Technische Hochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

ADT Portfolio

Implementierung des Lobbyregisters

Noah Raupold (5022097),
David Gläsle (5022114)

Eingereicht am: 17. November 2025

Inhaltsverzeichnis

1 Einleitung

Dieser Portfolioteil dokumentiert die vollständige Implementierung der Datenpipeline, die das Lobbyregister des Deutschen Bundestags automatisiert abruft, verarbeitet und in ein relationales PostgreSQL-Schema überführt. Während das erste Portfolio die konzeptionelle Modellierung, die Strukturierung der Entitäten und die theoretischen Überlegungen zur Datenorganisation behandelt, liegt der Fokus dieses Teils auf der technischen Realisierung der Pipeline und den dabei getroffenen Architekturentscheidungen.

Im Zentrum steht der gesamte Verarbeitungsprozess: das asynchrone Einsammeln der Registerdaten über die öffentliche API, die Normalisierung und strukturelle Transformation der JSON-Dokumente, die robuste und nachvollziehbare Persistierung im Datenbankschema sowie erste Mechanismen zur Beobachtbarkeit, etwa Fortschrittsberichte oder Metriken zur Lastverteilung. Die Pipeline soll nicht nur funktional korrekt sein, sondern auch skalierbar, fehlertolerant und reproduzierbar ausgeführt werden können. Dadurch entsteht eine technische Grundlage, auf der spätere Analysen, Erweiterungen und Optimierungen aufbauen können.

Die Projektstruktur ist klar abgegrenzt: Der gesamte Anwendungscode befindet sich unter `src/lobbyregister_ingestor`, einschließlich der Module zur API-Kommunikation, Transformation und Datenbankschnittstelle. Das relationale Schema liegt in `src/lobbyregister_ingestor/scheme.sql` und definiert die Tabellen, Constraints und Normalisierungsstufen. Infrastruktur, Containerisierung und Laufzeitumgebung sind im Repository-Root dokumentiert und ermöglichen eine reproduzierbare Ausführung sowohl lokal als auch in isolierten Umgebungen wie Docker oder Hochschul-VMs.

Durch diese Trennung von Konzept, Implementierung und Infrastruktur entsteht eine modular aufgebaute Pipeline, die sowohl experimentell als auch produktionsnah weiterentwickelt werden kann [1].

2 Technologie Stack

Die Implementierung basiert auf Python. Die Sprache stellt eine breite Auswahl an ausgereiften Bibliotheken für Netzwerkkommunikation, Datenverarbeitung und Asynchronität bereit, was die Entwicklungszeit reduziert und die Wartbarkeit erhöht. Besonders relevant ist die Unterstützung für asynchrone Programmierung: IO-lastige Anwendungen profitieren stark davon, Wartezeiten nicht blockierend zu behandeln. Durch den Einsatz von `httpx.AsyncClient` und `asyncio` können viele HTTP-Requests parallel abgearbeitet werden, ohne für jede Anfrage einen eigenen Thread zu erzeugen. Der Event-Loop überlappt dabei Wartezeiten auf Netzwerk- oder API-Antworten und erzielt so einen deutlich höheren Durchsatz bei geringerer Latenz. Im Vergleich zu klassischen Thread-Pools sinkt der Ressourcenverbrauch, da weniger Kontextwechsel und kein Thread-Overhead entstehen. Gleichzeitig bleibt der Code durch `async/await` gut lesbar und deterministisch testbar; Werkzeuge wie `pytest-asyncio` ermöglichen realistische Tests, ohne komplexe Thread-Synchronisation.

Für die Datenhaltung wurde PostgreSQL gewählt. PostgreSQL gilt als stabiler Industriestandard und bietet ein vollständiges relationales Modell mit starker Typisierung, Transaktionen, Foreign Keys und mächtigen Debug-Werkzeugen. Die bewusste Entscheidung für ein relational normalisiertes Schema (3NF) und gegen JSONB dient mehreren Zielen: Der Query-Planner erhält verlässliche Statistiken, was zu vorhersehbaren und reproduzierbaren Ausführungsplänen führt. Optimierungsmaßnahmen wie Indexdesign, Materialized Views oder Partitionierung lassen sich dadurch gezielt und messbar einsetzen. JSONB wäre zwar flexibel, erschwert aber die Kostenabschätzung für komplexe Abfragen und führt häufiger zu sequentiellen Scans oder unvorhersehbaren Plänen.

PostgreSQL bietet zudem eingebaute Analysewerkzeuge wie `EXPLAIN` und `auto_analyze`, die Lastverteilung, Kardinalitäten und Planentscheidungen sichtbar machen. Mit Extensions wie `pg_stat_statements` können „Hot Queries“ identifiziert werden, ohne dass strukturelle Änderungen am Schema nötig sind. Diese Kombination aus Stabilität, Transparenz und Erweiterbarkeit macht PostgreSQL besonders geeignet für Systeme, die langfristig wartbar bleiben und auf wachsende Datenmengen vorbereitet sein müssen.

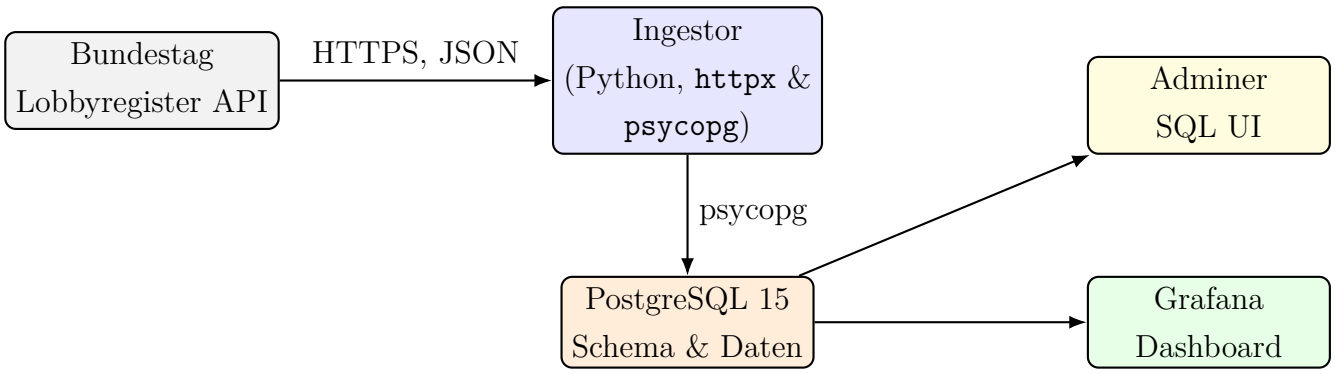

Abbildung 2.1: Container-Stack und Datenfluss (docker-compose.yml).

2.1 Konfigurierbarkeit

Die Konfiguration des Ingestors erfolgt entweder über eine `.env`-Datei im Root des Repositories oder über Environment-Variablen im `docker-compose.yml`. Dadurch kann die Pipeline ohne Codeänderungen in unterschiedlichen Umgebungen betrieben werden – lokal, in Docker oder in Cloud-Instanzen. Die zentralen Parameter steuern sowohl API-Verhalten als auch Datenbankzugriffe und Backpressure-Mechanismen.

Setzbare Einstellungen:

- `PG_DSN`: Vollständiger DSN-String für PostgreSQL. Falls nicht gesetzt, konstruiert der Ingestor automatisch einen DSN aus den einzelnen `POSTGRES_*`-Variablen.
- `POSTGRES_*`: Host, Port, Benutzer, Passwort und Datenbankname zur manuellen DSN-Zusammenstellung. Dies erleichtert flexible Deployments, ohne DSNs hartkodieren zu müssen.
- `LOBBY_API_*`: Basis-URL, Endpunkte und optionale Zugangstoken für die Bundestag-API. Diese Variablen steuern, wohin und wie der HTTP-Client seine Anfragen richtet.
- `HTTP_CONCURRENCY`: Obergrenze paralleler HTTP-Requests. Begrenzt sowohl die eigene Netzlast als auch die Anfragefrequenz gegenüber der API, um Timeouts und Rate-Limits zu vermeiden.
- `DB_WORKERS`: Anzahl parallel arbeitender Consumer, die Datenbank-Schreiboperationen ausführen. Erhöht die Schreibkapazität, ohne den Connection-Pool zu überlasten.
- Backoff-Parameter: Steuerung von Retry-Strategien (z. B. exponentieller Backoff, maximale Retry-Zahl). Verhindert aggressive Request-Stürme bei temporären API- oder Netzwerkfehlern.

- `PROGRESS_EVERY`: Intervall für Fortschrittslogs. Hilft beim Monitoring langer Läufe, insbesondere in Entwicklungs- und Lehrkontexten.
- `INGEST_QUEUE_SIZE`: Maximale Größe der internen `asyncio`-Queue zwischen Producer und Consumer. Bildet die zentrale Backpressure-Komponente: Überläuft die Queue, drosselt der Producer automatisch.

3 API-Gathering und ETL-Fluss

3.1 HTTP-Client

Der `LobbyregisterClient` (`api.py`) bildet die Schnittstelle zur öffentlichen API des Deutschen Bundestags. Kern ist ein `httpx.AsyncClient` mit explizit konfigurierten Connection-Limits, sodass die in den Umgebungsvariablen definierte `HTTP_CONCURRENCY` strikt eingehalten wird. Damit lässt sich steuern, wie viele parallele Requests das System gleichzeitig absetzt, was sowohl für die eigene Laststeuerung als auch für Fair Use gegenüber dem API-Endpunkt wichtig ist.

Wesentliche Aspekte der Implementierung:

- **Pagination**: Die Methode `iter_register_entries` folgt den vom Server bereitgestellten Cursor-Links und vermeidet doppelte Tokens, um konsistent voranzuschreiten. Pro Seite werden Meta-Daten wie `sourceDate` oder `totalResultCount` injiziert, um Kontext für spätere Schritte zu erhalten. Cursor-Pagination verhindert unnötige Übertragungen und reduziert Lastspitzen auf der API.
- **Robuste Payload-Prüfung**: `_extract_entries` akzeptiert ausschließlich Listen von Mappings. Wird ein abweichendes oder beschädigtes Format geliefert, löst der Client einen `ApiError` aus und gibt ein begrenztes Preview der empfangenen Daten aus. Fehlerzustände lassen sich dadurch schneller analysieren, ohne unkontrolliert große Antwortkörper zu loggen.
- **Retry/Backoff**: Fehler wie 5xx, 429 oder 408 werden mit einem exponentiellen Backoff erneut versucht, bis die definierte Obergrenze `HTTP_MAX_RETRIES` erreicht ist. 404 wird hingegen sofort als `ResourceNotFoundError` gewertet, da kein Retry sinnvoll wäre. Der Backoff ist gekappt, um eskalierende Wartezeiten zu vermeiden.

Durch Asynchronität wird der effektive Durchsatz erhöht: HTTP-Latenzen überlappen sich, ohne dass zusätzliche Threads nötig wären. Die vom Bundestag bereitgestellte API arbeitet cursorbasiert; jede Seite liefert einen Token, der beim nächsten Request wiederverwendet wird. Dieses Verfahren ist stabil gegenüber Datenänderungen zwischen Aufrufen und vermeidet die typischen Schwächen klassischer Offset-Pagination (z. B. große Skips, Race Conditions oder

inkonsistente Seiten). Zugleich wird nur der tatsächlich benötigte Ausschnitt übertragen, was Netzwerk- und CPU-Verbrauch reduziert.

3.2 Pipeline-Orchestrierung

Der Einstiegspunkt ist `python -m lobbyregister_ingestor` (`__main__.py`). Das Skript führt mehrere Schritte kontrolliert und asynchron aus:

1. Initiales Anlegen oder Aktualisieren des Schemas über `apply_schema`.
2. Aufbau eines `psycopg_pool.ConnectionPool` zur effizienten Verwaltung paralleler Datenbankverbindungen.
3. Starten eines Producers (API-Fetch) und mehrerer Consumer entsprechend `DB_WORKERS`, die Daten in die Datenbank schreiben.

Die Kommunikation zwischen Producer und Consumer erfolgt über eine `asyncio.Queue`. Diese entkoppelt die Geschwindigkeiten beider Systeme: Ein schneller HTTP-Client produziert Einträge, aber überflutet die Datenbank nicht, da die Queue nur eine begrenzte Kapazität besitzt und Backpressure erzeugt. Abbildung 3.1 visualisiert den Fluss.

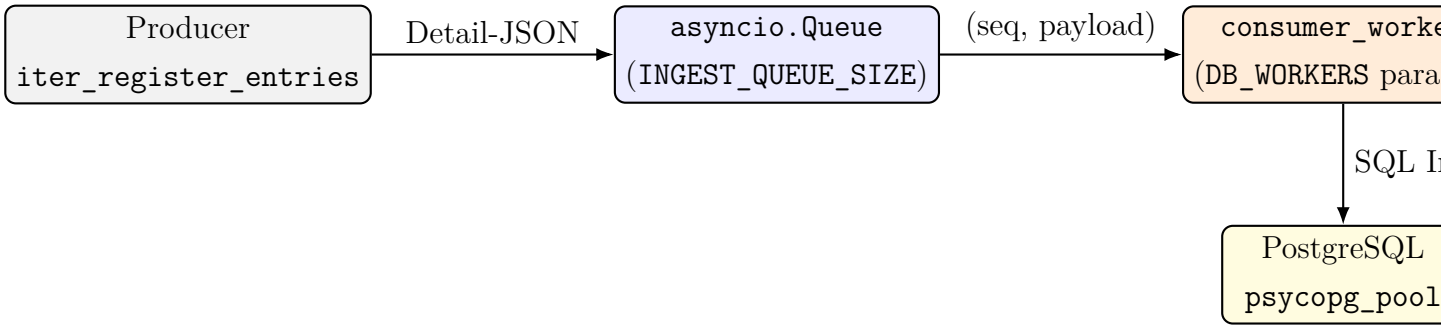

Abbildung 3.1: Asynchroner ETL-Fluss mit Producer/Consumer.

Robustheitsaspekte der Orchestrierung:

- **Transiente DB-Fehler**: Deadlocks oder Sperren führen zu automatischen Wiederholungen bis `MAX_DB_WRITE_RETRIES`. Bleibt ein Datensatz fehlerhaft, wird er übersprungen, der Lauf bleibt jedoch stabil.
- **Stop-Token**: Für jeden Consumer wird ein Stop-Token in die Queue gestellt. Dies verhindert Hängenbleiben, wenn der Producer beendet ist und keine neuen Elemente mehr eintreffen.

- **Fortschritt**: Die API liefert `totalResultCount`, wodurch ein Schätzwert für den Fortschritt des gesamten ETL-Laufs verfügbar ist.

3.3 Vom JSON zur Datenbank

Die Verarbeitung eines einzelnen Eintrags beginnt in `ingest_entry` (`writer.py`). Die Funktion ruft `upsert_register_entry` auf, entfernt veraltete Kindobjekte über `purge_children_for_entry` und iteriert anschließend die in `SECTION_HANDLERS` definierten Abschnittsverarbeiter (`mappings/registry.py`). Jeder Handler übernimmt die Transformation und das Schreiben eines spezifischen API-Teilbaums.

Beispiele für verarbeitete Abschnitte:

- **accountDetails**: Versionierung, Grunddaten, Zuordnung von Gesetzeskatalogen (`GL2022`, `GL2024`).
- **lobbyistIdentity**: Personen- und Organisationsdaten, Adressen, Kontaktinformationen und vertretungsberechtigte Personen.
- **activitiesAndInterests**: Interessensfelder, Formen der Lobbyarbeit sowie gemeldete regulatorische Projekte (`regulatory_projects.py`).
- **financialExpenses**, **donators**, **membershipFees**: Finanzielle Angaben, Betragsnormalisierung, Periodeninterpretation und strukturiertes Auflösen von Listen.
- **contracts**, **codeOfConduct**: Vertragsangaben, Compliance-Dokumentation und zugehörige Metadaten.

`mappings/common.py` stellt robuste Hilfsfunktionen bereit: Idempotenz über `upsert_code_label` und `insert_address`, Datumsnormalisierung mittels `normalize_year_month` sowie konsistente Reihenfolgen durch `ordinal`. Diese gemeinsame Basis verhindert Schema-Duplikationen und stellt sicher, dass alle Abschnitte unter denselben Regeln verarbeitet werden.

4 Relationales Datenmodell und Datenbankaspekte

4.1 Normalisierung und Entscheidung gegen JSONB

Das relationale Schema (`scheme.sql`) löst sämtliche Listenstrukturen der API in klar definierte Child-Tabellen auf. Jede Liste wird über einen `ordinal`-Wert in eine stabile Reihenfolge gebracht, sodass sowohl Idempotenz als auch Reproduzierbarkeit gewährleistet bleiben. Die Entscheidung gegen eine Speicherung der API-Blöcke in JSONB ist bewusst getroffen und folgt mehreren technischen Überlegungen:

- **Integrität**: Fremdschlüsselbeziehungen – etwa zwischen `address` und `country_label` – erzwingen Konsistenz auf Datenebene. Solche Garantien sind mit JSONB nur schwer oder gar nicht durchsetzbar, da die Struktur dort untypisiert vorliegt.
- **Bessere Planner-Statistiken**: PostgreSQL liefert für relationale Tabellen exakte oder gut schätzbare Kardinalitäten. Für JSONB-Felder hingegen sind Selektivitätsschätzungen oft ungenau, was zu suboptimalen Ausführungsplänen führt. Da spätere Analysen auf `EXPLAIN (ANALYZE)` basieren, ist ein strukturiertes Schema essenziell.
- **Einfache Deduplizierung**: Tabellen wie `code_label` nutzen `UNIQUE(domain, code)`, um doppelte oder inkonsistente Werte zu verhindern. Diese Form der Normalisierung ist in JSONB weder natürlich noch performant umsetzbar.

Die Kombination aus strikter Typisierung, expliziten Relationen und stabilen Kardinalitäten bildet die Grundlage für spätere Optimierungsschritte, etwa Indexdesign oder Materialized Views.

4.2 Tabellen-Hotspots

Das Modell trennt Kernobjekte, Dimensionen und Detailtabellen klar voneinander. Wesentliche Tabellen sind:

- **register__entry**: Zentrales Objekt des Schemas; enthält Identität, Quelle, Metadaten sowie den Primärschlüssel für sämtliche abhängigen Tabellen.
- **register__entry__version**: Versionierte Sachstände erlauben es, historische Unterschiede (z. B. zwischen `GL2022` und `GL2024`) zu erfassen, ohne ältere Daten zu überschreiben.
- **lobbyist__identity** und **client__***: Trennen Identitäten von Interessenvertretern und Auftraggebern. Adressen und Kontaktinformationen werden als wiederverwendbare Dimensionstabellen geführt, wodurch Redundanz minimiert wird.
- **regulatory__projects**: Erfasst gemeldete Projekte inkl. Ministerien. Der `ordinal`-Wert stellt die vom API gelieferte Reihenfolge her, was für deterministische Läufe wichtig ist.

4.3 Indizes und Performance-Tuning

Das Schema enthält Foreign-Key-Indizes, deren Existenz für performante Joins wesentlich ist. Darüber hinaus bieten sich weitere Indizes an:

- Eindeutigkeitsindex auf `register_entry.register_number` zur schnellen Identifikation von Einträgen.
- Zeitbasierte Indizes auf `donation(year)` und `financial_expense(year)` für typische Auswertungen über Jahre oder Berichtperioden.
- Optionale GIN-Indizes auf Textspalten, falls spätere Volltextsuchen oder Filterfunktionen vorgesehen sind.

Für Analyse und langfristiges Tuning stehen `pg_stat_statements` (zur Identifikation teurer Anfragen) sowie `auto_explain` (zum Loggen unerwarteter oder kostenintensiver Pläne) zur Verfügung. Durch die relationale Struktur lassen sich Hot-Queries klar isolieren und gezielt optimieren.

4.4 Mapper-Strategie und Beispiel

Die Pipeline nutzt eine modulare Mapper-Strategie: Jeder Abschnitt der API besitzt einen dedizierten Handler, der für Parsing, Normalisierung und Persistierung verantwortlich ist. Dies kapselt Logik lokal und reduziert die Kopplung zwischen Schema und höherer Pipeline.

Beispiel aus `funding.py`:

a) `load_financial_expenses`: Aggregiert und normalisiert Jahresangaben und schreibt sie in `financial_expense`.

b) `load_main_funding_sources`: Legt Hauptfinanzierungsquellen via `insert_returning` an und referenziert dazugehörige Beträge über einen stabilen `ordinal`-Wert.

c) `load_donators`: Verwendet `upsert_country`, um Länderangaben zu deduplizieren, bevor zugehörige Spenden erfasst werden.

Ändert sich ein API-Block strukturell, bleibt die notwendige Anpassung sauber auf den jeweiligen Handler und die betroffenen Schemaelemente beschränkt. Dadurch bleibt das Gesamtmodell wartbar und erweiterbar.

5 Betrieb

Die lokale Ausführung kann entweder vollständig containerisiert oder nativ auf dem Host-System erfolgen. Der Docker-Workflow bietet eine reproduzierbare Umgebung, in der alle Dienste – Datenbank, Adminer, Grafana und der Ingestor – in definierten Startreihenfolgen bereitgestellt werden. Ein einfacher Aufruf von `docker compose up -build` startet die gesamte Kette. Nach erfolgreichem Healthcheck der PostgreSQL-Instanz führt der Ingestor einen vollständigen Lauf aus; Adminer und Grafana stehen sofort zur manuellen Analyse und Visualisierung zur Verfügung.

5.1 Lokal

Für den nativen Betrieb ohne Container wird zunächst mit `uv sync` das Python-Umfeld eingerichtet. Anschließend genügt `docker compose up -d db adminer`, um ausschließlich die benötigte Datenbank und das Adminer-Interface in Containern bereitzustellen. Der eigentliche Ingestor läuft dann direkt über `uv run python -m lobbyregister_ingestor`. Das relationale Schema wird dabei bei Bedarf automatisch angelegt und bleibt anschließend stabil; wiederholte Läufe verursachen keine Migrationen. Dadurch eignet sich die lokale Umgebung gut für Entwicklung und Debugging, da Änderungen an Code und Mappern sofort getestet werden können, ohne den gesamten Stack neu aufzubauen.

5.2 Docker

Die Docker-Variante kapselt sämtliche Abhängigkeiten und ermöglicht eine vollständig isolierte Ausführung. Postgres, Adminer und Grafana laufen als dauerhafte Dienste; der Ingestor ist als Einmal-Job konzipiert, der nach Abschluss seines Laufes beendet wird. Persistente Docker-Volumes sichern Datenbankinhalte dauerhaft, sodass erneute Läufe nur neue oder geänderte Einträge schreiben. Ressourcenlimits und Healthchecks im Compose-File steuern Start, Überwachung und Verhalten der Container. Damit eignet sich dieser Modus sowohl für reproduzierbare Testläufe als auch für den finalen Betrieb auf Serverumgebungen.

6 Reflexion

Die Zerlegung der API-Domäne in klar abgegrenzte Mapper-Module hat sich als tragfähiges Architekturprinzip erwiesen. Jede Teilstruktur des JSON kann unabhängig verarbeitet, getestet und angepasst werden. Änderungen an der API – sei es neue Felder, neue Kodierungen oder strukturelle Modifikationen – betreffen damit nur den jeweils zuständigen Handler. Die Pipeline bleibt stabil, und Anpassungen bleiben lokal, ohne Kaskaden über das gesamte System auszulösen.

Der cursorbasierte Abruf der Bundestag-API erwies sich ebenfalls als vorteilhaft. Durch die serverseitig bereitgestellten Cursor-Token entstehen kleine, konsistente Batches; unnötige Offsets werden vermieden, und die Datenbank des Bundestags wird nicht mit großvolumigen Abfragen belastet. Das Verfahren ist zudem resilient gegenüber Änderungen zwischen einzelnen Requests, da keine großen Sprünge in der Ergebnismenge existieren.

Die Entscheidung für eine strikte dritte Normalform führt naturgemäß zu einer größeren Anzahl an Tabellen und Foreign Keys. Für die aktuelle Datenmenge überwiegen jedoch die verbesserten Integritätsgarantien, die klaren Kardinalitäten und die Vorhersagbarkeit von Abfrageplänen. Die Pipeline profitiert davon, dass jeder Wert genau strukturiert abgelegt ist und spätere Analysewerkzeuge zuverlässige Statistiken erhalten.

Python und AsyncIO boten während der Implementierung einen geeigneten Kompromiss aus Effizienz und Verständlichkeit. Die semantische Klarheit von `async/await` erleichtert die Nachvollziehbarkeit, während das Event-Loop-Modell genügend Parallelität liefert, um Netzwerkwartezeiten zu überlappen. Gerade im Lehrkontext ist diese Verbindung aus Transparenz, Einfachheit und funktionaler Leistungsfähigkeit ein wesentliches Kriterium.

7 Fazit

Die Kombination aus Python/AsyncIO, streng normalisiertem PostgreSQL-Schema und einem bewusst schlanken Docker-Stack hat sich als robuste Grundlage für eine transparente ETL-Pipeline erwiesen. Cursor-Pagination reduziert die externe API-Last, und die interne Queue schützt die Datenbank vor Überlast. Die klar getrennte Mapper-Architektur erhöht die Wartbarkeit und erleichtert Erweiterungen ebenso wie Fehleranalysen. Im Lehrkontext ist die Nachvollziehbarkeit wichtiger als maximale absolute Performance, und gerade diese strukturelle Klarheit ermöglicht ein präzises Verständnis der Datenflüsse und Transformationsschritte.

8 Ausblick

Der nächste Portfolioteil wird die Abfrageoptimierung und die Visualisierung der Daten in den Vordergrund stellen. Geplant sind geeignete Indizes, selektive Materialized Views und systematische Tuningrunden mit `EXPLAIN (ANALYZE)`. Die Nutzung von `pg_stat_statements` ermöglicht es dabei, Hot-Queries anhand realer Ausführungszeiten und Zugriffshäufigkeiten zu identifizieren. Darauf aufbauend sollen Grafana-Dashboards entstehen, etwa zu jährlichen Spendenvolumina, thematischen Aktivitätsfeldern oder Metriken der Pipeline-Laufzeit. Damit verschiebt sich der Schwerpunkt von der strukturellen Funktionsfähigkeit der ETL-Strecke hin zu performanten Auswertungen und anschaulichen Visualisierungen.

Literatur

[1] Noah Raupold und David Gläsle. *Lobbyregister Ingestor*. https://github.com/dav354/adt. Git-Repository. 2025.