WP2: Realizzazione Infrastruttura Data-Lake MAPS

1 Requisiti e Vincoli del Data Lake

1.1 Requisiti del Data Lake

Il data lake costituisce l'infrastruttura centrale del progetto, progettata per supportare l'acquisizione, la standardizzazione e l'analisi di dataset eterogenei su scala nazionale. I requisiti si articolano in due categorie: funzionali e non funzionali.

1.1.1 Requisiti funzionali

RF1: Acquisizione Dati Eterogenei

Il sistema deve supportare l'acquisizione di oltre 180 dataset provenienti da fonti pubbliche diverse, come censiti nel catalogo dati del progetto. La distribuzione per ente vede ISTAT come fonte principale con 113 dataset (62%), seguito dal Ministero dell'Istruzione con 35 dataset (19%). Le fonti complementari includono Agenas per le strutture sanitarie (6 dataset), ISPRA per i dati ambientali (6 dataset), la Ragioneria Generale dello Stato (4 dataset), il Ministero della Salute (3 dataset), il Ministero della Giustizia (2 dataset) e altri 8 enti pubblici con contributi minori. La varietà dei formati di distribuzione (CSV, XLSX, HTML, PDF) richiede capacità di parsing differenziate per ciascuna tipologia.

RF2: Gestione Serie Storiche

La dimensione temporale copre il periodo 2010-2025, per un totale di 15 anni di osservazioni. La granularità di riferimento è il livello comunale, che comporta la gestione di circa 8.000 comuni italiani. Il sistema deve gestire le discontinuità temporali introdotte dalla pandemia COVID-19 nel biennio 2020-2021, durante il quale molte rilevazioni hanno subito interruzioni o modifiche metodologiche. Inoltre, deve tracciare le fusioni e separazioni comunali avvenute nel periodo attraverso un sistema di lookup storico.

RF3: Standardizzazione Dati

Il processo di standardizzazione prevede diverse operazioni critiche. I codici ISTAT devono essere normalizzati per garantire la coerenza dei riferimenti territoriali. Le coordinate spaziali vengono uniformate al sistema di riferimento EPSG:32632 (WGS84 / UTM zone 32N). Il sistema deve distinguere tra valori mancanti e nulli semantici, applicando strategie di gestione differenziate. La riconciliazione delle fusioni e separazioni comunali garantisce la continuità delle serie storiche anche in presenza di modifiche amministrative.

RF4: Pattern Medallion

L'architettura dei dati segue il pattern Medallion, articolato in tre livelli distinti. Il livello Bronze

conserva i dati grezzi immutabili, esattamente come acquisiti dalle fonti. Il livello Silver applica la standardizzazione attraverso uno schema EAV (Entity-Attribute-Value) che gestisce la variabilità degli attributi comunali nel tempo. Il livello Gold produce data mart analitici ottimizzati per le applicazioni business, con tabelle denormalizzate e geometrie PostGIS pronte per l'interrogazione spaziale. I dettagli dell'implementazione sono descritti nel capitolo 2.

1.1.2 Requisiti non funzionali

RNF1: Completeness

Il sistema deve garantire una copertura pari o superiore al 95% dei comuni italiani per ogni dataset prioritario. La metrica di riferimento è la percentuale di comuni con dati completi rispetto al totale dei comuni esistenti nel periodo di riferimento.

RNF2: Accuracy

L'accuratezza richiede che almeno il 99,9% dei record contenga un codice ISTAT valido. La validazione si basa sul confronto con il dataset di riferimento dei confini amministrativi ufficiali pubblicati da ISTAT.

RNF3: Timeliness

I dataset con frequenza di aggiornamento mensile devono essere processati entro 5 giorni dalla loro disponibilità. Le pipeline ETL prioritarie devono completare l'elaborazione con latenza inferiore alle 24 ore.

RNF4: Lineage

Ogni record deve essere tracciabile attraverso l'intera catena di trasformazione, dalla fonte originale al livello Bronze, quindi al livello Silver e infine al livello Gold. Il sistema deve mantenere il versionamento delle fonti dati per consentire la ricostruzione storica delle trasformazioni.

RNF5: Scalabilità

Il sistema è dimensionato per supportare un volume di dati fino a 1TB nell'arco del triennio di progetto. Il throughput delle pipeline ETL deve essere pari o superiore a 100 record al secondo per garantire tempi di elaborazione accettabili.

RNF6: Governance

La governance richiede la catalogazione completa di tutti i dataset con metadati strutturati. Una dashboard in tempo reale deve monitorare la qualità dei dati, evidenziando anomalie e scostamenti rispetto alle soglie definite. Tutte le operazioni su dati sensibili devono essere registrate in log completi per scopi di audit.

1.2 Dati critici per MVP (Fase 1a)

La fase iniziale del progetto (mesi 1-4) si concentra sull'acquisizione e validazione di 5 dataset critici, che costituiscono la base informativa minima per lo sviluppo degli algoritmi SLO. La tabella seguente riassume le caratteristiche principali di ciascun dataset.

Dataset	Fonte	Formato	Frequenza	Priorità
Confini comunali + metadata	ISTAT	Shapefile/JSON	Annuale	Alta
Popolazione residente	ISTAT	CSV	Annuale	Alta
Matrici pendolarismo	ISTAT	CSV	Decennale	Alta
Rete trasporto pubblico	GTFS	ZIP	Mensile	Media
Strutture sanitarie	Min. Salute	Excel	Annuale	Media

1.3 Vincoli architetturali

L'architettura del sistema è soggetta a quattro vincoli fondamentali, derivanti sia da scelte strategiche che da requisiti tecnici specifici.

Vincolo V1: Self-Hosted Open Source

La scelta di utilizzare esclusivamente tecnologie open source in modalità self-hosted risponde all'esigenza di indipendenza dai fornitori e di prevedibilità dei costi. Lo stack tecnologico comprende PostgreSQL per lo storage primario, PostGIS per le operazioni spaziali, Prefect per l'orchestrazione delle pipeline, OpenMetadata per la governance dei dati e DuckDB per le analisi embedded. Sono esplicitamente esclusi i servizi managed offerti da BigQuery, Azure e AWS, che introdurrebbero dipendenze da fornitori e costi variabili difficili da prevedere.

Vincolo V2: Adeguatezza di Scala

La volumetria prevista si attesta nell'ordine di grandezza di 10^4 righe per 10^3 colonne, tipica dei progetti gestiti da DEPP. A questa scala, l'adozione di BigQuery rappresenterebbe un eccesso di capacità (overkill) rispetto alle reali necessità, con conseguente spreco di risorse e complessità gestionale non giustificata.

Vincolo V3: Standard Territoriali

PostGIS costituisce lo standard industriale de facto per la gestione di dati spaziali in ambiente PostgreSQL. Il sistema di riferimento adottato è EPSG:32632 (WGS84 / UTM zone 32N), che rappresenta il sistema standard per il territorio italiano e garantisce la compatibilità con i dataset geografici nazionali.

Vincolo V4: Compliance FAIR

I dataset rilasciati devono rispettare i principi FAIR. Findable: i dati devono essere catalogati con metadati strutturati che ne facilitino la scoperta. Accessible: le licenze adottate (CC-BY, CC0) garantiscono l'accesso libero e gratuito. Interoperable: i formati di distribuzione (GeoJSON, GeoParquet, CSV) sono standard aperti che assicurano l'interoperabilità con altri sistemi. Reusable: l'assegnazione di DOI garantisce la citabilità scientifica e la tracciabilità delle versioni.

1.4 Sfide specifiche

Il progetto affronta quattro sfide tecniche significative, ciascuna delle quali richiede soluzioni architetturali specifiche.

Sfida S1: Assenza Geolocalizzazione Puntuale

La maggior parte dei servizi analizzati (scuole, ospedali, uffici postali) non dispone di coordinate geografiche precise. L'approccio adottato si basa sulla rilevazione di presenza o assenza del servizio a livello comunale, rinunciando alla geolocalizzazione puntuale. Di conseguenza, i calcoli di isocrone operano su distanze municipality-to-municipality piuttosto che point-to-point.

Sfida S2: Evoluzione Confini Amministrativi

Nel periodo 2010-2025 si sono verificate circa 100 fusioni o separazioni comunali, che modificano la struttura territoriale di riferimento. La soluzione adottata prevede un lookup storico con validità temporale implementato secondo il pattern Slowly Changing Dimension Type 2 (SCD Type 2). Lo schema Silver include le colonne `valid_from` e `valid_to` che delimitano l'intervallo di validità di ciascuna configurazione territoriale.

Sfida S3: Gap Temporali COVID

I dati relativi al biennio 2020-2021 presentano discontinuità metodologiche dovute alle restrizioni imposte dalla pandemia. Il sistema applica un flag `covid_affected` ai dataset impattati, consentendo strategie di analisi differenziate. Le opzioni includono tecniche di imputation statistica per stimare i valori mancanti o l'esclusione esplicita dei periodi problematici dalle analisi longitudinali.

Sfida S4: Eterogeneità Formati

La presenza di 207 dataset distribuiti in formati diversi (CSV, XLSX, HTML, PDF) richiede uno stack di estrazione multistrato. Docling gestisce l'estrazione da PDF complessi con tabelle strutturate. Pandas elabora i file CSV ed Excel. BeautifulSoup effettua il parsing di tabelle HTML. La normalizzazione finale avviene attraverso lo schema EAV flessibile del livello Silver, che assorbe la variabilità strutturale delle fonti.

1.5 Obiettivi del WP2

Al termine del Work Package 2, il data lake dovrà soddisfare cinque obiettivi misurabili:

1. Acquisire gli oltre 180 dataset censiti nel catalogo dati del progetto, garantendo la diversità delle tipologie e dei formati di ingresso.
2. Garantire una copertura pari o superiore al 95% dei comuni italiani per tutti i dataset prioritari identificati nella fase MVP.
3. Fornire tracciabilità completa attraverso il lineage Bronze-Silver-Gold per ogni singolo record presente nel sistema.
4. Validare le pipeline ETL attraverso report di data quality che documentino le metriche di completezza, accuratezza e tempestività.
5. Preparare l'infrastruttura dati necessaria all'implementazione degli algoritmi SLO previsti nel Work Package 3.

1.6 Output attesi (Deliverable)

Il Work Package 2 produce cinque deliverable documentali:

- **D2.1.1**: Documento di Progettazione Tecnica del Data Lake
- **D2.1.2**: Solution Design dell'Architettura Cloud
- **D2.1.3**: Specifiche Infrastruttura di Hosting
- **D2.2**: Script ETL (Python/SQL) documentati e testati
- **D2.3**: Report di Validazione delle Pipeline ETL

2 Architettura tecnica Data-Lake

Deliverable D2.1.1: Documento Progettazione Tecnica Data-Lake

2.1 Pattern Medallion: Bronze → Silver → Gold

2.1.1 Visione d'insieme

L'architettura del Data Lake MAPS adotta il **pattern Medallion** (anche denominato Multi-Hop Architecture), best practice nell'ecosistema Modern Data Stack, che organizza i dati in **tre layer di progressivo raffinamento** con crescente livello di qualità, pulizia e business-readiness:

```
FONTI ESTERNE → BRONZE → SILVER → GOLD → APPLICAZIONI
```

2.1.2 Motivazioni della scelta

Per il progetto MAPS, l'architettura Medallion è particolarmente indicata per cinque motivazioni principali.

L'**eterogeneità delle fonti dati** costituisce il primo fattore determinante. Il progetto gestisce oltre 180 dataset provenienti da fonti pubbliche diverse (ISTAT, Ministeri, enti territoriali), distribuiti in formati multipli (CSV, Excel, PDF, JSON, HTML) con qualità variabile che spazia da dataset strutturati a documenti semi-strutturati. La stratificazione in layer progressivi permette di standardizzare gradualmente questa eterogeneità senza perdere le informazioni originali.

I **requisiti di audit e compliance** rappresentano il secondo elemento critico. Sebbene i dati siano pubblici, la ricerca scientifica richiede tracciabilità completa della provenienza e delle trasformazioni applicate. L'articolo 30 del GDPR impone inoltre la documentazione delle attività di trattamento dati. Il livello Bronze immutabile fornisce la "source of truth" originale necessaria per soddisfare questi requisiti.

La **complessità delle trasformazioni** richiede un'architettura articolata. Le pipeline operano in modalità multi-step, passando dall'acquisizione da fonti eterogenee (CSV, Excel, PDF, JSON, HTML) alla validazione, quindi alla normalizzazione in schema EAV e infine alle aggregazioni territoriali. La gestione di serie storiche 2010-2025 con fusioni e scissioni comunali aggiunge ulteriore complessità. La separazione logica tra acquisizione grezza (Bronze), pulizia (Silver) e business logic (Gold) semplifica lo sviluppo e il debugging delle trasformazioni.

Il **riutilizzo dei dati** giustifica l'introduzione di un layer intermedio validato. Gli stessi dataset vengono impiegati per analisi multiple (demografia, servizi, mobilità), rendendo inefficiente il reprocessing dalla fonte esterna ad ogni utilizzo. Il livello Silver funge da cache enterprise validata, accelerando le analisi downstream.

Performance e scalabilità costituiscono l'ultimo fattore abilitante. Le query analitiche operano su oltre 8.000 comuni con decine di attributi ciascuno, mentre le operazioni spaziali PostGIS risul-

Mermaid Diagram

Figure 1: Mermaid Diagram

tano computazionalmente intensive. Il livello Gold denormalizzato ottimizza le prestazioni delle interrogazioni frequenti, riducendo i tempi di risposta da secondi a millisecondi.

2.1.3 Data flow completo

I dati transitano dalle fonti esterne attraverso tre layer progressivi, orchestrati da Prefect. Il Bronze layer conserva i file originali; il Silver layer li normalizza in schema EAV su PostgreSQL; il Gold layer produce le tabelle analitiche con estensioni spaziali PostGIS. I dati pubblicati raggiungono il portale nazionale tramite CKAN, mentre OpenMetadata traccia la governance lungo l'intero flusso.

2.1.4 Bronze Layer: Raw Data Archive

Ruolo: Archivio immutabile dei dati originali esattamente come scaricati dalla fonte.

Principio chiave: "Non modificare mai, conservare sempre"

2.1.4.1 Caratteristiche tecniche

Aspetto	Specifica MAPS
Storage	File system locale `/data/bronze/` (server op-linkurious)
Formato	File originali senza trasformazioni (CSV, XLSX, PDF, JSON, HTML, Parquet)
Convenzione di nomenclatura	`/data/bronze/{fonte}/{anno}/{dataset}_{timestamp}.{ext}`
Politica di conservazione	Indefinita (storage cost è basso: ~50GB totali per 180 dataset)
Backup	Snapshot giornalieri via `backup.sh` script
Modalità di accesso	Write-once, read-rarely (solo per reprocessing o audit)

2.1.4.2 Struttura della Directory

```
/data/bronze/
+-- istat/
|   +-- 2024/
|   |   +-- popolazione_comuni_20240218.csv
|   |   +-- pendolarismo_matrix_20240218.csv
|   |   +-- confini_amministrativi_20240218.geojson
|   |   +-- _metadata/
|   |       +-- popolazione_comuni_20240218.json   # Metadata file
|   |       +-- checksums.sha256
|   +-- 2023/
|       +-- popolazione_comuni_20230315.csv
+-- minlavoro/
|   +-- 2024/
|       +-- tabacchi_adm_report_20240218.pdf
```

```
|       +-- _metadata/
|           +-- tabacchi_adm_report_20240218.json
+-- minsalute/
|   +-- 2023/
|       +-- strutture_sanitarie_asl_20231120.xlsx
|       +-- _metadata/
|           +-- strutture_sanitarie_asl_20231120.json
+-- minambiente/
    +-- 2024/
        +-- ato_gas_page_20240115.html          # ← HTML file
        +-- _metadata/
            +-- ato_gas_page_20240115.json
```

2.1.4.3 Tracciamento dei metadati Il tracciamento avviene su due livelli complementari: un file JSON locale affiancato a ciascun file acquisito, e una tabella centralizzata su PostgreSQL che registra lo stato di tutte le acquisizioni.

Ogni file Bronze ha un corrispondente file JSON con i metadati di acquisizione:

Esempio: `/data/bronze/istat/2024/_metadata/popolazione_comuni_20240218.json`

```
{
  "file_path": "/data/bronze/istat/2024/popolazione_comuni_20240218.csv",
  "fonte": "istat",
  "dataset": "popolazione",
  "anno_riferimento": 2024,
  "download_timestamp": "2024-02-18T03:00:15Z",
  "download_url": "https://www.istat.it/storage/cartografia/popolazione_comuni_2024.csv",
  "file_size_bytes": 3355482,
  "file_hash_sha256": "a3f5b8c9d2e1f4a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0",
  "mime_type": "text/csv",
  "encoding": "UTF-8",
  "rows_detected": 7901,
  "columns_detected": 12,
  "prefect_flow_run_id": "abc123-456-def-789",
  "ingestion_status": "completed"
}
```

Il file viene scritto dal flow Prefect al termine di ogni acquisizione e risiede nella sottodirectory `_metadata/` accanto al file dati corrispondente. Registra le informazioni operative essenziali: provenienza, hash di integrità, stato e identificativo del flow che ha eseguito il download. Svolge tre funzioni nell'architettura: consente il reprocessing mirato (in caso di errore nel layer Silver, la pipeline individua il file Bronze da rielaborare senza riscaricare dalla fonte); garantisce l'audit trail tramite hash SHA256; e fornisce i metadati tecnici che il flow pubblica in OpenMetadata tramite API, alimentando il catalogo di governance centralizzato.

Tracciamento su Database (PostgreSQL):

Mentre il JSON è locale al file system, la tabella `bronze.ingestion_log` centralizza lo stato di tutte le acquisizioni, consentendo query aggregate su fonti, dataset e stati. Funge da registro consultabile

da Prefect per gestire le riesecuzioni e garantire la deduplicazione tramite il vincolo di unicità sull'hash del file.

```sql
-- Schema: bronze
CREATE TABLE bronze.ingestion_log (
    id BIGSERIAL PRIMARY KEY,
    fonte VARCHAR(50) NOT NULL,
    dataset VARCHAR(100) NOT NULL,
    anno_rif INTEGER NOT NULL,
    file_path TEXT NOT NULL,
    file_size_bytes BIGINT,
    file_hash_sha256 CHAR(64),
    download_url TEXT,
    download_timestamp TIMESTAMP NOT NULL,
    prefect_flow_run_id UUID,
    status VARCHAR(20) NOT NULL,  -- 'completed', 'failed', 'in_progress'
    error_message TEXT,
    created_at TIMESTAMP DEFAULT NOW(),

    UNIQUE (fonte, dataset, anno_rif, file_hash_sha256)
);

-- Index per query frequenti
CREATE INDEX idx_ingestion_log_fonte_dataset ON bronze.ingestion_log(fonte, dataset);
CREATE INDEX idx_ingestion_log_status ON bronze.ingestion_log(status);
CREATE INDEX idx_ingestion_log_timestamp ON bronze.ingestion_log(download_timestamp DESC);
```

2.1.4.4 Garanzie del Bronze layer Il Bronze layer offre tre garanzie fondamentali: - **Immutabilità**: i file non vengono mai modificati dopo la scrittura; un nuovo download dello stesso dataset produce un file distinto con timestamp diverso, preservando la storia completa di tutte le acquisizioni. - **Idempotenza**: la riesecuzione di un flow non produce duplicati; se un file con lo stesso hash è già presente, il download viene saltato e la deduplicazione opera sul vincolo `(fonte, dataset, anno_rif, file_hash)`. - **Disaster recovery**: il Bronze layer costituisce la copia originale da cui ricostruire l'intera piattaforma; se i layer Silver o Gold si corrompono, è sufficiente rielaborare i file Bronze tramite lo script di backup dedicato.

2.1.4.5 Caso d'uso: HTML → Dati strutturati Alcune fonti pubbliche italiane pubblicano dati esclusivamente come tabelle HTML embedded in pagine web, senza mettere a disposizione file CSV o Excel scaricabili. È il caso di diversi registri regionali di strutture sanitarie accreditate, di elenchi di enti pubblici e di sezioni di portali ministeriali in cui i dati sono presentati per la consultazione online ma non esportabili direttamente. Questo tipo di fonte richiede un approccio di acquisizione dedicato, basato su HTTP GET della pagina e parsing con BeautifulSoup, prima di poter procedere alla normalizzazione dei dati.

Il flusso Bronze-HTML segue la stessa logica degli altri formati: la pagina HTML viene salvata integralmente nel Bronze layer prima di qualsiasi trasformazione.

Il vantaggio del pattern Bronze rispetto all'acquisizione diretta senza archiviazione intermedia è la resilienza: se il parser fallisce perché la struttura HTML della pagina è cambiata, è possibile

Mermaid Diagram

Figure 2: Mermaid Diagram

correggere il codice e rielaborare il file già salvato, senza tornare alla fonte. Se il server della fonte è temporaneamente irraggiungibile o la pagina viene rimossa, il Bronze layer conserva la copia originale e permette la riprocessazione. La storia delle versioni HTML acquisite nel tempo consente inoltre di rilevare modifiche retroattive ai dati pubblicati.

2.1.5 Silver Layer: Dati puliti e validati

Ruolo: Fonte di riferimento unica per tutti i processi analitici del progetto. I dati sono puliti, validati e normalizzati a partire dal Bronze layer.

Principio chiave: "Verifica prima di fidarti, poi archivia"

2.1.5.1 Caratteristiche tecniche

Aspetto	Specifica MAPS
Storage	PostgreSQL schema `silver`
Formato	Tabelle relazionali normalizzate (schema EAV)
Convenzione di nomenclatura	Schema `silver`, tabelle principali: `territory_types`, `territories`, `territory_identifiers`, `territory_names`, `territory_containments`, `territory_attributes`
Politica di conservazione	Versionamento temporale (valid_from, valid_to) per serie storiche
Backup	Snapshot giornalieri PostgreSQL + WAL archiving
Modalità di accesso	Read-heavy (query analitiche), write-moderate (batch ingestion)

2.1.5.2 Schema EAV (Entity-Attribute-Value) Il progetto MAPS acquisisce oltre 180 dataset con strutture eterogenee: popolazione, strutture sanitarie, infrastrutture di trasporto, dati ambientali. Adottare un modello relazionale tradizionale richiederebbe una tabella separata per ciascun dataset, rendendo difficile la gestione delle evoluzioni nel tempo e l'aggiunta di nuove fonti. Il modello EAV (Entity-Attribute-Value) risolve questo problema rappresentando ogni osservazione come una riga `(territory_id, attribute, value)`: lo schema rimane stabile al variare dei dataset acquisiti, e nuovi attributi si aggiungono senza modifiche alla struttura del database.

Un ulteriore requisito è emerso nel corso del progetto: i dati non sono sempre disponibili a livello comunale. Alcuni dataset sono pubblicati a livello provinciale o regionale; altri si riferiscono ad aggregazioni funzionali come i Sistemi Locali del Lavoro (SLL) o i nuovi Sistemi Locali Omogenei (SLO) che questo progetto si propone di definire. Lo schema Silver supporta quindi entità territoriali di qualsiasi tipo, non solo i comuni.

Il modello si articola in sei tabelle principali: `territory_types` per il registro dei tipi di entità; `territories` per il registro principale di tutte le entità territoriali, ciascuna con un `id` surrogato stabile che non cambia mai anche quando i codici ISTAT vengono riassegnati; `territory_identifiers` per i codici ISTAT temporali e altri identificatori esterni; `territory_names` per i nomi ufficiali e

Mermaid Diagram

Figure 3: Mermaid Diagram

alternativi con supporto multilingue; `territory_containments` per il contenimento gerarchico temporale (comune → provincia, con date precise per i cambi di provincia); e `territory_attributes` per tutti i valori osservati con tracciabilità completa. Gli eventi di variazione amministrativa (fusioni, incorporazioni, scissioni) sono tracciati tramite `territory_relationships`, che memorizza i legami predecessore/successore derivati dai dati ISTAT delle variazioni amministrative.

La tabella `territory_attributes` è il cuore dello schema Silver. Ogni riga rappresenta un singolo valore osservato per un'entità territoriale in un periodo di validità, con la tracciabilità completa della fonte. Le colonne `valid_from` e `valid_to` implementano il versionamento temporale secondo il pattern SCD Type 2. In modo cruciale, `territory_attributes` fa riferimento all'`id` surrogato stabile del territorio invece che direttamente al codice ISTAT: quando un comune cambia provincia e riceve un nuovo codice ISTAT, lo storico degli attributi viene preservato senza alcuna migrazione dei dati.

```
-- Registro dei tipi di entità territoriale
CREATE TABLE silver.territory_types (
    code             VARCHAR(30) PRIMARY KEY,
    label            VARCHAR(100) NOT NULL,
    authority        VARCHAR(100),          -- "ISTAT", "SNAI", "MAPS", ...
    coding_scheme    TEXT,
    hierarchy_level  INTEGER                -- 1=nazione ... 5=comune; NULL=aggregazione funzio
);

-- Registro principale - id surrogato stabile, non cambia mai
CREATE TABLE silver.territories (
    id           SERIAL PRIMARY KEY,
    type_code    VARCHAR(30) NOT NULL REFERENCES silver.territory_types(code),
    label        VARCHAR(255),
    valid_from   DATE,                      -- da eventi variazioni CS; NULL = inizio sconosciu
    valid_to     DATE,                      -- da eventi variazioni ES; NULL = ancora attivo
    end_reason   TEXT
);

-- Identificatori esterni temporali (codici ISTAT, catastali, ...)
CREATE TABLE silver.territory_identifiers (
    id            SERIAL PRIMARY KEY,
    territory_id  INTEGER NOT NULL REFERENCES silver.territories(id),
    scheme        VARCHAR(50) NOT NULL,     -- 'istat', 'catasto', 'fiscale', 'uts', ...
    identifier    VARCHAR(50) NOT NULL,
    valid_from    DATE,
    valid_to      DATE,
    fonte         VARCHAR(100)
);

-- Nomi temporali (ufficiali + bilingui + alternativi)
```

```
CREATE TABLE silver.territory_names (
    id            SERIAL PRIMARY KEY,
    territory_id  INTEGER NOT NULL REFERENCES silver.territories(id),
    name          TEXT NOT NULL,
    language      VARCHAR(10),              -- 'it', 'de', 'fr', 'sl', ...
    name_type     VARCHAR(50),              -- 'official', 'local', 'alias', ...
    valid_from    DATE,
    valid_to      DATE
);

-- Contenimento gerarchico temporale (comune → provincia, con date per i cambi di provincia)
CREATE TABLE silver.territory_containments (
    id            SERIAL PRIMARY KEY,
    member_id     INTEGER NOT NULL REFERENCES silver.territories(id),
    container_id  INTEGER NOT NULL REFERENCES silver.territories(id),
    valid_from    DATE,
    valid_to      DATE
);

-- EAV generalizzata - referenzia territory_id stabile, non il codice ISTAT
CREATE TABLE silver.territory_attributes (
    id            SERIAL PRIMARY KEY,
    territory_id  INTEGER NOT NULL REFERENCES silver.territories(id),
    type_code     VARCHAR(30) NOT NULL,     -- denormalizzato da territories per comodità
    attribute     VARCHAR(255) NOT NULL,
    value         TEXT,
    data_type     VARCHAR(50),              -- integer, float, string, boolean, date
    source        VARCHAR(100),
    valid_from    DATE,
    valid_to      DATE,                     -- NULL = valore corrente
    UNIQUE (territory_id, attribute, valid_from)
);

CREATE INDEX idx_ta_territory  ON silver.territory_attributes(territory_id);
CREATE INDEX idx_ta_type       ON silver.territory_attributes(type_code);
CREATE INDEX idx_ta_attribute  ON silver.territory_attributes(attribute);
CREATE INDEX idx_ta_source     ON silver.territory_attributes(source);
```

2.1.5.3 Esempio dati EAV Per rendere concreto il modello, si riporta un esempio reale: il comune di **AGLIÈ** (codice ISTAT 001001), con attributi provenienti da tre fonti distinte, affiancato da un record a livello provinciale e da un record relativo a un Sistema Locale del Lavoro. Tutti i tipi di entità coesistono nella stessa tabella grazie allo schema EAV generalizzato. La risoluzione territoriale passa sempre tramite l'`id` surrogato stabile; il codice ISTAT viene cercato in `territory_identifiers`.

```
-- Prima di tutto, risolvere territory_id per AGLIÈ (codice ISTAT 001001)
-- Questo join viene gestito da territory_resolver.py al momento della trasformazione
SELECT t.id FROM silver.territories t
```

```sql
JOIN silver.territory_identifiers ti ON ti.territory_id = t.id
WHERE t.type_code = 'comune' AND ti.scheme = 'istat' AND ti.identifier = '001001';
-- → id = 42 (esempio)

-- Comune AGLIÈ (territory_id=42) - attributi da tre fonti
INSERT INTO silver.territory_attributes
    (territory_id, type_code, attribute, value, data_type, source, valid_from)
VALUES
    (42, 'comune', 'popolazione',    '2635',       'integer', 'ISTAT',       '2024-01-01'),
    (42, 'comune', 'superficie_kmq', '13.98',      'float',   'ISTAT',       '2024-01-01'),
    (42, 'comune', 'num_tabacchi',   '2',          'integer', 'ADM',         '2024-01-01'),
    (42, 'comune', 'num_asl',        '1',          'integer', 'MinSalute',   '2023-01-01'),
    (42, 'comune', 'ato_gas_id',     'ATO-PIE-01', 'string',  'MinAmbiente', '2024-01-01');

-- Provincia di Torino (territory_id=107) - dataset a livello provinciale
INSERT INTO silver.territory_attributes
    (territory_id, type_code, attribute, value, data_type, source, valid_from)
VALUES
    (107, 'provincia', 'popolazione',    '2259523', 'integer', 'ISTAT', '2023-01-01'),
    (107, 'provincia', 'pil_pro_capite', '32400',   'integer', 'ISTAT', '2022-01-01');

-- Sistema Locale del Lavoro (territory_id=8) - aggregazione funzionale
INSERT INTO silver.territory_attributes
    (territory_id, type_code, attribute, value, data_type, source, valid_from)
VALUES
    (8, 'sll', 'addetti_industria', '15200', 'integer', 'ISTAT', '2021-01-01');
```

2.1.5.4 Trasformazioni Bronze → Silver Ogni dataset acquisito nel Bronze layer viene trasformato in Silver attraverso una pipeline Prefect standard in quattro fasi: parsing del file sorgente nel formato originale (CSV, Excel, PDF, HTML), pulizia e normalizzazione dei valori, validazione con Great Expectations, e caricamento nello schema EAV. Questa struttura è uniforme per tutti i dataset, indipendentemente dalla fonte.

```python
@task(name="bronze-to-silver-transform")
def transform_bronze_to_silver(
    bronze_file_path: str,
    entity_type: str,   # 'comune', 'provincia', 'regione', 'sll', ... - usato per risolvere te
    fonte: str,
    dataset: str,
    anno: int
):
    """
    Standard transformation pipeline: Bronze → Silver

    Steps:
    1. Parse: Read Bronze file (CSV/PDF/Excel/HTML)
    2. Clean: Normalize encoding, trim whitespace, fix typos
    3. Validate: Check data quality, enforce business rules
```

```
    4. Load: Risolvi territory_id e inserisci in silver.territory_attributes
    """

    # Step 1: Parse
    if bronze_file_path.endswith('.csv'):
        df = pd.read_csv(bronze_file_path)
    elif bronze_file_path.endswith('.pdf'):
        df = extract_pdf_tables(bronze_file_path)  # Docling
    elif bronze_file_path.endswith('.xlsx'):
        df = pd.read_excel(bronze_file_path)
    elif bronze_file_path.endswith('.html'):
        df = parse_html_to_dataframe(bronze_file_path)  # BeautifulSoup

    # Step 2: Clean
    df = clean_dataframe(df)

    # Step 3: Validate with Great Expectations
    validation_results = validate_dataframe(df, fonte, dataset)
    if not validation_results.passed:
        raise ValueError(f"Validation failed: {validation_results.errors}")

    # Step 4: Load to Silver (EAV generalizzata)
    load_to_territory_attributes(df, entity_type, fonte, anno)
```

2.1.5.5 Versionamento temporale Nel periodo 2010-2025 si sono verificate circa 100 fusioni o separazioni comunali. Quando due comuni si fondono in uno nuovo, i dati storici degli enti preesistenti rimangono validi per il periodo precedente alla fusione: non vengono cancellati né sovrascritti, ma delimitati temporalmente tramite le colonne `valid_from` e `valid_to`. Il nuovo comune ottiene i propri record a partire dalla data di istituzione. Le query possono quindi interrogare lo stato del territorio a qualsiasi data storica con un semplice filtro temporale.

L'esempio seguente mostra una fusione avvenuta nel 2019: i comuni 001234 e 001235 confluiscono nel nuovo comune 001236. Nel modello a identità surrogata, i due insediamenti originari mantengono le loro righe in `territories` (ora con `valid_to` impostato); il nuovo comune ottiene una nuova riga. La relazione è registrata in `territory_relationships`.

```
-- territory_id=10 → comune 001234 (valid_to 2019-01-01)
-- territory_id=11 → comune 001235 (valid_to 2019-01-01)
-- territory_id=12 → comune 001236 (valid_from 2019-01-01, valid_to NULL)

-- Attributi prima della fusione (valid_to chiude alla data della fusione)
INSERT INTO silver.territory_attributes
    (territory_id, type_code, attribute, value, source, valid_from, valid_to)
VALUES
    (10, 'comune', 'popolazione', '1500', 'ISTAT', '2018-01-01', '2019-01-01'),
    (11, 'comune', 'popolazione', '800',  'ISTAT', '2018-01-01', '2019-01-01');

-- Attributi dopo la fusione
INSERT INTO silver.territory_attributes
```

```
    (territory_id, type_code, attribute, value, source, valid_from, valid_to)
VALUES
    (12, 'comune', 'popolazione', '2300', 'ISTAT', '2019-01-01', NULL);

-- Query: Popolazione dell'insediamento con codice ISTAT 001234 nel 2018
SELECT ta.value FROM silver.territory_attributes ta
JOIN silver.territory_identifiers ti ON ti.territory_id = ta.territory_id
WHERE ti.scheme = 'istat' AND ti.identifier = '001234'
  AND ta.attribute = 'popolazione'
  AND '2018-12-31' BETWEEN ta.valid_from AND COALESCE(ta.valid_to, '9999-12-31');
-- Risultato: 1500
```

2.1.6 Gold Layer: Dati pronti per l'analisi

Ruolo: Dati ottimizzati per applicazioni specifiche — aggregati, denormalizzati, arricchiti con geometrie spaziali. È il layer che alimenta direttamente le dashboard, le API e gli algoritmi SLO.

Principio chiave: "Ottimizza per le query, non per lo spazio"

2.1.6.1 Caratteristiche tecniche

Aspetto	Specifica MAPS
Storage	PostgreSQL schema `gold` con estensioni PostGIS
Formato	Tabelle denormalizzate (wide table) con geometrie spaziali
Convenzione di nomenclatura	Schema `gold`, tabelle tematiche per dominio (comuni, attrattori, serie storiche)
Politica di conservazione	Snapshot aggiornati periodicamente (giornaliero o settimanale); ricostruibili dal Silver layer
Backup	Snapshot giornalieri (ma ricostruibili dal Silver layer)
Modalità di accesso	Sola lettura ad alta frequenza (query aggregate, accesso puntuale via API, scansioni complete per gli algoritmi SLO)

2.1.6.2 Data mart principali Un data mart è una vista tematica e ottimizzata dei dati, progettata per rispondere a un insieme specifico di domande analitiche. Nel Gold layer, ogni data mart è una tabella denormalizzata che aggrega in un'unica struttura informazioni provenienti da più fonti Silver, eliminando la necessità di join complessi al momento della query. Questo approccio sacrifica lo spazio di archiviazione — i dati vengono duplicati rispetto al Silver — a favore della velocità di interrogazione e della semplicità d'uso per chi sviluppa applicazioni o algoritmi.

Il Gold layer di MAPS definisce tre data mart principali, ciascuno orientato a un dominio di analisi distinto.

A. `gold.comuni_aggregati`

È la tabella centrale del Gold layer: una rappresentazione denormalizzata di ciascun comune italiano, con tutti gli attributi rilevanti raccolti in un'unica riga. Ogni record integra dati demografici da ISTAT, conteggi di servizi dai Ministeri, informazioni geografiche con geometria PostGIS, e gli attributi calcolati dall'algoritmo SLO (livello di attrattività, isocrona a 60 minuti). Questa struttura consente query analitiche complesse in tempi inferiori ai 10 ms, senza alcun join tra tabelle.

```sql
CREATE TABLE gold.comuni_aggregati (
    -- Identificativi
    codice_istat VARCHAR(6) PRIMARY KEY,
    denominazione VARCHAR(255) NOT NULL,
    denominazione_full VARCHAR(255),  -- Con sigla provincia

    -- Gerarchia amministrativa
    codice_regione CHAR(2) NOT NULL,
    denominazione_regione VARCHAR(100) NOT NULL,
    codice_provincia CHAR(3),
    denominazione_provincia VARCHAR(100),
    sigla_provincia CHAR(2),

    -- Attributi demografici (da ISTAT)
    popolazione_2024 INTEGER,
    popolazione_2023 INTEGER,
    popolazione_2022 INTEGER,
    crescita_popolazione_pct NUMERIC(5,2),  -- Calculated: (2024-2023)/2023*100

    -- Attributi geografici
    superficie_kmq NUMERIC(10,2),
    densita_abitanti_kmq NUMERIC(10,2),  -- Calculated: pop/superficie
    altitudine_m INTEGER,
    zona_altimetrica VARCHAR(50),  -- 'montagna', 'collina', 'pianura'

    -- Geometria PostGIS
    geometria GEOMETRY(MultiPolygon, 4326) NOT NULL,
    centroide GEOMETRY(Point, 4326),

    -- Servizi (da Ministeri)
    num_strutture_sanitarie INTEGER DEFAULT 0,
    num_asl INTEGER DEFAULT 0,
    num_scuole_primarie INTEGER DEFAULT 0,
    num_scuole_secondarie INTEGER DEFAULT 0,
    num_tabacchi INTEGER DEFAULT 0,
    num_uffici_postali INTEGER DEFAULT 0,

    -- Infrastrutture (da OpenData)
    ha_stazione_ferroviaria BOOLEAN DEFAULT FALSE,
    ha_casello_autostradale BOOLEAN DEFAULT FALSE,
    ha_aeroporto BOOLEAN DEFAULT FALSE,

    -- Ambiti territoriali (da registri pubblici)
    ato_gas_id VARCHAR(50),
    ato_gas_denominazione VARCHAR(255),
    ato_gas_gestore VARCHAR(255),
    ato_acqua_id VARCHAR(50),
    ato_rifiuti_id VARCHAR(50),
```

```
    -- Attributi DLS (Calculated)
    attractor_level VARCHAR(50),  -- 'metropolitan', 'urban', 'semi-urban', 'rural'
    cluster_dls_id INTEGER,
    isochrone_60min GEOMETRY(MultiPolygon, 4326),

    -- Metadata
    last_updated TIMESTAMP NOT NULL DEFAULT NOW(),
    data_completeness_pct NUMERIC(5,2),  -- % attributi popolati

    -- Constraints
    CONSTRAINT valid_codice CHECK (codice_istat ~ '^\d{6}$')
);

-- Spatial indexes
CREATE INDEX idx_gold_comuni_geometria ON gold.comuni_aggregati USING GIST(geometria);
CREATE INDEX idx_gold_comuni_centroide ON gold.comuni_aggregati USING GIST(centroide);
CREATE INDEX idx_gold_comuni_isochrone ON gold.comuni_aggregati USING GIST(isochrone_60min);

-- Attribute indexes
CREATE INDEX idx_gold_comuni_regione ON gold.comuni_aggregati(codice_regione);
CREATE INDEX idx_gold_comuni_provincia ON gold.comuni_aggregati(codice_provincia);
CREATE INDEX idx_gold_comuni_attractor ON gold.comuni_aggregati(attractor_level);
```

L'esempio di query seguente mostra come la struttura denormalizzata permetta di filtrare per ambito territoriale del gas e soglia di popolazione con una singola istruzione, senza join, e con un tempo di esecuzione stimato inferiore ai 10 ms.

```
-- Query: Comuni in ATO Gas "ATO-PIE-01" con popolazione > 5000
SELECT
    denominazione,
    popolazione_2024,
    ato_gas_gestore,
    num_tabacchi,
    attractor_level
FROM gold.comuni_aggregati
WHERE ato_gas_id = 'ATO-PIE-01'
  AND popolazione_2024 > 5000
ORDER BY popolazione_2024 DESC;

-- Execution: Index scan su ato_gas_id, no joins, < 10ms
```

B. `gold.dls_attractors`

Contiene i risultati dell'analisi degli attrattori territoriali prodotta dall'algoritmo SLO. Ogni riga rappresenta un comune classificato secondo il suo livello di attrattività (metropolitano, urbano, semi-urbano, rurale), con i punteggi parziali per categoria di servizio, la geometria dell'isocrona a 60 minuti e l'elenco dei comuni raggiungibili. Questa tabella è il principale output del Work Package 3 e alimenta direttamente le visualizzazioni cartografiche e il simulatore di politiche territoriali.

```sql
CREATE TABLE gold.dls_attractors (
    codice_istat VARCHAR(6) PRIMARY KEY REFERENCES gold.comuni_aggregati(codice_istat),
    denominazione VARCHAR(255) NOT NULL,

    -- Attractor classification
    attractor_level VARCHAR(50) NOT NULL,  -- 'metropolitan', 'urban', 'semi-urban', 'rural'
    attractor_score NUMERIC(5,2),  -- 0-100 score

    -- Service availability (weighted scores)
    servizi_sanitari_score NUMERIC(5,2),
    servizi_educativi_score NUMERIC(5,2),
    servizi_commerciali_score NUMERIC(5,2),
    servizi_trasporti_score NUMERIC(5,2),

    -- Isochrone analysis (60 min travel time)
    isochrone_60min GEOMETRY(MultiPolygon, 4326),
    comuni_raggiungibili_60min INTEGER[],  -- Array codici ISTAT
    popolazione_raggiungibile_60min INTEGER,

    -- Clustering
    cluster_id INTEGER NOT NULL,
    cluster_centroid GEOMETRY(Point, 4326),

    -- Metadata
    calculation_date TIMESTAMP NOT NULL DEFAULT NOW(),

    CONSTRAINT valid_attractor_level CHECK (attractor_level IN ('metropolitan', 'urban', 'semi-
);
```

C. Data mart aggiuntivi

I due data mart descritti sopra coprono i requisiti analitici identificati nella fase di progettazione. Il Gold layer è tuttavia progettato per accogliere ulteriori tabelle tematiche al crescere delle necessità del progetto. Nuovi data mart possono essere aggiunti senza modificare la struttura Silver, semplicemente definendo una nuova funzione di refresh che legge dal layer EAV e materializza i dati nella forma richiesta. Esempi possibili includono viste per l'analisi dei flussi di mobilità, tabelle di supporto per il simulatore di politiche territoriali (WP5), o aggregazioni tematiche per specifici indicatori di svantaggio territoriale.

2.1.6.3 Accesso analitico con DuckDB I data mart Gold sono la principale interfaccia di accesso ai dati del progetto, ma con modalità distinte a seconda del tipo di utilizzo. Le API REST e gli algoritmi SLO interrogano le tabelle Gold direttamente via PostgreSQL, sfruttando gli indici spaziali e la bassa latenza delle query puntuali. Per l'analisi esplorativa — notebook Jupyter, script Python dei ricercatori, verifica di ipotesi — si utilizza invece DuckDB in modalità embedded.

DuckDB legge le tabelle Gold di PostgreSQL tramite federation diretta, senza necessità di copiare i dati, con una sintassi SQL identica a quella standard:

```sql
import duckdb
```

```
con = duckdb.connect()
con.execute("INSTALL postgres; LOAD postgres;")
con.execute("""
    ATTACH 'dbname=maps_db host=localhost user=maps password=...'
    AS maps (TYPE postgres, READ_ONLY);
""")

# Query sul Gold layer: comuni con alta attrattività in area montana
result = con.execute("""
    SELECT denominazione, attractor_score, popolazione_2024, zona_altimetrica
    FROM maps.gold.dls_attractors a
    JOIN maps.gold.comuni_aggregati c USING (codice_istat)
    WHERE a.attractor_level = 'urban'
      AND c.zona_altimetrica = 'montagna'
    ORDER BY a.attractor_score DESC
    LIMIT 20
""").df()
```

Quando i dati devono essere distribuiti a ricercatori esterni o archiviati per analisi offline, il Gold layer può essere esportato in formato Parquet — un formato colonnare compresso che DuckDB legge direttamente, senza database, con prestazioni equivalenti a quelle in-memory.

2.1.6.4 Trasformazioni Silver → Gold Il popolamento del Gold layer avviene attraverso una funzione SQL schedulata da Prefect con cadenza giornaliera. La funzione esegue un ciclo completo di ricreazione della tabella: prima svuota il contenuto esistente con `TRUNCATE`, poi lo ricostruisce con una query di pivot che trasforma le righe EAV del Silver layer in colonne della tabella denormalizzata Gold. Al termine, un comando `ANALYZE` aggiorna le statistiche del query optimizer.

```
-- Refresh Gold tables da Silver (scheduled daily)
CREATE OR REPLACE FUNCTION gold.refresh_comuni_aggregati()
RETURNS void AS $$
BEGIN
    -- Truncate e rebuild (snapshot approach)
    TRUNCATE gold.comuni_aggregati;

    -- Pivot EAV → Wide table
    INSERT INTO gold.comuni_aggregati
    SELECT
        c.codice_istat,
        c.denominazione,
        c.geometria,
        ST_Centroid(c.geometria) AS centroide,

        -- Pivot attributi da Silver
        MAX(CASE WHEN e.attribute = 'popolazione_2024' THEN e.value::INTEGER END) AS popolazior
        MAX(CASE WHEN e.attribute = 'popolazione_2023' THEN e.value::INTEGER END) AS popolazior
        MAX(CASE WHEN e.attribute = 'superficie_kmq' THEN e.value::NUMERIC END) AS superficie_k

        -- Campi calcolati
```

```
        (MAX(CASE WHEN e.attribute = 'popolazione_2024' THEN e.value::NUMERIC END) -
         MAX(CASE WHEN e.attribute = 'popolazione_2023' THEN e.value::NUMERIC END)) /
         NULLIF(MAX(CASE WHEN e.attribute = 'popolazione_2023' THEN e.value::NUMERIC END), 0) *
            AS crescita_popolazione_pct,

        -- Ambiti territoriali
        MAX(CASE WHEN e.attribute = 'ato_gas_id' THEN e.value END) AS ato_gas_id,
        MAX(CASE WHEN e.attribute = 'ato_gas_gestore' THEN e.value END) AS ato_gas_gestore,

        NOW() AS last_updated
    FROM
        gold.comuni_anagrafica c
    LEFT JOIN silver.territory_identifiers ti ON ti.scheme = 'istat' AND ti.identifier = c.codi
    LEFT JOIN silver.territory_attributes e ON e.territory_id = ti.territory_id
            AND e.valid_to IS NULL  -- Solo dati correnti
    GROUP BY
        c.codice_istat, c.denominazione, c.geometria;

    -- Analyze per query optimizer
    ANALYZE gold.comuni_aggregati;
END;
$$ LANGUAGE plpgsql;

-- Schedule refresh (chiamato da Prefect flow daily)
SELECT gold.refresh_comuni_aggregati();
```

L'approccio con funzione SQL schedulata da Prefect è la scelta adottata per il progetto MAPS, in quanto mantiene la logica di trasformazione vicina al database e sfrutta le capacità spaziali di PostGIS. Esistono tuttavia approcci alternativi ugualmente validi.

Con dbt (data build tool) le trasformazioni Silver → Gold si esprimono come modelli SQL versionati, con dependency graph automatico, test integrati e documentazione generata; è la scelta preferibile quando il numero di modelli Gold cresce e la manutenibilità diventa prioritaria.

Con Prefect puro le trasformazioni avvengono in Python tramite task dedicati, eliminando la dipendenza da funzioni SQL nel database; è preferibile quando le trasformazioni richiedono logica complessa non esprimibile in SQL o quando si vuole un unico punto di controllo per l'intera pipeline.

2.1.7 Comparazione tra schema EAV e schema tradizionale

Il progetto MAPS acquisisce oltre 180 dataset da fonti eterogenee — ISTAT, Ministeri, registri pubblici, dati OpenData — ciascuno con attributi propri, copertura temporale variabile e disponibilità non uniforme tra comuni. Questa eterogeneità pone un problema di progettazione del Silver layer: come strutturare uno schema che possa accogliere attributi nuovi senza richiedere modifiche alla base dati, che registri la provenienza di ogni valore e che gestisca le variazioni nel tempo?

L'approccio tradizionale, basato su una tabella "wide" con una colonna per ogni attributo, risulta inadeguato in questo contesto. La tabella avrebbe centinaia di colonne, la maggior parte delle quali vuote per qualsiasi comune specifico, e richiederebbe una migrazione dello schema ogni volta che si aggiunge un nuovo dataset. La tracciabilità temporale sarebbe assente o richiederebbe strutture

parallele.

codice_istat	popolazione	n_asili	has_ospedale	n_scuole	ato_gas_id	…
001001	45230	12	true	?	?	…
001002	8420	2	false	?	?	…

Il modello EAV (Entity-Attribute-Value) risolve questi problemi invertendo la struttura: invece di avere una colonna per attributo, ogni attributo diventa una riga. Lo schema rimane fisso — tre colonne fondamentali più i metadati — mentre il contenuto cresce verticalmente all'aggiunta di nuovi dataset, senza alcuna modifica strutturale.

entity	attribute	value	valid_from	source
001001	popolazione	45230	2021-01-01	ISTAT
001001	n_asili_nido	12	2021-01-01	ISTAT
001001	has_ospedale	true	2015-01-01	MinSalute
001002	popolazione	8420	2021-01-01	ISTAT
001002	cod_pre_fusione	001045	2017-01-01	ISTAT

La colonna `source` garantisce la tracciabilità della provenienza per ogni singolo valore; le colonne `valid_from` e `valid_to` consentono di gestire serie storiche e variazioni comunali senza strutture parallele. Lo storage è efficiente perché vengono registrati solo gli attributi effettivamente disponibili, non i valori mancanti.

Il compromesso principale è la complessità delle query: per leggere un insieme di attributi relativi a un comune, è necessario un'operazione di pivot (`CASE WHEN`) che trasforma le righe EAV in colonne. Questa complessità è però confinata al layer di trasformazione Silver → Gold, invisibile a chi utilizza i data mart. Le prestazioni di lettura, potenzialmente penalizzate dal maggior numero di righe, sono mitigate da indici appropriati su `codice_istat` e `attributo`. I valori sono archiviati come testo e vengono convertiti al tipo corretto durante il popolamento del Gold layer.

Lo schema EAV nel Silver layer rappresenta il compromesso ottimale per MAPS: massima flessibilità in acquisizione, governabilità nel tempo, e nessun accoppiamento strutturale tra l'arrivo di nuovi dati e lo schema del database.

2.2 Stack tecnologico

La scelta dello stack tecnologico per MAPS è guidata da tre criteri: adeguatezza alla scala dei dati trattati, indipendenza da fornitori commerciali, e maturità degli strumenti. Il progetto lavora con circa 180 dataset a granularità comunale — un volume nell'ordine delle decine di gigabyte, non dei petabyte — che non richiede infrastrutture cloud di grandi dimensioni ma beneficia di strumenti specializzati per dati spaziali e analisi territoriale.

L'intero stack è composto da software open source, ciascuno scelto per un ruolo preciso e non sovrapponibile agli altri. PostgreSQL con l'estensione PostGIS è il nucleo del sistema: gestisce sia lo schema EAV del Silver layer sia le tabelle denormalizzate del Gold layer, con supporto nativo per operazioni geografiche come il calcolo delle isocrone. Prefect orchestra l'esecuzione delle pipeline, dalla raccolta dei dati Bronze fino al popolamento del Gold layer, con monitoraggio e gestione

Mermaid Diagram

Figure 4: Mermaid Diagram

degli errori. OpenMetadata cataloga internamente tutti i dataset, traccandone la lineage e la qualità; CKAN espone verso l'esterno i dataset approvati per la pubblicazione open data. Great Expectations presidia la qualità dei dati nel passaggio Bronze → Silver, mentre DuckDB offre ai ricercatori un'interfaccia analitica leggera e portatile sul Gold layer.

2.2.1 Componenti principali

Componente	Tecnologia	Versione	Ruolo
Storage primario	PostgreSQL + PostGIS	17 + 3.5	Master data, operazioni spaziali
Orchestrazione	Prefect	3.x	Scheduling, monitoring, lineage ETL
Analytics layer	DuckDB	1.x	Query analytics (federation da PostgreSQL)
Governance interna	OpenMetadata	1.x	Catalog interno, lineage, data quality, profiling
Catalogo open data	CKAN	2.11	Catalogo pubblico open data, DCAT-AP_IT
Data quality	Great Expectations	1.x	Validation, anomaly detection
PDF extraction	Docling	Latest	Estrazione tabelle da PDF (97.9% accuracy)
Object storage	MinIO (o GCS)	Latest	Raw files, PDFs, archivi Excel

2.2.2 Motivazioni delle scelte tecnologiche

Le motivazioni dettagliate delle scelte tecnologiche — confronti quantitativi tra le alternative considerate per ciascun componente, analisi dei costi e valutazione dei vincoli — sono documentate nell'Appendice A.

2.3 Architettura di deployment

I servizi della piattaforma MAPS sono distribuiti come container orchestrati su un cluster Kubernetes managed (DigitalOcean DOKS). Il database PostgreSQL + PostGIS è gestito come servizio managed dallo stesso provider, eliminando l'overhead operativo di backup, failover e patching. Lo storage persistente per il Bronze layer è fornito da volumi a blocchi. Il traffico HTTPS è terminato da un Load Balancer con certificati Let's Encrypt, distribuito verso i servizi esposti tramite Ingress Controller.

Il solution design completo — motivazioni della scelta infrastrutturale, dimensionamento, topologia dei servizi, sicurezza e stima dei costi — è documentato nel deliverable D2.1.2.

2.4 Data lineage e governance

La governance dei dati è una componente strutturale dell'architettura MAPS, non un'aggiunta successiva. Un progetto che integra oltre 200 dataset da fonti pubbliche eterogenee, li trasforma attraverso tre layer progressivi e li pubblica come open data a fini di ricerca scientifica, ha bisogno di sapere in ogni momento da dove proviene ciascun dato, quali trasformazioni ha subito, e con quale livello di qualità è arrivato a destinazione. Senza questa tracciabilità, i risultati analitici del WP3 — la mappa dei Sistemi di Vita Quotidiana, la classificazione degli attrattori — non sarebbero verificabili né riproducibili.

La governance si articola in tre dimensioni complementari: il tracciamento della lineage (da dove viene il dato e come è stato trasformato), la misurazione della qualità (quanto il dato è completo, coerente e affidabile), e la catalogazione (quali dataset esistono, chi ne è responsabile, come sono strutturati).

2.4.1 Data lineage

La lineage descrive il percorso di ciascun dato dalla fonte esterna fino all'utilizzo finale. Nel contesto dell'architettura Medallion, questo percorso attraversa almeno tre passaggi: l'acquisizione nel Bronze layer, la normalizzazione nel Silver layer, e l'aggregazione nel Gold layer. A ciascun passaggio possono corrispondere trasformazioni significative — parsing di un PDF, riconciliazione dei codici ISTAT per fusioni comunali, calcolo di aggregazioni spaziali — e la lineage deve documentarle tutte.

OpenMetadata è lo strumento scelto per il tracciamento della lineage. Si integra nativamente con PostgreSQL, da cui legge automaticamente la struttura degli schemi (tabelle, colonne, tipi), e con Prefect, da cui riceve i metadati di esecuzione delle pipeline (flow run, task run, stato, durata, errori). Questa doppia integrazione consente di ricostruire la catena completa: quale flow ha prodotto quale tabella, partendo da quali dati sorgente, in quale esecuzione, con quale esito.

La lineage si articola su tre livelli di granularità. Il livello di **dataset** traccia le dipendenze tra tabelle: la tabella Gold `comuni_aggregati` dipende dalla tabella Silver `territory_attributes`, che a sua volta dipende dai file Bronze di ciascuna fonte. Il livello di **colonna** traccia la provenienza dei singoli attributi: il campo `popolazione_residente` nel Gold layer deriva dall'attributo omonimo nel Silver layer, originato dal file CSV ISTAT della popolazione. Il livello di **pipeline** traccia le esecuzioni: quale run di Prefect ha aggiornato una data tabella, quando, con quale durata e con quale esito.

Il valore pratico della lineage emerge in diversi scenari operativi. Quando una fonte esterna pubblica

Mermaid Diagram

Figure 5: Mermaid Diagram

un aggiornamento, la lineage indica immediatamente quali tabelle Silver e Gold devono essere ricalcolate, evitando riesecuzioni inutili o, peggio, dimenticanze. Quando un'analisi produce risultati inattesi, la lineage consente di risalire la catena e verificare se la causa è un dato sorgente anomalo, un errore di trasformazione, o un cambiamento nella struttura della fonte. Quando un dataset viene pubblicato su CKAN, la lineage fornisce la documentazione metodologica richiesta dagli standard DCAT-AP_IT: provenienza, trasformazioni applicate, data di ultimo aggiornamento.

2.4.2 Data Quality

La qualità dei dati è verificata in modo sistematico a ogni passaggio della pipeline, con regole di validazione definite tramite Great Expectations. Le validazioni non sono controlli generici applicati uniformemente a tutti i dataset: ciascuna fonte ha un proprio profilo di qualità atteso, con soglie e regole calibrate sulla natura specifica dei dati.

Nel passaggio Bronze → Silver, le validazioni presidiano l'integrità strutturale. Per i dataset a granularità comunale, il controllo fondamentale è la **copertura**: il numero di comuni presenti nel file deve essere coerente con l'universo di riferimento (7.896 comuni al 2024, con variazioni annuali dovute a fusioni e scissioni). Vengono inoltre verificati la **validità dei codici ISTAT** (formato, lunghezza, appartenenza all'anagrafe comunale di riferimento), l'**assenza di duplicati**, la **coerenza dei tipi di dato** (valori numerici nei campi numerici, date in formato valido) e la **completezza** (percentuale di valori nulli per ciascun attributo).

Nel passaggio Silver → Gold, le validazioni si concentrano sulla **coerenza semantica**. Le aggregazioni territoriali devono produrre totali coerenti con i dati di partenza (ad esempio, la somma della popolazione dei comuni di una provincia deve corrispondere al dato provinciale). Le operazioni spaziali PostGIS devono generare geometrie valide (nessun self-intersection, copertura territoriale completa). Le serie storiche devono mantenere continuità: un comune che esisteva nel 2023 deve avere un successore nel 2024, o essere documentato come soppresso.

I risultati delle validazioni sono registrati in OpenMetadata come metriche di qualità associate a ciascuna tabella. Le metriche principali sono cinque: **completezza** (percentuale di valori non nulli), **unicità** (assenza di record duplicati), **validità** (conformità dei valori al dominio atteso), **coerenza** (compatibilità tra attributi correlati), e **tempestività** (distanza tra la data di pubblicazione della fonte e la data di acquisizione). Queste metriche sono consultabili nel catalogo interno e consentono di valutare a colpo d'occhio lo stato di salute di ciascun dataset.

Quando una validazione fallisce — ad esempio, un file CSV contiene solo 5.000 comuni anziché i 7.896 attesi — la pipeline si interrompe e il dato non viene propagato al layer successivo. L'errore viene registrato nel log di ingestion, segnalato nell'interfaccia di Prefect e associato alla tabella corrispondente in OpenMetadata. Il dato Bronze rimane disponibile per l'analisi del problema, ma il Silver layer è protetto da ingestioni parziali o corrotte.

2.4.3 Catalogazione

OpenMetadata funge da **catalogo interno** dell'intera piattaforma dati. Ogni tabella, in ciascuno dei tre layer, è registrata con i suoi metadati: struttura (colonne, tipi, vincoli), provenienza (fonte,

pipeline che l'ha generata), qualità (metriche dell'ultima validazione), responsabilità (owner del dataset), e documentazione (descrizione testuale, note metodologiche).

Il catalogo si popola in parte **automaticamente** — OpenMetadata legge la struttura delle tabelle PostgreSQL e i metadati delle esecuzioni Prefect — e in parte **manualmente**, per le informazioni che richiedono conoscenza del dominio: la descrizione di un attributo, la metodologia di calcolo di un indicatore, le avvertenze sull'uso di un dataset. Questa combinazione di automazione e curatela consente di mantenere il catalogo aggiornato senza un onere di documentazione eccessivo.

Il catalogo risponde a tre esigenze operative: - per il team di sviluppo, è lo strumento per capire quali dati sono disponibili, in quale formato, e con quale grado di affidabilità, prima di avviare un'analisi o una nuova pipeline; - per la governance del progetto, è il registro che documenta lo stato di avanzamento dell'ingestion: quanti dataset sono stati acquisiti, quanti validati, quanti pubblicati; - per la pubblicazione open data, è la fonte da cui CKAN attinge i metadati DCAT-AP_IT da associare ai dataset esportati nel catalogo pubblico.

2.5 Best practices

Le pipeline di ingestion del progetto MAPS seguono tre principi architetturali che garantiscono affidabilità e manutenibilità nel tempo: idempotenza delle operazioni, caricamento incrementale dei dati, e gestione controllata dell'evoluzione degli schemi.

2.5.1 Idempotenza

Ogni pipeline deve poter essere rieseguita in qualsiasi momento senza generare duplicati né corrompere i dati esistenti. Questo requisito è essenziale in un contesto dove le fonti dati sono eterogenee e i fallimenti parziali sono frequenti: un download interrotto, un file PDF malformato, un timeout di rete possono interrompere l'esecuzione a metà, e la pipeline deve poter ripartire dall'inizio senza effetti collaterali.

L'idempotenza è implementata a livello di database attraverso operazioni di upsert: quando un record viene inserito nel Silver layer, la chiave composta (codice ISTAT, attributo, fonte, dataset, anno di riferimento) determina se si tratta di un nuovo dato o di un aggiornamento. Se il record esiste già, il valore viene sovrascritto e il timestamp di aggiornamento viene registrato; se non esiste, viene creato. Il risultato è identico sia alla prima esecuzione sia a ogni esecuzione successiva: lo stato finale del database dipende esclusivamente dai dati sorgente, non dalla storia delle esecuzioni.

Nel Bronze layer l'idempotenza è garantita dalla struttura stessa del layer: i file scaricati sovrascrivono eventuali versioni precedenti nella stessa posizione del file system, identificata dalla combinazione fonte/dataset/anno. L'hash SHA-256 del file consente di verificare se il contenuto è effettivamente cambiato rispetto alla versione precedente, evitando trasformazioni inutili nel caso in cui la fonte non abbia pubblicato aggiornamenti.

2.5.2 Caricamento incrementale

La strategia di caricamento differisce tra Silver layer e Gold layer, riflettendo le diverse esigenze dei due strati.

Il Silver layer adotta un approccio incrementale con versionamento temporale. Quando un dataset viene aggiornato, la pipeline confronta i nuovi dati con quelli già presenti: i record nuovi vengono inseriti, quelli modificati vengono aggiornati con chiusura della versione precedente (campo `valid_to`)

e apertura di una nuova versione (campo `valid_from`), quelli invariati non vengono toccati. Questa strategia minimizza il volume delle scritture e preserva la storia delle modifiche, rendendo possibile interrogare lo stato dei dati a una qualsiasi data passata.

Il Gold layer adotta invece un approccio a snapshot: le tabelle denormalizzate vengono ricostruite integralmente a partire dai dati Silver ogni volta che un aggiornamento lo richiede. La ricostruzione completa è preferibile all'aggiornamento incrementale per due ragioni. La prima è la natura aggregata dei dati Gold: una modifica a un singolo record Silver può propagarsi su più tabelle Gold e su più righe di ciascuna tabella, rendendo l'aggiornamento puntuale complesso e fragile. La seconda è il volume contenuto: con circa 8.000 comuni e poche centinaia di attributi per tabella, la ricostruzione completa richiede pochi minuti e non giustifica la complessità aggiuntiva di un meccanismo incrementale.

2.5.3 Evoluzione degli schemi

L'architettura deve gestire due tipi di evoluzione nel tempo: l'aggiunta di nuove fonti dati e la modifica della struttura dei dati esistenti.

L'aggiunta di nuove fonti è il caso più frequente e più semplice. Lo schema EAV del Silver layer è progettato per accogliere nuovi attributi senza modifiche strutturali: un nuovo dataset si traduce in nuove righe nella tabella `territory_attributes` con un nuovo valore nel campo `attribute`, senza bisogno di aggiungere colonne o alterare tabelle. Nel Gold layer, l'aggiunta di un nuovo attributo richiede una nuova colonna nella tabella denormalizzata corrispondente, ma questa operazione è non-breaking: la colonna viene aggiunta con un valore predefinito e popolata dalla pipeline, senza impatto sulle query esistenti.

La modifica della struttura dei dati esistenti è meno frequente ma più delicata. Può accadere che una fonte ISTAT modifichi il formato di un file, aggiunga o rimuova colonne, o cambi la codifica dei valori tra un rilascio e il successivo. In questi casi, la pipeline di trasformazione Bronze → Silver deve essere aggiornata per gestire il nuovo formato, mantenendo la compatibilità con le versioni precedenti ancora presenti nel Bronze layer. Il versionamento temporale del Silver layer garantisce che i dati già acquisiti con il vecchio formato restino validi e interrogabili, mentre i nuovi dati seguono il formato aggiornato.

OpenMetadata registra ogni modifica di schema come evento di lineage, consentendo di tracciare quando una tabella è stata alterata, da quale pipeline, e per quale motivo. Questo tracciamento è particolarmente utile nelle fasi di debug: quando un'analisi produce risultati diversi da quelli attesi, la storia delle modifiche di schema consente di verificare se la causa è un cambiamento nella struttura dei dati piuttosto che nei dati stessi.

2.6 Metriche e monitoraggio

Un'architettura a tre layer con oltre 200 dataset e pipeline di trasformazione concatenate richiede un sistema di monitoraggio che consenta di rispondere in ogni momento a tre domande: **i dati stanno entrando correttamente?**, **le trasformazioni stanno funzionando?**, e **i dati prodotti sono utilizzabili?**

2.6.1 Indicatori di salute della piattaforma

Il monitoraggio si organizza lungo i tre layer dell'architettura Medallion, con indicatori specifici per ciascuno.

Per il Bronze layer, l'indicatore principale è **il tasso di successo delle acquisizioni**. Ogni esecuzione di una pipeline di download viene registrata nella tabella `bronze.ingestion_log` con l'esito (successo, fallimento, parziale), la dimensione del file scaricato e l'hash del contenuto. Il monitoraggio verifica che le acquisizioni pianificate vengano effettivamente eseguite e che i file scaricati abbiano dimensioni coerenti con le aspettative: un file CSV della popolazione ISTAT che pesa pochi kilobyte anziché i megabyte attesi segnala un problema alla fonte o nel download. La crescita dello storage Bronze è contenuta — nell'ordine di pochi gigabyte all'anno, data la frequenza di aggiornamento semestrale o annuale delle fonti — e viene monitorata per garantire che i volumi persistenti abbiano capacità sufficiente.

Per il Silver layer, gli indicatori chiave sono **la latenza di ingestion** e **la qualità dei dati**. La latenza misura il tempo che intercorre tra il completamento del download Bronze e la disponibilità del dato normalizzato nel Silver layer: per la maggior parte dei dataset (CSV, Excel) questo tempo è nell'ordine dei minuti; per i PDF complessi che richiedono estrazione con Docling può arrivare a qualche decina di minuti. Le metriche di qualità — completezza, unicità, validità — sono quelle prodotte dalle validazioni Great Expectations e registrate in OpenMetadata, come descritto nella sezione precedente. Un degrado di questi indicatori nel tempo segnala un cambiamento nella struttura o nella qualità della fonte, che richiede intervento sulla pipeline di trasformazione.

Per il Gold layer, il monitoraggio si concentra sulle **prestazioni delle query** e sulla **coerenza delle aggregazioni**. I tempi di risposta delle query analitiche sono tracciati tramite l'estensione PostgreSQL `pg_stat_statements`, con l'obiettivo di mantenere il 95esimo percentile sotto i 100 millisecondi per le interrogazioni più comuni. La ricostruzione delle tabelle Gold a partire dal Silver layer deve completarsi in tempi ragionevoli — nell'ordine dei minuti per il volume attuale del progetto — e il monitoraggio segnala eventuali rallentamenti che possono indicare una crescita dei dati oltre le soglie previste o query di aggregazione da ottimizzare.

2.6.2 Monitoraggio delle pipeline

Prefect fornisce nativamente il monitoraggio dell'esecuzione delle pipeline, con visibilità sullo stato di ciascun flow run (in esecuzione, completato, fallito), sui tempi di esecuzione di ciascun task, e sui log dettagliati in caso di errore. L'interfaccia web di Prefect consente di consultare la storia delle esecuzioni, identificare i task falliti e rieseguire le pipeline con un singolo intervento.

Il monitoraggio delle pipeline si integra con la lineage tracciata in OpenMetadata: quando un flow fallisce, è possibile risalire immediatamente a quali tabelle Silver e Gold non sono state aggiornate, e quindi quali dati a valle potrebbero essere obsoleti. Questa correlazione tra stato delle pipeline e stato dei dati è particolarmente utile quando più pipeline condividono tabelle di destinazione: il fallimento di una singola pipeline non compromette l'intero layer, ma il monitoraggio deve segnalare quali porzioni dei dati sono rimaste ferme all'ultimo aggiornamento riuscito.

2.6.3 Monitoraggio infrastrutturale

Al livello infrastrutturale, il monitoraggio del cluster Kubernetes e del database PostgreSQL copre le metriche classiche di utilizzo risorse: CPU, memoria, I/O disco, spazio di archiviazione. DigitalOcean fornisce metriche di base per i nodi del cluster DOKS e per il database managed; per un monitoraggio più dettagliato, lo stack Prometheus + Grafana può essere distribuito nel cluster come servizio aggiuntivo.

Le soglie di attenzione sono calibrate sui profili di carico descritti nella sezione di dimensionamento: un utilizzo CPU sostenuto oltre l'80% sui nodi worker durante le finestre di esecuzione ETL è

normale; lo stesso livello in assenza di pipeline attive indica un problema. L'occupazione disco del database viene monitorata con soglia al 70% della capacità allocata, lasciando margine per la crescita e per le operazioni di manutenzione PostgreSQL (vacuum, reindex) che richiedono spazio temporaneo.

3 Solution design architettura cloud

Deliverable D2.1.2: Solution Design Architettura Cloud

3.1 Principi guida

Il progetto MAPS tratta dati pubblici italiani a granularità comunale, con volumi nell'ordine delle decine di gigabyte e un team operativo ristretto. L'architettura cloud è progettata attorno a tre principi derivati da questo contesto.

Il primo è la prevedibilità dei costi. Le piattaforme cloud di grandi dimensioni (AWS, GCP, Azure) adottano modelli di pricing variabile — per query, per GB processato, per request — che rendono difficile stimare la spesa mensile in anticipo. Per un progetto di ricerca con budget definito, questo rischio è inaccettabile. L'infrastruttura scelta adotta tariffazione a canone fisso mensile per tutte le componenti principali.

Il secondo è la semplicità operativa. La complessità gestionale deve essere proporzionale alla scala del progetto. Servizi managed (cluster Kubernetes, database PostgreSQL) eliminano gli oneri di manutenzione del control plane, degli aggiornamenti di sicurezza e della gestione dei backup, lasciando al team la sola responsabilità dei workload applicativi.

Il terzo è l'adeguatezza alla scala. L'infrastruttura è dimensionata sui carichi di lavoro reali del progetto: circa 200 dataset a granularità comunale, pipeline ETL in modalità batch sporadica, un numero di utenti concorrenti nell'ordine delle unità. Non è necessario progettare per carichi enterprise.

3.2 Scelta del provider: DigitalOcean

L'infrastruttura è ospitata su DigitalOcean. Rispetto agli hyperscaler (AWS, GCP, Azure), DigitalOcean offre un'esperienza di gestione più lineare, una curva di apprendimento contenuta e un pannello di controllo che non richiede competenze cloud specialistiche. La tariffazione è a canone fisso mensile per tutte le risorse principali (nodi del cluster, database managed, storage a blocchi, load balancer), con assenza di meccanismi di pricing variabile.

DigitalOcean supporta le estensioni PostGIS nel servizio di database managed, requisito essenziale per le operazioni spaziali del progetto. Il catalogo di servizi copre tutti i componenti necessari all'architettura MAPS senza richiedere integrazioni con provider esterni.

3.3 Orchestrazione container: Kubernetes (DOKS)

I servizi della piattaforma sono distribuiti come container orchestrati da Kubernetes, nella versione managed offerta da DigitalOcean (DOKS — DigitalOcean Kubernetes Service).

La scelta di Kubernetes rispetto a Docker Compose è motivata da tre considerazioni. La prima è la gestione dichiarativa dell'infrastruttura: Kubernetes descrive lo stato desiderato del sistema in manifest versionabili, con convergenza automatica. Se un container si arresta viene ricreato; se un

nodo diventa indisponibile i carichi vengono redistribuiti su quelli rimanenti. Docker Compose non offre equivalenti per il self-healing né per la distribuzione su nodi multipli.

La seconda è la separazione dei carichi di lavoro. I servizi della piattaforma hanno profili di consumo risorse molto diversi: le pipeline ETL di Prefect sono CPU-intensive e sporadiche, PostgreSQL richiede I/O disco costante, OpenMetadata e CKAN hanno un consumo moderato ma continuo. Kubernetes permette di allocare risorse (CPU request e limit, memoria) per singolo servizio, evitando che un picco di un componente degradi gli altri.

La terza è la prospettiva di evoluzione. Il progetto prevede fasi successive che introdurranno nuovi servizi e un numero maggiore di utenti. Kubernetes supporta questa crescita senza richiedere una migrazione architetturale: aggiungere nodi al cluster o nuovi deployment è un'operazione ordinaria.

DOKS elimina la complessità di gestione del control plane (API server, etcd, scheduler), che resta a carico del provider, lasciando al team la sola gestione dei workload applicativi.

3.4 Database: managed vs self-hosted

La scelta tra un'istanza PostgreSQL gestita come servizio managed (DigitalOcean Managed Databases) e un'istanza self-hosted nel cluster Kubernetes ha implicazioni su costi, operatività e flessibilità.

Con il servizio managed, il provider gestisce backup automatici con point-in-time recovery, failover automatico, patching di sicurezza e aggiornamenti di versione. L'overhead operativo è minimo. Il limite principale è la minore flessibilità nella configurazione: parametri di tuning avanzati ed estensioni non standard potrebbero non essere disponibili. Il costo è superiore rispetto al self-hosted a parità di risorse allocate (indicativamente 20-30% in più), ma il risparmio in ore di manutenzione compensa ampiamente per un team ristretto.

Con PostgreSQL self-hosted nel cluster (tramite un operatore Kubernetes come CloudNativePG o Zalando Postgres Operator) si ottiene pieno controllo su configurazione, estensioni e tuning. L'onere operativo è maggiore: backup, monitoring, failover e aggiornamenti sono a carico del team.

Criterio	Managed Database	Self-Hosted (K8s Operator)
Backup e recovery	Automatico (daily, PITR)	Da configurare
Failover	Automatico	Gestito dall'operatore
PostGIS	Supportato	Pieno controllo
Tuning avanzato	Limitato	Completo
Costo (4 vCPU, 8 GB)	~$80/mese	~$48/mese (risorse cluster)
Overhead operativo	Basso	Medio-alto
Estensioni custom	Lista predefinita	Qualsiasi

Per la fase MVP il database managed è l'opzione raccomandata: riduce il rischio operativo e consente al team di concentrarsi sullo sviluppo delle pipeline. La migrazione a self-hosted resta possibile in qualsiasi momento, se emergessero requisiti di configurazione specifica non supportati dal servizio managed.

3.5 Dimensionamento dell'infrastruttura

3.5.1 Profilo dei carichi di lavoro

Le fonti dati del progetto hanno frequenza di aggiornamento tipicamente semestrale o annuale. Le pipeline ETL di Prefect operano quindi in modalità batch sporadica: l'esecuzione avviene quando una fonte pubblica un aggiornamento, o durante le fasi di primo caricamento massivo dei dataset. I picchi di CPU sono concentrati nelle fasi di parsing PDF con Docling e nelle trasformazioni spaziali PostGIS, ma si verificano in modo occasionale. I servizi web (CKAN, OpenMetadata, Prefect UI) hanno un consumo costante ma contenuto, con utenti concorrenti nell'ordine delle unità.

3.5.2 Configurazione del cluster

Un cluster con tre nodi di tipo General Purpose da 4 vCPU e 8 GB di RAM ciascuno è sufficiente per garantire disponibilità e distribuzione dei carichi. La distribuzione indicativa dei workload è la seguente:

Componente	CPU request	Memoria request	Storage	Note
PostgreSQL + PostGIS	2 vCPU	4 GB	100 GB (block storage)	Solo se self-hosted
Prefect Server	0.5 vCPU	1 GB	10 GB	UI e orchestrazione
Prefect Worker	2 vCPU	4 GB	50 GB (Bronze files)	Burst durante ETL
OpenMetadata	1 vCPU	2 GB	20 GB	Catalogo e lineage
CKAN + Redis	1 vCPU	2 GB	20 GB	Catalogo pubblico
Ingress Controller	0.25 vCPU	256 MB	-	Traffico HTTPS

In condizioni ordinarie il cluster richiede circa 7 vCPU e 13 GB di RAM, con picchi fino a 10 vCPU durante le esecuzioni ETL. Tre nodi da 4 vCPU / 8 GB (totale 12 vCPU / 24 GB) offrono il margine necessario per i picchi e per tollerare la perdita di un nodo senza interruzione del servizio.

3.5.3 Storage

DigitalOcean Volumes (block storage) forniscono lo storage persistente per il database e per il Bronze layer. Il volume database è dimensionato a 100 GB con possibilità di espansione a caldo. Il Bronze layer richiede circa 50 GB per i file originali, anch'esso espandibile. I volumi sono replicati dal provider con protezione contro il guasto del singolo disco.

3.5.4 Stima dei Costi Mensili

Mermaid Diagram

Figure 6: Mermaid Diagram

Voce	Configurazione	Costo mensile
DOKS cluster (control plane)	Managed	$12
3 nodi General Purpose	4 vCPU / 8 GB ciascuno	$192 ($64/nodo)
Managed PostgreSQL	4 vCPU / 8 GB, 100 GB storage	$80
Block storage (150 GB)	Bronze + dati applicativi	$15
Load Balancer	Ingress HTTPS	$12
Totale		**~$311/mese (~$3.700/anno)**

Se il database è self-hosted nel cluster, il costo del Managed PostgreSQL ($80/mese) viene eliminato, riducendo il totale a circa $231/mese (~$2.770/anno), a fronte di un maggiore impegno operativo.

3.6 Topologia dei servizi

I servizi sono organizzati in namespace Kubernetes distinti per separare i carichi applicativi dai dati persistenti. L'Ingress Controller gestisce il traffico HTTPS in ingresso e il routing verso i servizi esposti (Prefect UI, OpenMetadata, CKAN). I servizi interni (Prefect Worker, Redis) non sono esposti all'esterno e comunicano esclusivamente sulla rete interna del cluster.

3.7 Sicurezza

3.7.1 Isolamento di rete

Kubernetes Network Policies restringono la comunicazione tra pod: i servizi applicativi possono raggiungere il database, ma non comunicano tra loro se non necessario. L'Ingress Controller è l'unico punto di ingresso dall'esterno, con terminazione TLS e certificati gestiti automaticamente tramite cert-manager e Let's Encrypt.

3.7.2 Controllo degli accessi al database

PostgreSQL implementa un modello di autorizzazione a tre livelli: un ruolo `maps_writer` con permessi di scrittura sugli schemi Bronze e Silver, un ruolo `maps_reader` con accesso in sola lettura al Gold layer, e un ruolo `maps_api` con accesso limitato alle sole tabelle esposte dalle applicazioni web. Questa separazione garantisce che le pipeline ETL non possano alterare i dati pubblicati, e che i servizi pubblici non possano accedere ai dati grezzi o intermedi.

3.7.3 Gestione dei segreti

Le credenziali sono gestite tramite un sistema di secret management esterno al cluster (AWS Secrets Manager o equivalente), che funge da source of truth per tutti i segreti dell'infrastruttura. I segreti vengono iniettati nei pod Kubernetes tramite External Secrets Operator, che sincronizza automaticamente i valori verso Kubernetes Secrets. Le API key per i servizi esterni sono configurate come Prefect Secret Blocks, anch'essi alimentati dallo stesso meccanismo. Nessuna credenziale è archiviata nei manifest Kubernetes o nel repository del codice.

4 Specifiche infrastruttura hosting

Deliverable D2.1.3: Specifiche Infrastruttura Hosting

4.1 Contesto e premesse

L'infrastruttura MAPS viene costruita da zero su un account DigitalOcean dedicato al progetto GST-MAPS. Il provisioning utilizza Pulumi come strumento IaC (Infrastructure as Code) con lo stesso stack tecnologico descritto nel deliverable D2.1.2: DOKS per l'orchestrazione container, cert-manager per i certificati TLS, external-secrets per la gestione dei segreti, CloudNative PostgreSQL (CNPG) per il database.

Il dominio di riferimento è `maps.gransassotech.it` (o equivalente assegnato da GST). La gestione DNS avviene tramite AWS Route53, con un hosted zone dedicato al progetto.

4.2 Account e credenziali prerequisite

Prima di avviare il provisioning, devono essere disponibili:

Risorsa	Tipo	Note
Account DigitalOcean	Nuovo account GST o progetto separato	Con billing attivo
Token API DigitalOcean	Personal Access Token con scope read+write	Per Pulumi provider DO
Account AWS	Per Route53 e AWS Parameter Store	Account GST dedicato al progetto
Hosted zone Route53	Per il dominio `maps.gransassotech.it`	Zone ID necessario per Pulumi
IAM credentials AWS	Access Key + Secret Key con permessi Route53 + SSM	Per external-dns e external-secrets
Bucket S3	Per lo stato Pulumi	Es. `gst-maps-pulumi-state`
Chiave KMS AWS	Per cifratura stato Pulumi	Alias `pulumi-maps`

4.3 Struttura del progetto Pulumi

Il provisioning viene realizzato con uno stack Pulumi `maps` strutturato come segue:

```
pulumi/maps/
+-- __main__.py          # Entry point
+-- cluster.py           # Creazione cluster DOKS
+-- components.py        # Installazione componenti Helm
+-- security.py          # IAM, network policies, segreti
+-- dns.py               # Record DNS Route53
+-- services.py          # Namespace e configurazione servizi
+-- utils.py             # Helper
+-- Pulumi.yaml          # Definizione progetto
+-- Pulumi.maps.yaml     # Configurazione stack maps
```

```
+-- helm/
|   +-- maps/            # Valori Helm per lo stack maps
|       +-- cert-manager.yaml
|       +-- external-secrets.yaml
|       +-- cnpg.yaml
|       +-- ingress-nginx.yaml
|       +-- external-dns.yaml
|       +-- metrics-server.yaml
+-- manifests/
    +-- maps/            # Manifest Kubernetes aggiuntivi
        +-- cluster-issuer.yaml
```

4.4 Configurazione del cluster DOKS

4.4.1 Parametri del cluster

```yaml
# Pulumi.maps.yaml (estratto)
config:
  env: maps
  defaultDOregion: fra1
  route53Domains:
    - maps.gransassotech.it

  cluster:
    name: maps
    version: "1.33"          # Aggiornare alla versione stabile più recente disponibile
    control_plane_high_availability: false
    node_pools:
      - name: general-purpose
        size: s-2vcpu-4gb
        count: 2
      - name: workloads
        size: s-4vcpu-8gb
        autoscale:
          min: 1
          max: 3
```

4.4.2 Node pool: motivazioni

Il pool `general-purpose` (2 nodi fissi, 2vCPU/4GB) ospita i componenti di sistema: ingress-nginx, cert-manager, external-dns, external-secrets. Il pool `workloads` (1-3 nodi autoscaling, 4vCPU/8GB) ospita i servizi applicativi: Prefect, PostgreSQL (CNPG), OpenMetadata, CKAN. Il dimensionamento è coerente con quello descritto nel deliverable D2.1.2: in condizioni ordinarie 1 nodo workloads è sufficiente, con picchi ETL che attivano l'autoscaling fino a 3 nodi.

4.5 Componenti Helm da installare

Componente	Chart	Repo	Namespace	Note
metrics-server	metrics-server	kubernetes-sigs	kube-system	Metriche HPA
cert-manager	cert-manager	jetstack	cert-manager	CRDs abilitati
ingress-nginx	ingress-nginx	kubernetes-nginx	ingress-nginx	LoadBalancer DO
external-dns	external-dns	kubernetes-sigs	kube-system	Provider AWS Route53
external-secrets	external-secrets	external-secrets	external-secrets	CRDs abilitati
cnpg	cloudnative-pg	cloudnative-pg	cnpg-system	Operator PostgreSQL

I valori Helm per ciascun componente sono configurazioni standard per ognuno dei chart elencati, con i soli adattamenti relativi al dominio e alle credenziali del progetto.

4.5.1 Manifest: ClusterIssuer Let's Encrypt

```yaml
# manifests/maps/cluster-issuer.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: admin@gransassotech.org
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
      - http01:
          ingress:
            ingressClassName: nginx
```

4.6 Gestione DNS

4.6.1 Record richiesti

Record	Tipo	Destinazione
`maps.k8s.maps.gransassotech.it`	A	IP Load Balancer DOKS (gestito da Pulumi/external-dns)
`prefect.maps.gransassotech.it`	CNAME	`maps.k8s.maps.gransassotech.it`
`metadata.maps.gransassotech.it`	CNAME	`maps.k8s.maps.gransassotech.it`
`ckan.maps.gransassotech.it`	CNAME	`maps.k8s.maps.gransassotech.it`

Il record A del Load Balancer viene creato automaticamente da Pulumi tramite `setup_public_lb_dns_record()`. I record CNAME per i singoli servizi vengono creati automaticamente da `external-dns` alla lettura degli `Ingress` Kubernetes.

4.6.2 Credenziali IAM per external-dns

Pulumi crea automaticamente un utente IAM `external-dns-maps` con policy limitata a `route53:ChangeResourceRecordSets` sulla hosted zone di `maps.gransassotech.it`. Le credenziali vengono iniettate come Kubernetes Secret nel namespace `kube-system`.

4.7 Gestione segreti

4.7.1 Struttura AWS Parameter Store

I segreti applicativi sono archiviati in AWS Parameter Store con il seguente schema di path:

```
/maps/{service}/{parameter}

Esempi:
/maps/postgres/password
/maps/prefect/secret-key
/maps/openmetadata/jwt-secret
/maps/ckan/api-key
```

4.7.2 External Secrets Operator

Per ciascun servizio applicativo, Pulumi crea:

1. Un utente IAM `external-secrets-{service}-maps` con policy di accesso in lettura ai parametri `ssm:GetParameter*` sul path `/maps/{service}/*`
2. Le credenziali IAM come Kubernetes Secret nel namespace del servizio
3. Una risorsa `SecretStore` che referenzia le credenziali
4. Una risorsa `ExternalSecret` che mappa i parametri SSM verso Kubernetes Secrets

I segreti vengono sincronizzati con refresh interval di 1 ora.

4.7.3 Segreti gestiti direttamente da Pulumi

I segreti di sistema (non applicativi) vengono archiviati cifrati in `Pulumi.maps.yaml` tramite AWS KMS:

- `external-dns-credentials`: credenziali IAM per external-dns
- `gitlab-runner-secret`: token runner CI/CD (se configurato)

4.8 Database: CloudNative PostgreSQL (CNPG)

Il database PostgreSQL del progetto MAPS viene gestito tramite l'operatore CNPG, già installato nel cluster. La risorsa `Cluster` CNPG per MAPS specifica:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
```

```yaml
  name: maps-postgres
  namespace: maps
spec:
  instances: 2
  imageName: ghcr.io/cloudnative-pg/postgis:17
  postgresql:
    parameters:
      shared_buffers: "256MB"
      max_connections: "100"
  storage:
    size: 100Gi
    storageClass: do-block-storage
  backup:
    barmanObjectStore:
      destinationPath: s3://gst-maps-backups/postgres
      s3Credentials:
        accessKeyId:
          name: maps-backup-credentials
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: maps-backup-credentials
          key: SECRET_ACCESS_KEY
      wal:
        retention: "7d"
    retentionPolicy: "30d"
```

CNPG gestisce automaticamente: replica sincrona tra le 2 istanze, failover automatico, backup WAL continuo su S3, point-in-time recovery.

4.8.1 Storage per backup

È richiesto un bucket S3 dedicato (DigitalOcean Spaces o AWS S3) `gst-maps-backups` con le relative credenziali di accesso in scrittura per CNPG.

4.9 Namespace Kubernetes per MAPS

I servizi applicativi MAPS vengono distribuiti nel namespace `maps`:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: maps
  labels:
    environment: production
    project: gst-maps
```

4.10 Procedura di provisioning

4.10.1 Ordine di esecuzione

L'esecuzione di `pulumi up --stack maps` dalla directory `pulumi/maps/` esegue le seguenti operazioni nell'ordine:

1. Creazione cluster DOKS con i due node pool
2. Creazione policy di rete e utente IAM per external-dns
3. Creazione segreti Kubernetes di sistema
4. Installazione componenti Helm (in parallelo dove possibile)
5. Applicazione manifest Kubernetes (ClusterIssuer, ecc.)
6. Configurazione record DNS del Load Balancer su Route53
7. Configurazione namespace e segreti dei servizi applicativi

4.10.2 Prerequisiti locali

```
# Strumenti necessari sulla macchina di provisioning
pulumi >= 3.0
python >= 3.11
poetry
kubectl
doctl (DigitalOcean CLI)
aws CLI (configurato con profilo maps)
```

4.10.3 Comandi

```
# 1. Configurare profilo AWS
export AWS_PROFILE=maps

# 2. Login stato Pulumi su S3
pulumi login s3://gst-maps-pulumi-state

# 3. Selezione stack
pulumi stack select maps

# 4. Preview (verifica senza applicare)
pulumi preview

# 5. Applicazione
pulumi up
```

4.11 Accesso ai servizi dopo il deployment

Servizio	URL	Autenticazione
Prefect UI	`https://prefect.maps.gter.it`	Credenziali Prefect
OpenMetadata	`https://metadata.maps.gter.it`	Admin / JWT
CKAN	`https://ckan.maps.gter.it`	Account CKAN

Servizio	URL	Autenticazione
PostgreSQL	Interno al cluster (porta 5432)	Solo da pod interni

4.12 Stima dei costi DigitalOcean

Voce	Configurazione	Costo mensile
DOKS control plane	Managed	$12
Pool general-purpose	2x s-2vcpu-4gb	$48 ($24/nodo)
Pool workloads (min)	1x s-4vcpu-8gb	$48
Pool workloads (max)	3x s-4vcpu-8gb	$144
Block storage CNPG	100 GB	$10
Block storage Bronze	100 GB	$10
Load Balancer	Ingress HTTPS	$12
Spaces (backup)	250 GB	$5
Totale (min)		**~$145/mese**
Totale (max, picco ETL)		**~$241/mese**

La differenza rispetto alla stima del D2.1.2 ($311/mese) è dovuta all'adozione di CNPG self-hosted invece del Managed PostgreSQL ($80/mese), e al meccanismo di autoscaling che riduce i costi nelle fasi di inattività.

5 Pipeline ETL e Script Documentati

Deliverable D2.2: Script ETL (Python/SQL) documentati e testati

Il deliverable D2.2 è costituito dagli script ETL che acquisiscono, trasformano e validano i dati per il Data Lake MAPS. Gli script coprono circa 200 dataset provenienti da ISTAT e da istituzioni pubbliche, normalizzati a granularità comunale o sovracomunale e archiviati nel layer Silver EAV. Questo capitolo fornisce una panoramica dell'architettura delle pipeline e descrive le pipeline implementate. La guida completa per gli sviluppatori — convenzioni, pattern, procedure di deployment e un'implementazione di riferimento — è disponibile nell'Appendice A2 — Manuale operativo per lo sviluppo di pipeline ETL.

5.1 Processo di sviluppo delle pipeline

Ogni pipeline del Data Lake MAPS segue una struttura standardizzata a quattro flow che separa l'acquisizione dei dati dalla trasformazione, dalla validazione e dalla catalogazione. La separazione è deliberata: se la logica di trasformazione cambia, è sufficiente rieseguire solo il flow di trasformazione senza riscaricare i dati sorgente; se una regola di validazione viene aggiornata, viene avviato solo il flow di qualità. Questa indipendenza rende lo sviluppo iterativo significativamente più rapido e riduce il rischio di reintrodurre errori quando si corregge una singola fase.

Il processo di sviluppo di una nuova pipeline si articola in quattro fasi:

1. **Flow 1 — Ingestion**: lo sviluppatore identifica la fonte dati, implementa la logica di download o di scraping e scrive i file grezzi nel Bronze layer. Il flow si considera completo quando almeno un file ben formato è disponibile al percorso Bronze atteso e registrato in `bronze.ingestion_log`.
2. **Flow 2 — Transform**: lo sviluppatore legge i file Bronze, risolve gli identificatori territoriali tramite `territory_resolver`, mappa le colonne sorgente sugli attributi EAV e scrive in `silver.territory_attributes`. È il flow più specifico per ogni dataset.
3. **Flow 3 — Data quality**: le suite Great Expectations validano sia i file Bronze che i record Silver rispetto ai criteri di accettazione definiti per completezza, accuratezza e tempestività.
4. **Flow 4 — Metadata**: la pipeline registra i file Bronze e le tabelle Silver in OpenMetadata, creando un'entry di lineage completa dalla fonte al layer EAV.

Ogni pipeline risiede in una directory dedicata sotto `flows/{ente}/{dataset}/` nel repository `gst-maps-pipelines`, contenente i quattro file flow, un manifesto di deployment `prefect.yaml`, un `requirements.txt` e un `README.md` che descrive la fonte, il formato e la frequenza di aggiornamento. La convenzione di naming, le utility condivise, i pattern di lavoro in team e le procedure di deployment sono descritti in dettaglio nell'Appendice A2.

5.2 Pipeline implementate

5.2.1 Fondazione territoriale ISTAT

La fondazione territoriale è un insieme di quattro pipeline coordinate che devono essere eseguite in sequenza prima che qualsiasi pipeline attributo possa caricare dati in Silver. Tutte le pipeline attributo risolvono gli identificatori territoriali tramite `silver.territories`; questa tabella — insieme alle tabelle associate di identificatori, nomi, contenimenti e relazioni — è popolata esclusivamente dalla fondazione territoriale.

flows/istat/variazioni-amministrative scarica la storia completa delle variazioni amministrative ISTAT tramite l'API REST SITUAS all'indirizzo `situas-servizi.istat.it/publish`. SITUAS organizza i dataset per codice `pfun`; la pipeline richiede dieci dataset che coprono regioni (pfun 107, 106, 108), province (pfun 113, 112, 114) e comuni (pfun 129, 98, 104, 105), ciascuno come serie storica completa dal 17 marzo 1861. Il tipo di parametro varia per dataset: alcuni accettano un intervallo di date (`pdatada`/`pdataa`), altri una singola data di riferimento (`pdata`). La utility condivisa `situas_client.py` gestisce entrambe le varianti. Le risposte grezze vengono scritte nel Bronze come CSV sotto `data/bronze/istat/variazioni-amministrative/`.

flows/istat/province-regioni legge i file Bronze delle variazioni e popola `silver.territories`, `silver.territory_identifiers`, `silver.territory_names` e `silver.territory_containments` per ripartizioni, regioni e province. L'ordine di elaborazione è deliberato: prima le ripartizioni, poi le regioni, poi le province, poi gli eventi di variazione (CS = Costituzione, ES = Estinzione). Gli eventi di variazione che hanno interessato il nome o i confini di una provincia vengono applicati in linea, in modo che i timestamp `valid_from` e `valid_to` su ciascun record riflettano il ciclo di vita effettivo dell'ente amministrativo.

flows/istat/comuni è la trasformazione più complessa, articolata in due fasi. La fase 1 utilizza un algoritmo union-find per identificare catene di equivalenza: quando il codice ISTAT di un comune cambia a seguito di un evento AP (Assegnazione Provinciale) o RN (Ridenominazione), i comuni collegati vengono raggruppati sotto un unico `territory_id` stabile. La chiave stabile è il codice catastale Belfiore (`COD_CATASTO`), invariante rispetto alle riorganizzazioni provinciali. La

fase 2 scrive snapshot annuali di nomi, codici ISTAT e contenimenti per ciascun comune, con `valid_from`/`valid_to` derivati dall'evento di variazione corrispondente.

flows/istat/territory-corrections applica un insieme curato di correzioni che non possono essere derivate algoritmicamente dai dati sorgente SITUAS. Utilizza il codice Belfiore come chiave di ricerca ed esegue due operazioni: l'inserimento di nomi alias ADM per i comuni il cui nome ufficiale differisce dall'etichetta del registro amministrativo ADM (consentendo la risoluzione territoriale dai dataset ADM), e la correzione di anomalie di qualità dei dati note nel sorgente SITUAS (valori `valid_to` non aggiornati, record di contenimento mancanti).

Insieme, queste quattro pipeline stabiliscono un grafo completo e versionato temporalmente delle entità territoriali italiane dall'861 a oggi. Ogni pipeline attributo risolve i propri identificatori sorgente in valori `territory_id` tramite `territory_resolver` prima di scrivere in `silver.territory_attributes`.

5.2.2 ADM Tabaccherie

La pipeline ADM Tabaccherie acquisisce l'elenco dei rivenditori di tabacchi autorizzati dal portale web dell'Agenzia delle Dogane e dei Monopoli (ADM). Il portale utilizza un'interfaccia JSF/PrimeFaces che richiede la gestione del token `javax.faces.ViewState`; la utility `html_parser` gestisce questa complessità in modo trasparente. Per ciascun comune, la pipeline interroga il portale, analizza la tabella HTML risultante e scrive le pagine HTML grezze nel Bronze sotto `data/bronze/agenzie-dogane/tabacchi/`.

Il flow di trasformazione risolve gli identificatori comunali tramite `territory_resolver` utilizzando la sigla provinciale e il nome del comune dalla risposta ADM, quindi scrive un record EAV per attributo (numero di punti vendita, indirizzi) in `silver.territory_attributes`. Questa pipeline è l'implementazione di riferimento per il pattern attributo service-count utilizzato dagli altri dataset provenienti da ADM.

Le voci aggiuntive delle pipeline saranno aggiunte man mano che ciascuna pipeline raggiunge la produzione.

5.3 Data quality framework

5.3.1 Suite Great Expectations

Ogni pipeline include due suite Great Expectations: una che valida la struttura del file Bronze e una che valida i record Silver. La suite Bronze verifica che il file sorgente grezzo sia ben formato e contenga le colonne attese; la suite Silver verifica che i record EAV siano completi, coerenti e correttamente collegati a `silver.territories`.

L'esempio seguente mostra la suite Bronze per la pipeline ISTAT variazioni-amministrative:

```
# expectations/istat_variazioni_bronze.py
import great_expectations as gx

suite = gx.core.ExpectationSuite(name="istat_variazioni_bronze")

# Colonne richieste presenti
suite.add_expectation(
    gx.core.ExpectationConfiguration(
```

```
        expectation_type="expect_table_columns_to_match_set",
        kwargs={"column_set": ["COD_COM_STORICO", "COD_CATASTO", "COMUNE_IT",
                               "COD_PROV_STORICO", "DATA_INI_EFF", "DATA_FIN_EFF"]}
    )
)

# Il codice catastale non deve essere null (chiave stabile per le catene di equivalenza)
suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_not_be_null",
        kwargs={"column": "COD_CATASTO"}
    )
)

# Formato data
suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_match_regex",
        kwargs={
            "column": "DATA_INI_EFF",
            "regex": r"^\d{2}/\d{2}/\d{4}$"
        }
    )
)
```

La suite Silver valida che i record di `silver.territory_attributes` scritti da una pipeline attributo siano correttamente collegati:

```
# expectations/territory_attributes_silver.py
import great_expectations as gx

suite = gx.core.ExpectationSuite(name="territory_attributes_silver")

# Nessun territory_id null (ogni record deve risolvere a un territorio noto)
suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_column_values_to_not_be_null",
        kwargs={"column": "territory_id"}
    )
)

# Copertura: almeno il 95% dei comuni attivi deve avere un record
suite.add_expectation(
    gx.core.ExpectationConfiguration(
        expectation_type="expect_table_row_count_to_be_between",
        kwargs={
            "min_value": 7500,  # ~95% dei comuni (7.904 attivi al 2024)
            "max_value": 8500
        }
```

```
    )
)
```

5.4 Monitoring e alerting

5.4.1 Prefect dashboard

- **URL**: https://prefect.maps.gransassotech.it
- **Metrics**: Flow runs, task duration, failure rate
- **Alerting**: Email/Slack su failure

5.4.2 Logging

I log dei worker sono disponibili nell'interfaccia Prefect per ogni flow run e vengono scritti nel volume mappato `data/logs/workers/` sull'host.

[WIP] Questo capitolo sarà completato con: - Documentazione completa di tutte le pipeline prioritarie - Unit test per ogni task - Integration test end-to-end - Performance benchmarks

6 Report di Validazione Pipeline ETL

Deliverable D2.3: Report di Validazione Pipeline ETL

[TODO] Questo capitolo sarà completato dopo la piena implementazione e il testing (Task 2.3, M10-M12)

6.1 Metodologia di validazione

6.1.1 Criteri di accettazione

Le pipeline ETL devono soddisfare cinque classi di criteri di accettazione. L'integrità temporale è il requisito principale per il layer di fondazione territoriale, poiché tutte le pipeline attributo dipendono dalla correttezza del grafo `silver.territories`.

6.1.1.1 Integrità temporale (pipeline di fondazione)

- **Obiettivo**: tutti i record territoriali in `silver.territories` e nelle tabelle correlate portano timestamp `valid_from`/`valid_to` validi derivati dagli eventi di variazione ISTAT; non esistono finestre temporali sovrapposte per la stessa entità; ogni entità attualmente attiva ha `valid_to` IS NULL
- **Metrica**: conteggio delle violazioni di integrità in `territories`, `territory_identifiers`, `territory_names`, `territory_containments`
- **Soglia**: PASS se 0 violazioni; FAIL in caso di qualsiasi violazione

6.1.1.2 Copertura

- **Obiettivo**: tutti i comuni italiani (7.904 al 2024) rappresentati in `silver.territories` con almeno un record attivo
- **Metrica**: `(territori_con_record_attivo / totale_comuni_istat) x 100`
- **Soglia**: FAIL se <99%, WARNING se 99–99,9%, PASS se 100%

6.1.1.3 Accuratezza

- **Obiettivo**: tutti i record in `silver.territory_attributes` portano un `territory_id` valido che risolve a una riga in `silver.territories`
- **Metrica**: `(record_con_territory_id_valido / totale_record) x 100`
- **Soglia**: FAIL se <99,9%

6.1.1.4 Tempestività

- **Obiettivo**: le pipeline di acquisizione completano l'esecuzione entro le finestre di schedulazione definite
- **Metrica**: durata del flow run come riportata da Prefect
- **Soglia**: <24h per le pipeline di fondazione e prioritarie

6.1.1.5 Lineage

- **Obiettivo**: tracciabilità completa dall'URL sorgente → file Bronze → record Silver
- **Metrica**: % dei record Silver con `fonte`, `anno_rif` compilati e una voce corrispondente in `bronze.ingestion_log`
- **Soglia**: 100%

6.1.2 Pipeline di validazione

La suite Great Expectations per ogni pipeline viene eseguita come Flow 3 immediatamente dopo il flow di trasformazione. I risultati sono persistiti nel log del run Prefect. Un'aspettativa fallita blocca il Flow 4 (registrazione dei metadati) per evitare la registrazione di lineage per dati incompleti.

6.2 Validazione delle pipeline di fondazione

6.2.1 Verifiche di integrità temporale

Dopo l'esecuzione delle quattro pipeline di fondazione in sequenza, le seguenti asserzioni SQL devono restituire zero righe o zero conteggi.

```sql
-- (1) Nessun duplicato attivo per lo stesso type_code e id territorio
SELECT type_code, COUNT(*) AS duplicati
FROM silver.territories
WHERE valid_to IS NULL
GROUP BY type_code, id
HAVING COUNT(*) > 1;

-- (2) Nessuna inversione di date (valid_from successivo a valid_to)
SELECT id FROM silver.territories
WHERE valid_to IS NOT NULL AND valid_from > valid_to;

-- (3) Ogni territorio ha almeno un identificatore
SELECT t.id FROM silver.territories t
LEFT JOIN silver.territory_identifiers ti ON ti.territory_id = t.id
WHERE ti.id IS NULL;

-- (4) Ogni comune attivo è contenuto in una provincia
```

```sql
SELECT t.id FROM silver.territories t
WHERE t.type_code = 'COM'
  AND t.valid_to IS NULL
  AND NOT EXISTS (
    SELECT 1 FROM silver.territory_containments tc
    WHERE tc.territory_id = t.id AND tc.container_type_code = 'PRO'
  );
```

6.2.2 Verifica della copertura

```sql
SELECT
  COUNT(*)              AS comuni_attivi,
  7904                  AS baseline_istat_2024,
  ROUND(COUNT(*) * 100.0 / 7904, 2) AS copertura_pct
FROM silver.territories
WHERE type_code = 'COM'
  AND valid_to IS NULL;
```

Metrica	Obiettivo	Risultato	Stato
Comuni attivi	7.904	TBD	in attesa
Copertura	100%	TBD	in attesa
Violazioni temporali	0	TBD	in attesa
Identificatori orfani	0	TBD	in attesa
Violazioni contenimento	0	TBD	in attesa

6.2.3 Validazione delle correzioni territoriali

Correzione	Chiave di ricerca (COD_CATASTO)	Esito atteso	Stato
Baranzate (MI) valid_to non aggiornato	A618	valid_to IS NULL	in attesa
Castegnero (VI) alias ADM	C056	nome alias presente in territory_names	in attesa

6.3 Validazione delle pipeline attributo

6.3.1 ADM Tabaccherie

Metrica	Obiettivo	Risultato	Stato
Copertura comunale	>=95%	TBD	in attesa
Fallimenti di risoluzione	0%	TBD	in attesa
Tempo di esecuzione	<24h	TBD	in attesa
Completezza lineage	100%	TBD	in attesa

I risultati delle pipeline attributo aggiuntive saranno aggiunti man mano che ciascuna pipeline raggiunge la produzione.

6.4 Benchmark di performance

[TODO] Misurazioni reali dopo l'esecuzione completa sui dati di produzione

Pipeline	Record elaborati	Tempo di esecuzione
variazioni-amministrative (ingestion)	~500K eventi	TBD
province-regioni (transform)	~120 entità	TBD
comuni (transform)	~8.000 comuni x finestre temporali	TBD
territory-corrections	~20 patch	TBD
ADM Tabaccherie	~8.000 comuni	TBD

6.5 Problemi identificati e risoluzioni

6.5.1 Problema #1: Instabilità dei codici comunali nelle riorganizzazioni provinciali

Descrizione: i comuni che hanno cambiato codice ISTAT a seguito di eventi di Assegnazione Provinciale (AP) non potevano essere risolti solo tramite codice, producendo record duplicati o mancanti nelle implementazioni naive. **Soluzione**: il flow di trasformazione dei comuni utilizza un algoritmo union-find per raggruppare tutte le varianti di codice dello stesso comune fisico sotto un unico `territory_id` stabile, identificato dal codice catastale Belfiore (`COD_CATASTO`), invariante rispetto alle riorganizzazioni provinciali. Tutti i codici ISTAT storici vengono conservati come voci in `silver.territory_identifiers` con i relativi intervalli `valid_from`/`valid_to`. **Stato**: Risolto

6.5.2 Problema #2: Timeout nello scraping del portale ADM

Descrizione: il portale JSF/PrimeFaces ADM andava in timeout dopo 10 minuti sulle pagine che elencavano molti comuni per provincia. **Soluzione**: timeout aumentato a 30 minuti; implementato retry con backoff esponenziale nella utility `rate_limiter`. **Stato**: Risolto

6.6 Raccomandazioni

6.6.1 Breve termine (prima della chiusura del Task 2.3)

1. Eseguire la pipeline di fondazione completa sui dati di produzione e registrare i risultati nelle tabelle di cui sopra.
2. Aggiungere un vincolo FK a livello di database su `silver.territory_attributes.territory_id` per applicare l'integrità referenziale al layer di storage e non solo al layer applicativo.
3. Configurare gli alert Prefect (email o Slack) sui fallimenti delle pipeline di fondazione, in modo che le regressioni di qualità dei dati vengano rilevate prima dell'esecuzione delle pipeline downstream.

6.6.2 Medio termine (fase di hardening del Task 2.3)

1. Estendere le suite Great Expectations Silver a tutte le pipeline attributo man mano che raggiungono la produzione.
2. Schedulare le verifiche automatiche di integrità temporale (le asserzioni SQL della sezione §2) come flow Prefect giornaliero.
3. Implementare l'acquisizione incrementale per i dataset SITUAS per evitare il re-download completo ad ogni run della pipeline.

6.6.3 Lungo termine

1. Esporre i risultati di validazione delle pipeline di fondazione in OpenMetadata come metriche di qualità del dataset, rendendo visibile lo stato di salute del grafo territoriale nel catalogo dati.
2. Integrare le verifiche di integrità temporale nella pipeline CI in modo che le regressioni nella logica di trasformazione vengano rilevate prima del deployment.

6.7 Conclusioni

[TODO] Riepilogo finale al completamento del testing

La strategia di validazione del Data Lake MAPS tratta l'integrità temporale come il gate di qualità principale. Poiché ogni pipeline attributo risolve infine gli identificatori sorgente in valori `territory_id` mantenuti dalla fondazione territoriale, la correttezza di quella fondazione — espressa attraverso catene `valid_from`/`valid_to` coerenti e gerarchie di contenimento complete — è la condizione necessaria per qualsiasi asserzione di qualità downstream. I criteri di accettazione e le asserzioni SQL definiti in questo capitolo saranno valutati sui dati di produzione man mano che ciascuna pipeline completa la fase di hardening del Task 2.3.

7 Appendice A1 — Motivazioni delle scelte tecnologiche

Questa appendice documenta le motivazioni dettagliate delle scelte tecnologiche adottate per la piattaforma MAPS, con confronti quantitativi tra le alternative considerate per ciascun componente. Il documento è un supporto alla lettura del Documento di Progettazione Tecnica Data-Lake.

7.1 1. PostgreSQL + PostGIS vs BigQuery/Cloud Data warehouse

La scelta di PostgreSQL 17 con estensione PostGIS 3.5, in esercizio self-hosted, nasce dalla combinazione di tre fattori specifici al progetto MAPS. Il primo è la scala dei dati: il dataset di riferimento è composto da circa 10.000 comuni con fino a 1.000 attributi ciascuno — un volume nell'ordine delle decine di gigabyte, non dei petabyte per cui sono progettati i data warehouse cloud come BigQuery. Il secondo è la centralità delle operazioni spaziali: calcolo di isocrone, buffer geografici, intersezioni tra geometrie sono operazioni di routine nel progetto, e PostGIS è lo standard di settore per questo tipo di elaborazioni, con un ecosistema di strumenti e documentazione incomparabilmente più maturo rispetto a BigQuery GIS. Il terzo è il controllo dei costi: PostgreSQL ha costi infrastrutturali fissi e prevedibili (circa €2.400/anno), mentre BigQuery applica una tariffazione per query che, con pattern di accesso intensivi, può variare tra €600 e oltre €6.000/anno senza possibilità di pianificazione.

Criterio	PostgreSQL + PostGIS	BigQuery	Decisione
Scala dati	Ottimizzato 10 -10 righe	Ottimizzato petabyte	PostgreSQL (scala MAPS: ~10 comuni x ~10^3 attributi)
Operazioni spaziali	PostGIS = industry standard	BigQuery GIS = limitato	PostgreSQL (isocrone, buffer, intersezioni native)

Criterio	PostgreSQL + PostGIS	BigQuery	Decisione
Costi	Prevedibili (infra fissa)	Per-query pricing	PostgreSQL (~€2.4k/anno vs €600-6k+/anno imprevedibili)
Vendor lock-in	Zero	Alto	PostgreSQL (principio indipendenza)
Maturità	30+ anni	~15 anni	PostgreSQL (affidabilità consolidata)
Integrazione	Universale	Ecosistema GCP	PostgreSQL (funziona con ogni tool)

BigQuery sarebbe giustificato in scenari con volumi superiori ai 100 GB per singola query, con centinaia di utenti concorrenti, con necessità di un servizio completamente gestito senza overhead operativo, o con un budget dedicato all'analytics superiore ai €5.000 annui. Nessuna di queste condizioni vale per MAPS.

7.2 2. DuckDB vs BigQuery per analytics

Per l'analisi esplorativa dei dati, DuckDB in modalità embedded è la scelta adottata in alternativa a BigQuery. Il motivo principale è la capacità di federazione diretta con PostgreSQL: DuckDB legge le tabelle Gold senza necessità di esportare i dati, eliminando un passaggio ETL intermedio e mantenendo i dati nel sistema di riferimento. A questo si aggiunge l'assenza di costi di licenza — BigQuery fattura circa €5 per terabyte di dati analizzati — e la semplicità d'uso in contesti Python: DuckDB si installa come libreria e non richiede infrastruttura separata. Sui volumi di MAPS (circa 50 GB), le query analitiche tipiche si eseguono in meno di 10 ms, prestazioni equivalenti a quelle di BigQuery su dataset di questa dimensione.

Criterio	DuckDB	BigQuery	Decisione
Dimensionamento	GB scale	TB/PB scale	DuckDB (volumi MAPS: ~50GB)
Costi	Zero	€5/TB query	DuckDB (risparmio €600-1.200/anno)
Performance	ms su GB	ms su TB	DuckDB (query $<$ 10ms sul volume MAPS)
Federation	Legge da PostgreSQL	Richiede export	DuckDB (no ETL aggiuntivo)
Portabilità	File auto-contenuto	Vendor lock-in	DuckDB (export Parquet portabile)
Developer UX	Embedded Python	API REST	DuckDB (zero overhead setup)

7.3 3. Prefect vs Airflow/Dagster

Prefect 3.x è stato scelto come orchestratore delle pipeline in ragione del basso overhead di adozione rispetto alle alternative. Airflow è lo strumento più diffuso nel settore, ma richiede un'infrastruttura complessa, una curva di apprendimento ripida e un carico operativo elevato, difficilmente giustificabile per un progetto delle dimensioni di MAPS. Dagster offre un'architettura moderna e una buona esperienza utente, ma ha anch'esso una curva di apprendimento più impegnativa. Prefect consente di passare da zero a pipeline operative in poche ore, con decoratori Python semplici, un'interfaccia web moderna e un'architettura che separa nettamente server di orchestrazione e worker di esecuzione.

Criterio	Prefect	Airflow	Dagster	Decisione
Setup complexity	Basso	Alto	Medio	Prefect
Learning curve	Gentile	Ripida	Ripida	Prefect
Python-native	Completo	Parziale	Completo	Prefect
Time-to-value	Rapido (ore)	Lento (giorni)	Medio	Prefect
UI/UX	Moderna	Datata	Moderna	Prefect/Dagster
Overhead operativo	Basso	Alto	Medio	Prefect

7.4 4. Docling vs alternative PDF extraction

Una parte significativa dei dataset acquisiti dal progetto è distribuita in formato PDF, spesso con tabelle complesse. La libreria Docling di IBM, basata sul modello TableFormer, raggiunge una precisione del 97,9% sull'estrazione di tabelle da PDF in un benchmark indipendente su documenti finanziari complessi. Camelot e Tabula, le librerie tradizionalmente più diffuse per questo tipo di operazioni, sono entrambe in modalità manutenzione e non più in sviluppo attivo; le valutazioni comparative disponibili indicano prestazioni significativamente inferiori su tabelle complesse rispetto a Docling. Docling è rilasciata con licenza MIT, integra nativamente l'esportazione in formato Pandas DataFrame, ed è interamente self-hosted, senza dipendenze da servizi cloud. Per i PDF senza tabelle complesse viene utilizzato PyMuPDF come fallback, e pdfplumber per i documenti con layout multi-colonna o strutture particolarmente problematiche. Azure Document Intelligence, pur con prestazioni paragonabili (~95%), è stata esclusa per il vincolo di vendor lock-in e i costi imprevedibili.

Libreria	Accuracy	Licenza	Sviluppo attivo	Pandas nativo	Decisione
Docling (TableFormer AI)	97.9%	MIT	Sì (2025, LF AI)	Sì	Primary
pdfplumber	85-90%	MIT	Sì	Parziale	Fallback
PyMuPDF	75-80%	AGPL-3.0	Sì	Parziale	Fallback
Camelot	~70%	MIT	Maintenance mode	No	Escluso
Tabula	~70%	MIT	Maintenance mode	No	Escluso
Azure Doc Intelligence	~95%	Commercial	Sì	Sì	Escluso (vendor lock-in)

7.5 5. OpenMetadata vs DataHub/Atlan/Atlas

OpenMetadata 1.x è lo strumento scelto per la governance interna dei dati. Rispetto alle alternative, combina un costo infrastrutturale contenuto (circa €1.200/anno per una VM dedicata), un'installazione su singola macchina, un'interfaccia moderna e un'integrazione diretta con lo stack PostgreSQL/Prefect del progetto. DataHub richiede due o tre macchine virtuali e ha costi leggermente superiori; Atlan offre un'esperienza utente eccellente ma con licenze commerciali nell'ordine di €12.000-30.000/anno; Apache Atlas è progettato per ecosistemi Hadoop e richiederebbe oltre dieci macchine virtuali per funzionare correttamente, risultando del tutto sproporzionato. OpenMetadata è operativo in uno-due giorni e non introduce dipendenze da fornitori.

Criterio	OpenMetadata	DataHub	Atlan	Atlas	Decisione
Costo annuale	€1.2k (infra)	€1.8-2.4k	€12-30k (lic.)	€3-5k	OpenMetadata
Setup	1 VM	2-3 VMs	1 VM	11+ VMs	OpenMetadata
UI/UX	Moderna	Funzionale	Eccellente	Datata	OpenMetadata
Stack fit	PostgreSQL/Prefect	Buono	Buono	Hadoop-only	OpenMetadata
Vendor lock-in	Zero	Zero	Alto	Zero	OpenMetadata
Time-to-value	1-2 giorni	2-4 giorni	Medio	Settimane	OpenMetadata

7.6 6. CKAN per Open Data vs uso esteso OpenMetadata

La pubblicazione open data richiede uno strumento distinto da OpenMetadata, che è ottimizzato per la governance interna e non supporta lo standard DCAT-AP_IT né l'integrazione con il portale nazionale dati.gov.it. CKAN 2.11, self-hosted, è la scelta adottata per il catalogo pubblico: implementa nativamente DCAT-AP_IT, include un harvester verso dati.gov.it, espone un portale web ottimizzato per utenti non tecnici, e gestisce download multi-formato con anteprima dei dati. Estendere OpenMetadata per coprire questi requisiti richiederebbe uno sviluppo custom stimato in due o tre mesi-persona, senza garanzie di conformità agli standard.

Criterio	CKAN	OpenMetadata esteso	Decisione
Target audience	Pubblico esterno	Data operators interni	CKAN (separation of concerns)
Standard DCAT-AP_IT	Nativo	Non supportato	CKAN (requisito WP6 D6.3)
Integrazione dati.gov.it	Built-in harvester	Richiede sviluppo custom	CKAN (API standard CSW/DCAT)
Portale user-friendly	Web UI ottimizzata pubblico	UI tecnica data engineers	CKAN (esperienza utente)
Rate limiting & API keys	Nativo	Limitato	CKAN (controllo accessi pubblici)

Mermaid Diagram

Figure 7: Mermaid Diagram

Criterio	CKAN	OpenMetadata esteso	Decisione
Dataset downloads	Multi-formato con preview	Solo metadata	CKAN (full data access)
SEO e discoverability	Ottimizzato motori ricerca	Non ottimizzato	CKAN (findability pubblica)
Licensing metadata	Completo (Creative Commons)	Basic	CKAN (compliance open data)

I due strumenti svolgono ruoli complementari e non sovrapposti. OpenMetadata riceve i metadati tecnici da Prefect durante l'esecuzione delle pipeline, traccia la lineage Bronze → Silver → Gold, registra i risultati delle validazioni Great Expectations e fornisce al team interno un catalogo operativo. CKAN riceve dal Gold layer i soli dataset approvati per la pubblicazione, li arricchisce con metadati DCAT-AP_IT completi (licenza, copertura temporale, estensione geografica, metodologia) e li rende accessibili a cittadini, ricercatori e sviluppatori terzi, con harvesting automatico verso dati.gov.it.

Dal punto di vista dei costi, aggiungere CKAN all'infrastruttura comporta un incremento di circa €600/anno per la VM dedicata, a fronte di un risparmio di €15.000-20.000 in sviluppo custom che sarebbe altrimenti necessario per adeguare OpenMetadata agli standard open data.

Voce	OpenMetadata solo	OpenMetadata + CKAN	Delta
Infra (VM)	€1.2k/anno	€1.8k/anno	+€600/anno
Sviluppo custom	€15-20k (DCAT-AP_IT)	€0	-€15-20k
Manutenzione	Alta (custom code)	Bassa (standard stack)	Significativo

8 Appendice A2 — Manuale operativo per lo sviluppo di pipeline ETL

> **Deliverable D2.2 — Appendice**: Guida operativa per la costruzione e la manutenzione delle pipeline ETL nel Data Lake MAPS.

Questa guida è rivolta agli sviluppatori che entrano nel progetto e devono costruire una nuova pipeline dati o manutenere una esistente. Descrive il flusso di lavoro standard, le convenzioni e i pattern adottati in tutte le pipeline dell'infrastruttura MAPS. L'implementazione di riferimento è la pipeline Agenzia Dogane e Monopoli — tabaccherie; il codice sorgente completo è disponibile in `flows/agenzie-dogane/tabacchi/` nel repository.

8.1 1. Quickstart — Ambiente di sviluppo

Per avviare l'ambiente di sviluppo locale sono necessari Python 3.11, Docker e Docker Compose. I passi seguenti configurano il database, il server Prefect e l'interprete Python in circa dieci minuti.

8.1.1 1.1 Prerequisiti

- Python 3.11
- Docker 24+
- Docker Compose v2

8.1.2 1.2 Clona il repository

```
git clone https://gitlab.openpolis.io/openpolis/gst/gst-maps-pipelines.git
cd gst-maps-pipelines
```

8.1.3 1.3 Variabili d'ambiente

```
cp .env.example .env
```

I valori predefiniti in `.env.example` funzionano senza modifiche per lo sviluppo locale. L'unica variabile che vale la pena verificare è `BRONZE_BASE_PATH`:

```
# .env.example - commentata, non necessaria in sviluppo locale
# BRONZE_BASE_PATH=/data/bronze
```

In sviluppo locale i flow calcolano automaticamente il path Bronze come `{repo_root}/data/bronze` (derivato dalla posizione del file). In produzione, dentro il container `prefect-worker`, impostare `BRONZE_BASE_PATH=/data/bronze` — il volume Docker è già montato in quella posizione.

I flow caricano `.env` automaticamente all'avvio tramite `python-dotenv` (incluso in `requirements-dev.txt`): non è necessario esportare le variabili manualmente nella shell prima di eseguire uno script.

8.1.4 1.4 Avvio dei servizi

```
docker compose -f docker-compose.local.yml up -d
```

Avvia PostgreSQL 17 + PostGIS 3.5, il server Prefect e OpenMetadata (con il suo database dedicato ed Elasticsearch). Lo schema del database MAPS (schemi `bronze`, `silver`, `gold`, tabelle e indici) viene creato automaticamente al primo avvio tramite `postgres/init-scripts/01-init-schemas.sql`. Verificare che i servizi siano attivi:

```
docker compose -f docker-compose.local.yml ps
```

Se si modifica `docker-compose.local.yml` o `.env` dopo che i container sono già in esecuzione, `docker compose restart` non rilegge le variabili d'ambiente. Usare invece:

```
docker compose -f docker-compose.local.yml up -d --force-recreate
```

8.1.5 1.5 Installazione dipendenze Python

```
python3.11 -m venv .venv
source .venv/bin/activate
pip install -r requirements-dev.txt
```

Questo installa Prefect, psycopg2, requests, BeautifulSoup e `python-dotenv`. Il virtualenv va attivato ogni volta che si apre una nuova sessione di terminale (`source .venv/bin/activate`).

8.1.6 1.6 Configurazione Prefect

```
export PREFECT_API_URL=http://localhost:4200/api
prefect work-pool create default-pool --type process
```

Il work pool `default-pool` è il pool di esecuzione referenziato nel `prefect.yaml` di ogni pipeline.

8.1.7 1.7 Verifica

- Prefect UI: http://localhost:4200
- OpenMetadata UI: http://localhost:8585 (credenziali iniziali: `admin@open-metadata.org` / `admin`)
- Database: `psql -h localhost -U maps -d maps_db` (password: `maps_dev`)

Verificare che gli schemi e le tabelle siano stati creati correttamente:

```
-- schemi attesi: bronze, silver, gold
\dn

-- tabelle attese: bronze.ingestion_log, silver.territory_types,
--   silver.territories, silver.territory_containments,
--   silver.territory_identifiers, silver.territory_names,
--   silver.territory_attributes,
--   silver.service_types, silver.services, silver.service_identifiers,
--   silver.service_attributes, silver.service_categories
\dt bronze.*
\dt silver.*
```

L'ambiente è pronto. Il layer Bronze è un volume Docker (`bronze-data`) montato in `/data` all'interno dei container; i file grezzi scaricati dai flow saranno visibili in quel volume.

OpenMetadata impiega alcuni minuti per completare la migrazione del database al primo avvio; attendere che il container `maps-openmetadata` riporti lo stato `healthy` prima di aprire l'interfaccia.

8.2 2. Architettura delle pipeline

8.2.1 2.1 Prefect: architettura client/server

Prefect segue un'architettura client/server. Il **server** (container `maps-prefect-server`) tiene traccia delle esecuzioni, conserva i log e serve la UI su http://localhost:4200. Il **worker** (container `maps-prefect-worker`) è il processo che esegue materialmente i flow: interroga il server in cerca di lavoro assegnato al suo pool (`default-pool`) e avvia ogni flow in un subprocess.

Lo sviluppatore interagisce con il server tramite il client Prefect — la CLI e l'SDK Python — la cui destinazione è configurata dalla variabile d'ambiente `PREFECT_API_URL`. Durante lo sviluppo locale il client punta al server nel container; in produzione punterà al server remoto, senza che il codice dei flow cambi.

Un **deployment** associa un flow a un pool di worker e ne fissa i parametri di esecuzione (schedule, parametri di default). Una volta registrato, il flow può essere avviato dalla UI o da CLI senza che

Mermaid Diagram

Figure 8: Mermaid Diagram

lo sviluppatore debba avere il processo attivo localmente.

8.2.2 2.2 Pattern Medallion

Ogni pipeline MAPS segue il **pattern Medallion**: i file grezzi vengono scaricati nel Bronze layer, trasformati e validati nel Silver layer, infine aggregati nel Gold layer. Le pipeline sono orchestrate da Prefect e seguono una struttura standard a quattro flow.

```
Fonte esterna → Flow 1: Ingestion → Bronze (filesystem)
             → Flow 2: Transform  → Silver (PostgreSQL)
             → Flow 3: Quality    → Report di validazione
             → Flow 4: Metadata   → Catalogo OpenMetadata
```

Ogni flow è indipendente e può essere rieseguito senza dover rieseguire i flow precedenti. Questo è intenzionale: se cambia la logica di trasformazione, è sufficiente rieseguire il Flow 2 senza dover ri-scaricare i dati dalla fonte.

8.3 3. Struttura delle directory

Ogni pipeline risiede in una sottodirectory dedicata sotto `flows/`:

```
flows/
+-- utils/                        # Moduli condivisi tra tutte le pipeline
|   +-- bronze_writer.py          # save_to_bronze(), log_ingestion()
|   +-- rate_limiter.py           # rate_limited_request()
|   +-- html_parser.py            # parse_table(), extract_form_data()
|   +-- situas_client.py          # SituasClient - wrapper API ISTAT SITUAS
|   +-- territory_resolver.py     # build_comune_lookup_from_conn(), resolve_comune()
|   +-- service_writer.py         # upsert_service(), upsert_service_identifier(), ...
+-- {ente}/{dataset}/
    +-- 01_ingestion_flow.py      # Flow 1: download → Bronze
    +-- 02_transform_flow.py      # Flow 2: Bronze → Silver
    +-- 03_data_quality_flow.py   # Flow 3: validazione
    +-- 04_metadata_flow.py       # Flow 4: catalogo OpenMetadata
    +-- prefect.yaml              # Definizioni dei deployment (tutti e 4 i flow)
    +-- requirements.txt          # Dipendenze Python
    +-- README.md                 # Documentazione della pipeline
    +-- docs/
    |   +-- DEPLOY_GUIDE.md       # Istruzioni di deployment passo per passo
    |   +-- KNOWN_ISSUES.md       # Storico dei problemi e relative soluzioni
    +-- tests/
        +-- test_flow.py          # Test unitari
```

Convenzioni di denominazione: - Directory: minuscolo con trattini (`agenzie-dogane/tabacchi/`) - File dei flow: sempre numerati `01_`, `02_`, `03_`, `04_` - README: obbligatorio, descrive scopo, URL della fonte, formato e frequenza di aggiornamento

8.4 4. Pipeline territoriali ISTAT

La directory `flows/istat/` contiene le pipeline fondazionali che popolano il modello territoriale Silver. Tutte le altre pipeline dati dipendono dalla presenza di questi dati.

Pipeline	Directory	Fonte	Copertura
Province, regioni, ripartizioni	`flows/istat/province-regioni/`	SITUAS pfun=64/68/71 + variazioni pfun=106–108, 112–114	~107 province, 20 regioni, 5 ripartizioni per anno (2000–oggi)
Comuni	`flows/istat/comuni/`	SITUAS pfun=61 + variazioni pfun=129	~7.900 comuni per anno (2000–oggi)
Variazioni amministrative	`flows/istat/variazioni-amministrative/`	SITUAS pfun=106–108, 112–114, 98, 104, 105, 129	Solo ingestione — scarica la storia completa dal 1861 nel Bronze
Territory corrections	`flows/istat/territory-corrections/`	Dataset — nomi alias ADM e correzioni di qualità	Flow idempotente; va eseguito dopo che i comuni sono popolati

L'ordine di esecuzione è importante. Il processamento delle variazioni è integrato nei flow di trasformazione di province-regioni e comuni. I dati Bronze delle variazioni devono essere ingeriti prima. Il flow territory-corrections va eseguito per ultimo, dopo che i comuni sono popolati.

```
Passo 1 - Ingestione (qualsiasi ordine, eseguibili in parallelo):
  istat-province-regioni → Flow 1 (ingestion)
  istat-comuni           → Flow 1 (ingestion)
  istat-variazioni       → Flow 1 (ingestion)   ← deve completarsi prima del Passo 2

Passo 2 - Trasformazione (ordine vincolato):
  istat-province-regioni → Flow 2 (transform)   ← legge bronze province + bronze variazioni
  istat-comuni           → Flow 2 (transform)   ← legge bronze comuni + bronze variazioni + sil

Passo 3 - Correzioni (dopo che i comuni sono popolati):
  istat-territory-corrections → Flow 1 (corrections)
```

8.4.1 4.1 Esecuzione durante lo sviluppo

Ogni file di flow accetta `--anni` come argomento da riga di comando. I flow possono essere eseguiti direttamente via Python (con il virtualenv attivo) o via Docker (raccomandato):

```
# Opzione A: Python direttamente (virtualenv attivo)

# Passo 1: Ingestione (qualsiasi ordine)
python flows/istat/variazioni-amministrative/01_ingestion_flow.py
```

```
python flows/istat/province-regioni/01_ingestion_flow.py
python flows/istat/comuni/01_ingestion_flow.py

# Passo 2: Trasformazione (ordine vincolato - bronze variazioni deve esistere prima)
python flows/istat/province-regioni/02_transform_flow.py
python flows/istat/comuni/02_transform_flow.py

# Passo 3: Correzioni (dopo che i comuni sono popolati)
python flows/istat/territory-corrections/01_corrections_flow.py

# Solo anni specifici
python flows/istat/province-regioni/01_ingestion_flow.py --anni 2024 2025 2026

# Opzione B: Docker (raccomandato)

# Passo 1: Ingestione
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/variazioni-amministrative && python 01_ingestion_flow.py"
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/province-regioni && python 01_ingestion_flow.py"
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/comuni && python 01_ingestion_flow.py"

# Passo 2: Trasformazione (province-regioni prima, poi comuni)
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/province-regioni && python 02_transform_flow.py"
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/comuni && python 02_transform_flow.py"

# Passo 3: Correzioni
docker exec maps-prefect-worker bash -c \
  "cd /flows/istat/territory-corrections && python 01_corrections_flow.py"
```

8.4.2 4.2 Esecuzione via deployment Prefect

Tutte le pipeline ISTAT sono registrate sul server Prefect senza schedule automatico; le esecuzioni vengono avviate manualmente. Per registrare o ri-registrare dopo una modifica al `prefect.yaml`:

```
docker exec maps-prefect-worker bash -c "cd /flows/istat/province-regioni && prefect deploy --a
docker exec maps-prefect-worker bash -c "cd /flows/istat/comuni && prefect deploy --all"
docker exec maps-prefect-worker bash -c "cd /flows/istat/variazioni-amministrative && prefect d
docker exec maps-prefect-worker bash -c "cd /flows/istat/territory-corrections && prefect deplo
```

Per avviare una run via CLI (rispettare l'ordine di esecuzione):

```
export PREFECT_API_URL=http://localhost:4200/api

# Passo 1: Ingestione (qualsiasi ordine)
prefect deployment run 'istat-variazioni-ingestion/istat-variazioni-01-ingestion'
prefect deployment run 'istat-province-regioni-ingestion/istat-pr-01-ingestion' \
```

```
  --param anni='[2024,2025,2026]'
prefect deployment run 'istat-comuni-ingestion/istat-comuni-01-ingestion' \
  --param anni='[2024,2025,2026]'

# Passo 2: Trasformazione (attendere il completamento dell'ingestione; province-regioni prima d
prefect deployment run 'istat-province-regioni-transform/istat-pr-02-transform' \
  --param anni='[2024,2025,2026]'
prefect deployment run 'istat-comuni-transform/istat-comuni-02-transform' \
  --param anni='[2024,2025,2026]'

# Passo 3: Correzioni (dopo il completamento della trasformazione dei comuni)
prefect deployment run 'istat-territory-corrections/istat-territory-corrections-01'
```

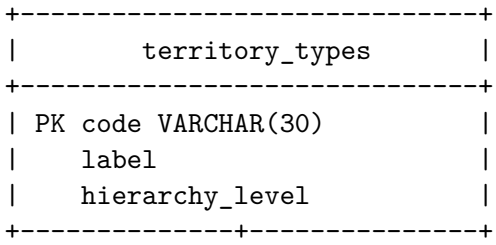
Tutti i flow ISTAT sono idempotenti: rieseguire una trasformazione su un Bronze invariato produce un output Silver identico.

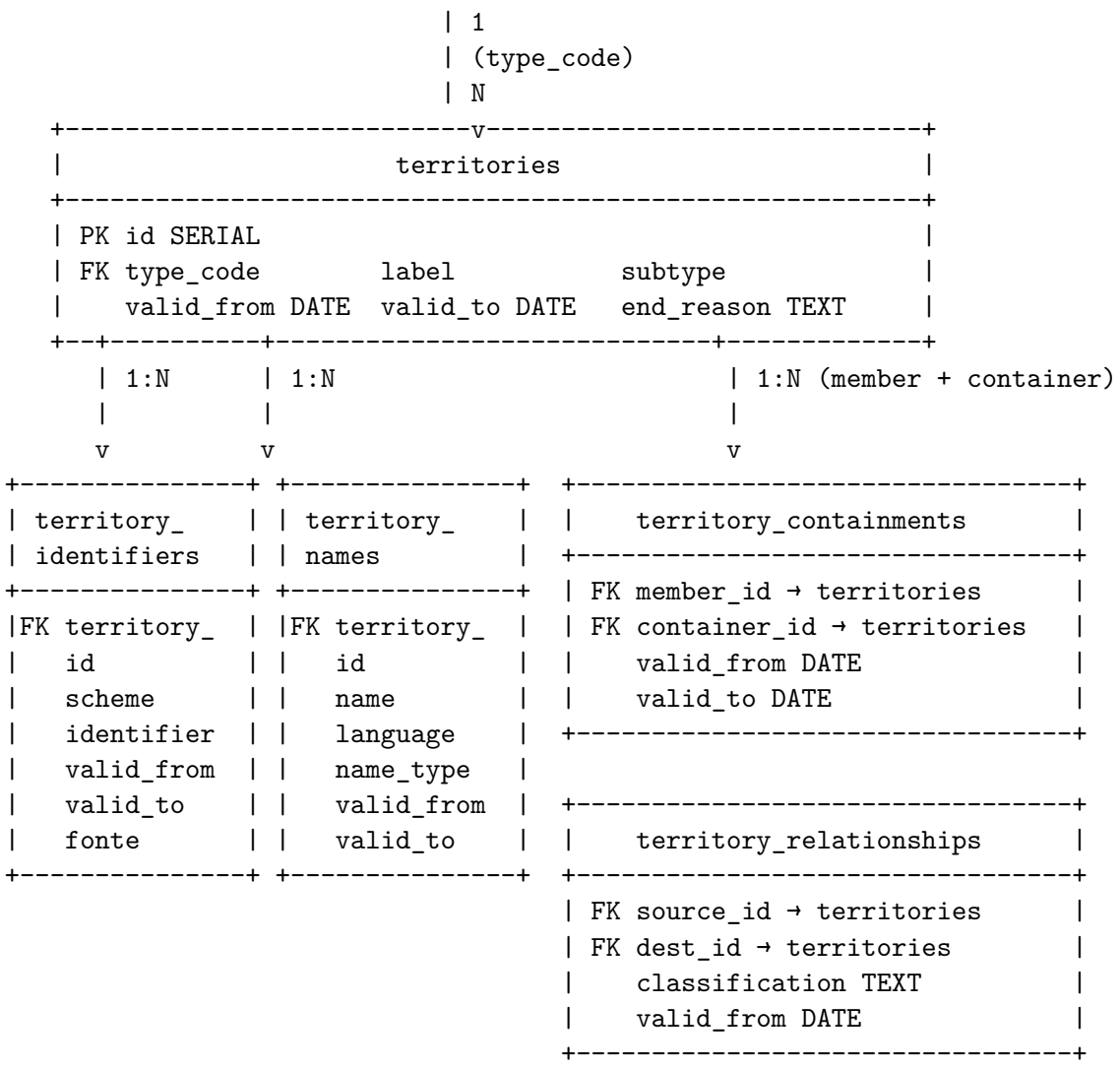
8.5 5. Silver layer

8.5.1 5.1 Modello delle entità territoriali

Il Silver layer adotta un modello a identità surrogata stabile per i territori. Ogni territorio ha un `id` interno che non cambia mai; i codici ISTAT sono memorizzati come identificatori in `silver.territory_identifiers`. Questo elimina la duplicazione dei record quando i comuni cambiano provincia — il comune rimane una sola riga, con più periodi di codice ISTAT.

Tabella	Scopo
`silver.territory_types`	Registro dei tipi di entità — pre-popolato con 9 tipi
`silver.territories`	Registro principale: `id` surrogato stabile, `type_code`, `label`, date di ciclo di vita (`valid_from`/`valid_to`)
`silver.territory_identifiers`	Identificatori temporali: codici ISTAT (`scheme='istat'`), codici catastali, codici fiscali, codici UTS
`silver.territory_names`	Nomi temporali con supporto multilingue (italiano e bilinguismi/locali)
`silver.territory_containments`	Contenimento gerarchico temporale (comune → provincia, con date precise per i cambi di provincia)
`silver.territory_relationships`	Legami predecessore/successore da eventi variazioni (ES, CS, RN, AQES, CECS)





I territori vengono risolti per codice ISTAT tramite un join su `territory_identifiers`:

```
-- Trovare il territorio per il codice ISTAT '092001'
SELECT t.id, t.label FROM silver.territories t
JOIN silver.territory_identifiers ti ON ti.territory_id = t.id
WHERE t.type_code = 'comune' AND ti.scheme = 'istat' AND ti.identifier = '092001';

-- In quale provincia si trovava Arbus il 1° gennaio 2015?
SELECT p.label FROM silver.territory_containments tc
JOIN silver.territories p ON p.id = tc.container_id AND p.type_code = 'provincia'
JOIN silver.territories t ON t.id = tc.member_id
WHERE t.type_code = 'comune' AND t.label = 'Arbus'
  AND (tc.valid_from IS NULL OR tc.valid_from <= '2015-01-01')
  AND (tc.valid_to IS NULL OR tc.valid_to > '2015-01-01');
```

`valid_from = NULL` indica un inizio di validità sconosciuto (precedente ai dati disponibili). `valid_to = NULL` indica che il territorio è ancora attivo.

8.5.2 5.2 Tabella EAV

Tutte le pipeline scrivono in `silver.territory_attributes`, la tabella EAV generalizzata che supporta qualsiasi tipo di entità territoriale.

```
INSERT INTO silver.territory_attributes
    (territory_id, type_code, attribute, value, data_type, source, valid_from, valid_to)
VALUES
    (42,  'comune',    'popolazione',   '2635',  'integer', 'ISTAT', '2024-01-01', NULL),
    (107, 'provincia', 'n_tabaccherie', '245',   'integer', 'ADM',   '2024-01-01', '2025-12-31'
    (8,   'sll',       'addetti',       '15200', 'integer', 'ISTAT', '2021-01-01', NULL);
```

Campo	Descrizione	Esempi
`territory_id`	FK verso `silver.territories(id)` — surrogato stabile, sopravvive ai cambi di codice ISTAT	risolto tramite `territory_resolver`
`type_code`	Tipo di entità (denormalizzato da `territories`)	`'comune'`, `'provincia'`, `'regione'`, `'sll'`, `'slo'`
`attribute`	Nome dell'attributo	`'popolazione'`, `'n_tabaccherie_adm'`
`value`	Valore memorizzato come testo	`'2635'`
`data_type`	Suggerimento per il cast del tipo	`'integer'`, `'float'`, `'string'`, `'boolean'`
`source`	Origine del dato	`'ISTAT'`, `'ADM'`, `'MinSalute'`
`valid_from`	Inizio del periodo di validità	`'2024-01-01'`
`valid_to`	Fine del periodo di validità — `NULL` indica il valore corrente	`NULL`

Il pattern `ON CONFLICT ... DO UPDATE` garantisce la riesecuzione idempotente:

```
execute_values(
    cur,
    """
    INSERT INTO silver.territory_attributes
        (territory_id, type_code, attribute, value,
         data_type, source, valid_from, valid_to)
    VALUES %s
    ON CONFLICT (territory_id, attribute, valid_from)
    DO UPDATE SET value = EXCLUDED.value, updated_at = NOW()
    """,
    eav_records
)
```

8.5.3 5.3 Modello dei servizi locali

`silver.territory_attributes` memorizza una riga per ogni *(territorio, attributo, periodo)*: è ottimizzata per statistiche territoriali aggregate come conteggi di popolazione o numero di farmacie per comune. Non è progettata per memorizzare record individuali di punti di interesse.

Diverse fonti pubblicano elenchi di servizi locali individuali (farmacie, ospedali, scuole, tabaccherie e altri). Conservare i record individuali nel Silver offre vantaggi concreti rispetto alla sola scrittura di conteggi aggregati: gli stessi dati grezzi possono essere aggregati per comune, provincia, ASL o SLL senza dover re-ingerire dalla fonte; i record provenienti da più dataset possono essere deduplicati tramite `service_identifiers` usando codici esterni come codice fiscale o codice HSP; attributi specifici del servizio possono essere memorizzati senza modifiche allo schema.

Tabella	Scopo
`silver.service_types`	Registro dei tipi di servizio — pre-popolato con 5 tipi (`farmacia`, `ospedale`, `scuola`, `biblioteca`, `tabaccheria`)
`silver.services`	Una riga per ogni servizio individuale, con `territory_id` risolto, date di ciclo di vita e un vincolo di unicità per upsert idempotenti
`silver.service_identifiers`	Codici esterni per servizio (`codice_fiscale`, `codice_ministeriale`, `codice_hsp`, …) — stesso identificatore significa stesso servizio logico tra dataset diversi
`silver.service_attributes`	EAV per metriche specifiche del servizio (`posti_letto`, `num_studenti`, …) — rispecchia `territory_attributes` ma legato a una riga di servizio
`silver.service_categories`	Tag tipizzati multivalore (`tipo_ospedale: DEA II livello`, `livello_scuola: primaria`, …)

La scrittura dei servizi da una pipeline avviene tramite gli helper in `utils/service_writer.py`:

```
from utils.territory_resolver import build_comune_lookup_from_conn, resolve_comune
from utils.service_writer import (
    upsert_service, upsert_service_identifier,
    upsert_service_attribute, upsert_service_category,
)

conn = _get_db_conn()
lookup = build_comune_lookup_from_conn(conn)

with conn.cursor() as cur:
    for record in source_records:
        # 1. Risolvere il territorio
        match = resolve_comune(lookup, sigla=record["provincia"], nome=record["comune"])
        territory_id = match[0] if match else None

        # 2. Upsert della riga servizio
```

```
        service_id = upsert_service(
            cur,
            type_code="farmacia",
            name=record["denominazione"],
            address=record["indirizzo"],
            territory_id=territory_id,
            source="AIFA",
            valid_from="2024-01-01",
        )

        # 3. Allegare identificatore esterno (per deduplicazione cross-source)
        upsert_service_identifier(cur, service_id=service_id,
                                  scheme="codice_fiscale", identifier=record["cf"])

        # 4. Memorizzare metrica specifica del servizio
        upsert_service_attribute(cur, service_id=service_id,
                                 attribute="tipo_farmacia", value=record["tipo"],
                                 data_type="text", source="AIFA", valid_from="2024-01-01")

        # 5. Aggiungere categoria
        upsert_service_category(cur, service_id=service_id,
                                scheme="tipo_farmacia", value=record["tipo"])

conn.commit()
conn.close()
```

Una volta che i record individuali sono nel Silver, i conteggi per territorio sono una semplice query GROUP BY che può alimentare `silver.territory_attributes` come statistiche aggregate.

8.6 6. La pipeline di riferimento: tabaccherie ADM

La pipeline Agenzia Dogane e Monopoli — tabaccherie è la pipeline di riferimento perché è la più complessa tra quelle sviluppate: richiede scraping HTML di un portale JSF con paginazione Ajax, gestione della sessione, rate limiting e normalizzazione dei codici territoriali. Una pipeline che scarica direttamente un file CSV o XLSX segue lo stesso schema ma con un Flow 1 molto più semplice.

8.6.1 6.0 Sviluppo locale

Prima di registrare un deployment su Prefect, ogni flow può essere eseguito direttamente come script Python. Questo è il modo normale di lavorare: scrivere il flow, eseguirlo, correggere gli errori, rieseguirlo — senza toccare il server.

Ogni file di flow deve contenere un blocco `if __name__ == "__main__"` con `argparse` in modo che i parametri possano essere passati da riga di comando senza modificare il sorgente:

```
# 01_ingestion_flow.py
from prefect import flow

@flow
```

```python
def ingestion_flow(anno: int = 2025):
    ...

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--anno", type=int, default=2025)
    args = parser.parse_args()
    ingestion_flow(anno=args.anno)
```

Usare sempre lo stesso metodo di esecuzione all'interno di una run di pipeline. Python scrive i file Bronze in `{repo_root}/data/bronze/` sull'host; Docker scrive in `/data/bronze/` dentro il container. Se si ingerisce con Python e si trasforma con Docker (o viceversa), il flow di trasformazione non troverà i file Bronze. In `docker-compose.local.yml` la directory Bronze è montata in bind da `./data`, quindi entrambi i metodi condividono lo stesso path — ma solo se `BRONZE_BASE_PATH` è impostato in modo coerente (v. §1.3).

Opzione A — Python direttamente (il virtualenv deve essere attivo):

```
source .venv/bin/activate
cd flows/agenzie-dogane/tabacchi/

python 01_ingestion_flow.py              # usa i parametri predefiniti
python 01_ingestion_flow.py --anno 2024  # override a runtime
```

Opzione B — Docker (raccomandato — replica esattamente l'ambiente di produzione):

```
docker exec maps-prefect-worker bash -c \
  "cd /flows/agenzie-dogane/tabacchi && python 01_ingestion_flow.py"

docker exec maps-prefect-worker bash -c \
  "cd /flows/agenzie-dogane/tabacchi && python 01_ingestion_flow.py --anno 2024"
```

Se `PREFECT_API_URL` è impostata e il server è in esecuzione, la run appare nella UI con log e stato. Se il server non è disponibile, il flow gira comunque come script Python normale.

L'iterazione tipica è:

1. Scrivere o modificare il flow
2. Eseguire via Docker (o Python direttamente per modifiche rapide)
3. Correggere errori, ripetere
4. Quando funziona, procedere al deployment (§8)

Ogni flow può essere sviluppato e testato indipendentemente dagli altri. Il Flow 2 può essere testato leggendo un file Bronze già presente su disco, senza che il Flow 1 sia terminato.

8.6.2 6.0.1 Boilerplate standard di un flow

Ogni file di flow inizia con la stessa intestazione in quattro parti. Copiarla verbatim e adattare l'indice `parents[N]` e la costante `FONTE`.

```python
import os
import sys
```

```
from pathlib import Path

try:
    from dotenv import load_dotenv
    load_dotenv(Path(__file__).resolve().parents[3] / ".env")   # repo root
except ImportError:
    pass

import psycopg2
from psycopg2.extras import execute_values
from prefect import flow, task, get_run_logger

sys.path.insert(0, str(Path(__file__).resolve().parents[2]))    # aggiunge flows/ al path
from utils.bronze_writer import save_to_bronze, log_ingestion   # e altri utils se necessario

FONTE = "nome-ente"                                             # corrisponde alla directory Br
_REPO_ROOT = Path(__file__).resolve().parents[3]
BRONZE_BASE = os.getenv("BRONZE_BASE_PATH", str(_REPO_ROOT / "data" / "bronze"))


def _get_db_conn():
    return psycopg2.connect(
        host=os.getenv("POSTGRES_HOST", "localhost"),
        port=int(os.getenv("POSTGRES_PORT", 5432)),
        database=os.getenv("POSTGRES_DB", "maps_db"),
        user=os.getenv("POSTGRES_USER", "maps"),
        password=os.getenv("POSTGRES_PASSWORD", "maps_dev"),
    )
```

L'indice `parents[N]` dipende dalla profondità della directory: `parents[2]` da `flows/{ente}/{dataset}/` raggiunge `flows/` (dove si trova `utils/`); `parents[3]` raggiunge la root del repository (dove si trova `.env`). `_get_db_conn()` è definita localmente in ogni flow, non importata da utils, in modo che ogni script sia autocontenuto.

8.6.3 6.1 Flow 1 — Ingestion (Bronze layer)

Il Flow 1 scarica i dati grezzi dalla fonte esterna e li salva nel filesystem Bronze tramite l'utility `save_to_bronze`. Registra l'operazione in `bronze.ingestion_log`.

```
from prefect import flow, task
from prefect.tasks import task_input_hash
from datetime import timedelta
import httpx
from utils.bronze_writer import save_to_bronze, log_ingestion

@task(
    retries=3,
    retry_delay_seconds=[60, 300, 900],
    cache_key_fn=task_input_hash,
```

```
    cache_expiration=timedelta(hours=24)
)
def download_fonte(anno: int) -> dict:
    url = "https://example.gov.it/data/dataset.xlsx"
    response = httpx.get(url, follow_redirects=True, timeout=300)
    response.raise_for_status()

    file_info = save_to_bronze(
        data=response.content,
        fonte="nome-ente",
        dataset="nome_dataset",
        anno=anno,
        ext="xlsx"
    )
    log_ingestion(file_info, n_record=0, url_source=url)
    return file_info
```

`save_to_bronze` crea la directory, scrive i byte grezzi, calcola il checksum SHA256 e restituisce un dizionario con `file_path`, `size_bytes`, `checksum`, `fonte`, `dataset` e `anno`. `log_ingestion` inserisce la riga corrispondente in `bronze.ingestion_log`.

Il Bronze layer contiene sempre la fonte grezza esattamente come ricevuta dall'origine: byte HTTP per file CSV, XLSX e PDF; HTML grezzo per le pipeline di scraping. La trasformazione in strutture dati avviene nel Flow 2.

Convenzione del path Bronze: `{repo_root}/data/bronze/{ente}/{anno}/{dataset}.{ext}`
Esempi: `/data/bronze/istat/2024/popolazione_residente.xlsx`, `/data/bronze/minsalute/2024/asl_boun`

Portali JSF/PrimeFaces. Alcuni portali pubblici italiani usano JavaServer Faces con paginazione Ajax. Ogni risposta contiene un token `javax.faces.ViewState` che deve essere ritrasmesso nella richiesta successiva. Usare una `requests.Session` per mantenere i cookie, estrarre il ViewState con BeautifulSoup e usare `rate_limited_request()` da `utils/rate_limiter.py`:

```
from utils.rate_limiter import rate_limited_request

def _get_viewstate(html: str) -> str:
    from bs4 import BeautifulSoup
    vs = BeautifulSoup(html, "html.parser").find("input", {"name": "javax.faces.ViewState"})
    return vs.get("value", "") if vs else ""

session = requests.Session()
response = rate_limited_request(
    url, method="POST",
    data={
        "javax.faces.ViewState": viewstate,
        "javax.faces.partial.ajax": "true",
        "javax.faces.partial.execute": "formBusca:regione",
        "javax.faces.partial.render": "formBusca:provincia",
        "formBusca:regione": regione_code,
    },
```

```
    session=session,
    delay=2,
)
```

Per un esempio completo vedere `flows/agenzie-dogane/tabacchi/01_ingestion_flow.py`.

8.6.4 6.2 Flow 2 — Transform (Silver layer)

Il Flow 2 legge dal Bronze, normalizza i dati e scrive in `silver.territory_attributes`. È il flow più specifico per ogni dataset.

Scheletro standard:

```
@task
def trasforma_e_carica(file_info: dict, entity_type: str) -> int:
    # 1. Leggere il file sorgente
    if file_info['file_path'].endswith('.xlsx'):
        df = pd.read_excel(file_info['file_path'])
    elif file_info['file_path'].endswith('.csv'):
        df = pd.read_csv(file_info['file_path'], encoding='utf-8-sig', sep=';')

    # 2. Risolvere territory_id e costruire i record EAV
    conn = _get_db_conn()
    lookup = _build_territory_lookup(conn)   # (sigla, label_upper) → (territory_id, type_code)
    eav_records = []
    for _, row in df.iterrows():
        match = lookup.get((row['SIGLA'].upper(), row['COMUNE'].upper()))
        if not match:
            continue
        territory_id, type_code = match
        eav_records.extend([
            (territory_id, type_code, 'attributo_1', str(row['COL_A']),
             'integer', fonte, f'{anno}-01-01', None),
            (territory_id, type_code, 'attributo_2', str(row['COL_B']),
             'string',  fonte, f'{anno}-01-01', None),
        ])

    # 3. UPSERT nel Silver EAV
    try:
        with conn.cursor() as cur:
            execute_values(cur, """
                INSERT INTO silver.territory_attributes
                    (territory_id, type_code, attribute, value,
                     data_type, source, valid_from, valid_to)
                VALUES %s
                ON CONFLICT (territory_id, attribute, valid_from)
                DO UPDATE SET value = EXCLUDED.value, updated_at = NOW()
            """, eav_records)
        conn.commit()
    finally:
```

```
        conn.close()

    return len(eav_records)
```

Risoluzione del territorio: le pipeline devono risolvere `territory_id` da `silver.territories` prima di scrivere in `silver.territory_attributes`. Usare `_build_territory_lookup()` (v. §6.0.1) o il modulo condiviso `territory_resolver.build_comune_lookup_from_conn()` per costruire un dizionario `(sigla_automobilistica, label_upper) → (territory_id, type_code)`. I record il cui comune non può essere risolto vengono registrati come warning e saltati — comportamento atteso quando le pipeline fondazionali ISTAT non sono ancora state eseguite.

8.6.5 6.3 Flow 3 — Data quality

Il Flow 3 valida sia i file Bronze che i record Silver. Usa Great Expectations per la validazione strutturata e produce un report.

Validazione Bronze (verificare il file grezzo prima di fidarsi della trasformazione):

```
import great_expectations as gx

suite = gx.core.ExpectationSuite(name=f"{fonte}_{dataset}")

suite.add_expectation(gx.core.ExpectationConfiguration(
    expectation_type="expect_column_to_exist",
    kwargs={"column": "colonna_codice_entita"}
))
suite.add_expectation(gx.core.ExpectationConfiguration(
    expectation_type="expect_column_values_to_not_be_null",
    kwargs={"column": "colonna_codice_entita", "mostly": 0.99}
))
```

Validazione Silver (verificare la correttezza dei record EAV):

```
suite.add_expectation(gx.core.ExpectationConfiguration(
    expectation_type="expect_column_values_to_not_be_null",
    kwargs={"column": "territory_id"}
))
suite.add_expectation(gx.core.ExpectationConfiguration(
    expectation_type="expect_column_values_to_be_between",
    kwargs={"column": "valore_cast", "min_value": 0, "max_value": 10_000_000}
))
```

Strategia di errore: i fallimenti non critici vengono registrati come warning; i fallimenti critici nella validazione Silver bloccano il Flow 4 (catalogo).

8.6.6 6.4 Flow 4 — Catalogo metadata

Il Flow 4 registra i file Bronze e la tabella Silver in OpenMetadata, creando una voce di lineage dalla fonte all'EAV.

```
@task
def catalog_bronze_files(bronze_path: str, fonte: str, dataset: str):
```

```
    """Scansiona la directory Bronze e registra i file in OpenMetadata."""
    from metadata.ingestion.ometa.ometa_api import OpenMetadata
    from metadata.generated.schema.entity.services.connections.metadata.openMetadataConnection
        OpenMetadataConnection
    )
    # ... configurazione ingestion OpenMetadata e creazione lineage
```

Prerequisiti: `pip install 'openmetadata-ingestion[datalake]'` e variabile d'ambiente `OPENMETADATA_JWT_TOKEN` impostata (generare dalla UI OpenMetadata → Settings → Bots → ingestion-bot).

8.7 7. Utility condivise

Sei moduli in `flows/utils/` sono disponibili per tutte le pipeline.

`utils/bronze_writer.py` — scrittura Bronze standardizzata con checksum e log:

```
from utils.bronze_writer import save_to_bronze, log_ingestion

file_info = save_to_bronze(data=response.content, fonte="agenzie-dogane", dataset="tabacchi", a
log_ingestion(file_info, n_record=0, url_source=url)
```

`utils/rate_limiter.py` — rate limiting globale thread-safe per il web scraping:

```
from utils.rate_limiter import rate_limited_request

response = rate_limited_request(url, method="POST", delay=2)
```

`utils/html_parser.py` — parsing HTML condiviso:

```
from utils.html_parser import parse_table, extract_form_data

data = parse_table(soup, table_id="results")
```

`utils/situas_client.py` — wrapper API ISTAT SITUAS:

```
from utils.situas_client import SituasClient

client = SituasClient()
catalogue = client.get_catalogue()              # lista di 74 dataset
records = client.get_all_records(pfun=64, pdata="01/01/2024")   # paginato
records = client.get_records(pfun=61, pdata="01/01/2024")       # singola pagina
```

`get_all_records()` gestisce la paginazione iterando pagine da 500 record. Alcuni pfun — in particolare pfun=61 (comuni) — restituiscono tutti i risultati in un'unica risposta e non supportano la paginazione per offset; usare `get_records()` direttamente per quelli.

`utils/territory_resolver.py` — lookup condiviso (sigla, comune) → `territory_id`:

```
from utils.territory_resolver import build_comune_lookup_from_conn, resolve_comune

# Costruire il lookup una volta per run (una query DB)
lookup = build_comune_lookup_from_conn(conn)
```

```
# Risolvere ogni record - lookup O(1) su dizionario
match = resolve_comune(lookup, sigla="MI", nome="Milano")
if match:
    territory_id, type_code = match
```

Il lookup cerca sia in `silver.territories.label` che in `silver.territory_names`, che include nomi alternativi, forme bilingui e alias ADM. Se `resolve_comune` restituisce `None`, il territorio non è stato trovato — registrare un warning e continuare.

`utils/service_writer.py` — helper upsert idempotenti per `silver.services` e le tabelle satellite:

```
from utils.service_writer import (
    upsert_service,             # → int (service id)
    upsert_service_identifier,  # allegare un codice esterno
    upsert_service_attribute,   # memorizzare una metrica tipizzata (EAV)
    upsert_service_category,    # allegare un tag categorico
)
```

Tutte le funzioni accettano un cursore psycopg2 aperto. Il chiamante gestisce la connessione e chiama `conn.commit()`. Per esempi d'uso vedere §5.3.

Usare queste utility invece di reimplementare pattern comuni in ogni pipeline.

8.8 8. Deployment con Prefect

Un deployment trasforma un flow in una risorsa pianificata e monitorata sul server Prefect. Lo sviluppatore definisce `prefect.yaml` una volta; le esecuzioni successive avvengono tramite UI o CLI senza toccare il codice.

8.8.1 8.1 Struttura di prefect.yaml

Ogni pipeline definisce quattro deployment in `prefect.yaml`. La sezione `pull` è obbligatoria in tutte le pipeline: imposta la directory di lavoro e installa le dipendenze del flow a runtime. Questo è necessario perché il container `prefect-worker` usa l'immagine base `prefecthq/prefect:3-python3.11`, che non pre-installa pacchetti specifici della pipeline come `psycopg2-binary`.

```
name: {ente}-{dataset}
prefect-version: "3.*"

work_pool:
  name: default-pool

pull:
  - prefect.deployments.steps.set_working_directory:
      directory: /flows/{ente}/{dataset}
  - prefect.deployments.steps.pip_install_requirements:
      directory: /flows/{ente}/{dataset}
      requirements_file: requirements.txt

deployments:
  - name: {ente}-{dataset}-01-ingestion      # Flow 1 - download → Bronze
```

```yaml
    entrypoint: 01_ingestion_flow.py:ingestion_flow
    schedules: []                          # nessuno schedule automatico - avviare manualmen
    parameters:
      anno: 2025
    work_pool:
      name: default-pool

  - name: {ente}-{dataset}-02-transform    # Flow 2 - Bronze → Silver
    entrypoint: 02_transform_flow.py:transform_flow
    schedules: []
    parameters:
      anno: 2025
    work_pool:
      name: default-pool

  - name: {ente}-{dataset}-03-quality      # Flow 3 - monitoraggio qualità
    entrypoint: 03_data_quality_flow.py:quality_flow
    schedules: []
    parameters:
      anno: 2025
    work_pool:
      name: default-pool

  - name: {ente}-{dataset}-04-metadata     # Flow 4 - aggiornamento catalogo
    entrypoint: 04_metadata_flow.py:metadata_flow
    schedules: []
    parameters:
      anno: 2025
    work_pool:
      name: default-pool
```

Il prefisso numerico (`01`, `02`, …) mantiene i deployment in ordine di esecuzione nella UI di Prefect. `schedules: []` rimuove esplicitamente qualsiasi schedule — omettere la chiave non elimina gli schedule precedentemente registrati.

8.8.2 8.2 Registrazione dei deployment

I deployment devono essere registrati dall'interno del container `prefect-worker`, non dalla shell dell'host. Questo perché Prefect registra il path della directory di lavoro al momento della registrazione e lo usa letteralmente al momento dell'esecuzione. Il worker esegue i flow in `/flows/{ente}/{dataset}/` (il mount del volume Docker); registrare dall'host registrerebbe il path dell'host che non esiste dentro il container.

```
docker exec maps-prefect-worker bash -c \
  "cd /flows/{ente}/{dataset} && prefect deploy --all"
```

Dopo la registrazione, i deployment sono visibili su http://localhost:4200/deployments.

8.8.3 8.3 Quando re-fare il deploy

Il worker Prefect legge il codice dal filesystem ad ogni esecuzione:

- **Modifiche al codice dei flow** (logica, bug fix): non richiedono `prefect deploy --all`. Il worker eseguirà il file aggiornato alla prossima run.
- **Modifiche a `prefect.yaml`** (nuovi deployment, cambi di schedule, nuovi parametri, sezione `pull`): richiedono `prefect deploy --all` eseguito dall'interno del container.
- **Rimozione di uno schedule**: impostare `schedules: []` in `prefect.yaml` e ri-registrare. Omettere la chiave non rimuove uno schedule esistente.

8.8.4 8.4 Avviare un deployment

Dalla UI — andare su http://localhost:4200/deployments, cliccare un deployment, poi "Run". Il form dei parametri è pre-compilato con i valori predefiniti e può essere modificato prima di ogni run.

Dalla CLI:

```
export PREFECT_API_URL=http://localhost:4200/api

# Eseguire con parametri predefiniti
prefect deployment run '{flow-name}/{ente}-{dataset}-01-ingestion'

# Override dei parametri a runtime
prefect deployment run '{flow-name}/{ente}-{dataset}-01-ingestion' \
  --param anno=2024

# Elencare tutti i deployment registrati
prefect deployment ls
```

Il nome CLI segue il pattern `{flow-name}/{deployment-name}`. Il nome del flow viene dal decoratore `@flow(name=...)`; il nome del deployment da `prefect.yaml`. Verificare i nomi esatti con `prefect deployment ls` o dalla UI.

Non avviare un worker Prefect locale (`prefect worker start`) quando si usa lo stack Docker. Il container `maps-prefect-worker` fa già polling su `default-pool`. Un worker locale concorrente prenderebbe in carico le run ma fallirebbe perché i path `/flows/` non esistono sull'host.

Monitoraggio: Prefect UI su http://localhost:4200.

8.9 9. Flusso di lavoro di squadra

Ognuno dei quattro flow può essere sviluppato e testato indipendentemente. Una suddivisione naturale per un team di due sviluppatori che lavorano sulla stessa pipeline:

Sviluppatore A	Sviluppatore B
Flow 1: Ingestion (specifico per la fonte, spesso scraping o API) Flow 4: Catalogo metadata	Flow 2: Transform (normalizzazione EAV, risoluzione territorio) Flow 3: Data quality (aspettative GX, rilevamento anomalie)

Sviluppatore A	Sviluppatore B
`README.md` + `prefect.yaml`	`tests/test_flow.py`

Criterio di passaggio di consegne tra i flow: il Flow 1 è considerato completo quando almeno un file Bronze è scritto nel path corretto e il suo checksum è registrato in `bronze.ingestion_log`. Il Flow 2 può iniziare con quel file indipendentemente dal completamento del Flow 1.

Tracciamento su GitLab: aprire un issue per ogni flow. Collegarli all'issue principale della pipeline. Chiudere ogni sub-issue quando il flow raggiunge i criteri di accettazione Bronze/Silver (v. §10).

8.10 10. Checklist per fase

8.10.1 Flow 1 (Ingestion) completo quando:

- ☐ File scaricato nel path Bronze corretto (`/data/bronze/{ente}/{anno}/`)
- ☐ Checksum SHA256 registrato in `bronze.ingestion_log`
- ☐ Logica di retry attiva (`retries=3` con backoff)
- ☐ `README.md` documenta URL della fonte, formato e frequenza di aggiornamento

8.10.2 Flow 2 (Transform) completo quando:

- ☐ Record inseriti in `silver.territory_attributes` con il `type_code` corretto
- ☐ `ON CONFLICT DO UPDATE` attivo (riesecuzioni idempotenti)
- ☐ `territory_id` risolto per tutti i record (comuni non risolti registrati e contati)
- ☐ Riesecuzione su Bronze invariato produce output Silver identico

8.10.3 Flow 3 (Quality) completo quando:

- ☐ Validazione Bronze copre: schema, tasso di null, formati dei valori
- ☐ Validazione Silver copre: presenza di `territory_id`, range dei valori, campi temporali
- ☐ Report prodotto ad ogni run (pass o fail)
- ☐ Soglia di fallimento definita e documentata

8.10.4 Flow 4 (Metadata) completo quando:

- ☐ File Bronze visibili in OpenMetadata
- ☐ Lineage Bronze → Silver creato
- ☐ Schema estratto e disponibile nell'interfaccia del catalogo

8.11 11. Errori frequenti

`ON CONFLICT` fallisce con "no unique constraint": il vincolo UNIQUE su `(territory_id, attribute, valid_from)` deve essere presente. Verificare con `\d silver.territory_attributes` in psql.

`territory_id` è NULL / nessun record risolto: le pipeline fondazionali ISTAT (`flows/istat/`) devono essere eseguite prima di qualsiasi pipeline EAV. Se `_build_territory_lookup()` restituisce un dizionario vuoto, `silver.territories` non è ancora popolato.

`valid_from` causa una chiave duplicata: due righe per la stessa entità/attributo/anno con date `valid_from` diverse. Usare una data fissa (es. `f"{anno}-01-01"`) invece di `CURRENT_DATE` per garantire l'idempotenza.

Il Flow 2 legge un file Bronze obsoleto: la cache dei task Prefect potrebbe restituire un risultato in cache. Disabilitare la cache per quel task in quella run, oppure eliminare la voce dalla UI di Prefect.

JWT OpenMetadata scaduto: ruotare il token dalla UI di OpenMetadata (Settings → Bots) e aggiornare la variabile d'ambiente `OPENMETADATA_JWT_TOKEN` nel container prefect-worker.

La run del flow va in crash con `FileNotFoundError: /flows/{ente}/{dataset}`: il deployment è stato registrato dalla shell dell'host invece di dall'interno del container. Ri-registrare usando `docker exec maps-prefect-worker bash -c "cd /flows/... && prefect deploy --all"` (v. §8.2).

La run del flow va in crash con `ModuleNotFoundError: No module named 'psycopg2'`: la sezione `pull` è assente o incompleta in `prefect.yaml`. Assicurarsi che entrambi i passi `set_working_directory` e `pip_install_requirements` siano presenti (v. §8.1), poi ri-registrare il deployment.

La run è pianificata ma non si avvia mai: un worker locale (`prefect worker start`) è in esecuzione insieme al worker Docker e si aggiudica la run, poi va in crash. Arrestare i processi worker locali e lasciare che `maps-prefect-worker` gestisca l'esecuzione.

`get_all_records()` solleva `JSONDecodeError`: il pfun restituisce tutti i record in un'unica risposta e non supporta la paginazione per offset tramite `pdatada`. Usare `get_records()` direttamente (v. §7 — `situas_client.py`).