Predictive Analysis as a Service

Fachpraktikum Softwareentwicklung mit Methoden der Künstlichen Intelligenz

Nikolaos Psycharis
Hamza Gueni
Joshua Allius

Erstellt: 26. Februar 2026 *

Fernuniversität Hagen

Zusammenfassung

Im Rahmen des Fachpraktikums „Softwareentwicklung mit Methoden der Künstlichen Intelligenz“ wurde das Projekt „Predictive Analysis as a Service“ (PAaaS) erstellt. Dabei handelt es sich um eine skalierbare Web-Anwendung, um die Anwendung verschiedener KI-Algorithmen zur Analyse diverser Datensätze zu vereinfachen. In einer containerisierten Microservice-Architektur wird unter anderem FastAPI, React und Celery / Redis verwendet, um das „MLCore“ genannte Backend anzusteuern. MLCore bietet dabei viele Möglichkeiten für predictive analysis und reduction analysis. Neben erweiterten Analyseoptionen liegen mögliche Verbesserungen im Bereich der Einstiegshilfe, der internen Datenspeicherung und der allgemeinen Nutzungsleichtigkeit vor.

*Kontakt:
nikolaos.psycharis@studium.fernuni-hagen.de
hamza.gueni@studium.fernuni-hagen.de
joshua.allius@fernuni-hagen.de

1 Übersicht

Zunächst wird kurz die Idee von PAaaS und der Aufbau betrachtet. Dabei wird auf die einzelnen Komponenten und deren Zusammenspiel in Form von Containerisierung eingegangen. Auch wird der Workflow während der Projekt-Entwicklung dargestellt. Anschließend wird detailliert auf den Aufbau der Datenbank und die Strukturierung der Daten eingegangen. Danach folgt eine detaillierte Darstellung von MLCore, also dem Modell-Training und der eigentlichen „Predictive Analysis“. Abschließend wird auf einige Erweiterungen und Ausbaumöglichkeiten eingegangen. Diese stellen interessante Optionen für eine Weiterentwicklung dar, welche jedoch im Zeitrahmen des Projektes nicht mehr umgesetzt werden konnten.

1.1 Idee

Das Ziel hinter PAaaS ist die Schaffung eines Nutzerinterfaces für verschiedene Methoden der Predictive Analysis. Dabei sollte das Tool leicht bedienbar, modular und skalierbar sein. Zuerst wurde das Backend sowie die API spezifiziert, mit einem Frontend was anschließend auf dieser API aufbaut. Ebenso wichtig war bei der Entwicklung ein Fokus auf allgemein gebräuchliche Tools, um die spätere Weiterentwicklung, Wartung und das Deployment zu vereinfachen.

2 Struktur

Bei dem Design von PAaaS wurde ein großer Fokus auf Modularisierung und Skalierbarkeit gelegt. Ebenso wurden primär weit verbreitete Tools eingesetzt, um die Weiterentwicklung und Wartung des Projektes zu vereinfachen. Folglich sind die einzelnen Komponenten über bestimmte Schnittstellen miteinander verbunden. Innerhalb dieser ist eine Modifizierung oder ein Austausch der Komponenten unabhängig der anderen Komponenten umzusetzen. Dabei wurde Docker als Containerisierungslösung eingesetzt, da Docker eine sehr weit verbreitete Lösung darstellt [1].

2.1 Komponenten

Die einzelnen Komponenten des Projekts werden als containerisierte Microservices ausgerollt. Dabei werden die folgenden Komponenten verwendet:

1. Frontend Node: eine React-basierte Weboberfläche zur vereinfachten Nutzung durch den Anwender
2. API Node: eine FastAPI-basierte API, über welche sämtliche Funktionen des Projektes angestoßen werden können
3. Worker Node: ein Python-Container, in dem die MLCore-bezogenen Tasks ausgeführt werden
4. MySQL: eine SQL-basierte Datenbank
5. Redis: Messaging Host für Zuweisung von Celery Tasks von / an verschiedene API / Worker Nodes
6. Flower: eine Weboberfläche zum Redis-Monitoring

Besonders sind dabei die Frontend, API und Worker Nodes zu betrachten. Die verbleibenden Komponenten sind unmodifizierte 3rd-party Softwares.

Die Frontend Node stellt eine in Docker gewrappte React-Applikation dar. React stellt ist dabei ein weit verbreitetes JavaScript-Framework zur Erstellung von interaktiven Weboberflächen. Auf das Frontend wird in der Sektion 6 näher eingegangen.

API Nodes stellen dabei eine in Docker gewrappte FastAPI dar. FastAPI ist ein Python-Framework zur Erstellung von REST-APIs. Die API wird entweder über das Frontend oder direkt durch einen versierten Nutzer angesteuert und löst entsprechende Celery-Tasks aus. Die FastAPI-Schnittstelle wird in der Untersektion 6.2 beschrieben.

Worker Nodes basieren auf einem Python-Dockerimage und bieten die Kernfunktionen von MLCore an, also die tatsächliche Datenverarbeitung. Diese Funktionen werden über Celery Tasks ausgeführt. In der Sektion 5 wird vertiefend auf MLCore eingegangen.

MySQL wurde aufgrund der weiten Verbreitung des Tools verwendet. So war MySQL 2023 die zweitmeistverwendeste Datenbank nach Oracle [2]. Oracle wurde allerdings aufgrund der Lizenzbedingungen für dieses Projekt nicht in Betracht gezogen. Zudem wurde speziell eine SQL-basierte Datenbank verwendet, um die Vorteile einer effizienten Speicherung für strukturierte Metadaten zu nutzen. Dabei ist zu beachten, dass in der aktuellen Version Nutzer-Uploads nur eingeschränkt in der Datenbank gespeichert werden (siehe Sektion 7). Für eine direkte Speicherung z.B. von .csv-Uploads wären auch dokumentbasierte Datenbanken wie beispielsweise MongoDB in Betracht zu ziehen gewesen [3].

Redis und Celery sind verbreitete Tools für Messaging. Diese Tools wurden aufgrund bestehender Erfahrungen im Team sowie der leichten Implementierung in Python gewählt. Flower wurde folglich als Monitoring-Tool für Redis gewählt. Celery lässt sich als Python-Modul einbinden. Dabei können Funktionen (in MLCore) als Task markiert werden. An anderer Stelle wird dann ein Aufruf implementiert. So löst hier ein Aufruf eines FastAPI-Endpunkts nach einer

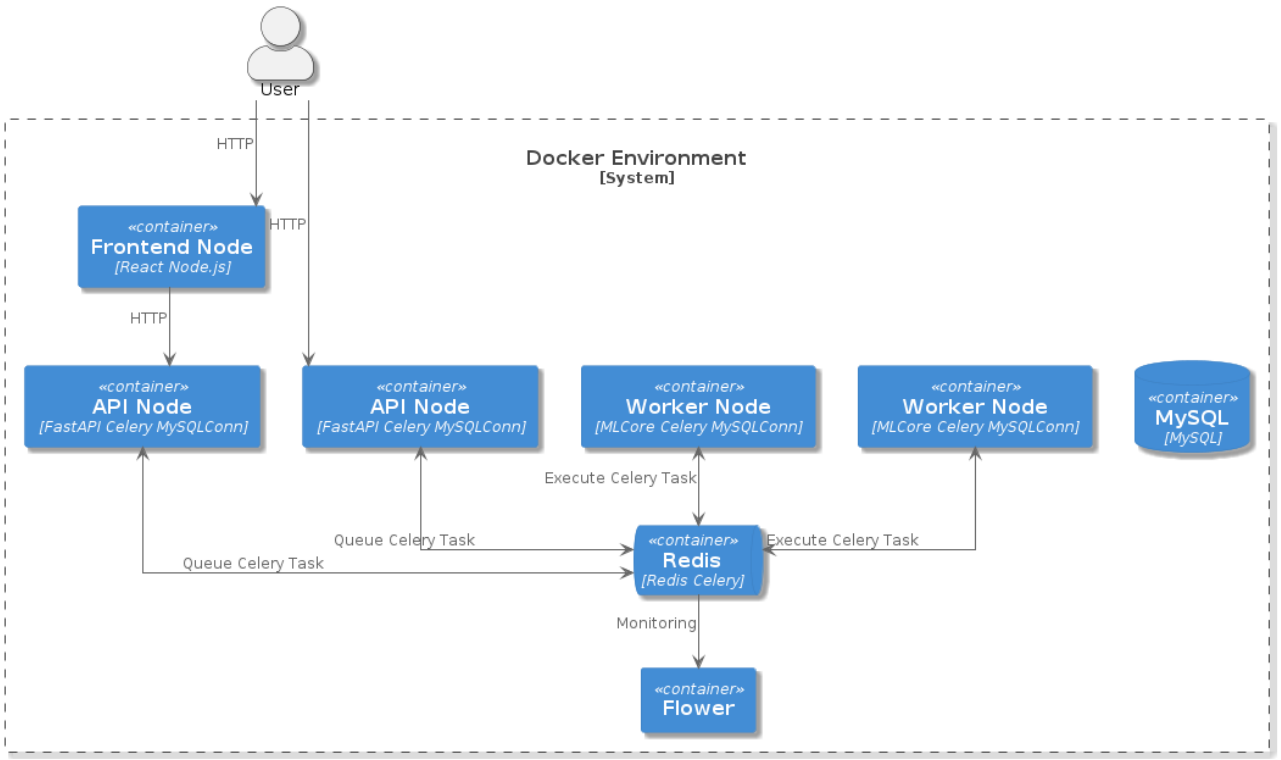

Abbildung 1: *Containerisierte Projekt-Struktur*

einfachen Datenvalidierung einen bestimmten Task mit entsprechenden Parametern auf. Da sich sowohl die API Node als auch die Worker Node bereits bei dem Redis-Server als Celery-Knoten gemeldet haben, wird der Task-Aufruf an den Redis-Server gesendet. Von dort wird der Aufruf an eine Worker Node weitergeleitet. Durch die getaggte Funktion stellt sich die Worker Node nämlich als valide Instanz für den entsprechenden Task-Aufruf bereit.

2.2 Containerisierung

Docker ist ein verbreitetes Tool zu Containerisierung von Anwendungen und Microservices. Auf einem Host wird die sogenannte Docker-Engine installiert. Über diese können isolierte Instanzen von Images als Container erstellt werden. Die Images selber werden durch eine „Dockerfile“ beschrieben [4]. Dabei wird meist ein bereits vorhandenes Image um die gewünschte Funktionalität erweitert, z.B. indem der Projektcode in dem Image eingebunden wird. Dieses Vorgehen nennt sich „Image Layering“[5].

Ein so beschriebenes Image wird anschließend gebaut und kann dann leicht auf anderen Host-Systemen verwendet werden. Dazu können die Images beispielsweise in einem Repository (öffentlich) zur Verfügung gestellt werden oder als „tar-Ball“ exportiert werden [6][7]. Dabei muss der Entwickler des Image wenige Vorkehrungen treffen um das Image auf einer Vielzahl an Hosts verwendbar zu machen: die Docker-Engine übernimmt einen Großteil des Aufwands.

Ebenso kann Docker genutzt werden, um mehrere Container zu deployen, zu vernetzen und diese von außen erreichbar gestalten. Dazu wird „Docker compose“ genutzt. Hierbei wird die Struktur eines „Compose-Stacks“ beschrieben. Hierbei werden die einzelnen Images, die Konfigurationen dieser sowie Netzwerke und Volumes beschrieben.

Ein Beispiel, wie die einzelnen Komponenten/-Container des Projektes in einem Compose-Stack verknüpft werden können, findet sich in Abbildung 2. Der `frontend`-Service (die Frontend-Node) baut ein Docker-Image basierend auf der `DockerfileFrontend` und ist über Port 3000 erreichbar. Er benötigt den `api`-Service (API-Node) zum Start. Analog dazu funktioniert die API-Node mit Umgebungsvariablen zur Konfiguration der Datenbank und des Redis-Servers. Diese hängt von dem `redis_server` ab und ist über Port 42000 erreichbar. Die Worker-Node wird hier mehrfach repliziert erstellt, um die Skalierbarkeit mit Celery / Redis zu zeigen. Neben den restlichen Services werden noch mehrere Volumes definiert, um die Daten persistent zu speichern.

```
services:
  frontend:
    build:
      context: .
      dockerfile: DockerfileFrontend
    ports:
      - "3000:3000"
    environment:
      VITE_API_URL: api:42000
    restart: "unless-stopped"
    depends_on:
      - api
  api:
    build:
      context: .
      dockerfile: DockerfileApi
    volumes:
      - testdata:/code/worker/testdata
    environment:
      REDISSERVER: redis://redis_server:6379
      DELAY_DB_CONN_ON_STARTUP: 5
      DB_HOST: db
      DB_PORT: "3306"
      DB_NAME: "team1_db"
      DB_USER: "team1_user"
      DB_PASS: "team1_pass"
      SEED_TEST_DATA: "false"
      TEST_DB_NAME: "team1_db"
      TEST_DATA_PATH: "/code/db/test_db.txt"
    links:
      - "db:db"
    ports:
      - "42000:80"
    restart: "unless-stopped"
    depends_on:
      - redis_server

  worker:
    build:
      context: .
      dockerfile: DockerfileWorker
    volumes:
      - model-data:/models
      - testdata:/code/worker/testdata
    environment:
      REDISSERVER: redis://redis_server:6379
      DB_HOST: db
      MODEL_BASE_PATH: /models
      # C_FORCE_ROOT: "true"
    links:
      - "db:db"
    deploy:
      mode: replicated
      replicas: 3
    restart: "unless-stopped"
    depends_on:
      - redis_server
  redis_server:
    image: redis

  flower:
    image: mher/flower
    command:
      ["celery", "--broker=redis://redis_server:6379", "flower", "--port=5555"]
    ports:
      - "5555:5555"
    depends_on:
      - redis_server

  db:
    image: mysql:8.0
    container_name: fachpraktikum-mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: safe123
      MYSQL_DATABASE: team1_db # auto-create DB
      MYSQL_USER: team1_user # auto-create user
      MYSQL_PASSWORD: team1_pass
    ports:
      - "3306:3306"
      - "33060:33060"
    volumes:
      - db-data:/var/lib/mysql

volumes:
  db-data:
  model-data:
  testdata:
```

Abbildung 2: *Beispiel: docker-compose.yml*

3 Workflow

Im Rahmen der Entwicklung wurde GitHub für Source Code Management verwendet. Dabei wurden einzelne Feature Branches erstellt, an denen eines oder mehrere Mitglieder des Projektes entwickeln konnten. Anschließend wurde im Rahmen des Pull Requests (PRs) ein Review durch die restlichen Mitglieder durchgeführt.

Ebenso wurde GitHub Actions verwendet, um die PRs um eine Continuous Integration (CI) Pipeline zu ergänzen. Dabei wurden der Python Code durch Pytest abgedeckt. Für das TypeScript basierte Frontend sowie die Docker Container wurde in einem zweiten CI-Job geprüft, ob der Buildprozess für das Frontend sowie die Docker Images erfolgreich beendet. Zum aktuellen Zeitpunkt wurde kein Continuous Deployment umgesetzt. Damit wurden die erstellten Images also weder veröffentlich noch auf einer staging Umgebung deployed.

Nur wenn beide Pipeline-Jobs, also pytest und der Build-Prozess, erfolgreich abgeschlossen sowie wenn ein Review durchgeführt wurde, wurde der PR umgesetzt. Dieser typische git-Workflow hat dabei die unabhängige Arbeit der Teammitglieder stark vereinfacht.

4 Datenbank

`PAaaS` folgt einer Kette. Eine Nutzerin oder ein Nutzer legt ein Dataset an. Ein Upload wird als Dataset-Version gespeichert. Darauf wird ein Machine-Learning-Problem definiert. Aus dem Training entstehen Modelle, und aus Prediction-Runs entstehen Ausgaben. Solange das Projekt klein ist, bleibt diese Kette übersichtlich. Sobald mehrere Datasets, Modelle und Jobs parallel existieren, wird die Nachvollziehbarkeit wichtiger als die einzel- nen Schritte. Dann geht es um Fragen wie: Welche Dataset-Version hat ein Modell trainiert. Welches Modell hat eine Vorhersage erzeugt. Wer hat einen Job angestoßen. Die Datenbank hält diese Beziehungen explizit fest. Jedes Objekt wird einmal gespeichert und über stabile Identifier mit dem vorherigen Schritt verbunden. Abbildung 3 zeigt die fachliche Lineage. Abbildung 5 zeigt, wie diese Lineage als Tabellen und Beziehungen umgesetzt ist.

4.1 Grundlagen und Vorgehen

Dieser Abschnitt führt die Begriffe ein, die später im Text vorkommen, und beschreibt das Vorgehen im Projekt. Eine Datenbank speichert Daten so, dass sie nach dem Ende eines Programms weiterhin verfügbar sind. Eine relationale Datenbank organisiert Daten in Tabellen. Tabellen haben Zeilen und typisierte Spalten. Die Menge aus Tabellen, Spalten und Regeln nennt man Schema. MySQL ist ein relationales Datenbankmanagementsystem. Es speichert und fragt Daten mit SQL ab und kann Regeln wie Schlüsselbeziehungen direkt im Schema erzwingen [8].

Nicht-relationale Datenbanken speichern Daten in

anderen Strukturen, zum Beispiel als Dokumente, Key-Value-Paare oder Graphen. In PAaaS sind die Kernobjekte und ihre Beziehungen stabil. Datasets, Versionen, Probleme, Modelle, Jobs und Predictions haben eine klare Struktur. Dafür passt ein relationales Schema gut.

Ein Primärschlüssel identifiziert eine Zeile eindeutig. Ein Fremdschlüssel speichert den Primärschlüsselwert einer anderen Tabelle und drückt damit eine Beziehung aus. FremdschlüsselschützendieIntegrität, weilsieVerweiseaufnichtexistierendeZeilenverhindern [8]. Im Schema werden UUIDs als Identifier verwendet. Eine UUID ist ein standardisiertes 128-Bit-Format, das für unabhängige Generierung gedacht ist [9]. Im Projekt werden UUIDs als `CHAR(36)` gespeichert. Das macht sie beim Debugging lesbar, auch in Logs und in SQL-Abfragen.

Ein Teil der Informationen in PAaaS hat keine feste Struktur. Dataset-Profile hängen von den Spalten eines Uploads ab. Modellmetriken hängen vom Algorithmus und der Aufgabe ab. JSON ist ein standardisiertes Format für strukturierte Daten mit Objekten und Arrays [10]. MySQL bietet dafür einen nativen JSON-Datentyp. Damit lassen sich solche Metadaten speichern, ohne ständig neue optionale Spalten anzulegen [11].

PAaaS speichert außerdem Verweise auf große Artefakte, etwa CSV-Dateien, Modell-Binaries und Prediction-Outputs. Deren Speicherorte werden als URI abgelegt. Eine URI ist ein standardisierter Bezeichner für Ressourcen mit klarer Syntax [12]. So bleibt die Datenbank auf strukturierte Daten und Metadaten fokussiert, während Dateien in der Storage-Schicht liegen.

Im Projekt wurde ein Lineage-First-Ansatz gewählt. Jeder Schritt im Workflow wird als eigene Tabellenzeile gespeichert. Jedes spätere Objekt speichert den Identifier des Schrittes, von dem es abhängt. Damit wird Nachvollziehbarkeit zu einer Query-Aufgabe. FüreinzelneUpdatesspieltzusätzlichKonsistenzübermehrereZeileneineRolle. Transaktionen bündeln mehrere SQL-Statements zu einer Einheit. Isolation Levels beschreiben, wie parallele Transaktionen miteinander interagieren [13].

4.2 Schema und Implementierung

Dieser Abschnitt verbindet den Workflow mit dem konkreten Schema und dem Python-Zugriff.

Abbildung 3 zeigt die fachliche Kette. Das Schema bildet diese Kette mit Fremdschlüsseln nach. Eine Dataset-Version zeigt auf ihr Dataset. Ein Problem zeigt auf eine Dataset-Version. Ein Modell zeigt auf ein Problem. Jobs und Predictions zeigen auf das jeweilige Modell. Jobs speichern zusätzlich, wer die Aktion angefordert hat.

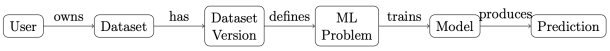

Abbildung 3: *Konzeptionelle Lineage von user-owned Datasets bis zu Predictions.*

Abbildung 5 folgt derselben Reihenfolge wie der Workflow. Dadurch wird auch klar, warum Joins für Detailansichten naheliegend sind. Wenn die UI zu einem Modell zusätzlich den Dataset-Namen anzeigen soll, gibt es dafür einen sauberen Join-Pfad.

Der Python-Zugriff ist in db.py umgesetzt und wird von der API genutzt. Das Modul folgt dem Connection- und Cursor-Muster, wie es in der Python DB-API 2.0 beschrieben ist [14] Die meisten Funktionen folgen einem einfachen Rhythmus. Create-Funktionen schreiben eine Zeile und geben die UUID zurück. Get-Funktionen lesen eine Zeile über den Primärschlüssel. Update-Funktionen ändern nur die übergebenen Felder. Für Listenansichten sind Filter und Pagination eingebaut, damit die API nicht ganze Tabellen in den Speicher lädt.

Eine Operation ist kritischer als normale CRUD-Aufrufe. Pro ML-Problem soll es genau ein Production-Modell geben. Ein Wechsel betrifft mehrere Zeilen und läuft daher als Transaktion. Dabei wird die ML-Problem-Zeile gesperrt, ein vorheriges Production- ModellbeiBedarfarchiviert,das neue Modell als Production markiert und der Production-Pointer aktualisiert. Abbildung 4 zeigt die Schritte. Transaktionen und Isolation-Regeln liefern die nötige Konsistenz bei parallelen Requests [13].

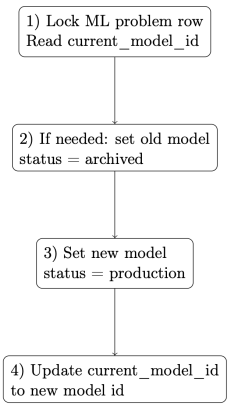

Abbildung 4: *Production-Model-Switch als konsistente Datenbankoperation.*

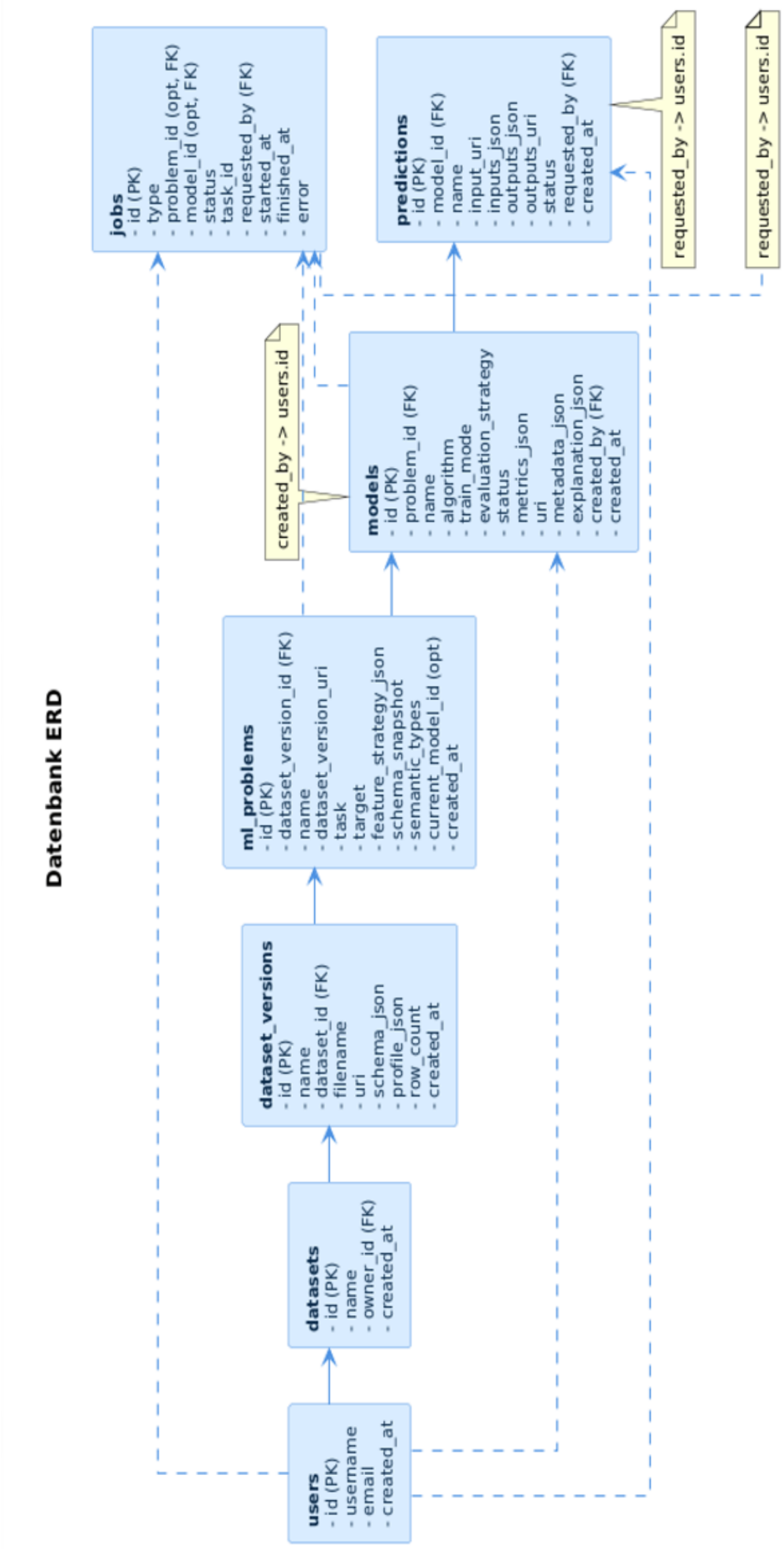

Abbildung 5: *Entity-Relationship-Diagramm des* **PAaaS***-Datenbankschemas.*

4.3 Evaluation

Die Evaluation fokussiert auf funktionale Korrektheit. Der wichtigste Nachweis ist ein lokaler Smoke Test, der einen vollständigen Happy Path ausführt. Er legt eine Nutzerin oder einen Nutzer an, erzeugt ein Dataset, eine Dataset-Version, ein ML-Problem, ein Modell, einen Trainingsjob und eine Prediction. Danach werden Datensätze wieder gelesen und es wird geprüft, ob die Identifier passen und die Fremdschlüsselbeziehungen gültig sind. Damit ist gezeigt, dass das Schema die beabsichtigte Insert-Reihenfolge unterstützt und die Helper-Funktionen konsistent arbeiten.

Zwei SQL-Abfragen verdeutlichen das Traceability-Ziel. Abbildung 6 listet Modelle zu einem ML-Problem. Abbildung 7 rekonstruiert die Lineage hinter einer Prediction über Joins ent- lang der Workflow-Tabellen.

Diese Evaluation ist bewusst praxisnah. Sie zeigt, dass die Datenbank den Projekt-Workflow Ende zu Ende abbildet. Sie misst keine Performance unter Last und deckt keine umfangreichen Parallelitäts- oder Stress-Szenarien ab.

5 MLCore

MLCore ist das Subsystem für maschinelles Lernen, implementiert unter `src/mlcore`. Es fokussiert sich auf Data Profiling und Vorverarbeitung, Model Training, Evaluation und Explainability der Modelle sowie Vorhersage Workflows.

5.1 Designziele

Das `mlcore` Package enthält die Ende-zu-Ende-Logik für maschinelles Lernen, die aus den Workers ausgeführt wird. Es ist verantwortlich sowohl für das Data Profiling und die Vorverarbeitung als auch für die Modellauswahl durch vordefinierte Presets sowie für das Training, die Evaluation der Modelle, ihre Explainability und die Vorhersage.

Dieses Package ist gezielt von der API-Logik getrennt. Verschiedene Aufgaben werden von den Workers behandelt und durch Datenbank-Helpers in der Datenbank dokumentiert und gespeichert. Die Trennung des MLCores von der API ist gezielt, um die Verantwortlichkeiten der einzelnen Komponenten klar zu begrenzen und um die Skalierbarkeit sowie die Wiederverwendbarkeit der ML-Komponente für Batch-Training und Vorhersage zu realisieren. [15][16]

Die Hauptdesignziele des Komponenten sind:

- **Wiederverwendbarkeit (Reusability)**: Vordefinierte Preset-basierte Pipelines enthalten das Trainings- und Vorverarbeitungs-logik für verschiedene Klassifikations- und Regressionsmodelle. Diese Pipelines werden sowohl für das Training als auch für die Vorhersage verwendet.
- **Erweiterbarkeit (Extensibility)**: Presets werden dynamisch aus mlcore/presets geladen. Das ermöglicht es, durch die Erstellung neuer Presets, neue Algorithmen als separate Modulen einfach einzufügen, ohne die Trainings- oder Vorhersagelogik anzupassen.
- **Testbarkeit (Testability)**: Profiling, Evaluation und Explainability sind als eigene Module erstellt, um eine einfache und gezielte Testbarkeit sicherzustellen.

5.2 Data Profiling und Vorverarbeitung

Data Profiling Das `mlcore/profile/profiler` berechnet statistische Daten und semantische Typen für die Spalten einer CSV-Datei. Für jede Spalte werden die folgenden Informationen berechnet:

- der semantische Typ aus den Datentypen
  - numeric
  - categorical
  - boolean
  - datetime
  - unknown
- der Prozentanteil der fehlenden Werte, die Kardinalität und eine Zusammenfassung der Verteilung.
- ob alle Werte der Spalte gleich sind (konstante Spalte) oder ob die Spalte id-ähnlich ist (inkl. sequenzähnliche Integer-IDs) zur Exklusion.
- ob die Spalte sehr hohe Kardinalität hat, um eine Meldung an den Benutzer zu geben.
- eine vorgeschlagene Analyseart (Klassifikation oder Regression).

Das resultierende Profil enthält eine Datensatzzusammenfassung, die id-ähnliche Spalten („id_candidates“), einen Exklusionsvorschlag(„exclude_suggestions“) und Metadaten pro Spalte. Später wird dieses Profil für die Vorverarbeitung und das semantische Typing während des Trainingsprozesses verwendet.

Vorverarbeitung Das Training beginnt beim Laden der CSV-Datei einer Dataset-Version durch `data_reader.get_dataframe_from_csv`. Wenn es schon ein Profil für diese Dataset-Version gibt, liest der Trainer dieses ebenfalls ein, und wenn nicht, führt der Trainer die Funktion `profiler.suggest_profile` aus, um eines zu berechnen. Dann wird die Vorverarbeitung durch `data_reader.preprocess_dataframe` ausgeführt. Die Vorverarbeitungsschritte sind die folgenden:

```
SELECT id, name, algorithm, status, created_at
FROM models
WHERE problem_id = %s
ORDER BY created_at DESC;
```

Abbildung 6: *SQL: `models` für ein bestimmtes Problem ausgeben.*

```
SELECT
p.id AS prediction_id,
m.id AS model_id,
mp.id AS problem_id,
dv.id AS dataset_version_id,
d.id AS dataset_id,
d.name AS dataset_name
FROM predictions p
JOIN models m ON m.id = p.model_id
JOIN ml_problems mp ON mp.id = m.problem_id
JOIN dataset_versions dv ON dv.id = mp.dataset_version_id
JOIN datasets d ON d.id = dv.dataset_id
WHERE p.id = %s;
```

Abbildung 7: *SQL: Ein `prediction` zu dem zugehörigen `dataset` zurückverfolgen.*

- Der Benutzer kann durch die Feature-Strategie des ML Problems auswählen, welche Spalten inkludiert und welche exkludiert werden sollen. Das Default-Verhältnis der Feature-Stategie ist „auto“. Das bedeutet, dass alle Spalten als inkludiert und die Spalten aus „exclude_suggestions“als exkludiert berücksichtigt werden. Dieses Verhältnis kann der Benutzer durch Anpassung der Feature-Strategie beeinflussen und nur die ausgewählten inkludierten und exkludierten Spalten berücksichtigen.
- Die Zeilen mit fehlenden oder Null-Werten in der Zielspalte werden entfernt.
- Die Daten werden in Merkmale X und Zielvariable y getrennt und zurückgegeben.

Am Ende werden die Spalten durch `data_reader.get_semantic_types` mithilfe des Profils in kategoriale, numerische und boolische Listen unterschieden, die dann im ColumnTransformer in jedem Preset verwendet werden, um eine konsistente Behandlung für jeden Merkmalstyp sicherzustellen.

5.3 Training Pipeline Architektur

Das Training wird durch `trainer.train` ausgeführt und folgt einer strukturierten Pipeline:

1. Die Dataset-Version, Zielspalte und der Analysetyp werden aus der Datenbank geladen.
2. Das Dataset-Version-Profil wird geladen oder aus der CSV-Datei neu erstellt.
3. Die Daten werden vorverarbeitet und in X und y getrennt.
4. Die semantischen Typen der Merkmale werden mithilfe des Profils bestimmt, da sie für die Konfiguration der Vorverarbeitungsschritte vor dem Modelltraining benötigt werden .
5. Die Daten werden in Train/Test mit train_test_split getrennt (inkl. Statifikation für Klassifikation).
6. Label Encoding wird auf die Zielspalte angewendet (nur für Klassifikation).
7. Das Preset wird geladen und die Modell-Pipeline für den ausgewählten Algorithmus wird gebaut.
8. Das Modell wird trainiert.
9. Das Modell wird mit den Testdaten evaluiert.
10. Die Cross-Validierungsmetriken werden berechnet (wenn gefordert).
11. Eine Explainability-Zusammenfassung wird mithile des SHAP-Frameworks berechnet (wenn gefordert).
12. Das Modell und seine Metadaten (inkl. Metriken, Cross-Validierung und Explainability-Zusammenfassung) werden gespeichert.

Preset Integration Die Presets werden dynamisch durch `preset_loader.loader` geladen, der Module aus <algorithm>.py lädt. Jedes Preset bietet eine `build_model()` Funktion, die die folgenden Objekte

zurückgibt:

- Eine scikit-learn-Pipeline, bestehend aus einem `ColumnTransformer` und einem `Estimator`.
- Ein Metadaten-Dictionary, das auf `metadata_presets.py` basiert.

Dieser Prozess standardisiert die Vorverarbeitungs- und Trainingsprozesse für alle unterstützten Algorithmen und hält nur die estimator-spezifischen Parameter modular, um eine konsistente Struktur zu gewährleisten.

Holdout Evaluation (Train-Test-Split) Nach dem Training des Modells werden Vorhersagen für die Testdaten erzeugt, um die Modellmetriken zu bestimmen. Für Klassifikationsaufgaben werden die Vorhersagen mithilfe des `LabelEncoder` wieder in die ursprünglichen Klassenlabels zurücktransformiert, und anschliessend werden die Metriken auf Basis der dekodierten Labels berechnet.

Cross-Validation Wenn eine Cross-Validation gefordert wird (`evaluation_strategy = ''cv''`), wird zusätzlich durch mlcore/metrics/cv_calculator eine Cross-Validation-Metrik berechnet:

- Für Klassifikation wird die Macro-F1-Metrik mithilfe des Stratified K-Fold-Verfahrens berechnet.
- Für Regression wird die R^2-Metrik mithilfe des K-Fold-Verfahrens berechnet.

Falls das „auto"-Preset ausgewählt wurde, wird im Rahmen der Modellauswahl bereits eine Cross-Validation-Metrik über die verschiedenen Modelle hinweg berechnet, um das „beste"Modell zu bestimmen. In diesem Fall verwendet der Trainer diese intern-berechnete Cross-Validation-Zusammenfassung, anstatt die Cross-Validation erneut auszuführen.

5.4 Modell-Presets und Modellauswahl

Preset-basiertes Design Die Presets enthalten verschiedene Algorithmusoptionen und Vorverarbeitungslogik unter einer konsistenten Struktur, um zu gewährleisten, dass sie in standardisierter Weise angewendet werden können. Sie unterstützen verschiedene `train_modes` („fast", „balanced", „accurate"), die die Hyperparameter der Modelle anpassen, wie z.B. die Anzahl der Estimatoren oder die Regularisierungsparameter usw.

Implementierte Modellfamilien

- `Linear / Logistic Modelle`
  - Klassifikation: `LogisticRegression`
  - Regression: `LinearRegression`, `Ridge`, `Ridge` in Kombination mit `PolynomialFeatures`

  Begründung: Schnelle Baseline-Modelle mit hoher Interpretierbarkeit und robuster Leistung bei linear separierbaren Daten.
- `Entscheidungsbaum-basierte Modelle`
  - Klassifikation: `RandomForestClassifier`, `ExtraTreesClassifier`
  - Regression: `RandomForestRegressor`

  Begründung: Nichtlineare Modellierung, robust gegenüber Merkmalskalierung sowie hohe Leistungsfähigkeit bei Datensätzen mit gemischten Merkmaltypen.
- `Gradient-Boosting Modelle`
  - Klassifikation: `HistGradientBoostingClassifier`
  - Regression: `HistGradientBoostingRegressor`

  Begründung: Hohe Leistungsfähigkeit bei tabularischen Daten bei gleichzeitig vertretbaren Trainingskosten. Diese Modelle erfordern dichte Eingabematrizen. Daher wird die Ausgabe des OneHotEncodings in diesen Presets explizit als „dense"konfiguriert.
- `XGBoost Modelle`[17]
  - Klassifikation: `XGBClassifier`
  - Regression: `XGBRegressor`

  Begründung: Sehr hohe Leistungsfähigkeit bei tabularischen Daten durch starke Regularisierung und gradientbasiertes Tree-Boosting.
- ,,`Auto`''-Preset
  - Klassifikation: `AutoClassifier`
  - Regression: `AutoRegressor`

  Begründung: Sie evaluieren mehrere Kandidatenmodelle aus einigen der zuvor genannten Algorithmen mithilfe von Cross-Validation-Metriken und wählen den Estimator mit der höchsten Leistung aus.

Klassifikationspresets gegenüber Regressionspresets Klassifikationspresets verwenden probabilistische Ausgaben sowie die Macro-F1-Metrik, um Klassenungleichgewichte angemessen zu berücksichtigen. Regressionspresets verwenden die R^2-Metrik und bieten optional zusätzliche `PolynomialFeatures`, um leichte Nonlinearitäten zu modellieren, während die Interpretierbarkeit weitgehend erhalten bleibt.

5.5 Evaluierung und Metriken

Klassifikationsmetriken Das `mlcore/metrics/metrics_calculator` berechnet für Klassifikation:

- Genauigkeit („Accuracy")
- Präzision („Precision") (macro)
- Sensitivität („Recall") (macro)

- F1 (macro)

Macro-averaging wird verwendet, um die Übergewichtung von Mehrheitsklassen zu vermeiden und eine gleichmässige Bewertung aller Klassen sicherzustellen.

Regressionsmetriken Das `mlcore/metrics/metrics_calculator` berechnet für Regression:

- Mean Absolute Error („MAE“)
- Mean Squared Error („MSE“)
- Root Mean Squared Error („RMSE“)
- R^2
- Mean Absolute Percentage Error („MAPE“)

Diese Metriken bieten eine komplementäre Sicht auf die Fehlersensitivität und das relative Fehlerverhältnis.

Einschränkung der Metriken

- Die Macro-F1-Metrik gewichtet jede Klasse gleich und kann daher die Leistung auf dominanten Klassen unterschätzen.
- R^2 kann bei stark nichtlinearen Zusammenhängen oder schief verteilten Zielvariablen die Modellgüte falsch darstellen.
- MAPE wird instabil, wenn die tatsächlichen Zielwerte nahe Null liegen.

5.6 Explainability

Die Explainability wird in `mlcore/explain` implementiert und basiert vollständig auf dem SHAP-Framework. Der Trainer erstellt einen SHAP-Explainer im transformierten Merkmalsraum nach Anwendung des ColumnTransformer, wobei die Merkmalsnamen aus dem trainierten ColumnTransformer übernommen werden.

SHAP Workflow

- Ein Referenzdatensatz wird als Stichprobe aus den Trainingsdaten erzeugt.
- Ein Erklärungsdatensatz wird als Stichprobe aus den Testdaten erzeugt.
- Der shap.Explainer wird entweder für `predict_proba` (für Klassifikation) oder für `predict` (für Regression) konfiguriert.
- In `summary_calculator.py` wird eine statistische Zusammenfassung erstellt, die Folgendes enthält:
  - Globale Mean Absolute SHAP-Werte pro Merkmal zur Identifikation der Top-k-wichtigsten Merkmale (Standardwert $k = 30$).
  - Aggregation auf Elternebene zur Gruppierung von durch OneHotEncoding erzeugten Merkmalen.
  - Quantil-Zusammenfassungen von SHAP-Werten und Merkmalswerten (dieses Feature wurde verworfen und nicht weiterverwendet).

Merkmalsnamen und Gruppierung In `get_feature_names` werden die Merkmalsnamen aus der Vorverarbeitungspipeline rekonstruiert:

- One-Hot-encodierte kategoriale Merkmale werden als ,,`col=value`‘‘-Einträge dargestellt.
- Numerische und boolesche Merkmale werden mit ihren ursprünglichen Namen dargestellt.
- Eine parallele Liste feature_parents erhält die Gruppierung nach der ursprünglichen Spalte.

5.7 Vorhersage Pipeline Architketur

Die Vorhersage wird durch `predictor.predict` ausgeführt:

1. Die Eingabedaten werden aus einer CSV-Datei oder einem In-Memory-DataFrame geladen.
2. Das ausgewählte Modell wird geladen:
   - Wenn `model_id=,,production‘‘`, wird das aktuelle Produktionsmodell des gegebenen ML-Problems geladen. Um diese Option zu verwenden, muss bereits ein Modell für dieses ML-Problem mit dem Status “Production” gesetzt sein.
   - Andernfalls wird das Modell mit der gegebenen ID geladen.
3. Die Modellmetadaten und das gespeicherte Zielschema werden für die Datenvalidierung geladen.
4. Die Zielspalte wird aus den Eingabedaten verworfen, wenn sie vorhanden ist.
5. Das Schema der Eingabedaten wird validiert:
   - Spalten mit leerem oder dupliziertem Namen werden verworfen.
   - Alle Merkmale aus dem `feature_order` des Schemas sind erforderlich.
   - Zusätzliche Spalten werden geloggt und anschliessend ignoriert, indem eine Neuordnung gemäss dem gespeicherten `feature_order` erfolgt.
6. Die Vorhersage erfolgt mithilfe der geladenen Pipeline.
7. Für Klassifikation werden die Vorhersagen durch `label_classes` zurücktransformiert, sofern diese in den Metadaten gespeichert sind.
8. Die Ergebnisse werden in einer Vorhersagezusammenfassung mit Eingabe- und Ausgabedaten gespeichert.

Die Vorhersageausgabe enthält die verwendeten Eingabedaten, die vorhergesagten Labels (für Klassifikation) sowie eine Kopie der Modellmetadaten zur Nachvollziehbarkeit und Kontrolle.

5.8 Einschränkungen und mögliche Erweiterungen

Skalierbarkeitseinschränkungen

- Training und Explainability werden im Hauptspeicher (in-memory) und auf der CPU ausgeführt und skalieren daher nur eingeschränkt für sehr grosse Datensätze.
- SHAP-Berechnungen können auch bei Verwendung von Stichproben rechenintensiv sein, insbesondere bei grossen Merkmalsräumen.

ifecycle Verwaltung

- Modell- und Vorhersageversionen werden über Metadaten und Datenbankeinträge verwaltet, jedoch existiert keine automatisierte Retention- oder Retirement-Strategie zur Bereinigung veralteter Modelle oder Vorhersagen, was langfristig zu unnötigem Speicherverbrauch führen kann.
- Es ist keine automatische Überwachung der Modelle und ihrer Leistung in der Produktionsumgebung implementiert. Veränderungen in den Vorhersageeingabedaten oder Leistungsabfälle der Modelle könnten daher unentdeckt bleiben.

6 WebUI

Das Frontend implementiert eine Dashboard-basierte Single-Page-Application („SPA“) zur Verwaltung von Datasets, Dataset-Versionen, ML-Problemen, Modellen und Vorhersagen. Es wurde für Anwender entwickelt, die Analysen und ML-Workflows ausführen möchten, ohne direkt mit den Backend-Services interagieren zu müssen.

Ein SPA-Dashboard ist in diesem Kontext geeignet, da es häufige Wechsel zwischen Listen- und Detailansichten unterstützt, Zustände durch URL-Query-Parameter beibehalten kann und interaktive Aktualisierungen ohne vollständige Seitenneuladung ermöglicht. Dadurch bleibt der Anwendungskontext auch bei Aktualisierungen erhalten.

Das Dashboard ist durch asynchrone ML-Workflows eingeschränkt. Trainings- und Vorhersageprozesse sind langlaufende Hintergrundaufgaben, und aus diesem Grund können die Listenansichten nicht immer direkt aktualisiert werden, da die zuvor genannten Backend-Prozesse nicht sofort abgeschlossen werden. Um dieses Problem zu vermeiden und eine aktuelle Darstellung sicherzustellen, werden nicht-blockierende Formulare, visuelle Rückmeldungen („toasts“) sowie Event-Driven-Aktualisierungsmechanismen eingesetzt. Dies beeinflusst die Zustandsverwaltung und Aktualisierungslogik im Frontend, da Änderungen erst nach Abschluss der Hintergrundprozesse im Frontend durch Event-Streams synchronisiert werden.

6.1 TechStack

- `React` und `Typescript`: React bietet ein komponentenbasiertes Modell für komplexe Dashboards mit wiederverwendbaren Komponenten, was für dieses Projekt geeignet ist, da es viele ähnliche Seiten und Komponenten gibt.
  TypeScript erzwingt eine starke Typisierung für alle Komponenten und Funktionen, was zu einer konsistenten und fehlerreduzierten Codebasis führt.
- `Vite`: Vite wird verwendet, da es schnelle Builds in der Entwicklungsumgebung und optimierte Builds in der Produktionsumgebung bietet.
- `React Router`: Die Routing-Schicht bindet die hierarchische Seitenstruktur (dataset → dataset version → ml problem → model → prediction) mit einer verschachtelten Pfadstruktur, was tiefe Verlinkungen und eine leichtere Navigation ermöglicht.
- `Tailwind CSS` und `shadcn` (RadixUI-basierte Komponenten): Tailwind bietet ein konsistentes Styling zwischen allen Seiten und Komponenten des Dashboards. `shadcn`-Komponenten bieten vordefinierte UI-Elemente mit intefrierter Funktionalität, die zusammen mit Tailwind eine konsistente Darstellung des Dashboards gewährleisten.
- `Recharts`: Recharts wird ausschliesslich für die Darstellung der Explainability verwendet und bietet interaktive und flüssige Diagramme.
- `Zod` und `React Hook Form`: Zod ermöglicht die Erstellung von Validierungsschemata für alle Formulare des Dashboards. React Hook Form bietet kontrollierte Formulare mit minimierten Re-Renders und klarer Fehlerbehandlung.

Die Auswahl dieser Werkzeuge unterstützt die Maintainability, Accesibility, Interaktivität, Konsistenz und Fehlerbehandlung, die dieses Dashboard benötigt, um eine stabile Benutzererfahrung zu gewährleisten.

6.2 FastAPI

Das Frontend kommuniziert mit einer FastAPI, welche auf den API-Nodes hinterlegt ist. Dabei wurde FastAPI als Framework genutzt, da es gut in Python

integrierbar ist und somit gut mit Celery / Redis kombinierbar ist. Außerdem bietet FastAPI standardmäßig eine interaktive „Swagger UI“-Dokumentation. Diese lässt sich unter `/docs` auf der API-Node abrufen.

Diese interaktive Dokumentation war besonders in den frühen Phasen der Projektentwicklung (ohne Frontend) hilfreich, da hier leicht die Funktionalität des Backends geprüft werden kann.

6.3 Webanwendungsarchitektur

Routing und Seitenstruktur Das Routing ist in `frontend/src/routes/index.tsx` definiert und wird mithilfe von `createBrowserRouter` implementiert. Die Routing-Struktur ist in zwei Bereiche gegliedert: einen hierarchischen Hauptbereich sowie aggregierte Übersichtsseiten zur vereinfachten Navigation. Der Hauptbereich enthält die zentrale Anwendungslogik und bildet die hierarchische Domänestruktur der Datenbank ab.

Hauptbereich:

- `/` → Login-Seite (derzeit nur als Platzhalter implementiert und inaktiv, da im Backend keine Authentifizierungslogik für Benutzer implementiert ist)
- `/dashboard` → Übersicht
- `/dashboard/datasets` → Liste aller Datensätze
- `/dashboard/datasets/:datasetId` → Liste der Versionen eines Datensatzes
- `/dashboard/datasets/:datasetId/:datasetVersionId` → Liste der ML-Probleme einer Datensatz-Version
- `/dashboard/datasets/:datasetId/:datasetVersionId/:problemId` → Liste der Modelle eines ML-Problems
- `/dashboard/datasets/:datasetId/:datasetVersionId/:problemId/:modelId` → Liste der Vorhersagen eines Modells
- `/dashboard/datasets/:datasetId/:datasetVersionId/:problemId/:modelId/:predictionId` → Details einer Vorhersage

Zusätzlich existieren aggregierte Seiten zur vollständigeren Übersicht über die Datenbank sowie zur vereinfachten Navigation.

Aggregierte Übersichtsseiten:

- `/dashboard/dataset-versions` → Liste aller Datensatz-Versionen
- `/dashboard/ml-problems` → Liste aller ML-Probleme
- `/dashboard/models` → Liste aller Modelle
- `/dashboard/predictions` → Liste aller Vorhersagen
- `/dashboard/jobs` → Liste aller Celery-Jobs. Diese Seite wurde nicht weiterentwickelt und ist daher nicht über die Sidebar erreichbar. Der Status asynchroner Aufgaben wird stattdessen in die Modell- und Vorhersagelisten integriert, um den aktuellen Zustand von Hintergrundprozessen sichtbar zu machen.
- Nach Auswahl eines Eintrags in diesen Tabellen wird der Benutzer zur entsprechenden Detailseite im Hauptbereich weitergeleitet.

Diese Struktur ermöglicht ein konsistentes Navigationsmodell für den Benutzer. Jede verschachtelte Seite enthält detailliertere Informationen zu einem einzelnen Eintrag, während die aggregierten Listenansichten eine globale Sicht auf die Datenbank zur Verwaltung bieten.

Layout und Navigation Das Dashboard-Layout verwendet eine persistente Sidebar mit kollabierbarem Verhalten sowie einen konsistenten Hauptinhaltsbereich. Die Sidebar wird mithilfe einer `shadcn`-Komponente erstellt und unterstützt folgende Funktionen:

- Collapse/Expand-Toggle
- Benutzerbereich (derzeit nur als Platzhalter implementiert und inaktiv, da im Backend keine Benutzerverwaltungslogik implementiert ist)
- Theme-Toggle (Hell/Dunkel)
- Hervorhebung der aktiven Route

Zusätzlich wird auf den Hauptbereichsseiten eine Breadcrumb-Navigation mithilfe von `shadcn`-Breadcrumbs integriert, um einen klaren und nachvollziehbaren Navigationskontext zu bieten.

Ordnerstruktur Die Codebasis folgt einer klaren inhaltlichen Trennung der Ordner:

- `pages/`: Seiten auf Routing-Ebene mit Datenabruf und Anwendungslogik
- `lib/actions/`: API-Funktionen pro Domänenentität
- `components/`: domänenspezifische UI-Elemente (Tabellen, Formulare, Detailansichten (Tabs))
- `components/ui/`: wiederverwendbare UI-Elemente und `shadcn`-Komponenten

Diese Struktur bietet Wiederverwendbarkeit und erlaubt eine klare Fokussierung der Seiten auf die Anwendungslogik statt auf UI-Details.

6.4 Zustandsverwaltung und Datenfluss

Serverseitiger Zustand Das Dashboard verwendet einen serverseitigen Zustand für Entitätslisten und Detailansichten. Seiten rufen Daten über Aktionsfunktionen ab, die den Datenabruf und die Fehlerbehandlung standardisieren. Der Zustand wird lokal

in jeder Seitenkomponente mithilfe der React-Hooks `useState` und `useEffect` verwaltet. Zusätzlich wird mithilfe von `useCallback` eine stabile Wiederabruflogik gewährleistet.

Zustandsverwaltung über URL-Query-Parameter für Filter und Pagination Filterung, Sortierung und Pagination werden über URL-Query-Parameter mithilfe des React-Router-Hooks `useSearchParams` verwaltet. Dies ermöglicht:

- teilbare, reproduzierbare URLs
- zustandlose paginierte Listenkomponenten

Pagination und Auswahl der Seitengrösse passen die Query-Parameter an und lösen einen erneuten Datenabruf aus. Dieses Muster vermeidet eine globale Zustandsverwaltung sowie den vollständigen Abruf aller Daten, da der Listenzustand direkt aus den URL-Parametern und den Backend-Antworten rekonstruiert werden kann.

Umgang mit grossen tabellarischen Daten Tabellen werden mit Pagination und Filterung statt clientseitiger Virtualisierung dargestellt. Es werden sinnvolle Seitengrössen (Standardwert 20, konfigurierbar auf 10 oder 50) verwendet, die mit der serverseitigen Pagination kombiniert werden, um den Speicherverbrauch auf Client-Seite zu reduzieren. Dieses Muster verbessert die Skalierbarkeit bei grossen Datenmengen.

6.5 ML- und CRUD-Funktionen

Training Workflow Das Training wird durch die Train-Komponente initialisiert, die ein Sheet (Seitenpanel) mit einem validierten Formular öffnet. Das Formular unterstützt folgende Felder:

- Problem-ID (sofern diese nicht bereits aus den URL-Parametern übernommen wird)
- Modellname
- Auswahl des Train-Modus (fast, balanced, accurate)
- Auswahl des Algorithmus-Presets
- Evaluierungsstrategie (cv, holdout)
- Explainability-Toggle

Eine erfolgreiche Übermittlung löst eine Rückmeldung („toast“) aus und schliesst das Formular. Anschliessend wird die Modellliste aktualisiert, um das neu erzeugte Modell anzuzeigen.

Vorhersage Workflow Die Vorhersage verwendet eine ähnliche Sheet-Komponente. Sie unterstützt drei Eingabemodi für die Daten:

- Hochladen einer neuen CSV-Datei
- Verwendung einer existierenden CSV-Datei (derzeit inaktiv)
- Eingabe im JSON-Format

Das Formular wird mithilfe von Zod validiert, um sicherzustellen, dass eine Eingabequelle angegeben ist und eine Modell- oder Problem-ID vorhanden ist. Eine erfolgreiche Übermittlung löst ebenfalls eine Rückmeldung aus und aktualisiert die entsprechenden Datenansichten.

Jobstatus und Rückmeldung Das UI verwendet Toast-Rückmeldungen, um Ereignisse der Trainings- und Vorhersageprozesse darzustellen. Seiten abonnieren Job-Events und zeigen Rückmeldungen wie „Training finished successfully“oder „Prediction failed“, um unmittelbares Feedback für den Benutzer zu geben, ohne eine manuelle Aktualisierung erforderlich ist.

CRUD-Funktionen Die übrigen CRUD-Funktionen folgen einem ähnlichen Muster wie die Trainings- und Vorhersageformulare. Die Erstellung und Aktualisierung von Daten erfolgt ebenfalls über Sheet-Komponenten.

Besondere Bedeutung kommt den Löschfunktionen zu, die eine schriftliche Bestätigung durch den Benutzer erfordern, bevor der Vorgang ausgeführt wird.

Fehlerbehandlung Die API-Aktionen geben typisierte Erfolgs- oder Fehlerantworten zurück. Formulare interpretieren Validierungsfehler und zeigen diese mithilfe von `FieldError` direkt bei den entsprechenden Eingabefeldern an. Netzwerk- und Backend-Fehler werden über Toast-Rückmeldungen mit verständlichen Meldungen dargestellt.

6.6 Explainability Visualisierung

Explainability wird auf der Modell-Detailseite dargestellt. Das Frontend erwartet eine strukturierte Explainabilty-Zusammenfassung aus dem Backend, die folgende Informationen enthält:

- Merkmalswichtigkeiten (globale Mean Absolute SHAP-Werte)
- Aggregation auf Elternebene (Gruppierung nach ursprünglichen Spalten)
- Zusammenfassungen pro Klasse für Klassifikationsaufgaben

Zwei Visualisierungskomponenten behandeln die jeweiligen ML-Aufgabentypen:

- Regression: Ein Balkendiagramm mit einem Toggle zur Umschaltung zwischen gruppierter Ansicht (Aggregation auf Elternebene) und ungruppierter Ansicht (einzelne Merkmale).
- Klassifikation: Eine tab-basierte Klassenauswahl sowie derselbe Toggle zur Umschaltung zwi-

schen gruppierter und ungruppierter Ansicht pro Klasse.

Die Diagramme werden mithilfe von Recharts dargestellt. Dabei werden Tailwind-Variablen verwendet, um ein themenkonsistentes Styling zu gewährleisten. Dieses Muster wird aus folgenden Gründen gewählt:

- Gruppierte Ansichten reduzieren visuelles Rauschen bei kategorialen Merkmalen mit hoher Kardinalität.
- Klassenbasierte Tabs vermeiden visuelles Rauschen bei Multi-Klassen-Aufgaben.

6.7 Echtzeit-Aktualisierung

Das Frontend verwendet Server-Sent Events („SSE“) mithilfe von EventSource, um Job-Aktualisierungen vom Endpoint `/events/stream` zu empfangen.

Seiten, die Modelle oder Vorhersagen darstellen, abonnieren die Events `job.completed` und `job.failed`. Diese werden nach Job-Typ und relevanten Identifikatoren gefiltert, bevor eine Datenaktualisierung ausgelöst wird.

Die Event-Handlers werden innerhalb von `useEffect` registriert und bei Komponenten-Unmouut wieder entfernt, um Memory-Leaks zu vermeiden.

6.8 Design und visuelle Konsistenz

Das UI wird grundsätzlich mithilfe einer konsistenten Komponentenbasis aus `shadcn` erstellt. Tailwind-Utility-Klassen implementieren Layout, Abstände und Typografie mit einem einheitlichen und vorhersehbaren Verhalten.

Das Design von Sidebar, Breadcrumb-Navigation, Navigationsstruktur und Überschriften folgt einem wiederkehrenden Stil. Filter- und Aktionsschaltflächen und Eingabefelder sind auf den Listenseiten konsistent positioniert.

Zusätzlich bieten Leerzustandsseiten verständliche Rückmeldungen für den Benutzer, inkl. klarer Handlungsaufforderungen (sofern möglich). Das Seitenlayout ist durchgehend einheitlich aufgebaut: eine Hauptliste bildet den Hauptinhalt der Seite, während Tabs zugehörige Unterbereiche oder zusätzliche Details strukturieren.

Der `ThemeProvider` aus `shadcn` ermöglicht einen Hell-/Dunkelmodus und speichert die Auswahl im lokalen Speicher. Das Theme steuert sowohl Tailwind-Variablen als auch Diagrammfarben, um ein konsistenes Aussehen des gesamten UI und der Visualisierungen sicherzustellen.

7 Perspektive

Dieser Abschnitt beschreibt Erweiterungen, die aus dem aktuellen Design naheliegend folgen. Das sind Punkte, die im Projektumfang sinnvoll gewesen wären, aber zeitlich nicht mehr umgesetzt wurden.

7.1 CSV-Daten direkt in der Datenbank speichern

Im aktuellen Setup bleiben Uploads als Dateien erhalten. Die Datenbank speichert dazu URIs und Metadaten. Eine Alternative ist, die Inhalte eines Datasets zusätzlich in relationale Tabellen zu laden.

Der größte Vorteil ist die Abfragbarkeit. Daten lassen sich direkt mit SQL filtern und aggregieren. Das hilft bei Dataset-Exploration, Validierungsregeln oder schnellen Preview-Abfragen. Auf häufig genutzten Spalten lassen sich Indizes anlegen, was typische Filter deutlich beschleunigt. Für das Importieren größerer Datenmengen bietet MySQL außerdem Bulk-Load-Funktionen wie LOAD DATA [8].

Ein zweiter Vorteil sind stärkere Integritätsprüfungen. Spaltentypen, NOT NULL-Constraints und einfache Checks reduzieren Unklarheiten, bevor die Daten in die ML-Pipeline gehen.Gleichzeitig steigt der Aufwand für Ingestion, Backups und Schema-Management, vor allem wenn Datasets sehr groß oder sehr unterschiedlich strukturiert sind.

7.2 Authentication und Authorization

`PAaaS` speichert Nutzer und Ownership. Wenn die Plattform von mehreren Nutzern geteilt wird, wird Zugriffskontrolle wichtiger. Drei gängige Ansätze sind dafür relevant:

- Basic Authentication: Der Client sendet Benutzername und Passwort bei jedem Request im HTTP-Header. Das Verfahren ist einfach und weit verbreitet. Passwörter werden dabei zum zentralen Sicherheitsfaktor und sollten nur zusammen mit Transportverschlüsselung eingesetzt werden [18].
- Token-basierte Authentifizierung mit JWT: Nach dem Login wird ein signiertes Token ausgegeben. Der Client sendet das Token bei Requests, und der Server prüft die Signatur. JWT ist standardisiert und wird häufig für stateless APIs genutzt [19].
- OAuth 2.0 Authorization Flows: OAuth 2.0 wird genutzt, wenn Authentifizierung und Autorisierung an einen Identity Provider ausgelagert werden. Es definiert Flows zur Ausgabe von Access Tokens und zur Verwaltung delegierter Zugriffe. Das ist hilfreich, wenn externe Login-Anbieter

genutzt werden oder mehrere Services eine gemeinsame Auth-Schicht teilen [20].

Neben der Authentifizierung braucht es Autorisierung, also die Frage, was ein Nutzer darf. Für `PAaaS` beginnt das bei Dataset-Rechten und einfachen Rollen, etwa read-only versus write. Das ließe sich durch Rollen-Zuordnungen und Access Rules ergänzen, die auf `users` und `datasets` referenzieren.

7.3 MLCore

In MLCore sind besonders folgende Punkte als mögliche Erweiterungen und Ausbaumöglichkeiten zu betrachten:

- Einführung eines kontinuierlichen Modell-Monitorings zur Überwachung von Datenverteilungen, Vorhersageverhalten und Leistungsmetriken in der Produktionsumgebung.
- Erweiterung der Presets und "Auto"-Presets um systematische Hyperparameteroptimierung, z.B. durch GridSearchCV, RandomizedSearchCV oder Optuna.
- Einführung von speicher- und rechenoptimierten Trainingsverfahren, z.B. durch Batch-Verarbeitung oder GPU-Unterstützung zur Verbesserung der Skalierbarkeit.

7.4 Frontend

Auch im Bereich des Frontends / der WebUI sind Verbesserungsmöglichkeiten vorhanden:

- Jobs-Seite: Die Jobs-Seite ist derzeit lediglich ein Platzhalter und bietet keine detaillierten Echtzeit-Ansichten der Hintergrundprozessen.
- Auswahl existierender CSV-Dateien: Die Formulare zur Erstellung von Datensatz-Versionen sowie zur Vorhersage enthalten einen Modus zur Verwendung einer existierenden CSV-Datei. Dieser ist aktuell deaktiviert, da noch keine vollständige Unterstützung für gespeicherte CSV-Dateien implementiert ist.
- Explainability-Diagramme: Wie bereits im MLCore-Explainability-Kapitel erwähnt, waren zusätzliche Visualisierungen (z.B. Quantil-Zusammenfassungen) vorgesehen. Diese wurden aufgrund zeitlicher Einschränkungen nicht umgesetzt, und sind eine mögliche Erweiterung zur vertieften Analyse.

Anmerkung zur Nutzung von KI-Tools

In diesem Bericht wurden KI-Tools („ChatGPT“) zur Fehlerkorrektur und sprachlichen Ausgestaltung verwendet. „GPT-5.2 Thinking“wurde zur Unterstützung bei Formulierungen, Struktur und Grammatik eingesetzt. Der Projekt-Code verwendet in Teilen Code-Snippets, bei denen eine KI-Verwendung nicht ausgeschlossen werden kann (zum Beispiel durch in Suchmaschine eingebundene Tools wie „Google Search“, „Microsoft Copilot“, „duck.ai“)

Abbildungsverzeichnis

Literatur

[1] Michael Sollfrank u. a. “Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation”. In: *IEEE Transactions on Industrial Informatics* 17.5 (Mai 2021), S. 3566–3576. ISSN: 1941-0050. DOI: 10.1109/TII.2020.3022843. URL: https://ieeexplore.ieee.org/abstract/document/9187833 (besucht am 01. 02. 2026).

[2] *DB-Engines Ranking*. DB-Engines. URL: https://db-engines.com/en/ranking (besucht am 25. 06. 2023).

[3] *Document Stores - DB-Engines Encyclopedia*. URL: https://db-engines.com/en/article/Document+Stores (besucht am 28. 06. 2023).

[4] David Reis u. a. “Developing Docker and Docker-Compose Specifications: A Developers' Survey”. In: *IEEE Access* 10 (2022), S. 2318–2329. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3137671.

[5] *Understanding the Image Layers*. Docker Documentation. 4:29:50 +0530 +0530. URL: https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/ (besucht am 03. 02. 2026).

[6] *Docker Hub Container Image Library | App Containerization*. URL: https://hub.docker.com (besucht am 03. 02. 2026).

[7] *Docker Image Save*. Docker Documentation. 11:23:46 +0100 +0100. URL: https://docs.docker.com/reference/cli/docker/image/save/ (besucht am 03. 02. 2026).

[8] *MySQL 8.0 Reference Manual*. URL: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/ (besucht am 23. 02. 2026).

[9] Paul J. Leach, Rich Salz und Michael H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. Request for Comments RFC 4122. Internet Engineering Task Force, Juli 2005. 32 S. DOI: 10.17487/RFC4122. URL: https://datatracker.ietf.org/doc/rfc4122 (besucht am 23. 02. 2026).

[10] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Internet Engineering Task Force, Dez. 2017. 16 S. DOI: 10.17487/RFC8259. URL: https://datatracker.ietf.org/doc/rfc8259 (besucht am 23. 02. 2026).

[11] *13.5 The JSON Data Type*. URL: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/json.html (besucht am 23. 02. 2026).

[12] Tim Berners-Lee, Roy T. Fielding und Larry M. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Request for Comments RFC 3986. Internet Engineering Task Force, Jan. 2005. 61 S. DOI: 10.17487/RFC3986. URL: https://datatracker.ietf.org/doc/rfc3986 (besucht am 23. 02. 2026).

[13] *17.7.2 InnoDB Transaction Model*. URL: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/innodb-transaction-model.html (besucht am 23. 02. 2026).

[14] *PEP 249 – Python Database API Specification v2.0 | Peps.Python.Org*. Python Enhancement Proposals (PEPs). URL: https://peps.python.org/pep-0249/ (besucht am 23. 02. 2026).

[15] Fabian Pedregosa u. a. “Scikit-Learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12 (null 1. Nov. 2011), S. 2825–2830. ISSN: 1532-4435. URL: https://dl.acm.org/doi/10.5555/1953048.2078195 (besucht am 18. 02. 2026).

[16] Scott M. Lundberg und Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 4. Dez. 2017, S. 4768–4777. ISBN: 978-1-5108-6096-4. URL: https://dl.acm.org/doi/10.5555/3295222.3295230 (besucht am 18. 02. 2026).

[17] Tianqi Chen und Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. New York, NY, USA: Association for Computing Machinery, 13. Aug. 2016, S. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: https://dl.acm.org/doi/10.1145/2939672.2939785 (besucht am 18. 02. 2026).

[18] Julian Reschke. *The 'Basic' HTTP Authentication Scheme*. Request for Comments RFC 7617. Internet Engineering Task Force, Sep. 2015. 15 S. DOI: 10.17487/RFC7617. URL: https://datatracker.ietf.org/doc/rfc7617 (besucht am 23. 02. 2026).

[19] Michael B. Jones, John Bradley und Nat Sakimura. *JSON Web Token (JWT)*. Request for Comments RFC 7519. Internet Engineering Task Force, Mai 2015. 30 S. DOI: 10.17487/RFC7519. URL: https://datatracker.ietf.org/doc/rfc7519 (besucht am 23. 02. 2026).

[20] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Request for Comments RFC 6749. Internet Engineering Task Force, Okt. 2012. 76 S. DOI: 10.17487/RFC6749. URL: https://datatracker.ietf.org/doc/rfc6749 (besucht am 23. 02. 2026).