VALUTAZIONE DELLE SCELTE PROGETTUALI (THREADING, CODE, DISACCOPPIAMENTO)

La progettazione del MAS adotta alcune scelte specifiche per garantire **responsività**, **robustezza** e **manutenibilità** del sistema.

Sul piano del **threading**:

- il listener MQTT opera in modo indipendente rispetto agli agenti, inoltrando gli eventi al router interno senza bloccare l'elaborazione dei messaggi già in coda;
- gli agenti possono essere eseguiti:
    - ciascuno su un proprio thread dedicato, che esegue il loop di consumo della coda;
    - oppure tramite un meccanismo cooperativo centralizzato che cicla sugli agenti e sulle loro code.

  In entrambi i casi, l'obiettivo è evitare che chiamate potenzialmente lente (es. verso il LLM o il backend) blocchino l'elaborazione di altri messaggi.

Le **code interne** sono il meccanismo principale di disaccoppiamento:

- separano la **produzione** di messaggi (ad esempio l'invio di un `ESCALATION_REQUEST`) dal loro **consumo** (la logica del coordinatore);
- rendono il sistema più tollerante a **picchi di carico**, in quanto i messaggi possono accumularsi temporaneamente in coda senza perdere informazioni;
- consentono di modellare il MAS secondo uno stile **event-driven**, naturale per sistemi reattivi e distribuiti.

Il **disaccoppiamento tra componenti** è ulteriormente rafforzato da:

- una chiara separazione tra:
    - osservazione degli eventi (MQTT → router → agenti),
    - decisione (agenti + LLM),
    - persistenza e presentazione (backend + DB + dashboard);
- l'uso di **interfacce REST** tra MAS, backend e gateway LLM:
    - gli agenti non si occupano della logica interna di persistenza o del parsing delle risposte del modello;
    - il gateway LLM incapsula i dettagli dell'interazione con il motore linguistico;
    - il backend incapsula i dettagli della struttura del database.

Queste scelte progettuali permettono di:

- sostituire o aggiornare in modo indipendente:
    - il modulo di persistenza,

    - l’engine LLM,
    - il layer di sensing;
- ridurre il rischio che modifiche in un componente introducano effetti collaterali imprevisti negli altri, mantenendo il MAS come **core logico coeso ma poco accoppiato** al resto dell’infrastruttura.

6. FLUSSI FUNZIONALI END-TO-END

6.1. SELEZIONE DEGLI SCENARI RAPPRESENTATIVI

I flussi funzionali end-to-end sono descritti attraverso due **scenari rappresentativi** che coprono i comportamenti fondamentali del sistema:

- uno scenario di **evento non critico**, in cui il sistema svolge compiti di monitoraggio, normalizzazione e persistenza, senza attivare escalation o coordinamento tra quartieri;
- uno scenario di **evento critico con escalation e coordinamento multi-quartiere**, in cui entrano in gioco sia il **CityCoordinatorAgent** sia il **LLM**, con generazione di azioni distribuite su più distretti.

Questi scenari:

- esercitano l'intera pipeline sensore → **MQTT broker** → **MAS** → **backend** → **dashboard**;
- evidenziano la differenza tra:
    - gestione **puramente locale** di fenomeni ordinari;
    - gestione **globale e cooperativa** di fenomeni critici, con supporto di un componente LLM.

I diagrammi di sequenza associati ai due scenari formalizzano le interazioni tra i componenti e le tempistiche con cui vengono prese le decisioni, mostrando come i diversi servizi cooperino in un flusso continuo dalla generazione dell'evento all'osservabilità lato operatore.

6.2. SCENARIO 1 – GESTIONE DI UN EVENTO NON CRITICO

Questo scenario descrive il comportamento del sistema in condizioni di **operatività ordinaria**, in cui gli eventi urbani non superano soglie di criticità e vengono trattati come dati di monitoraggio continuo. L'obiettivo è mettere in evidenza il percorso completo dell'informazione, dalla generazione del dato da parte del sensore fino alla visualizzazione nella dashboard, senza l'intervento di meccanismi di escalation o coordinamento multi-quartiere.

6.2.1. DESCRIZIONE DELLO SCENARIO

Lo scenario di base riguarda la gestione di un **evento non critico** generato in un singolo quartiere. Esempio tipico: un sensore di traffico nel *Quartiere 1* rileva un livello di traffico moderato, al di sotto delle soglie che definiscono una condizione critica o di congestione.

Il flusso funzionale è il seguente:

- Un **simulatore di sensore di traffico** nel servizio `sims` produce periodicamente una misura, ad esempio:
    - quartiere: `quartiere1`;
    - tipo di sensore: `traffic`;

    - valore: livello di flusso veicolare (ad esempio veicoli/ora);
    - severità preliminare: `low` o `medium`.
- Il simulatore pubblica l'evento sul broker **MQTT** (`mqtt-broker`) utilizzando un topic strutturato, ad esempio:
  `city/quartiere1/traffic`.
- Il servizio mas-core è sottoscritto ai topic city/+/+ tramite il listener MQTT interno. Alla ricezione del messaggio:
    - il **router interno** esegue il parsing del payload JSON;
    - costruisce un oggetto evento interno coerente con il modello `Event`;
    - individua il **DistrictMonitoringAgent** responsabile di `quartiere1`.
- Il **DistrictMonitoringAgent** di quartiere1:
    - aggiorna il proprio contesto locale (buffer di eventi recenti);
    - valuta la severità dell'evento secondo regole deterministiche e constata che rientra nell'area di **non criticità** (traffico moderato, nessuna condizione anomala persistente).
- L'agente invia una richiesta HTTP al **backend web** (`web-backend`) verso l'endpoint `/api/events`, serializzando l'evento in conformità al modello `Event`.
  Il backend:
    - valida i dati ricevuti;
    - inserisce una nuova riga nella tabella `events` del database SQLite;
    - aggiorna eventuali viste interne utilizzate dalla dashboard.
- La **dashboard web** legge periodicamente gli eventi dal backend, ad esempio:
    - mostrando gli ultimi N eventi per quartiere;
    - evidenziando la severità `low`/`medium`, ma senza alcuna marcatura di allarme.
- L'operatore, accedendo alla dashboard, può:
    - osservare il flusso di eventi ordinari;
    - verificare che il quartiere sia in condizioni di funzionamento normale;
    - usare questi dati come base storica per analisi successive.

In questo scenario non viene attivata alcuna **escalation** verso il coordinatore, né vengono coinvolti il **CityCoordinatorAgent** o il **LLM**. Il MAS opera in modalità *monitoraggio locale + persistenza*, con visualizzazione immediata degli eventi.

6.2.2. DIAGRAMMA DI SEQUENZA – SCENARIO BASE (EVENTO NON CRITICO)

Il **diagramma di sequenza** relativo allo scenario base rappresenta i seguenti partecipanti principali:

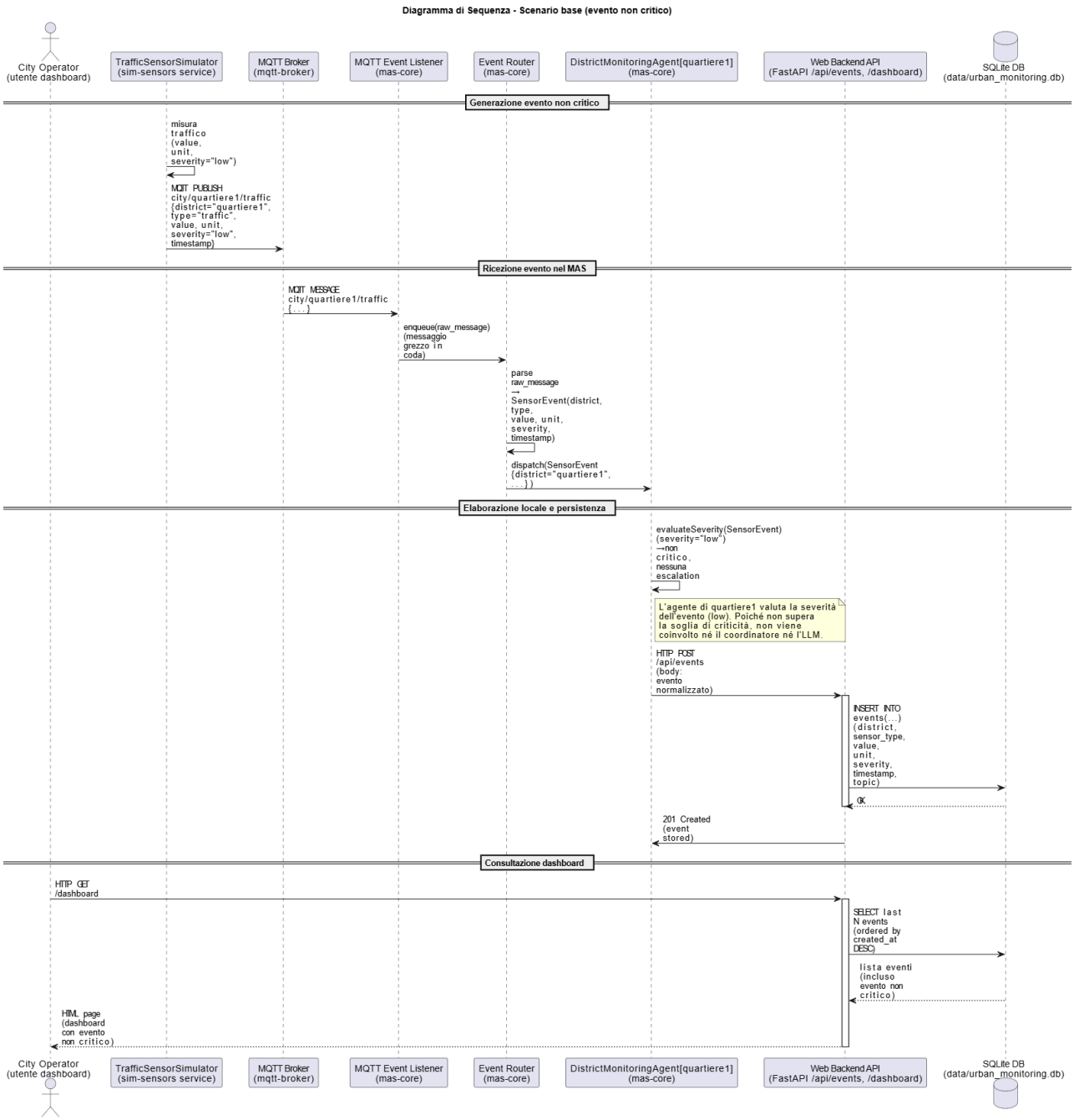
- `TrafficSensorSimulator` (all'interno del servizio `sims`);
- `MQTT Broker` (Eclipse Mosquitto);
- `MAS Listener/Router` (nel servizio `mas-core`);
- `DistrictMonitoringAgent[quartiere1]`;
- `Web Backend` (API FastAPI);
- `Database` (SQLite);
- `Operatore` (via dashboard web).

La sequenza di interazioni evidenzia:

- il **publish** dell'evento da parte del simulatore sul topic `city/quartiere1/traffic`;
- la **consegna del messaggio** al listener MQTT e il passaggio al router interno;
- la **chiamata interna** dal router all'agente di quartiere, con passaggio dell'oggetto evento;
- la **chiamata HTTP** `POST /api/events` dall'agente verso il backend per la persistenza;
- la scrittura del record nella tabella `events` e la conferma (ad esempio un `201 Created`);
- il **pull** dei dati da parte della dashboard (ad esempio via `GET /events` o endpoint equivalenti) e la successiva visualizzazione lato operatore.

Il diagramma mette in evidenza che:

- le uniche interazioni esterne all'ambiente MAS sono:
    - MQTT per il trasporto dell'evento grezzo;
    - HTTP/REST per la persistenza e la successiva visualizzazione;
- l'agente di quartiere prende decisioni **locali** senza coinvolgimento di altri agenti o servizi, operando come filtro intelligibile tra dominio IoT e dominio applicativo.



6.2.3. ANALISI DELLA PIPELINE SENSORE → MAS → BACKEND → DASHBOARD

La pipeline funzionale dello scenario base può essere letta come una **catena di trasformazioni**:

1. **Generazione del dato grezzo**
   Il simulatore produce una misura elementare, che porta con sé:
   - contesto geografico (quartiere);
   - contesto semantico (tipo di sensore);
   - valore e unità.

2. **Trasporto e normalizzazione tecnica**
   Il messaggio viene trasportato via MQTT; quindi, normalizzato dal **router del MAS** in una struttura dati interna coerente con il modello `Event`.
   In questa fase vengono:
   - effettuati controlli sul payload (esistenza dei campi richiesti);
   - applicate eventuali trasformazioni preliminari (es. mapping dei valori di severità).
3. **Valutazione locale dell'evento**
   Il **DistrictMonitoringAgent**:
   - colloca l'evento in un contesto temporale (finestra recente del quartiere);
   - applica regole locali di severità;
   - decide che il fenomeno è **ordinario** e non richiede escalation.
4. **Persistenza applicativa**
   L'evento viene inviato al **backend** e salvato nel database:
   - la tabella `events` funge da *registro persistente* dello stato urbano;
   - i campi indicizzati (`district`, `sensor_type`, `severity`) permettono query efficienti per filtraggio e aggregazione.
5. **Esposizione all'operatore**
   La **dashboard** interroga periodicamente il backend:
   - il quartiere appare come *monitorato* ma non *allarmato*;
   - l'operatore può verificare che l'attività sia sotto controllo, sfruttando l'interfaccia per navigare storici, grafici, tabelle.

Questo flusso esprime il comportamento **nominale** del sistema in condizioni non critiche, dimostrando la capacità di:

- integrare mondo IoT e mondo applicativo;
- registrare in modo strutturato tutti gli eventi;
- offrire **osservabilità** in tempo reale senza introdurre complessità decisionale non necessaria.

6.3. SCENARIO 2 – GESTIONE DI UN EVENTO CRITICO CON ESCALATION E COORDINAMENTO MULTI-QUARTIERE

Questo scenario descrive il comportamento del sistema in presenza di **condizioni anomale ad alta severità**, per le quali risulta necessario passare dalla gestione locale alla **gestione coordinata a livello cittadino**. Il flusso funzionale coinvolge l'intero stack logico: MAS locale e globale, gateway LLM, motore LLM esterno, backend e dashboard, evidenziando i punti in cui si attivano l'escalation e la pianificazione multi-quartiere.

6.3.1. DESCIZIONE DELLO SCENARIO CRITICO

Lo scenario avanzato riguarda la gestione di un **evento critico** che, per intensità o contesto, richiede:

- una **escalation** dal livello di quartiere al livello di città;
- un **coordinamento multi-quartiere**, potenzialmente supportato dal LLM.

Un esempio concreto è un picco di traffico nel *Quartiere 2* con valori nettamente superiori alla soglia di congestione, in presenza di altri quartieri confinanti (ad esempio *Quartiere 1* e *Quartiere 3*) che presentano margini di capacità residuale.

Il flusso funzionale si articola nei seguenti passaggi:

- Un **simulatore di traffico** nel servizio `sims` genera un evento con:
    - quartiere: `quartiere2`;
    - tipo di sensore: `traffic`;
    - valore: molto elevato;
    - severità preliminare: `high`.

  Il simulatore pubblica il messaggio sul topic `city/quartiere2/traffic`.
- Il **listener MQTT** del `mas-core` riceve il messaggio e il router:
    - costruisce l'oggetto evento interno;
    - lo indirizza al `DistrictMonitoringAgent[quartiere2]`.
- L'agente di `quartiere2`:
    - aggiorna il proprio contesto locale;
    - rileva, tramite regole e soglie, che:
        - la severità è superiore alla soglia di criticità;
        - il fenomeno ha potenziale impatto non solo sul distretto locale, ma anche sugli assi viari che lo collegano ad altri quartieri.
- L'agente:
    - invia l'evento al **backend** (`/api/events`) per la persistenza (come nello scenario base);
    - decide che la sola gestione locale non è sufficiente e valuta l'escalation:
        - può coinvolgere il **LLM** tramite il gateway (`/llm/decide_escalation`), inviando:
            - l'evento in forma strutturata;
            - il contesto locale recente (altri eventi di traffico, eventuali trend);

        - riceve una risposta JSON con:
            - severità normalizzata,
            - indicazione se effettuare escalation,
            - motivazione sintetica.
        - sulla base di questa risposta (e/o di regole locali), decide di **escalare**.
- L'agente costruisce un messaggio interno `ESCALATION_REQUEST` e lo invia al **CityCoordinatorAgent**, includendo:
    - evento critico;
    - motivazione (es. suggerimento del LLM);
    - eventuali indicatori aggiuntivi.
- Il **CityCoordinatorAgent**:
    - integra la richiesta nello **stato globale** della città;
    - costruisce una rappresentazione sintetica dello stato dei quartieri (traffico e, se rilevante, altri parametri);
    - decide di calcolare un **piano di coordinamento multi-quartiere**:
        - può applicare regole di fallback;
        - oppure invocare il **LLM** tramite il gateway (`/llm/plan_coordination`), fornendo:
            - l'evento critico sorgente;
            - lo stato aggregato dei quartieri (es. indice di congestione per ciascuno).
- Il **LLM Gateway**:
    - costruisce il prompt e il payload per il motore LLM;
    - invia la richiesta al **LLM Engine** (Ollama o equivalente);
    - riceve una risposta strutturata che, ad esempio, suggerisce:
        - deviazione del traffico dal `quartiere2` verso `quartiere1`;
        - mantenimento di un certo livello di attenzione su `quartiere3`;
    - valida il formato JSON e restituisce al coordinatore un **piano di coordinamento** normalizzato.
- Il coordinatore traduce il piano in una serie di messaggi `COORDINATION_COMMAND`, uno per ciascun quartiere interessato:
    - per `quartiere2`, ad esempio: azione `REDIRECT_TRAFFIC` con motivazione;

    - per `quartiere1`, ad esempio: azione `HANDLE_REDIRECTED_FLOW` con motivazione.
- Ogni **DistrictMonitoringAgent** coinvolto:
    - riceve il comando,
    - applica la logica locale corrispondente (ad esempio aggiornando il proprio stato, predisponendo contromisure, loggando il cambiamento di strategia),
    - registra l'azione nel backend (`/api/actions`), includendo:
        - `source_district`, `target_district`, `action_type`;
        - `reason` (indicante l'uso del LLM o la policy adottata);
        - `event_snapshot` con il contesto dell'evento critico.
- La **dashboard web**:
    - mostra non solo gli eventi critici, ma anche le **azioni di coordinamento** applicate ai quartieri;
    - rende evidente il legame temporale e causale tra evento critico in `quartiere2` e azioni in `quartiere1` e `quartiere2`.

Lo scenario mette in luce la capacità del sistema di **passare da una prospettiva locale a una visione globale**, con generazione di piani multi-quartiere sostenuti dal contributo di un LLM.

6.3.2. DIAGRAMMA DI SEQUENZA – SCENARIO AVANZATO (EVENTO CRITICO, ESCALATION E COORDINAMENTO CON LLM)

Il **diagramma di sequenza** dello scenario avanzato arricchisce quello dello scenario base introducendo:

- il **CityCoordinatorAgent** come partecipante aggiuntivo;
- il **LLM Gateway** e il **LLM Engine** come componenti esterni consultivi;
- uno o più **DistrictMonitoringAgent** target oltre a quello sorgente.

Le principali interazioni rappresentate includono:

- il flusso iniziale sensore → MQTT → MAS, identico a quello dello scenario non critico;
- la chiamata dall'agente di quartiere al **LLM Gateway** (`/llm/decide_escalation`) con contesto locale, e la risposta con decisione di escalation;
- l'invio del messaggio `ESCALATION_REQUEST` dal DistrictMonitoringAgent al CityCoordinatorAgent;
- la chiamata del coordinatore al **LLM Gateway** (`/llm/plan_coordination`) con stato globale della città;

- la chiamata interna del gateway verso il **LLM Engine** e la successiva validazione della risposta;
- la generazione di più messaggi `COORDINATION_COMMAND` verso i diversi agenti di quartiere;
- la registrazione delle azioni nel backend e la successiva visualizzazione sulla dashboard.

Il diagramma evidenzia il ruolo del LLM come **componente di supporto**:

- il MAS non delega al LLM l'intera responsabilità del processo decisionale;
- il LLM fornisce **suggerimenti strutturati** che vengono integrati in un flusso di messaggi e decisioni comunque gestito in modo esplicito dagli agenti.

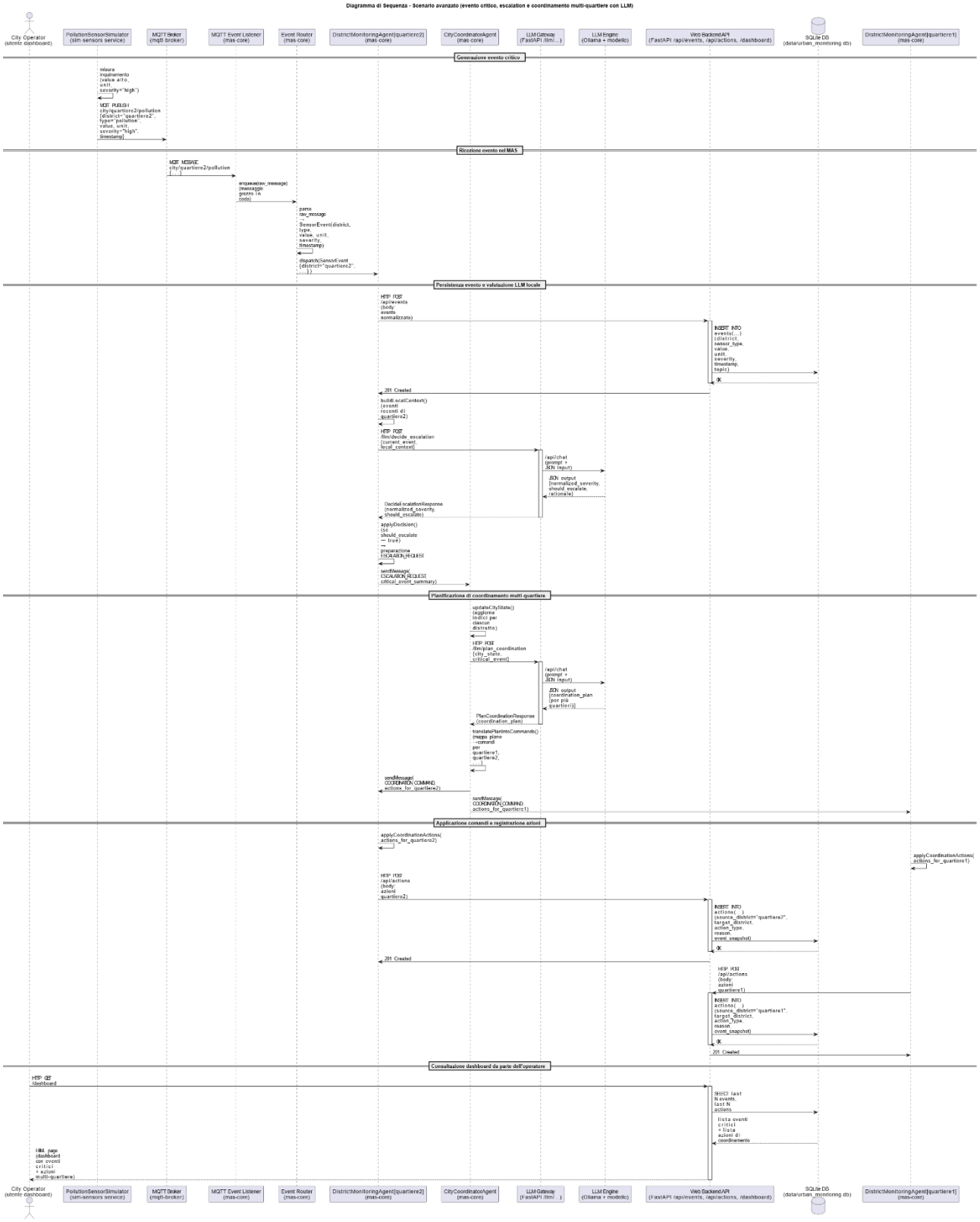


Diagramma di Sequenza - Scenario avanzato (evento critico, escalation e coordinamento multi-quartiere con LLM)
City Operator (utente dashboard)
PollutionSensorSimulator (sim-sensors service)
MQTT Broker (mqtt broker)
MQTT Event Listener (mas-core)
Event Router (mas-core)
DistrictMonitoringAgent[quartiere2] (mas-core)
CityCoordinatorAgent (mas-core)
LLM Gateway (FastAPI /llm/...)
LLM Engine (Ollama + modello)
Web Backend API (FastAPI /api/events, /api/actions, /dashboard)
SQLite DB (data/urban_monitoring.db)
DistrictMonitoringAgent[quartiere1] (mas-core)
Generazione evento critico
misura inquinamento (value alto, unit, severity="high")
MQTT PUBLISH city/quartiere2/pollution {district="quartiere2", type="pollution", value, unit, severity="high", timestamp}
Ricezione evento nel MAS
MQTT MESSAGE city/quartiere2/pollution
on_message(raw_message) (message: stringa in JSON)
parse raw_message
SensorEvent(district, type, value, unit, severity, timestamp)
dispatch(SensorEvent) (district="quartiere2")
Persistenza evento e valutazione LLM locale
HTTP POST /api/events (body: evento normalizzato)
INSERT INTO events(...) (district, sensor_type, value, unit, severity, timestamp, topic)
OK
201 Created
buildLocalContext() (eventi recenti di quartiere2)
HTTP POST /llm/decide_escalation (current_event, local_context)
/api/chat (prompt + JSON input)
JSON output (normalized_severity, should_escalate, rationale)
LocalEscalationResponse (normalized_severity, should_escalate)
applyDecision() [se should_escalate == true]
preparazione ESCALATION_REQUEST
sendMessage(ESCALATION_REQUEST, critical_event_summary)
Pianificazione di coordinamento multi quartiere
updateCityState() (aggiorna indici per ciascun distretto)
HTTP POST /llm/plan_coordination (city_state, critical_event)
/api/chat (prompt + JSON input)
JSON output (coordination_plan per più quartieri))
PlanCoordinationResponse (coordination_plan)
translatePlanIntoCommands() (mappa piano --comandi per quartiere1, quartiere2,
sendMessage(COORDINATION_COMMAND, actions_for_quartiere2)
sendMessage(COORDINATION_COMMAND, actions_for_quartiere1)
Applicazione comandi e registrazione azioni
applyCoordinationActions(actions_for_quartiere2)
applyCoordinationActions(actions_for_quartiere1)
HTTP POST /api/actions (body: azioni quartiere2)
INSERT INTO actions(...) (source_district="quartiere2", target_district, action_type, reason, event_snapshot)
OK
201 Created
HTTP POST /api/actions (body: azioni quartiere1)
INSERT INTO actions(...) (source_district="quartiere1", target_district, action_type, reason, event_snapshot)
OK
201 Created
Consultazione dashboard da parte dell'operatore
HTTP GET /dashboard
SELECT last N events, last N actions
lista eventi critici + lista azioni di coordinamento
HTML page (dashboard con eventi critici + azioni multi-quartiere)

6.3.3. ANALISI DEL COORDINAMENTO TRA QUARTIERI

Il coordinamento multi-quartiere implementato nello scenario avanzato si basa su alcuni principi progettuali chiave:

- **centralizzazione della decisione globale**
  Il **CityCoordinatorAgent** è l'unico componente che:
  - mantiene una vista completa dello stato dei quartieri;
  - valuta gli impatti sistemici di un evento critico;
  - decide se e come distribuire le azioni su più distretti.
- **localizzazione dell'attuazione**
  Sebbene la strategia sia calcolata a livello centrale, l'attuazione è **localizzata**:
  - ciascun DistrictMonitoringAgent riceve un comando strutturato;
  - traduce il comando nel proprio contesto locale;
  - registra l'azione come parte del proprio flusso operativo.
- **uso del LLM per la generazione di piani complessi**
  Il LLM viene sfruttato per:
  - sintetizzare relazioni non banali tra quartieri (es. pattern di traffico su più distretti);
  - proporre piani di coordinamento che rispettino vincoli e obiettivi multipli.
    L'output del LLM, in forma JSON, viene interpretato come una **bozza di piano**, che il coordinatore traduce in messaggi interni verso gli agenti.
- **tracciabilità delle decisioni**
  Ogni azione generata nel contesto del piano:
  - viene registrata in `actions` con esplicita indicazione del `source_district`, `target_district` e `reason`;
  - preserva un `event_snapshot` dell'evento critico che l'ha originata.
    Questo consente, anche a posteriori, di analizzare se e come una decisione globale abbia influenzato l'andamento dei singoli quartieri.
- **separazione tra logica globale e locale**
  Le **regole locali** restano sotto il controllo degli agenti di quartiere, che:
  - possono integrare i comandi globali con la propria conoscenza interna;
  - possono adottare comportamenti aggiuntivi coerenti con il proprio stato locale.

Questa articolazione permette al sistema di realizzare un **coordinamento distribuito ma coerente**, in cui l'intervento del LLM arricchisce il livello di intelligenza globale senza annullare la responsabilità esplicita degli agenti.

6.4. CONFRONTO TRA GESTIONE DI EVENTI NON CRITICI E CRITICI

Il confronto tra scenario non critico e scenario critico evidenzia alcune differenze strutturali nei flussi end-to-end:

- **profondità del flusso decisionale**
  - negli eventi non critici:
    - il flusso si ferma al **livello di quartiere**;
    - il MAS opera come filtro e normalizzatore, persiste l'evento e rende i dati osservabili;
  - negli eventi critici:
    - il flusso si estende al **livello cittadino**, coinvolgendo il coordinatore;
    - può essere innestata la fase di consultazione del **LLM** per l'escalation e la pianificazione.
- **coinvolgimento degli attori**
  - scenario non critico:
    - coinvolge: simulatore, MQTT broker, listener/router, DistrictMonitoringAgent, backend, DB, dashboard;
  - scenario critico:
    - aggiunge: CityCoordinatorAgent, LLM Gateway, LLM Engine, altri agenti di quartiere destinatari dei comandi.
- **tipo di output generato**
  - scenario non critico:
    - produce principalmente **record di eventi** (`events`), utilizzati per monitoraggio e analisi;
  - scenario critico:
    - produce sia **eventi** sia **azioni** (`actions`), con forte enfasi sulla tracciabilità delle decisioni e sulle relazioni tra quartieri.
- **complessità della pipeline e della tracciabilità**
  - nel caso non critico:
    - la pipeline è più lineare, con poche trasformazioni;
    - la tracciabilità è centrata sulla registrazione degli eventi;
  - nel caso critico:
    - la pipeline include più passaggi decisionali, sia deterministici sia supportati da LLM;

- la tracciabilità deve coprire:
    - lo stato globale della città,
    - le raccomandazioni del LLM,
    - le azioni distribuite sui quartieri.

Questa differenziazione rende esplicito come il sistema sia progettato per **adattare il proprio comportamento** alla severità e alla natura degli eventi, mantenendo comune l'infrastruttura tecnica (sensori, MQTT, MAS, backend) ma variando la profondità del ragionamento e il livello di coordinamento attivato.

7. INTEGRAZIONE DEL LLM NEL PROCESSO DECISIONALE

7.1. OBIETTIVO DELL'INTEGRAZIONE NEL PROCESSO DECISIONALE

L'integrazione di un **modello linguistico di grandi dimensioni (LLM)** nel sistema è orientata a potenziare le capacità decisionali del **MAS** senza snaturarne l'architettura né delegare al modello il controllo diretto del dominio operativo.

Gli obiettivi principali sono:

- **Supporto decisionale avanzato in condizioni di ambiguità**
  Gli eventi urbani generati dai sensori possono collocarsi in aree di incertezza (valori prossimi alle soglie, pattern temporali non banali, combinazioni di fenomeni). Il LLM viene impiegato per:
  - affinare la **valutazione della severità** di un evento;
  - fornire una raccomandazione strutturata sulla necessità di **escalation** al livello cittadino;
  - esporre una **motivazione testuale sintetica** a supporto dell'interpretazione.
- **Generazione di piani di coordinamento multi-quartiere**
  In presenza di eventi critici con impatto potenzialmente diffuso, il LLM viene sfruttato per:
  - analizzare uno **stato globale** della città, composto da indicatori per distretto;
  - proporre un **piano di azioni coordinato**, espresso come elenco strutturato di comandi da applicare a diversi quartieri;
  - tenere conto di vincoli e obiettivi multipli (ad esempio bilanciamento del carico tra distretti).
- **Mantenimento del controllo esplicito da parte degli agenti**
  Il LLM non interagisce direttamente con:
  - sensori;
  - database;
  - attuatori.
    Viene utilizzato come *componente consulenziale* all'interno di un flusso controllato dagli agenti, che:
  - decidono quando interrogarlo;
  - interpretano le risposte;
  - assumono la responsabilità finale delle azioni intraprese.
- **Strutturazione, tracciabilità e auditabilità delle decisioni**
  L'uso del LLM è incapsulato in protocolli basati su **JSON tipizzati**:
  - ogni richiesta specifica il contesto in maniera esplicita;

- ogni risposta viene validata e trasformata in oggetti coerenti con il dominio (decisione di escalation, piano di coordinamento).
  Ciò consente di tracciare in database e nei log:
- quando e perché il LLM è stato consultato;
- in che modo il suo contributo ha influenzato le decisioni del MAS.

7.2. ARCHITETTURA DEL LLM GATEWAY E INTERFACCIAMENTO CON IL MODELLO

L'integrazione del LLM è incapsulata in un micro-servizio dedicato, denominato **LLM Gateway**, implementato come applicazione **FastAPI** e progettato per fornire un'interfaccia REST stabile al sistema multi-agente.

Le responsabilità principali del LLM Gateway sono:

- **Astrazione del motore LLM sottostante**
  Il gateway espone endpoint applicativi stabili, indipendenti da:
  - tecnologia effettiva del modello (Ollama, altro motore locale o remoto);
  - nome concreto del modello utilizzato.

  Parametri come `LLM_API_BASE` e `LLM_MODEL_`NAME vengono letti da variabili d'ambiente tramite una configurazione centralizzata (LLMSettings), permettendo la sostituzione del modello senza modificare il MAS.
- **Definizione di API specializzate per gli use case**
  Vengono definiti due endpoint principali:
  - `POST /llm/decide_escalation` per supportare la **decisione locale di escalation**;
  - `POST /llm/plan_coordination` per supportare la **pianificazione del coordinamento multi-quartiere**.
    Entrambi utilizzano schemi Pydantic per input e output, garantendo contratti espliciti.
- **Costruzione dei prompt e orchestrazione delle chiamate al LLM**
  Il gateway:
  - costruisce il **prompt testuale** che descrive il ruolo del modello (supporto al monitoraggio urbano) e specifica il formato di risposta richiesto;
  - serializza il contesto operativo in JSON e lo incorpora nel prompt o nel payload;
  - invia la richiesta HTTP al motore LLM, applicando timeout configurabili;
  - gestisce eventuali errori di rete o di servizio.
- **Validazione e normalizzazione delle risposte**
  Le risposte testuali del LLM:
  - vengono analizzate per individuare il blocco JSON atteso;

- sono parse-ate in oggetti Python;
- sono validate tramite i modelli Pydantic di output (campi obbligatori, tipi, domini di valori ammessi).

In caso di errore, il gateway restituisce una risposta strutturata di fallimento al MAS, che potrà attivare politiche di fallback.

- **Logging e osservabilità**
  Il gateway registra:
  - richieste ricevute (tipologia, distretti coinvolti);
  - esito delle chiamate al LLM (successo, timeout, errore di formato);
  - tempi di risposta.
    Queste informazioni sono utilizzate per diagnosticare problemi e per analizzare l'impatto dell'uso del LLM sul comportamento del sistema.

Il LLM Gateway rappresenta quindi uno strato di **decoupling** tra il dominio MAS/IoT e l'infrastruttura di inferenza, riducendo la complessità visibile agli agenti e garantendo stabilità dei contratti applicativi.

7.3. PROGETTAZIONE DEI PROMPT E DEI FORMATI JSON DI INPUT/OUTPUT

La progettazione dei **prompt** e dei formati **JSON** di input/output è cruciale per trasformare il LLM in un componente prevedibile e integrabile.

I principi adottati sono:

- **Strutturazione rigorosa del contesto in input**
  Gli oggetti passati al LLM sono modellati tramite schemi Pydantic e includono:
  - per la decisione di escalation:
    - sezione `event` con campi:
      - `district`, `sensor_type`, `value`, `unit`, `severity`, `timestamp`;
    - sezione `local_context` con:
      - elenco di `recent_events` sintetici;
      - eventuali indicatori aggregati del quartiere;
  - per la pianificazione del coordinamento:
    - sezione `critical_event` con l'evento che ha originato l'escalation;
    - sezione `city_state` con:
      - elenco di quartieri e relativi indicatori (es. indice di traffico, stato locale).

- **Definizione di formati JSON di output vincolati**
  I formati di risposta sono progettati per essere facilmente mappati sul dominio MAS:
  - per `decide_escalation`:
    - `normalized_severity`: stringa appartenente a un insieme finito (ad esempio `low`, `medium`, `high`, `critical`);
    - `should_escalate`: booleano;
    - `rationale`: testo breve che spiega la decisione;
  - per `plan_coordination`:
    - `actions`: lista di azioni, ciascuna con:
      - `source_district`;
      - `target_district`;
      - `action_type` (tra valori ammessi, ad esempio `REDIRECT_TRAFFIC`, `INCREASE_ALERT_LEVEL`, ecc.);
      - `reason`.
- **Prompt istruitivi e vincolanti**
  I prompt forniti al LLM specificano in modo esplicito:
  - il ruolo del modello come **supporto decisionale** per un sistema di monitoraggio urbano;
  - il fatto che la risposta **deve essere esclusivamente JSON** nel formato indicato;
  - i domini di valori ammessi per i campi chiave (severità, tipi di azione);
  - il divieto di inserire testo extra al di fuori del blocco JSON.
- **Riduzione dell'ambiguità semantica**
  I nomi dei campi e i valori ammessi sono scelti per essere:
  - semanticamente chiari e stabili nel tempo;
  - coerenti con i modelli interni del MAS (`Event`, `Action`).
    Ciò riduce il rischio di interpretazioni divergenti tra output del LLM e logica lato agente.

Questa progettazione consente al sistema di utilizzare il LLM come generatore di **decisioni strutturate**, riducendo al minimo l'interpretazione testuale a carico del codice applicativo.

7.4. PUNTI DI INTEGRAZIONE NEL CICLO DECISIONALE

L'integrazione del LLM nel ciclo decisionale avviene in due punti distinti e ben delimitati:

- nel processo di **valutazione locale** degli eventi da parte dei DistrictMonitoringAgent;

- nel processo di **coordinamento globale** gestito dal CityCoordinatorAgent.

In entrambi i casi, l'invocazione del LLM è facoltativa e condizionata dal contesto:

- se le regole deterministiche sono sufficienti (ad esempio soglie nettamente superate o non raggiunte), l'agente può prendere decisioni senza consultare il modello;
- nelle situazioni di maggiore incertezza o complessità, il LLM viene utilizzato per arricchire il processo decisionale.

7.4.1. DECISIONE LOCALE DI ESCALATION (`decide_escalation`)

Nel flusso di gestione di un evento, ogni **DistrictMonitoringAgent** svolge i seguenti passi:

- riceve l'evento dal router interno, corredato di:
  - quartiere;
  - tipo di sensore;
  - valore misurato;
  - severità preliminare;
  - timestamp;
- aggiorna il proprio **contesto locale**, includendo l'evento in un buffer temporale;
- applica regole deterministiche di prima valutazione (soglie, trend locali).

Quando l'evento si colloca in una zona di **criticità potenziale**, l'agente può:

- costruire un oggetto `EscalationDecisionRequest` contenente:
  - `event`: descrizione completa dell'evento;
  - `local_context`: eventi recenti e indicatori sintetici del quartiere;
- inviare il payload al LLM Gateway via `POST /llm/decide_escalation`.

Il gateway:

- costruisce il prompt, interroga il LLM e restituisce un `EscalationDecisionResponse` contenente:
  - `normalized_severity`;
  - `should_escalate`;
  - `rationale`.

L'agente utilizza tale risposta per:

- aggiornare la severità interna, se ritenuto coerente con il contesto;
- decidere se emettere un messaggio `ESCALATION_REQUEST` al CityCoordinatorAgent;

- registrare in eventuali log interni o nel campo `reason` di future azioni il fatto che la decisione è stata **supportata dal LLM**.

Se la chiamata al gateway fallisce o l'output non è valido, l'agente attiva le **politiche di fallback**, come descritto nel capitolo sulla robustezza.

7.4.2. PIANIFICAZIONE DEL COORDINAMENTO MULTI-QUARTIERE (`plan_coordination`)

Quando il CityCoordinatorAgent riceve una o più richieste di escalation, costruisce una vista **globale** dello stato della città e può:

- applicare regole deterministiche per situazioni semplici;
- oppure delegare al LLM la generazione di un piano di coordinamento più articolato.

Nel secondo caso, il coordinatore:

- costruisce un oggetto `CoordinationPlanRequest` contenente:
  - `critical_event`: evento che ha originato l'escalation (quartiere sorgente, tipo di sensore, severità);
  - `city_state`: elenco dei quartieri con i relativi indicatori (es. livello di traffico).
- invia la richiesta al LLM Gateway via POST /llm/plan_coordination.

Il gateway:

- costruisce il prompt, interroga il LLM, e restituisce un `CoordinationPlanResponse` che contiene una lista di `CoordinationAction`:
  - ciascuna azione specifica `source_district`, `target_district`, `action_type`, `reason`.

Il CityCoordinatorAgent:

- traduce ogni `CoordinationAction` in un messaggio interno `COORDINATION_COMMAND` verso il DistrictMonitoringAgent target;
- può arricchire o adattare il piano in base a vincoli locali o regole aggiuntive;
- assicura che le azioni generate siano coerenti con le capacità operative dei quartieri coinvolti.

Le azioni applicate vengono poi registrate nel backend (`/api/actions`), mantenendo traccia del legame tra piano LLM e implementazione concreta nel MAS.

7.5. VALIDAZIONE E PARSING DELLE RISPOSTE DEL LLM

Le risposte del LLM sono sottoposte a un processo di **validazione multilivello**, concepito per evitare che output non conformi compromettano il comportamento del sistema.

Le fasi principali sono:

- **Estrazione del blocco JSON**
  Il LLM può produrre testo contenente spiegazioni e il blocco JSON richiesto. Il LLM Gateway si occupa di:
  - individuare il segmento che rappresenta il JSON;
  - rimuovere eventuali elementi non strutturati (commenti, testo introduttivo o conclusivo).
- **Parsing in oggetto Python**
  Il blocco JSON viene parse-ato tramite la libreria standard, trasformandolo in dizionario o lista di dizionari.
  Eventuali errori di parsing (JSON malformato) vengono intercettati e tradotti in un errore strutturato (`error_type = "invalid_output"`).
- **Validazione tramite modelli Pydantic**
  Il dizionario viene passato ai modelli Pydantic di output:
  - `EscalationDecisionResponse` per `decide_escalation`;
  - `CoordinationPlanResponse` per `plan_coordination`.
    La validazione verifica:
  - presenza di tutti i campi obbligatori;
  - tipo dei valori (stringhe, booleani, liste);
  - appartenenza a domini chiusi (severità, tipi di azione, nomi di quartiere ammessi).
- **Normalizzazione e sanificazione dei valori**
  Se necessario, il gateway può applicare piccole normalizzazioni:
  - trasformazioni di case (`HIGH` → `high`);
  - trimming di spazi o caratteri non significativi.
    Qualsiasi correzione più invasiva viene evitata, privilegiando la segnalazione di errore al MAS.
- **Gestione dell'errore verso il MAS**
  Se una qualsiasi fase fallisce (timeout, errore di rete, parsing, validazione), il gateway:
  - non restituisce una risposta parzialmente consistente;
  - invia al MAS un oggetto di errore con:
    - `error_type`;
    - `error_message` sintetico.

  Gli agenti, ricevendo un errore strutturato, possono attivare il corrispondente flusso di fallback.

Questa catena di validazione garantisce che il MAS interagisca sempre con oggetti **internamente coerenti**, riducendo al minimo il rischio di comportamenti imprevedibili derivanti da diverse forme di output del LLM.

7.6. IMPATTI DELL'USO DEL LLM SU QUALITA' E SPIEGABILITA' DELLE DECISIONI

L'integrazione del LLM produce effetti significativi sulla **qualità** e sulla **spiegabilità** delle decisioni prese dal sistema.

Dal punto di vista della **qualità delle decisioni**:

- la valutazione della severità beneficia di una capacità di **contestualizzazione** che combina:
    - valori numerici;
    - pattern temporali (sequenze di eventi recenti);
    - informazioni qualitative incluse nei campi descrittivi;
- la pianificazione multi-quartiere può tenere conto di **relazioni non banali** tra distretti, come:
    - pattern combinati di traffico;
    - capacità residuali;
    - possibili effetti collaterali di spostamenti di carico.

Ciò consente di generare scenari di risposta più articolati rispetto a quelli ottenibili con sole regole statiche, soprattutto in condizioni di complessità crescente.

Dal punto di vista della **spiegabilità**:

- le risposte del LLM includono sempre una componente testuale (`rationale` o `reason`) che:
    - esplicita i motivi della classificazione della severità;
    - giustifica le azioni di coordinamento proposte;
- queste motivazioni vengono integrate:
    - nei campi `reason` delle entità Action, spesso con riferimenti al contenuto del ragionamento LLM;
    - nei log del LLM Gateway e del MAS.

L'operatore può così:

- analizzare non solo **che cosa** il sistema ha deciso, ma anche **perché** lo ha fatto;
- confrontare le decisioni supportate dal LLM con le decisioni derivate da fallback deterministici, individuando eventuali pattern ricorrenti o incoerenze.

Resta comunque cruciale il fatto che:

- il LLM non è un *oracolo infallibile*;
- le sue raccomandazioni sono sempre sottoposte alla **mediazione esplicita** degli agenti;
- l'architettura prevede meccanismi strutturati di fallback e tracciabilità, in modo che l'impatto di eventuali errori o bias del modello sia identificabile e confinato all'interno di un framework di governance chiaro.

8. ROBUSTEZZA E GESTIONE DEI FALLIMENTI

8.1. REQUISITI DI ROBUSTEZZA E TOLLERANZA AI GUASTI

La progettazione della robustezza è orientata a garantire che il sistema mantenga un comportamento **coerente, osservabile e controllato** anche in presenza di malfunzionamenti parziali, con particolare attenzione alle dipendenze esterne legate al **LLM Gateway** e al **LLM Engine**.

I requisiti principali possono essere sintetizzati nei seguenti punti:

- **Continuità del monitoraggio locale**
  Il flusso sensore → MQTT → MAS → backend deve rimanere operativo anche in caso di:
  - indisponibilità del LLM Engine;
  - errori di comunicazione tra MAS e LLM Gateway;
  - output non validi del modello.

  Il sistema deve poter continuare a ricevere, valutare e registrare eventi, anche in modalità degradata.
- **Degrado controllato delle funzionalità avanzate**
  Le funzionalità che dipendono dal LLM (decisioni di escalation raffinate, piani di coordinamento multi-quartiere) devono degradare in modo **prevedibile**:
  - mantenendo la capacità di prendere decisioni tramite regole deterministiche;
  - evitando stalli o comportamenti silenti;
  - evitando l'emissione di azioni incoerenti o parzialmente informate.
- **Isolamento dei guasti LLM dal resto dell'architettura**
  Guasti del LLM Engine o del LLM Gateway non devono propagarsi:
  - né verso il broker MQTT;
  - né verso il backend e la dashboard;
  - né verso altri sottosistemi del MAS non coinvolti nelle decisioni LLM.

  L'impatto deve rimanere confinato ai punti di integrazione (decide_escalation, plan_coordination).
- **Esplicitazione degli errori e tracciabilità dei fallback**
  Ogni fallimento nella catena di chiamate LLM deve essere:
  - rilevabile e loggato (tipologia, messaggio, contesto);
  - reso visibile agli agenti sotto forma di errore strutturato;
  - riflesso, quando rilevante, nelle azioni registrate, attraverso motivazioni che evidenzino l'uso di regole di fallback.

- **Assenza di dipendenze rigide dal LLM per la sicurezza operativa**
  Le decisioni che riguardano la mitigazione di eventi critici non devono dipendere esclusivamente dal LLM.
  Il sistema deve sempre disporre di **strategie alternative** basate su soglie e regole deterministiche, in grado di garantire un comportamento prudente anche in caso di indisponibilità prolungata del modello.

8.2. GESTIONE DEGLI ERRORI DI COMUNICAZIONE CON LLM GATEWAY E LLM ENGINE

La comunicazione verso il LLM è articolata su due livelli distinti:

1. **MAS → LLM Gateway**
2. **LLM Gateway → LLM Engine**

Ognuno di questi livelli è dotato di meccanismi specifici di gestione degli errori.

Errore MAS → LLM Gateway

Quando un agente (DistrictMonitoringAgent o CityCoordinatorAgent) invoca un endpoint del LLM Gateway:

- la chiamata HTTP è effettuata tramite un client con:
  - **timeout espliciti**;
  - gestione delle eccezioni di rete (connessione rifiutata, host non raggiungibile);
  - interpretazione dei codici di stato HTTP.
- i possibili casi gestiti includono:
  - timeout lato MAS;
  - codici HTTP 5xx (errore lato gateway);
  - codici HTTP 4xx (errore di richiesta, ad esempio payload non valido);
  - impossibilità di risolvere il nome del servizio nella rete Docker.

In caso di errore, il MAS non riceve un'eccezione grezza, ma una **rappresentazione strutturata del fallimento**, ad esempio un oggetto con campi:

- `error_type` (es. `"network_error"`, `"timeout"`, `"gateway_5xx"`);
- `error_message` sintetico;
- eventuali dettagli diagnostici loggati ma non esposti all'esterno.

Questa codifica permette agli agenti di distinguere chiaramente:

- errori temporanei di comunicazione;
- errori logici di richiesta;
- indisponibilità generale del gateway.

Errore LLM Gateway → LLM Engine

All'interno del LLM Gateway, la comunicazione verso il LLM Engine è gestita separatamente:

- il gateway costruisce il prompt e invia una richiesta HTTP al servizio LLM, con:
    - timeout dedicato (`LLM_TIMEOUT_SECONDS`);
    - gestione di errori di connessione, DNS, handshake, ecc.;
    - verifica del codice di risposta HTTP (ad esempio 200 vs 5xx).
- i fallimenti considerati comprendono:
    - **timeout** nella risposta del modello;
    - **errori di rete** (connessione impossibile, host non raggiungibile);
    - **errori applicativi** (HTTP 5xx);
    - **risposte non parse-abili** o non conformi al JSON atteso.

Il gateway intercetta questi errori e:

- li registra nei log, con dettaglio del contesto (tipo di richiesta, distretti coinvolti, timestamp);
- li traduce in errori strutturati verso il MAS, senza propagare dettagli interni del motore LLM;
- evita sempre di restituire dati parzialmente validi o incoerenti.

Questa doppia barriera (validazione nel gateway e gestione degli errori lato agente) consente di gestire separatamente:

- i problemi di collegamento tra MAS e gateway;
- i problemi specifici di inferenza o infrastruttura del LLM Engine.

8.3. POLITICHE DI FALLBACK LATO DistrictMonitoringAgent

Il **DistrictMonitoringAgent** è il primo attore che può subire l'impatto dell'indisponibilità del LLM e, di conseguenza, è dotato di politiche di fallback esplicite per preservare la capacità di gestire gli eventi.

Le politiche sono articolate su tre casi principali:

- **Caso 1 – Evento chiaramente non critico**
  Se la severità preliminare e le regole deterministiche locali indicano in modo univoco che l'evento è non critico:
    - l'agente **non interroga** il LLM;
    - registra l'evento nel backend come evento ordinario;
    - non genera né escalation né azioni correttive.

In questo caso, l'indisponibilità del LLM è irrilevante.

- **Caso 2 – Evento chiaramente critico oltre soglia**
  Se i valori dell'evento superano soglie definite come estremamente critiche (ad esempio valori fuori scala o condizioni evidentemente anomale):
    - l'agente può decidere di **escalare direttamente** al CityCoordinatorAgent, senza attendere il contributo del LLM;
    - la severità è marcata come elevata o critica secondo le regole interne;
    - viene inviata una richiesta `ESCALATION_REQUEST` con una motivazione che esplicita il ruolo delle soglie deterministiche.

  Anche in questo caso, il LLM viene bypassato per ridurre latenza e dipendenze.

- **Caso 3 – Evento in zona di incertezza con LLM non disponibile o non affidabile**
  Questo è il caso in cui il LLM sarebbe normalmente interrogato, ma il LLM Gateway restituisce un errore (timeout, network error, invalid output).
  La politica di fallback prevede:
    - **Rilevamento dell'errore**
      L'agente riceve una risposta strutturata che indica il fallimento (es. `error_type = "timeout"`).
    - **Applicazione di regole deterministiche di backup**
      L'agente applica una combinazione di:
        - soglie basate sul valore corrente;
        - analisi del numero e della severità di eventi recenti;
        - eventuali contatori di eventi ripetuti oltre un certo intervallo temporale.
          Sulla base di queste regole, l'agente decide se:
        - generare un'escalation prudenziale;
        - mantenere la gestione al solo livello di quartiere.
    - **Annotazione della modalità di decisione**
      Se viene generata un'azione o un'escalation:
        - il campo reason delle azioni registrate potrà includere un riferimento esplicito al fallback, ad esempio:
            - `"fallback_rule: escalation without llm"`;
            - `"llm_error: timeout, applied threshold-based policy"`.
        - ciò consente, in fase di analisi, di distinguerle da azioni basate su raccomandazioni LLM.

In tutte le situazioni, il DistrictMonitoringAgent deve rimanere in grado di **produrre una decisione**, anche se meno raffinata, evitando di lasciare eventi ambigui non trattati a causa di problemi nel sottosistema LLM.

8.4. POLITICHE DI FALLBACK LATO CityCoordinatorAgent

Il **CityCoordinatorAgent** è responsabile della vista globale e del coordinamento multi-quartiere. Le politiche di fallback sono orientate a preservare un comportamento prudente in caso di eventi critici, anche quando la pianificazione avanzata tramite LLM non è disponibile.

Le principali modalità di fallback sono:

- **Piano deterministico di base senza LLM**

  Quando `plan_coordination` non è disponibile o restituisce un errore, il coordinatore può adottare un **schema di coordinamento semplificato**, ad esempio:

  - limitare l'azione al solo quartiere critico, senza ridistribuire il carico verso altri distretti;
  - applicare regole di default preconfigurate, come:
    - marcature di allerta per il quartiere critico;
    - azioni conservative (riduzione della capacità, limitazioni locali), senza coinvolgere infrastrutture di distretti adiacenti.

  Questo approccio riduce il rischio di generare azioni non ponderate su altri quartieri quando lo stato globale non è stato analizzato dal LLM.

- **Bloccare il piano multi-quartiere e registrare la criticità**

  In scenari in cui il contesto è particolarmente complesso e la pianificazione richiede una valutazione globale che non si ritiene opportuno svolgere con sole regole deterministiche, il coordinatore può:

  - evitare del tutto di emettere azioni di coordinamento multi-quartiere;
  - registrare nel backend:
    - l'evento critico;
    - l'informazione che il piano LLM non è stato disponibile;
    - eventuali azioni minime di mitigazione locale.

  Questa strategia privilegia la **trasparenza** rispetto all'adozione di piani potenzialmente inadeguati.

- **Annotazione sistematica della modalità di decisione**

  Le azioni generate in assenza di LLM vengono registrate in `actions` con motivazioni che esplicitano la strategia utilizzata, ad esempio:

  - `"fallback_rule: local-only mitigation"`

- `"llm_error: coordination plan unavailable"`

Ciò permette, a posteriori, di:

- distinguere chiaramente le decisioni derivate da piani LLM da quelle adottate per fallback;
- valutare se i piani deterministici siano stati sufficienti o se sia opportuno affinare le politiche di fallback.

L'obiettivo è garantire che il CityCoordinatorAgent **non generi mai piani di coordinamento arbitrari** in assenza di un'adeguata base informativa, mantenendo invece una condotta prudenziale e tracciabile.

8.5. DIAGRAMMA DI SEQUENZA – SCENARIO DI FALLIMENTO LLM E COMPORTAMENTO DI FALLBACK

Lo **scenario di fallimento LLM** è rappresentato da un diagramma di sequenza dedicato, che descrive in modo puntuale l'interazione tra i principali attori: simulatori, broker MQTT, MAS, LLM Gateway, LLM Engine, backend.

La sequenza può essere riassunta nei seguenti passaggi logici:

1. **Generazione e pubblicazione dell'evento**
   - Il simulatore (`SensorSimulator`) produce un evento critico o potenzialmente critico per un determinato quartiere.
   - L'evento viene pubblicato sul broker MQTT (`mqtt-broker`) sul topic `city/<district>/<sensor_type>`.
2. **Ricezione nel MAS e inoltro all'agente di quartiere**
   - Il listener MQTT di `mas-core` riceve il messaggio e lo trasforma in un evento strutturato.
   - Il router interno indirizza l'evento al `DistrictMonitoringAgent` competente.
3. **Decisione di consultare il LLM**
   - L'agente valuta l'evento e lo colloca in una zona di incertezza.
   - Viene costruita una richiesta `EscalationDecisionRequest` e inviata a `LLM` Gateway tramite `POST /llm/decide_escalation`.
4. **Chiamata fallita verso il LLM Engine**
   - Il LLM Gateway riceve la richiesta e tenta di chiamare il LLM Engine.
   - Si verifica un fallimento (timeout, errore di rete o output non valido).
   - Il gateway logga l'errore e restituisce al MAS una risposta strutturata di errore (`error_type`, `error_message`).
5. **Attivazione del fallback lato DistrictMonitoringAgent**
   - Il DistrictMonitoringAgent riceve l'errore dal LLM Gateway.

- Attiva le regole deterministiche di fallback per decidere se escalare o meno.
- Se viene decisa l'escalation:
  - invia un messaggio `ESCALATION_REQUEST` al CityCoordinatorAgent;
  - registra in `actions` o nei log locali la motivazione che indica la mancata disponibilità del LLM.

6. **Eventuale coinvolgimento del CityCoordinatorAgent**
   - Il CityCoordinatorAgent riceve l'eventuale richiesta di escalation.
   - Tenta, se configurato, di chiamare `LLM Gateway` su `/llm/plan_coordination` per generare un piano multi-quartiere.
   - Se anche questa chiamata fallisce, attiva le proprie politiche di fallback:
     - applica un piano deterministico ridotto;
     - oppure limita l'intervento al solo quartiere critico, registrando la mancata disponibilità LLM.
7. **Persistenza e visualizzazione**
   - Gli eventi vengono registrati in `events`.
   - Le azioni (sia locali sia di coordinamento in fallback) vengono registrate in `actions` con:
     - motivazioni che fanno riferimento a `fallback_rule` o `llm_error`;
     - `event_snapshot` del contesto.
   - La dashboard riflette l'intera catena di eventi e azioni, consentendo all'operatore di riconoscere rapidamente che le decisioni sono state prese **senza supporto LLM**.

Il diagramma rende evidente che:

- il fallimento del LLM è gestito come parte normale del flusso decisionale;
- la logica di fallback è esplicitamente codificata negli agenti;
- la tracciabilità del comportamento è preservata in ogni passaggio.

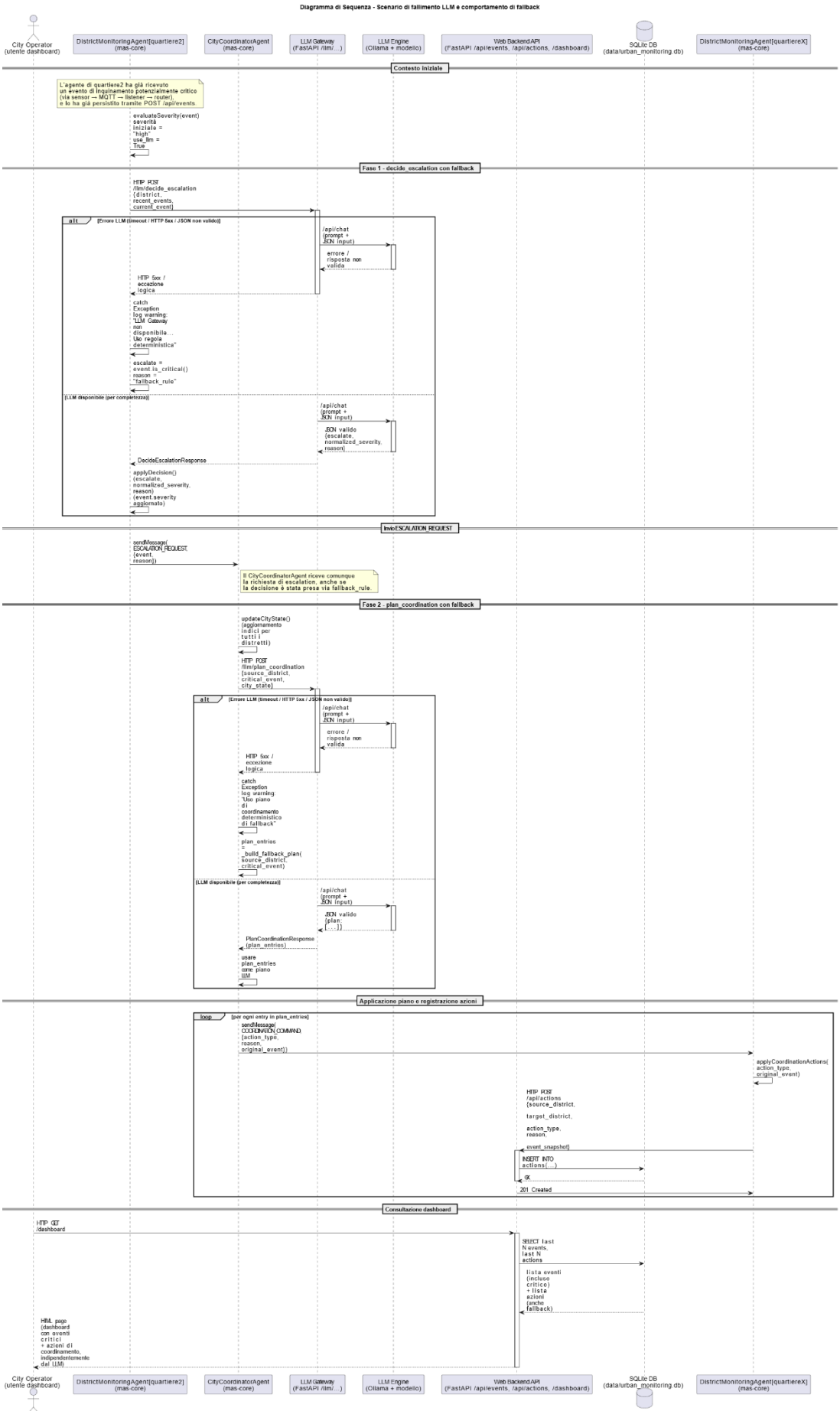


Diagramma di Sequenza - Scenario di fallimento LLM e comportamento di fallback
City Operator (utente dashboard)
DistrictMonitoringAgent[quartiere2] (mas-core)
CityCoordinatorAgent (mas-core)
LLM Gateway (FastAPI /llm/...)
LLM Engine (Ollama + modello)
Web Backend API (FastAPI /api/events, /api/actions, /dashboard)
SQLite DB (data/urban_monitoring.db)
DistrictMonitoringAgent[quartiereX] (mas-core)
Contesto iniziale
L'agente di quartiere2 ha già ricevuto un evento di inquinamento potenzialmente critico (via sensor → MQTT → listener → router) e lo ha già persistito tramite POST /api/events.
evaluateSeverity(event)
severità iniziale = "high"
use_llm = True
Fase 1 - decide_escalation con fallback
HTTP POST /llm/decide_escalation {district, recent_events, current_event}
alt
[Errore LLM (timeout / HTTP 5xx / JSON non valido)]
/api/chat (prompt + JSON input)
errore / risposta non valida
HTTP 5xx / eccezione logica
catch Exception log warning: "LLM Gateway non disponibile... Uso regola deterministica"
escalate = event.is_critical()
reason = "fallback_rule"
[LLM disponibile (per completezza)]
/api/chat (prompt + JSON input)
JSON valido {escalate, normalized_severity, reason}
DecideEscalationResponse
applyDecision() {escalate, normalized_severity, reason} (event.severity aggiornato)
Invio ESCALATION_REQUEST
sendMessage(ESCALATION_REQUEST, {event, reason})
Il CityCoordinatorAgent riceve comunque la richiesta di escalation, anche se la decisione è stata presa via fallback_rule.
Fase 2 - plan_coordination con fallback
updateCityState() (aggiornamento indici per tutti i distretti)
HTTP POST /llm/plan_coordination {source_district, critical_event, city_state}
alt
[Errore LLM (timeout / HTTP 5xx / JSON non valido)]
/api/chat (prompt + JSON input)
errore / risposta non valida
HTTP 5xx / eccezione logica
catch Exception log warning "Uso piano di coordinamento deterministico di fallback"
plan_entries = _build_fallback_plan(source_district, critical_event)
[LLM disponibile (per completezza)]
/api/chat (prompt + JSON input)
JSON valido {plan: [...]}
PlanCoordinationResponse {plan_entries}
usare plan_entries come piano LLM
Applicazione piano e registrazione azioni
loop
[per ogni entry in plan_entries]
sendMessage(COORDINATION_COMMAND {action_type, reason, original_event})
applyCoordinationActions(action_type, original_event)
HTTP POST /api/actions {source_district, target_district, action_type, reason, event_snapshot}
INSERT INTO actions(...)
OK
201 Created
Consultazione dashboard
HTTP GET /dashboard
SELECT last N events, last N actions
lista eventi (incluso critico) + lista azioni (anche fallback)
HTML page (dashboard con eventi critici + azioni di coordinamento, indipendentemente dal LLM)

8.6. VALUTAZIONE COMPLESSIVA DELL'AFFIDABILITÀ DEL SISTEMA

L'affidabilità del sistema, con particolare riferimento all'integrazione del LLM, può essere valutata lungo tre dimensioni principali: **continuità operativa**, **controllo del rischio** e **trasparenza del comportamento**.

- **Continuità operativa**

  Il sistema è in grado di:

  - continuare a ricevere e registrare eventi in tutti gli scenari;
  - mantenere una capacità di decisione autonoma a livello di quartiere mediante regole deterministiche;
  - operare anche in presenza di indisponibilità prolungata del LLM Engine o del LLM Gateway.

  Il contributo del LLM è trattato come **opzionale e arricchente**, non come dipendenza rigida per il funzionamento di base.

- **Controllo del rischio**

  La presenza di fallback strutturati:

  - evita che decisioni critiche vengano demandate completamente a un componente potenzialmente instabile o non deterministico;
  - riduce il rischio di azioni incoerenti in caso di output non validi;
  - limita l'impatto dei guasti LLM all'area funzionale direttamente collegata (escalation raffinata, pianificazione avanzata), lasciando intatti gli altri flussi.

  Le azioni adottate in fallback sono progettate per essere **conservative**, privilegiando comportamenti prudenti (escalation precauzionali o mitigazioni locali) anziché scelte aggressive o basate su informazioni incomplete.

- **Trasparenza e auditabilità**

  L'uso sistematico di:

  - log strutturati nel LLM Gateway e nel MAS;
  - campi `reason` nelle azioni che esplicitano l'uso di LLM o di regole di fallback;
  - `event_snapshot` persistiti nel backend;

  consente di ricostruire con precisione:

  - se un dato comportamento è stato influenzato dal LLM o meno;
  - in quale punto della catena si è verificato un eventuale guasto;
  - quali strategie di fallback sono state attivate e con quali effetti.

Rimangono naturalmente alcune potenziali fonti di indisponibilità (ad esempio arresto del backend o del broker MQTT) che esulano dall'integrazione LLM, ma all'interno del perimetro considerato il sistema mostra caratteristiche di **affidabilità coerenti con un**

contesto in cui un componente di AI generativa viene integrato come supporto decisionale, senza compromettere la governabilità del comportamento complessivo.

9. DETTAGLI DI IMPLEMENTAZIONE

9.1. STRUTTURA DEL REPOSITORY E ORGANIZZAZIONE DEI MODULI

L'implementazione è organizzata come un **sistema multi-servizio** contenuto in un unico repository, strutturato per riflettere i diversi ruoli architetturali:

- una sezione dedicata alla **simulazione dei sensori** e alla produzione di eventi IoT;
- una sezione dedicata al **core multi-agente** (MAS) e alle sue logiche di coordinamento;
- una sezione per il **backend web** e la **dashboard** di osservazione;
- una sezione per il **LLM Gateway** che funge da frontiera verso il modello linguistico;
- la configurazione di **Docker Compose** per il deployment locale.

A livello logico, la struttura può essere schematizzata come segue:

- `docker/`
  Contiene il file `docker-compose.yml` e, ove necessario, file di configurazione aggiuntivi (ad esempio per il broker MQTT o per parametri specifici dei servizi). Questa cartella rappresenta il **punto di orchestrazione** del sistema in ambiente containerizzato.
- `sims/`
  Contiene i **simulatori di sensori**, implementati in Python, insieme a moduli di supporto (configurazione, utilità di publishing MQTT).
  I file sono organizzati in modo da distingure i diversi scenari di simulazione (es. traffico, inquinamento) e da mantenere riutilizzabili le componenti comuni (ad esempio la logica di connessione al broker MQTT).
- `mas-core/`
  Contiene il **cuore del sistema multi-agente**, comprensivo di:
  - listener MQTT;
  - router interno degli eventi;
  - definizione delle classi agente (DistrictMonitoringAgent, CityCoordinatorAgent);
  - logiche di interazione con backend e LLM Gateway.
    La struttura interna è organizzata per moduli (ad esempio `mqtt_listener`, `router`, `agents`, `messages`, `services`) in modo da separare:
  - la gestione del trasporto dei messaggi;
  - la logica applicativa degli agenti;
  - i servizi di integrazione (HTTP client verso backend e gateway).

- `web-backend/`
  Contiene l'applicazione web basata su **FastAPI** (o framework equivalente), con:
  - modelli ORM (SQLAlchemy) per `Event` e `Action`;
  - schemi Pydantic per la validazione I/O;
  - router HTTP per le API (`/api/events`, `/api/actions`, eventuali endpoint di aggregazione/statistiche);
  - logica di rendering della **dashboard** (template HTML, asset statici).
- `llm_gateway/`
  Contiene il micro-servizio che integra il **LLM** nel sistema:
  - configurazione del client verso il motore LLM;
  - definizione degli schemi Pydantic per i payload di richiesta/risposta;
  - implementazione degli endpoint `/llm/decide_escalation` e `/llm/plan_coordination`.
- eventuali directory aggiuntive (es. `docs/`)
  Utilizzate per i **diagrammi architetturali** e di sequenza, nonché per materiale di supporto alla documentazione.

La struttura è pensata per ottenere:

- una chiara **separazione delle responsabilità** tra simulazione, MAS, backend e integrazione LLM;
- la possibilità di eseguire ciascun componente in isolamento durante le fasi di sviluppo e test;
- una mappatura diretta tra componenti architetturali e directory del repository.

9.2. IMPLEMENTAZIONE DEI SIMULATORI DI SENSORI E CONFIGURAZIONE MQTT

I simulatori di sensori, ospitati nel servizio `sims`, sono implementati in Python e utilizzano la libreria `paho-mqtt` per la pubblicazione degli eventi verso il broker MQTT.

Ogni simulatore rappresenta un **tipo di sensore** (ad esempio traffico o inquinamento) e genera eventi secondo una logica parametrizzabile:

- la **frequenza di emissione** è tipicamente regolata tramite un intervallo di sleep configurabile;
- i **valori generati** seguono andamenti pseudo-casuali o deterministici, eventualmente modulati per simulare:
  - picchi di traffico;
  - condizioni di inquinamento persistente;
  - variazioni diurne/notturne.

Gli eventi vengono pubblicati come messaggi JSON sul broker MQTT, con una struttura coerente con il modello `Event` (district, sensor_type, value, unit, severity preliminare, timestamp). I **topic MQTT** seguono una convenzione gerarchica del tipo:

- `city/<district>/<sensor_type>`

ad esempio:

- `city/quartiere1/traffic`
- `city/quartiere2/pollution`

tale convenzione consente al listener del MAS di sottoscriversi a pattern generali (`city/+/+`) e di instradare i messaggi verso gli agenti appropriati.

La **configurazione del broker MQTT** è delegata a un container dedicato (tipicamente basato sull'immagine `eclipse-mosquitto`), definito nel `docker-compose.yml`. I parametri principali includono:

- porte di ascolto (esposizione interna alla rete Docker e, opzionalmente, verso l'host);
- eventuali file di configurazione montati come volume (per definire permessi, autenticazione, ACL, se necessari);
- politiche di QoS utilizzate dai simulatori, generalmente **QoS 0** per semplicità, con assenza di retained message se non richiesto dal dominio applicativo.

I simulatori leggono i parametri di configurazione (host, port, topic base, intervallo di pubblicazione) da **variabili d'ambiente** o da file di configurazione dedicati, in modo da permettere un facile riuso degli stessi script in contesti diversi.

9.3. IMPLEMENTAZIONE DEL MAS (LISTENER MQTT, ROUTER, AGENTI, PERSISTENZA)

Il servizio `mas-core` ospita il **sistema multi-agente** vero e proprio. La sua implementazione è articolata in più componenti cooperanti:

Listener MQTT

Il listener MQTT è responsabile della **sottoscrizione** ai topic sensoriali e della trasformazione dei messaggi grezzi in eventi interni:

- utilizza `paho-mqtt` per connettersi al broker definito in Docker Compose (host e porta configurati via variabili d'ambiente);
- si sottoscrive a un pattern di topic del tipo `city/+/+`;
- per ogni messaggio ricevuto:
    - decodifica il payload JSON;
    - esegue un controllo di base sulla presenza dei campi necessari;
    - costruisce una struttura dati interna (ad esempio un dizionario o un oggetto Python) rappresentante l'evento;

- inoltra l'evento al **router interno** tramite una coda (es. `queue.Queue`).

In questo modo, il listener è **disaccoppiato** dalla logica degli agenti, limitandosi a garantire il collegamento tra MQTT e il dominio applicativo.

Router interno degli eventi

Il router funge da **punto di smistamento** tra il listener MQTT e gli agenti:

- consuma gli eventi dalla coda riempita dal listener;
- determina il **quartiere di destinazione** in base al campo `district` (o, in alternativa, al topic MQTT di origine);
- seleziona o istanzia il **DistrictMonitoringAgent** corrispondente al quartiere;
- inserisce nell'input queue dell'agente un messaggio di tipo "nuovo evento", contenente:
  - l'evento strutturato;
  - eventuali metadati aggiuntivi (es. timestamp di arrivo nel MAS).

Il router può mantenere un **registro degli agenti** (es. mappa `{district → agente}`) per evitare creazioni ripetute e per garantire la persistenza del contesto locale all'interno di ciascun agente.

Agenti di quartiere e coordinatore

Gli agenti sono implementati come oggetti con un proprio **loop di elaborazione** che consuma messaggi da una coda interna:

- ogni `DistrictMonitoringAgent`:
  - mantiene un **buffer di eventi recenti** e stato locale;
  - esegue la **valutazione di severità**;
  - decide se un evento sia non critico, critico locale o da escalare;
  - interagisce con:
    - il backend (`/api/events`, `/api/actions`) per persistenza;
    - il LLM Gateway (`/llm/decide_escalation`) quando previsto;
    - il CityCoordinatorAgent, tramite messaggi interni (`ESCALATION_REQUEST`).
- il `CityCoordinatorAgent`:
  - mantiene una vista **aggregata** dei quartieri (stato globale sintetico);
  - riceve richieste di escalation;
  - decide piani di coordinamento, eventualmente consultando il LLM Gateway (`/llm/plan_coordination`);

    - emette comandi di coordinamento verso i DistrictMonitoringAgent sotto forma di messaggi `COORDINATION_COMMAND`.

La gestione dei messaggi interni avviene tramite **code dedicate** per ciascun agente, consentendo una forma di concorrenza controllata e un chiaro modello di **event loop**.

Persistenza tramite backend web

Il MAS non interagisce direttamente con il database, ma delega la persistenza al **backend web**:

- per ogni evento da registrare viene inviata una richiesta HTTP `POST /api/events`, con payload conforme al modello `Event`;
- per ogni azione viene inviata una richiesta `POST /api/actions`, con payload conforme al modello `Action` (incluso `event_snapshot`).

L'implementazione delle chiamate HTTP utilizza librerie come requests o httpx, con:

- gestione dei timeout;
- gestione dei codici di risposta HTTP;
- logging degli errori, che alimentano le logiche di fallback descritte nelle sezioni precedenti.

9.4. IMPLEMENTAZIONE DEL BACKEND WEB E DELLA DASHBOARD

Il servizio `web-backend` implementa, in un'unica applicazione, sia le **API di persistenza e consultazione** sia la **dashboard** per la visualizzazione degli eventi e delle azioni.

Strato di persistenza (database e ORM)

La persistenza è gestita tramite **SQLite** e **SQLAlchemy**:

- `database.py` definisce:
    - l'engine di connessione al file SQLite (tipicamente montato su volume Docker per persistenza tra riavvii);
    - la `SessionLocal` per la gestione delle transazioni;
    - la Base ORM da cui derivano i modelli.
- `models.py` definisce le classi ORM:
    - `Event` con i campi descritti nel capitolo di modellazione (id, district, sensor_type, value, unit, severity, timestamp, topic, created_at);
    - `Action` con i campi id, source_district, target_district, action_type, reason, event_snapshot, created_at.
- `schemas.py` (o `schemas/*.py`) definisce gli **schemi Pydantic** per i DTO:
    - schemi di input per creazione di eventi/azioni;
    - schemi di output per restituzione verso MAS e dashboard.

- `crud.py` o moduli equivalenti incapsulano le operazioni di accesso al dato:
    - creazione di un evento;
    - creazione di un'azione;
    - interrogazione degli ultimi N eventi o azioni;
    - eventuali aggregazioni (es. conteggi per quartiere o per severità).

API REST

L'applicazione FastAPI espone endpoint come:

- `POST /api/events`
  Riceve un evento dal MAS, valida il payload tramite Pydantic, lo persiste tramite SQLAlchemy e restituisce l'oggetto creato o un identificativo.
- `GET /api/events`
  Restituisce una lista di eventi, con supporto a parametri di filtro (es. per quartiere, severità, intervallo temporale).
- `POST /api/actions`
  Analogamente per le azioni, con persistenza e restituzione dell'azione registrata.
- `GET /api/actions`
  Restituisce l'elenco delle azioni filtrabili per sorgente, target, tipo di azione, intervallo temporale.

Possono essere presenti endpoint aggiuntivi per **statistiche** (ad esempio `/api/stats/overview`) che aggregano dati su eventi e azioni.

Dashboard web

La dashboard è implementata con **template HTML** (ad esempio Jinja2) e asset statici (CSS, JavaScript):

- una vista principale può presentare:
    - tabella degli ultimi eventi, filtrabile per distretto e tipo di sensore;
    - tabella o pannello dedicato alle ultime azioni (con source/target, tipo di azione, reason);
    - indicatori sintetici (conteggio di eventi per severità, numero di azioni recenti, ecc.).
- eventuali script JavaScript possono invocare le API `/api/events` e `/api/actions` per aggiornare dinamicamente i dati, realizzando una dashboard **semi-real-time**.

La separazione tra API e viste consente:

- utilizzo del backend da parte sia del MAS sia della dashboard;
- futura estendibilità verso altre interfacce (ad esempio client esterni) senza modificare la logica interna.

9.5. IMPLEMENTAZIONE DEL LLM GATEWAY (API, SCHEMI PYDANTIC, CHIAMATE AL MODELLO)

Il servizio `llm_gateway` incapsula completamente l'interazione con il **modello linguistico**, mettendo a disposizione del MAS un'interfaccia REST stabile e tipizzata.

Configurazione e setup

- `config.py` definisce una classe di configurazione (ad esempio `LLMSettings`) che legge da variabili d'ambiente:
    - `LLM_API_BASE`: URL base del motore LLM (es. istanza Ollama o servizio HTTP compatibile);
    - `LLM_MODEL_NAME`: nome del modello da utilizzare;
    - `LLM_TIMEOUT_SECONDS`: timeout per le chiamate al modello.

La configurazione è caricata all'avvio dell'applicazione FastAPI, assicurando che i parametri siano **centralizzati e modificabili** senza intervenire sul codice.

Schemi Pydantic per I/O

- `schemas.py` definisce gli schemi per i payload:
    - per `/llm/decide_escalation`:
        - `EventContext`: rappresentazione strutturata dell'evento e del contesto locale;
        - `EscalationDecisionRequest`: wrapper che contiene `event` e `local_context`;
        - `EscalationDecisionResponse`: output atteso (severità normalizzata, booleano `should_escalate`, `rationale`).
    - per `/llm/plan_coordination`:
        - `CityState`: rappresentazione dello stato globale (distretti e relativi indicatori);
        - `CoordinationPlanRequest`: input contenente evento critico e stato città;
        - `CoordinationAction`: singola azione del piano (source_district, target_district, action_type, reason);
        - `CoordinationPlanResponse`: lista di `CoordinationAction`.

La validazione Pydantic garantisce che il MAS riceva solo **strutture coerenti e tipizzate**.

Chiamate al modello LLM

I metodi associati agli endpoint seguono uno schema comune:

1. Ricezione della richiesta del MAS, validata da Pydantic.

2. Costruzione del **prompt** e del payload verso il LLM:
    - serializzazione in JSON delle sezioni `event`, `local_context`, `city_state`;
    - definizione di istruzioni chiare sul formato della risposta (JSON con campi specifici, domini di valori limitati).
3. Invio della richiesta HTTP al motore LLM, utilizzando ad esempio `httpx`:
    - `POST` verso un endpoint come `/api/generate` o equivalente;
    - passaggio del modello, prompt e parametri di inferenza;
    - applicazione di un timeout.
4. Ricezione della risposta testuale del LLM:
    - individuazione del blocco JSON all'interno del testo;
    - parsing in oggetto Python;
    - validazione rispetto ai modelli di output Pydantic.
5. Ritorno della risposta normalizzata al MAS:
    - in caso di successo, restituzione di `EscalationDecisionResponse` o `CoordinationPlanResponse`;
    - in caso di errore, sollevamento controllato di un'eccezione HTTP (es. 503) con un corpo strutturato che indica `error_type` ed `error_message`.

Questa implementazione rende il LLM Gateway un componente **riutilizzabile** e **sostituibile**: cambiando `LLM_API_BASE` e `LLM_MODEL_NAME` è possibile utilizzare differenti back-end di inferenza senza alterare il codice del MAS.

9.6. CONFIGURAZIONE DI DOCKER COMPOSE E MODALITÀ DI DEPLOYMENT LOCALE

La composizione dei micro-servizi è orchestrata tramite un file `docker-compose.yml` situato nella directory `docker/`. Tale configurazione definisce i servizi principali:

- `mqtt-broker`
    - immagine basata su `eclipse-mosquitto`;
    - esposizione della porta MQTT interna alla rete Docker (e, opzionalmente, verso l'host);
    - montaggio di eventuali file di configurazione e volumi per la persistenza.
- `web-backend`
    - build da Dockerfile locale nella directory `web-backend`;
    - definizione di variabili d'ambiente per:
        - percorso del file SQLite;

        - parametri di logging;
    - dipendenza dal servizio che ospita il database (se separato) o uso di un file locale montato come volume.
- `llm-gateway`
    - build da Dockerfile in `llm_gateway`;
    - variabili d'ambiente `LLM_API_BASE`, `LLM_MODEL_NAME`, `LLM_TIMEOUT_SECONDS`;
    - dipendenza dal servizio che espone il motore LLM (se containerizzato) o utilizzo dell'host tramite host.docker.internal.
- `mas-core`
    - build da Dockerfile nella directory `mas-core`;
    - variabili d'ambiente per:
        - host e porta del broker MQTT;
        - base URL del `web-backend`;
        - base URL del `llm-gateway`;
    - dipendenze dichiarate verso `mqtt-broker`, `web-backend` e `llm-gateway` per garantire un ordine logico di avvio.
- `sims`
    - build da Dockerfile nella directory `sims` oppure uso di un'immagine Python generica con `sims` montato come volume;
    - variabili d'ambiente per:
        - host e porta del broker MQTT;
        - frequenze di pubblicazione;
        - elenco dei distretti e dei sensori attivi.

Tutti i servizi sono collegati a una **rete Docker comune**, che consente l'indirizzamento tramite i nomi di servizio (ad esempio `mqtt-broker`, `web-backend`, `llm-gateway`) senza dover conoscere indirizzi IP specifici.

La configurazione prevede:

- volumi per:
    - il file di database SQLite, in modo da preservare gli eventi e le azioni tra riavvii dei container;
    - eventuali log persistenti o configuration file del broker MQTT;

- mapping delle porte per l’accesso dall’host:
    - porta HTTP del `web-backend` per la dashboard (ad esempio `127.0.0.1:8000`);
    - eventuale esposizione del broker MQTT verso l’esterno, se necessario.

L’avvio locale dell’intero sistema è così centralizzato in un’unica orchestrazione: Docker Compose si occupa di **istanziare e connettere** tutti i servizi, permettendo di osservare:

- il flusso continuo degli eventi dai simulatori al MAS;
- la persistenza nel backend;
- la visualizzazione in dashboard;
- l’interazione con il LLM attraverso il gateway, il tutto all’interno di un ambiente isolato ma rappresentativo di un deployment reale.

10. VALIDAZIONE, OSSERVABILITA' E UTILIZZO DEL SISTEMA

10.1. MODALITÀ DI AVVIO E CONFIGURAZIONE DEL SISTEMA

L'avvio del sistema avviene tramite l'orchestrazione dei micro-servizi definita in **Docker Compose**, che mette a disposizione un ambiente uniforme per tutti i componenti: `mqtt-broker`, `sims`, `mas-core`, `web-backend`, `llm-gateway` ed eventuale container del motore LLM.

L'architettura di configurazione si basa su tre principi:

- uso sistematico di **variabili d'ambiente** per tutti i parametri operativi rilevanti;
- isolamento dei servizi all'interno della **rete Docker** comune;
- possibilità di abilitare/disabilitare singoli componenti (ad esempio il LLM) senza modificare il codice.

Gli elementi di configurazione principali sono:

- **Broker MQTT (`mqtt-broker`)**
  - Host e porta sono visibili agli altri container tramite il nome logico del servizio (ad esempio `mqtt-broker:1883`).
  - Opzionalmente vengono utilizzati file di configurazione per profili più avanzati (autenticazione, ACL), montati come volumi.
- **Servizio `sims` (simulatori)**
  - Parametri chiave:
    - indirizzo del broker MQTT (`MQTT_HOST`, `MQTT_PORT`);
    - frequenza di pubblicazione (intervallo tra le misure, ad esempio `PUBLISH_INTERVAL_SECONDS`);
    - elenco dei distretti simulati e dei tipi di sensore attivi.
  - I simulatori utilizzano tali parametri per inizializzare la connessione al broker e calibrare i pattern temporali e i valori generati.
- **Servizio `mas-core`**
  - Parametri fondamentali:
    - endpoint del broker MQTT (`MQTT_HOST`, `MQTT_PORT`);
    - endpoint del backend web (`WEB_BACKEND_BASE_URL`);
    - endpoint del LLM Gateway (`LLM_GATEWAY_BASE_URL`);
    - eventuali parametri di tuning per il polling interno o per la dimensione delle code di messaggi.

    - La configurazione governa:
        - le modalità di iscrizione ai topic sensoriali (`city/+/+`);
        - il comportamento dei client HTTP verso backend e gateway;
        - l'abilitazione o meno di alcune funzionalità (ad esempio utilizzo del LLM per specifici scenari).
- **Servizio** `web-backend`
    - Parametri principali:
        - percorso del file di database SQLite (montato come volume);
        - configurazione di logging (livello, formato, destinazione);
        - eventuali opzioni di CORS o di configurazione delle rotte.
    - Tali parametri determinano la persistenza dei dati e la disponibilità della dashboard via HTTP.
- **Servizio** `llm-gateway`
    - Parametri di configurazione:
        - `LLM_API_BASE`: URL del motore LLM;
        - `LLM_MODEL_NAME`: modello da utilizzare;
        - `LLM_TIMEOUT_SECONDS`: timeout per le chiamate al modello;
        - eventuali chiavi o token, se il motore LLM è esposto tramite API autenticata.
    - Queste impostazioni permettono di puntare a diversi back-end di inferenza senza modificare il MAS.

L'utilizzo del sistema da parte dell'operatore prevede tipicamente:

- l'avvio dell'intero stack tramite Docker Compose;
- l'accesso alla **dashboard** esposta dal `web-backend` tramite browser;
- la parametrizzazione delle simulazioni (ad esempio scenari di traffico moderato o critico) agendo sui parametri del servizio `sims`.

La scelta di centralizzare l'avvio e la configurazione in Docker Compose consente di riprodurre fedelmente l'ambiente di esecuzione e di garantire che tutti i servizi partano in uno stato coerente.

10.2. STRUMENTI DI OSSERVAZIONE E LOGGING

Il sistema utilizza una combinazione di **logging applicativo** e **strumenti di osservazione a livello di servizio** per permettere il monitoraggio operativo e l'analisi dei comportamenti.

Gli assi principali di osservabilità sono:

- **Logging dei micro-servizi**
    - Ogni servizio emette log strutturati con livelli (*info*, *warning*, *error*, eventualmente *debug*), che includono:
        - timestamp;
        - identificativo del servizio;
        - messaggio descrittivo;
        - eventuali dettagli di contesto (distretto, tipo di sensore, identificativo dell'evento).
    - In particolare:
        - `sims` logga la generazione dei messaggi MQTT e, in modalità più verbosa, i valori simulati;
        - `mas-core` logga:
            - arrivo e routing degli eventi;
            - decisioni di severità;
            - chiamate verso backend e LLM Gateway, con esito e tempi di risposta;
            - attivazione di politiche di fallback;
        - `web-backend` logga:
            - richieste HTTP ricevute;
            - operazioni di persistenza riuscite o fallite;
            - eccezioni interne;
        - `llm-gateway` logga:
            - richieste ricevute dal MAS;
            - invocazioni al motore LLM;
            - errori di rete, di formato o di validazione.
- **Dashboard web**
    - La dashboard costituisce lo strumento primario di **osservabilità funzionale**:
        - visualizzazione degli **eventi** con filtraggio per quartiere, severità, tipo di sensore;
        - visualizzazione delle **azioni** con evidenza dei legami source/target tra quartieri;

        - evidenziazione grafica di eventi critici e di azioni derivanti da escalation e coordinamento.
    - Attraverso la dashboard è possibile cogliere rapidamente:
        - la presenza di anomalie ricorrenti in specifici quartieri;
        - la frequenza di azioni di coordinamento;
        - l'effetto delle politiche di fallback.
- **Osservazione dell'infrastruttura Docker**
    - È possibile ispezionare lo stato dei container per verificare:
        - se tutti i servizi sono in esecuzione;
        - se alcuni container si riavviano frequentemente (sintomo di errori interni);
        - l'utilizzo di risorse (CPU, memoria) in caso di carichi più intensi.
- **Analisi dei log per la diagnosi**
    - La correlazione tra:
        - log dei servizi;
        - dati persistiti in `events` e `actions`;
        - comportamento visualizzato in dashboard;
          permette di condurre diagnosi mirate su:
        - malfunzionamenti del LLM;
        - problemi nella catena di persistenza;
        - eventuali anomalie nella simulazione.

L'insieme di questi strumenti rende possibile un ciclo completo di **osservazione → diagnosi → taratura** dei parametri del sistema (ad esempio soglie di severità, timeout del LLM, frequenza dei simulatori).

10.3. STRATEGIE DI TEST SUGLI SCENARI PRINCIPALI

Le strategie di test sono costruite attorno ai tre scenari funzionali chiave:

- gestione di **eventi non critici**;
- gestione di **eventi critici** con LLM attivo;
- gestione di **eventi critici** in condizioni di LLM non disponibile, con attivazione delle politiche di fallback.

Ogni scenario viene testato seguendo principi comuni:

- **controllo dell'input** tramite configurazione dei simulatori (`sims`), in modo da generare pattern di eventi coerenti con lo scenario;

- **osservazione multi-livello**:
    - dashboard per la vista funzionale;
    - log dei servizi per la vista operativa;
    - database per la verifica della persistenza (eventi e azioni);
- **criteri di verifica** precisi:
    - presenza o assenza di escalation;
    - presenza o assenza di azioni di coordinamento;
    - coerenza dei campi `reason` e `event_snapshot` nelle azioni;
    - corretta attivazione o meno delle chiamate verso il LLM Gateway.

10.3.1. TEST DEGLI SCENARI NON CRITICI

Gli scenari non critici sono progettati per verificare il comportamento del sistema in modalità di **monitoraggio ordinario**, senza attivazione di escalation o coordinamento multi-quartiere.

Gli step logici del test includono:

- **Configurazione dei simulatori**
    - impostazione dei simulatori in modo che:
        - generino valori di traffico e/o inquinamento chiaramente al di sotto delle soglie critiche;
        - non creino pattern di eventi che suggeriscano anomalie persistenti;
    - verifica nei log di sims che i valori generati siano coerenti con il profilo non critico desiderato.
- **Monitoraggio del MAS**
    - osservazione dei log di mas-core per verificare che:
        - gli eventi vengano ricevuti dal listener MQTT e instradati ai DistrictMonitoringAgent;
        - i DistrictMonitoringAgent valutino la severità come **bassa o moderata**;
        - non vengano generati messaggi di tipo `ESCALATION_REQUEST` verso il CityCoordinatorAgent;
        - non vengano effettuate chiamate a `/llm/decide_escalation` o `/llm/plan_coordination`.

- **Verifica sul backend e sulla dashboard**
    - interrogazione del backend (via API o tramite dashboard) per verificare che:
        - tutti gli eventi simulati risultino registrati nella tabella `events`;
        - non siano presenti record nella tabella `actions` legati a tali eventi;
        - la dashboard mostri i quartieri come monitorati senza marcature di criticità persistenti.
    - verifica della consistenza temporale (ordine degli eventi, timestamp coerenti, frequenza di inserimento).

Il criterio di accettazione per questo scenario è che il sistema:

- non effettui escalation;
- non generi azioni di coordinamento;
- continui a fornire una rappresentazione stabile e coerente del comportamento urbano nella modalità di funzionamento standard.

10.3.2. TEST DEGLI SCENARI CRITICI CON LLM ATTIVO

Gli scenari critici con LLM attivo sono pensati per validare la corretta interazione tra:

- simulazione di eventi ad alta severità;
- MAS (agenti di quartiere e coordinatore);
- LLM Gateway e motore LLM;
- backend e dashboard.

Le fasi di test comprendono:

- **Preparazione dell'ambiente**
    - verifica che il motore LLM sia effettivamente attivo e raggiungibile all'URL configurato in `LLM_API_BASE`;
    - verifica da log del `llm-gateway` che le richieste di prova al modello producano risposte ben formate.
- **Generazione di eventi critici**
    - configurazione dei simulatori per produrre:
        - valori di traffico o inquinamento oltre le soglie critiche in uno o più quartieri;
        - pattern di eventi che suggeriscano una situazione anomala persistente;
    - verifica, nei log di `sims`, della coerenza e ripetibilità dei valori critici simulati.

- **Osservazione del comportamento degli agenti**
    - nei log di `mas-core`:
        - verifica che i DistrictMonitoringAgent effettuino chiamate al LLM Gateway (`/llm/decide_escalation`) quando l'evento è in zona grigia o marcato come critico;
        - verifica che, ricevuta la risposta del LLM, l'agente:
            - aggiorni la severità normalizzata;
            - decida un'eventuale escalation, indicando il motivo legato al LLM;
        - verifica che il CityCoordinatorAgent:
            - riceva richieste `ESCALATION_REQUEST`;
            - invochi il LLM Gateway su `/llm/plan_coordination` per pianificare il coordinamento multi-quartiere;
            - traduca il piano in messaggi `COORDINATION_COMMAND` verso gli agenti target.
- **Verifica della persistenza e della visualizzazione**
    - controllo, nel backend:
        - che gli eventi critici siano registrati in `events` con severità coerenti con le decisioni LLM + regole;
        - che le azioni di coordinamento siano registrate in `actions` con:
            - `source_district` e `target_district` coerenti con il piano;
            - `action_type` appartenente ai valori ammessi;
            - `reason` che includa riferimenti al LLM (ad esempio prefissi o pattern concordati).
    - osservazione della dashboard per verificare che:
        - gli eventi critici siano evidenziati;
        - le azioni di coordinamento siano correttamente visualizzate e correlabili agli eventi origine.

L'obiettivo del test è verificare che:

- il LLM venga effettivamente utilizzato nei punti previsti;
- le sue raccomandazioni vengano trasformate in piani e azioni coerenti;
- il sistema rimanga trasparente rispetto al ruolo del LLM grazie alla tracciabilità in `actions` e nella dashboard.

10.3.3. TEST DEGLI SCENARI CON LLM NON DISPONIBILE (FALLBACK)

Gli scenari con LLM non disponibile sono fondamentali per validare le politiche di **robustezza e fallback** descritte nelle sezioni precedenti.

Le principali fasi di test sono:

- **Induzione controllata del guasto LLM**
  - disattivazione del motore LLM (ad esempio arresto del relativo container o servizio);
  - oppure impostazione intenzionalmente errata di `LLM_API_BASE` nel `llm-gateway`, in modo da simulare un errore di rete persistente;
  - verifica, nei log del llm-gateway, che le richieste in arrivo non possano essere inoltrate con successo e che vengano generati errori strutturati (error_type, error_message).
- **Generazione di eventi critici**
  - configurazione dei simulatori per produrre eventi analoghi a quelli del caso 10.3.2 (valori oltre soglia, situazioni anomale in uno o più quartieri).
- **Osservazione del comportamento degli agenti**
  - nei log di `mas-core`:
    - verifica che i DistrictMonitoringAgent tentino comunque la chiamata al LLM Gateway;
    - verifica che le risposte del gateway indichino in modo esplicito l'errore (`timeout`, `network_error`, `invalid_output`, ecc.);
    - verifica che, a fronte di tali errori, gli agenti attivino le **politiche di fallback**, ovvero:
      - applichino regole deterministiche basate su soglie per decidere l'escalation;
      - formulino richieste ESCALATION_REQUEST o evitino l'escalation in base alle regole interne, senza bloccare il flusso.
- **Controllo della persistenza e della motivazione delle azioni**
  - verifica in `actions` che:
    - le azioni generate in questa modalità riportino nel campo `reason` un'indicazione esplicita del fallback (esempi: stringhe che includano `fallback_rule` o `llm_error`);
    - gli `event_snapshot` associati permettano di comprenderne il contesto.
  - verifica in events che gli eventi critici continuino a essere registrati regolarmente, senza perdita dovuta al guasto del LLM.

- **Osservazione sulla dashboard**
    - controllo che:
        - la dashboard continui a visualizzare eventi e azioni;
        - il comportamento complessivo del sistema resti coerente (assenza di blocchi o inconsistenze funzionali), pur in assenza del contributo del LLM.

Questi test certificano che l'indisponibilità del LLM non pregiudica la **continuità operativa** del sistema e che le decisioni adottate in fallback rimangano tracciabili e distinguibili da quelle supportate dal modello.

10.4. EVIDENZE QUALITATIVE SUL COMPORTAMENTO DEL SISTEMA

Le osservazioni raccolte durante l'esecuzione degli scenari di test permettono di individuare alcune evidenze qualitative significative sul comportamento del sistema.

Dal punto di vista del **monitoraggio ordinario**, il sistema mostra una buona stabilità:

- gli eventi non critici vengono acquisiti e persistiti con continuità;
- la dashboard consente di interpretare in modo immediato l'andamento temporale dei fenomeni urbani;
- il MAS si comporta come filtro intelligente, evitando escalation inutili e mantenendo un carico decisionale proporzionato alla severità effettiva.

Per quanto riguarda gli **scenari critici con LLM attivo**, emergono alcuni elementi di interesse:

- le raccomandazioni prodotte dal LLM, incapsulate attraverso il LLM Gateway, consentono di articolare piani di coordinamento multi-quartiere che, nei casi osservati, risultano coerenti con le informazioni disponibili nei payload di input;
- l'interazione tra CityCoordinatorAgent e LLM produce una catena di decisioni che si traduce in azioni distribuite, registrate in `actions` con motivazioni esplicite;
- la dashboard rende percepibile il legame causa-effetto tra evento critico iniziale, consultazione del LLM, piano di coordinamento e azioni applicate.

Negli **scenari con LLM non disponibile**, le evidenze qualitative mostrano:

- la regolare attivazione dei meccanismi di fallback, sia a livello di quartiere sia a livello di coordinamento;
- l'assenza di blocchi nella pipeline sensore → MAS → backend → dashboard;
- la capacità del sistema di mantenere un comportamento prudente ma operativo, con decisioni basate su regole deterministiche quando il supporto del LLM non è accessibile.

Dal punto di vista dell'**esperienza dell'operatore**, la combinazione fra:

- una dashboard in grado di rappresentare eventi e azioni con adeguato livello di dettaglio;
- una tracciabilità puntuale delle decisioni (campi `reason`, `event_snapshot`, log del LLM Gateway e del MAS);

permette di comprendere non solo *che cosa* sia accaduto in un determinato intervallo temporale, ma anche *come* e *perché* il sistema abbia preso determinate decisioni, distinguendo le scelte supportate dal LLM da quelle derivate da fallback o da regole locali.

11. CONCLUSIONI

11.1. SINTESI DELLE PRINCIPALI SCELTE ARCHITETTURALI

L'architettura realizzata si fonda su una chiara **separazione delle responsabilità** e sull'adozione di tecnologie eterogenee, orchestrate in modo coerente per affrontare il problema del monitoraggio urbano e della gestione di eventi critici.

Al livello più esterno, i **simulatori di sensori** forniscono un flusso controllato di eventi, modellando differenti fenomeni urbani (ad esempio traffico, inquinamento) e veicolandoli verso il sistema tramite **MQTT**. L'uso di un broker dedicato consente di disaccoppiare la produzione degli eventi dalla loro elaborazione, introducendo un primo livello di robustezza e permettendo una facile estensione ad altre tipologie di sensori o di distretti.

Il **core del sistema multi-agente (MAS)** rappresenta il cuore logico dell'architettura. Il listener MQTT e il router interno costituiscono un layer di adattamento tra mondo IoT e mondo applicativo, trasformando i messaggi grezzi in eventi strutturati indirizzati ai **DistrictMonitoringAgent**. Ogni agente di quartiere mantiene una vista locale del proprio distretto e applica regole di valutazione della severità, integrando, quando opportuno, il contributo del LLM. Il **CityCoordinatorAgent** introduce una vista globale della città, consentendo di passare da decisioni puramente locali a strategie di coordinamento multi-quartiere.

La **persistenza applicativa** è delegata a un **backend web** dedicato, che espone API REST per la registrazione e la consultazione di eventi e azioni. L'uso combinato di SQLAlchemy e Pydantic garantisce coerenza tra modello dati persistito e contratti API, mentre la **dashboard web** rende osservabile il comportamento del sistema, consentendo all'operatore di interpretare in modo immediato la dinamica urbana e le azioni intraprese.

L'integrazione del **LLM** è incapsulata in un **LLM Gateway** autonomo, che funge da frontiera tra il dominio applicativo e il motore di inferenza. Gli endpoint dedicati alla decisione di escalation e alla pianificazione del coordinamento multi-quartiere operano su payload strettamente strutturati e restituiscono output validati, trattando il modello linguistico come un *consulente* piuttosto che come un elemento che controlla direttamente il flusso.

L'intero ecosistema è orchestrato tramite **Docker Compose**, che fornisce un ambiente uniformemente riproducibile per `sims`, `mqtt-broker`, `mas-core`, `web-backend`, `llm-gateway` e motore LLM. La scelta di una composizione multi-servizio containerizzata rende il sistema modulare, isolando i guasti e permettendo di eseguire e testare singoli componenti in modo indipendente.

11.2. VALUTAZIONE DEL COMPORTAMENTO DEL SISTEMA RISPETTO AI REQUISITI

Il comportamento osservato del sistema può essere confrontato con i requisiti funzionali e non funzionali individuati nelle fasi di analisi.

Sul piano **funzionale**, il sistema:

- è in grado di **acquisire eventi da sensori distribuiti** tramite MQTT, preservando le informazioni di contesto (quartiere, tipo di sensore, valore, severità preliminare, timestamp);
- realizza un **monitoraggio locale per quartiere** attraverso i DistrictMonitoringAgent, che valutano gli eventi in base a regole deterministiche e a contesti temporali recenti;
- distingue in modo esplicito tra:
    - **eventi non critici**, trattati come flusso di monitoraggio ordinario;
    - **eventi critici**, per i quali vengono attivati meccanismi di escalation e, se necessario, di coordinamento multi-quartiere;
- supporta **azioni locali e coordinate**:
    - azioni locali associate a singoli quartieri, originate da regole interne o da input del coordinatore;
    - azioni di coordinamento multi-distretto, originate da piani globali che possono essere supportati dal LLM;
- garantisce la **persistenza** degli eventi e delle azioni, abilitando analisi successive e tracciabilità decisionale.

Rispetto ai requisiti di integrazione con un **LLM**:

- il LLM viene utilizzato nei due punti decisionali chiave:
    - raffinamento della severità e decisione di escalation a livello di quartiere;
    - generazione di piani di coordinamento a livello cittadino;
- il contributo del modello linguistico è sempre mediato dal LLM Gateway, che impone schemi di input/output rigorosi, riducendo al minimo l'impatto di output non strutturati;
- le decisioni finali restano sotto il controllo esplicito degli agenti, che possono accettare o adattare le raccomandazioni ricevute.

Dal punto di vista **non funzionale**, il sistema mostra:

- un buon livello di **robustezza** grazie a:
    - isolamento dei servizi tramite container;
    - uso di politiche di fallback in caso di indisponibilità del LLM;
    - gestione esplicita degli errori nelle chiamate HTTP e nelle interazioni con il broker MQTT;
- un livello adeguato di **osservabilità**:
    - log strutturati nei diversi servizi;

    - dashboard che rende visibili eventi e azioni, distinguendo scenari ordinari da scenari critici;
    - persistenza di `event_snapshot` e `reason` nelle azioni, utile per analisi ex post;
- una **scalabilità logica** ragionevole, almeno sul piano architetturale:
    - la separazione tra simulazione, MAS, backend e LLM Gateway consente di estendere il numero di quartieri, di sensori o di piani di coordinamento senza modificare i principi di base dell'architettura.

Alcuni **vincoli e semplificazioni** restano tuttavia presenti:

- il modello urbano è deliberatamente semplificato, con un numero limitato di quartieri e di fenomeni monitorati;
- la simulazione dei sensori, pur flessibile, non riproduce la complessità di un ambiente reale (rumore, guasti del dispositivo, fenomeni stocastici più ricchi);
- l'integrazione con il LLM, pur rigorosa dal punto di vista dei contratti, non copre l'intero spettro di possibili errori o bias del modello, e dipende dalla qualità del modello sottostante.

Nel perimetro definito dal progetto e dalle sue assunzioni, il sistema soddisfa in modo coerente i requisiti identificati, fornendo un quadro operativo credibile per l'uso congiunto di MAS, IoT e LLM in contesto urbano simulato.

11.3. CONSIDERAZIONI FINALI

L'insieme delle componenti sviluppate dimostra la possibilità di costruire un sistema di **monitoraggio urbano multi-agente** che integri in modo strutturato:

- flussi IoT basati su **MQTT**;
- logiche distribuite di **percezione e decisione** a livello di quartiere e di città;
- un livello di **intelligenza aumentata** basato su LLM, confinato in un gateway dedicato;
- un'infrastruttura di **persistenza e osservabilità** che consente sia il monitoraggio in tempo reale sia l'analisi retrospettiva.

L'approccio adottato si caratterizza per alcune scelte metodologiche rilevanti:

- la volontà di mantenere il **MAS come attore centrale** del processo decisionale, evitando di delegare al LLM il controllo diretto del sistema;
- la definizione di **protocolli interni espliciti** (messaggi tra agenti, formato dei payload verso backend e gateway), che rendono verificabile ogni passaggio;
- la cura nella **modellazione dei dati** e nell'uso di schemi tipizzati per le API, riducendo gli spazi di ambiguità e favorendo l'estensibilità.

L’utilizzo del LLM all’interno di un contesto governato da regole e agenti consente di beneficiare delle capacità di ragionamento del modello linguistico, mitigandone al contempo alcune fragilità attraverso:

- la validazione sistematica degli output;
- la presenza di **fallback deterministici** in caso di errore o indisponibilità;
- la tracciabilità delle decisioni e delle motivazioni associate.

Il risultato complessivo è un sistema che, pur operando su un ambiente simulato e con ipotesi controllate, offre un **dimostratore concreto** di come tecniche di intelligenza artificiale basate su LLM possano essere integrate in architetture distribuite orientate agli eventi, senza rinunciare a requisiti di chiarezza, robustezza e governabilità del comportamento.