# Technische Übersicht

Das Projekt **Arzttarif-Assistent** automatisiert die Analyse von ärztlichen Behandlungs­texten anhand zweier LLM-Stufen und anschliessender Regelprüfungen. Die Anwendung ist als Flask-Server strukturiert und lädt beim Start umfangreiche Katalog‑ und Regel­daten (load\_data). Über den Endpunkt /api/analyze-billing werden Nutzer­eingaben verarbeitet und darauf basierend **Einzelleistungen** (TARDOC) sowie **Pauschalen** ermittelt.

# Hauptprozesse

## Vorverarbeitung & Kontextaufbau

1. Eingaben (inputText, optionale ICD/GTIN, Alter, Geschlecht) werden bereinigt.
2. **Synonymkatalog** (falls aktiviert) erweitert Suchbegriffe; sonst werden nur Rohbegriffe verwendet.
3. **RAG / Embeddings** (falls aktiviert) ranken Katalogeinträge; ohne RAG erfolgt Token-Frequency-Ranking.
4. Durch expand\_compound\_words und extract\_keywords werden Suchterm‑Varianten erzeugt, ggf. unter Nutzung des Synonymkatalogs.

## LLM-Stufe 1 – Identifikation von LKN

* Funktion: call\_gemini\_stage1
* Prompt: get\_stage1\_prompt
* Eingabe: aufbereiteter Katalogauszug + Nutzereingabe
* Ausgabe: JSON mit
  + identified\_leistungen (LKN, Typ, Menge)
  + extracted\_info (Dauer, Alter usw.)
  + begruendung\_llm (Begründung)
* Zweck: Erkennen aller relevanten Leistungspositionen

### Vorbereitung der Eingabe

Vor dem LLM‑Aufruf wird der Behandlungstext linguistisch aufbereitet. Dazu zählen:

* Aufsplitten zusammengesetzter Wörter (expand\_compound\_words)
* Schlüsselwort-Extraktion und optionales Synonym‑Mapping
* Sammeln explizit erwähnter LKN‑Codes, um den Kontext nicht leer zu lassen

Diese Schritte erzeugen sowohl einen bereinigten Behandlungstext als auch eine Token‑Menge, die als Suchanker für den Katalog dient.

### Aufbau des Katalogkontextes

Aus den gerankten LKNs wird Zeile für Zeile ein Kontextstring generiert:

LKN: <Code>, Typ: <Typ>, Beschreibung: <Beschreibung>, MedizinischeInterpretation: <Synonyme…>

Dieser String wird später in den Prompt eingefügt; seine Länge wird über eine naive Token‑Funktion protokolliert

### HTTP‑Payload & Modelleinstellungen

call\_gemini\_stage1 setzt den vollständigen Prompt als text in der Google‑Gemini‑API ein. Wichtige Felder:

* contents[0].parts[0].text – der prompt (Anweisungen + Katalogkontext + Behandlungstext)
* generationConfig:
  + response\_mime\_type = application/json
  + temperature = 0.05
  + maxOutputTokens = 65536  
    So wird ein deterministisches, rein JSON‑basiertes Ergebnis angestrebt

### Promptstruktur und Zweck

Der Prompt (deutsche Variante) besteht aus:

1. Aufgabenbeschreibung und Katalogkontext
2. Sechs schrittweise Anweisungen (Analyse, LKN‑Identifikation, Mengenregeln, Validierung, Kontext‑Extraktion, JSON‑Ausgabe)
3. Vorgabe eines exakten JSON‑Schemas mit Feldern identified\_leistungen, extracted\_info und begruendung\_llm
4. Wiederholung des Behandlungstextes

Diese Struktur zwingt das Modell, jede Tätigkeit zu finden, Mengen zu berechnen und die Resultate strikt als JSON zurückzugeben

### Rückgabestruktur (Felddefinitionen)

Die Antwort wird validiert und normalisiert:

* identified\_leistungen: Liste von Objekten mit lkn (string), typ (string), beschreibung (string), menge (int)
* extracted\_info:
  + dauer\_minuten, menge\_allgemein, alter, anzahl\_prozeduren (int | null)
  + geschlecht, seitigkeit (str | null; Default seitigkeit="unbekannt")
* begruendung\_llm: kurzer Erklärungstext

Fehlende oder falsch typisierte Werte werden auf Defaults gesetzt oder konvertiert, um Konsistenz zu gewährleisten

### Tokenumfang & Limitationen

* **Prompt-Instruktionen**: ca. 700–1 000 Tokens (abhängig von Sprache).
* **Katalogkontext**: bis zu 500 LKN-Zeilen à ~20–40 Tokens ⇒ typischerweise 10 000–20 000 Tokens.
* **Behandlungstext**: meist < 200 Tokens.

Die Tokenzählung nutzt eine Regex-basierte Heuristik (count\_tokens); tatsächliche Modell-Tokens können abweichen.  
**Limitationen:**

* Bei sehr grossen Kontexten droht Zeit- oder Kostenanstieg; es existiert kein hartes Limit ausser der Modellgrenze.
* Die naive Tokenzählung kann Fehleinschätzungen verursachen (Über-/Unterschätzung).
* Gemischte Sprachen und umfangreiche Anweisungen erhöhen die Komplexität; das Modell kann dennoch inkonsistente oder nicht-parsbare JSONs liefern, weshalb zusätzliche Validierung nötig ist.

### Optimierungsansätze

* Kontextbeschränkung oder dynamisches Trimmen, um Tokenkosten zu reduzieren.
* Exaktere Tokenberechnung (modell-spezifische Tokenizer).
* Modularere Aufbereitung des Prompttextes, um Wartung und Tests zu vereinfachen.
* Frühzeitige JSON-Validierung durch Schema oder Function Calling, um Parserfehler zu minimieren.

## Regelprüfung Einzelleistungen

* Modul: regelpruefer\_einzelleistungen.pruefe\_abrechnungsfaehigkeit
* Prüflogik:
  + Mengenbeschränkung (REGEL\_MENGE)
  + Kumulations‑/Exklusivregeln
  + Patienten­bedingungen (Alter, Geschlecht, GTIN)
  + Diagnosepflicht (ICD)
  + Pauschalenausschlüsse
* Nicht bestandene Regeln führen zu Fehlermeldungen; Mengen werden ggf. reduziert.
* Ausgabe: Liste regelkonformer LKN (rule\_checked\_leistungen\_list)

## Pauschalenprozess

### LKN-Mapping (LLM-Stufe 2)

* Nur für regelkonforme LKN des Typs **E/EZ**.
* Funktion: call\_gemini\_stage2\_mapping
* Kandidaten-LKNs stammen aus:
  + PAUSCHALEN\_Leistungspositionen.json
  + Bedingungen (PAUSCHALEN\_Bedingungen.json)
  + Tabelleninhalte (PAUSCHALEN\_Tabellen.json)
* Ziel: Mapping auf LKN, die Bedingungen einer Pauschale erfüllen könnten.

### Kandidatensuche & Ranking

* Potentielle Pauschalencodes werden aus LKN-Zuordnungen und Regelbedingungen gesammelt.
* LLM-Stufe 2 (call\_gemini\_stage2\_ranking) kann die Kandidaten gewichten und sortieren.
* Ergebnis: Liste ranked\_codes (beste Pauschalen zuerst)

### Strukturierte Bedingungen & Hauptprüfung

* Hauptlogik in regelpruefer\_pauschale:
  + check\_pauschale\_conditions / evaluate\_pauschale\_logic\_orchestrator
  + check\_single\_condition prüft atomare Bedingungen (ICD, GTIN, LKN, Patienten­bedingungen, Seitigkeit, Anzahl etc.)
  + Gruppenlogik (UND/ODER) möglich.
* determine\_applicable\_pauschale wählt die bestbewertete, regelkonforme Pauschale:
  + Bedingungsergebnisse werden HTML-basiert dokumentiert.
  + Score: v.a. Taxpunkte, alternativ LLM-Ranking.
* TARDOC-Abrechnung (prepare\_tardoc\_abrechnung) sammelt abrechenbare Einzelleistungen.

## Ergebnisaufbau

* Response enthält:
  + ggf. gefundene Pauschale samt Regel‑HTML
  + Liste regelkonformer Einzelleistungen
  + Zeitmessung, Loginformationen
* Feedback-Endpunkte zur Qualitätssicherung sind vorhanden.

**Betriebsmodi: Synonymkatalog & RAG**

| **Merkmal** | **Mit Synonymkatalog / RAG** | **Ohne Synonymkatalog / RAG** |
| --- | --- | --- |
| Suchterm-Expansion | expand\_query nutzt SynonymCatalog (mehr Treffer, Gefahr von Ausreissern) | Nur originaler Nutzertext |
| Ranking | Embedding-basierte Ähnlichkeit (SentenceTransformer) | Token-Frequenz-basierter Vergleich |
| Fehlerquellen | Falsche Synonyme, mangelnde Abdeckung | Geringere Treffergenauigkeit |
| Datenbedarf | synonyms.json + Embedding-Datei | Nur Leistungs­katalog |

**Umgang mit Pauschalen vs. Einzelleistungen**

* **Einzelleistungen**: LLM-Stufe 1 findet LKN, Regel­prüfung reduziert/eliminiert unzulässige Leistungen. Für TARDOC-Leistungen wird anschliessend eine Abrechnungsliste erstellt.
* **Pauschalen**:
  1. Ermittlung möglicher Pauschalencodes anhand von LKN & Bedingungen
  2. Optionales LLM-Mapping zusätzlicher LKN
  3. Strukturierte Bedingungs­prüfung
  4. Optionales LLM-Ranking zur Priorisierung
  5. Ergebnis enthält detaillierte HTML-Darstellung der erfüllten/fehlenden Bedingungen.