



WHZ Westsächsische
Hochschule Zwickau
University of Applied Sciences

Fakultät Physikalische Technik/Informatik
Studiengang Data Science

BACHELORARBEIT

Konzeption und Implementierung eines KI-gestützten Support-Agenten auf Basis von Retrieval-Augmented Generation

*Design and Implementation of an AI-Powered Support Agent Based on
Retrieval-Augmented Generation*

Tymur Abkiliamov

Matrikelnummer: 48080

Kennnummer: 222250

Erstgutachter:

Prof. Dr. rer. nat. Mike Espig

Betreuerin von Seiten des Unternehmens: Dr. Aida Farahani

Abgabedatum: 17.06.2026

Kontakt Daten	Telefonnummer: +49 151 41494264
Anschrift: Bauernweg 1	E-Mail: tymur.abkiliamov.mjy@whz.de;
09366 Stollberg/Erzgeb.,	timurabkiliamov@gmail.com
Sachsen	

Aufgabenstellung

Die zunehmende Komplexität von Softwareprodukten und der steigende Bedarf an schnell, qualitativ hochwertigem Kundensupport erfordern innovative Lösungen zur Automatisierung von Support-Prozessen.

Moderne Sprachmodelle (Large Language Models) bieten in Kombination mit Retrieval-Augmented Generation (RAG) die Möglichkeit, Support-Anfragen automatisiert und kontextbezogen zu beantworten, indem relevante Informationen aus bestehenden Wissensdatenbanken abgerufen werden.

Die Aufgabe dieser Arbeit ist die Entwicklung eines KI-gestützten Support-Agenten als Webanwendung. Dabei sollen Daten aus verschiedenen Quellen (Zendesk, Zendesk Help Center, Confluence, Jira, Slack, PDF-Dokumentationen) in einer semantischen Suchstruktur zusammengeführt und über eine React-basierte Benutzeroberfläche zugänglich gemacht werden.

Im Rahmen dieser Arbeit sind die folgenden Punkte zu erarbeiten:

- Literaturrecherche zu RAG-Systemen und LLM-gestützten Support-Lösungen
- Konzeption der Systemarchitektur (Backend, Frontend, Datenquellen)
- Implementierung der Datenextraktion aus Zendesk, Confluence, Jira und Slack
- Aufbau einer semantischen Suchstruktur mittels FAISS und Sentence-Transformers
- Entwicklung eines React-Frontends mit Anbindung an das Backend
- Integration und Vergleich verschiedener lokaler LLM-Modelle
- Evaluation des Systems anhand realer Support-Anfragen
- Auswertung und weiterführende Fragestellungen

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VI
Kurzzeichenverzeichnis	VIII
Abkürzungsverzeichnis	IX
Einleitung	1
Motivation	1
Zielsetzung	2
Aufbau der Arbeit	2
1. Grundlagen	4
1.1. Natural Language Processing	4
1.1.1. Tokenisierung und Subword-Einheiten	4
1.2. Klassisches und dichtes Information Retrieval	5
1.2.1. Klassische Verfahren: TF-IDF und BM25	5
1.2.2. Vektorräume, Kosinusähnlichkeit und dichtes Retrieval	5
1.3. Large Language Models	6
1.3.1. Transformer-Architektur	6
1.3.2. Autoregressive Generierung	7
1.3.3. Decoding-Strategien	7
1.3.4. Few-Shot Learning und In-Context Learning	8
1.3.5. Self-Consistency	8
1.3.6. Quantisierung	9
1.3.7. Grammar-constrainer Generierung mit GBNF	10
1.4. Retrieval-Augmented Generation	11
1.4.1. RAG-Pipeline	11
1.4.2. Embedding-Modelle	11
1.4.3. Vektorindexierung mit FAISS	12
1.5. Zweistufiges Retrieval und Reranking	12
1.5.1. Bi-Encoder vs. Cross-Encoder	13
1.5.2. Maximal Marginal Relevance	13
1.6. Klassifikation und Evaluationsmetriken	14
1.6.1. Konfusionsmatrix	14
1.6.2. Accuracy, Precision, Recall, F1	14
1.6.3. Annotation und Ground Truth	15
1.7. Webtechnologien	15

2. Stand der Technik	17
2.1. Aktuelle Praxis im technischen Kundensupport	17
2.1.1. Kommerzielle Support-Plattformen mit integrierter KI	17
2.1.2. Atlassian Roovo als unternehmensweiter Wissensassistent	20
2.1.3. ChatGPT Custom GPTs als geteilte Wissensagenten	22
2.1.4. Zwischenfazit: Werkzeugfragmentierung als Kernproblem	23
2.2. Open-Source-Frameworks für RAG-Pipelines	24
2.3. Aktuelle RAG-Forschung jenseits des Standardansatzes	24
2.4. Abgrenzung und Positionierung	26
3. Konzept und Architektur	30
3.1. Anforderungsanalyse	30
3.1.1. Funktionale Anforderungen	30
3.1.2. Nicht-funktionale Anforderungen	31
3.2. Systemarchitektur	31
3.3. Datenquellen	33
3.3.1. Zendesk	33
3.3.2. Confluence – Analyse und bewusste Nicht-Implementierung	50
3.3.3. Jira – Analyse und bewusste Nicht-Implementierung	51
3.3.4. Slack – Analyse und bewusste Nicht-Implementierung	52
3.3.5. Zendesk Help Center TKI (HC)	53
3.3.6. Externe Wissensquellen: Vitruvi	56
3.3.7. TKI-Produkte-Wissensbasis (TKI-Pr-KB)	67
3.4. Datenverarbeitungspipeline	69
3.5. Semantische Suche	70
3.6. LLM-Integration und Antwortgenerierung	71
3.7. Konfidenzbewertung	71
4. Implementierung	73
4.1. Projektstruktur	73
4.2. Backend	74
4.2.1. Zentrale Konfiguration	74
4.2.2. Zendesk-Pipeline	75
4.2.3. PDF-Verarbeitung (shared)	84
4.2.4. Externe Vitruvi-Pipeline	84
4.2.5. TKI-Pr-KB-Pipeline	88
4.2.6. HC-Pipeline	90
4.2.7. FAISS-Index und Embedding-Pipeline	92
4.2.8. Cross-Encoder Reranking	93
4.2.9. FastAPI-Server und Streaming-Pipeline	93
4.2.10. Queue-Management	94
4.3. Antwortpfad und Benutzerführung	94
4.3.1. Erste Ansicht: Arbeitsbereich beim Aufruf	94
4.3.2. Einstellungen und Personalisierung	96
4.3.3. Beispielhafte Antwort und Ergebnisdarstellung	98
4.3.4. Live-Phasen während der Anfrage	100

4.3.5. Quellenrückführung über Zitationsanker	101
4.3.6. Editierbarkeit und Verantwortungskette	102
4.3.7. Diagnose-Panel	102
4.3.8. Produktfilterung	103
4.4. Funktionale Korrektheit des Endprodukts	103
4.4.1. Iterative Verfeinerung des Antwortpfads	103
4.4.2. Exemplarische Anfragen und Antworten	105
4.4.3. Latenz und Reaktivität	106
4.4.4. Komponententests und Validierungsinfrastruktur	113
4.5. Pro-Pipeline-Hilfsfunktionen	115
5. Ergebnisse und Evaluation	117
5.1. Testdatensatz und Methodik	117
5.1.1. RAGAS-Metriken	117
5.2. Qualitätsfilterung der Zendesk-Tickets	118
5.2.1. Datensätze und Test-Set-Hierarchie	118
5.2.2. Vergleich mit regelbasierten Baselines	120
5.2.3. Iterationsverlauf	122
5.2.4. Quantisierungseffekt Q4_K_M → Q5_K_M	125
5.2.5. Baseline – Qwen 2.5 32B	127
5.2.6. Modellvergleich	127
5.2.7. v2-Versuch: synthetisch kalibrierte Overrides	128
5.2.8. v3: realdaten-kalibrierte Overrides und Ergebnisse	130
5.2.9. Methodische Erkenntnis: leere applied_overrides	133
5.2.10. Wechsel zurück zu Llama 3.1 8B Q5_K_M	135
5.2.11. Vergleich aller eingesetzten Konfigurationen	137
5.2.12. Externe Validierung durch Frontier-LLMs via VS Code Copilot	139
5.2.13. Statistische Robustheitsanalyse	142
5.2.14. Fehleranalyse	147
5.2.15. Diskussion	148
5.3. RAG-Datenunit-Extraktion	148
5.3.1. Quality-Gate auf Golden-23	148
5.3.2. Produktivlauf über 5 411 USEFUL-Tickets	149
5.3.3. Quality-Weighted Retrieval – Übergang in die Suche	152
5.4. Qualität der semantischen Suche	153
5.4.1. Smoke-Test mit kanonischen Anfragen	154
5.4.2. Index-Eigenschaften und Latenz	154
5.4.3. Qualitative Beobachtungen aus den Smoke-Tests	155
5.5. Qualität der generierten Antworten	156
5.5.1. Test-Set-Konstruktion	156
5.5.2. Judge-Konfiguration	157
5.5.3. Interpretationsschwellen	158
5.5.4. RAGAS-Ergebnistabelle für die finale Konfiguration	158
5.5.5. Quellbezogene Diskussion	159
5.5.6. Bezug zu FA-2 und FA-5	160
5.5.7. Limitationen der RAGAS-Evaluation	161

5.6. Systemperformance	161
5.7. Vergleich verschiedener Konfigurationen	163
5.8. Diskussion	163
5.8.1. Stärken des Systems	163
5.9. Limitationen und Threats to Validity	164
5.9.1. Interne Validität	164
5.9.2. Externe Validität	165
5.9.3. Konstrukt-Validität	166
5.9.4. Statistische Konklusionsvalidität	166
5.9.5. Operative Einschränkungen	167
6. Zusammenfassung und Ausblick	169
6.1. Zusammenfassung	169
6.2. Ausblick	170
Literaturverzeichnis	172
7. Übersicht verwendeter Hilfsmittel	178
A. Anhang	180
A.1. Technologieübersicht	180
A.2. API-Endpunkte	181
A.3. Umgebungsvariablen	182
A.4. Per-Ticket-Vergleichsmatrix Golden 45	182
A.5. Externe Frontier-LLM-Validierung – Vergleichsmatrix Golden 45	182
A.6. Kanonische Produktliste	184
A.7. Vollständige Quellcode-Listings zum Implementierungskapitel	184
A.7.1. Zendesk-Ticket-Extraktion mit Cursor-Paginierung und Checkpointing	186
A.7.2. Deterministische Pre-Filter-Regeln	187
A.7.3. Override P4 – abandoned → solved bei kundenseitiger Auflösung . .	188
A.7.4. Self-Consistency-Aggregation	188
A.7.5. LLM-Initialisierung für Qwen 2.5 32B	189
A.7.6. SSE-Streaming-Pipeline	189
A.7.7. Thin Wrapper über die produktive RAG-Pipeline (RAGAS-Evaluation)	190
A.7.8. Lokaler RAGAS-Judge mit Llama-3.1-Chat-Template	192
A.8. RAGAS-Test-Set – repräsentative Beispiele	193

Abbildungsverzeichnis

2.1.1.	Zendesk AI Agent	18
2.1.2.	Intercom Fin in der Live-Chat-Oberfläche	19
2.1.3.	Freshdesk Freddy AI	19
2.1.4.	Salesforce Einstein for Service	20
2.1.5.	Atlassian Rovo: integrierte Such- und Chat-Oberfläche im Atlassian-Stack .	21
2.1.6.	Geteilte Custom GPTs in einem Unternehmens-Workspace	23
3.2.1.	Übersicht der Systemarchitektur	32
3.3.1.	Architektur der Qualitätsfilterung der Zendesk-Tickets	36
3.3.2.	Streamlit-Annotationsanwendung – Kopfbereich mit globaler Statistik	41
3.3.3.	Konversationsansicht mit rollen- und sichtbarkeitsbasierter Farbkodierung .	42
3.3.4.	Annotationsbuttons USEFUL / NOT USEFUL	43
3.3.5.	Architektur der RAG-Datenunit-Extraktionspipeline	45
3.3.6.	Confluence <i>Software Knowledge Base</i> – interne Sondierungsansicht	51
3.3.7.	Jira-Projekt <i>Support</i> – interne Sondierungsansicht	52
3.3.8.	Slack-Workspace – beispielhafte Kanaldarstellung	53
3.3.9.	TKI Help Center – öffentliche Einstiegsseite	54
3.3.10.	Aufteilung des Vitruvi Developer-Portals in zwei Sektoren	57
3.3.11.	Titelseite der Vitruvi Customer Terms of Service v4.0	58
3.3.12.	Startseite des Vitruvi Help Center	59
3.3.13.	Sektions-Übersicht der Vitruvi Knowledge Base	61
3.3.14.	Manuelle Verifikation der Antwort zu Frage 1	64
3.3.15.	Manuelle Verifikation der Antwort zu Frage 2	65
3.3.16.	Manuelle Verifikation der Antwort zu Frage 3	66
4.3.1.	Arbeitsbereich beim ersten Aufruf	95
4.3.2.	Einstellungs-Panel mit Modell- und Retrieval-Parametern	97
4.3.3.	Beispielhafte Antwortdarstellung	99
4.3.4.	Live-Phasen der Antwortpipeline	100
4.4.1.	Antwort auf eine prozedurale HC-Anfrage	107
4.4.2.	Anfrage mit mittlerer Konfidenz und sichtbarem Diagnose-Panel	108
4.4.3.	Mehrschrittige Workflow-Antwort	109
4.4.4.	Quellenübergreifende Vitruvi-Anfrage	110
4.4.5.	Übersichts-Anfrage über die TKI-Pr-KB	111
4.4.6.	Definitorische Vitruvi-ToS-Anfrage mit Diagnose-Panel	112

Tabellenverzeichnis

1.1.	Konfusionsmatrix für eine binäre Klassifikation	14
2.1.	Konkrete Support- und Wissens-Systeme im Vergleich zum eigenen Ansatz .	27
2.2.	Erweiterte RAG-Varianten im Vergleich zum eigenen Ansatz	28
3.1.	API-Endpunkte des Backends	33
3.2.	Outcome-Buckets der Qualitätsfilterung	37
3.3.	Confidence-Overrides der Qualitätsfilterung	39
3.4.	Felder eines RAG-Datenunits (Stage-2-Output)	47
3.5.	Quality-Flag-Werte des RAG-Datenunit-Schemas	49
3.6.	Quality-Gate-Schwellen vor dem Full-Run	50
3.7.	HC-Korpus nach Kategorien	55
3.8.	Verteilung des Vitruvi Help Center Korpus	60
3.9.	Bild-Strategien für das Vitruvi Help Center	62
3.10.	Übersicht der externen Vitruvi-Wissensquellen	68
4.1.	Modul-Aufteilung der RAG-Extraktionspipeline	82
4.2.	Vergleich der drei Chunker für externe Vitruvi-Quellen	86
4.3.	Individuell justierbare Parameter im Einstellungs-Panel	98
4.4.	Semantische Zitations-Tags je Quelltyp	101
4.5.	Exemplarische Anfragen mit Konfidenz und Bewertung	106
4.6.	Typische Phasen-Latenzen unter Server-Konfiguration	113
5.1.	Datensätze für die Evaluation der Qualitätsfilterung	119
5.2.	Hardware-Konfigurationen während der Experimente	119
5.3.	Vergleich regelbasierter Baselines mit dem LLM-Klassifikator	121
5.4.	Iterationsverlauf der Qualitätsfilterung (alle reproduzierbaren Commits)	123
5.5.	Isolierter Quantisierungseffekt Q4_K_M $\rightarrow$ Q5_K_M auf Golden 45	125
5.6.	Votum-Verteilung der Self-Consistency ($N = 3$, Llama 8B Q4_K_M) auf Golden 45.	126
5.7.	Baseline-Ergebnisse Qwen 2.5 32B Q4_K_M (Single-Pass, $T = 0, 0$)	127
5.8.	Bucket-Verteilung der Modellvorhersagen (Qwen 2.5 32B Baseline)	128
5.9.	Modellvergleich – alle eingesetzten Konfigurationen	129
5.10.	Vergleich Baseline (v1) vs. Versuch v2 auf Golden 45	130
5.11.	Tickets mit unterschiedlicher v1/v2-Klassifikation	130
5.12.	Falsch-negative Tickets der Baseline – Gruppierung nach Rückgewinnbarkeit	131
5.13.	v3 Konfusionsmatrizen (Qwen 2.5 32B, Single-Pass)	132
5.14.	Vergleich v1 Baseline vs. v3 (Qwen 2.5 32B)	132
5.15.	Per-Ticket-Flips Set 200, v1 $\rightarrow$ v3	132

5.16.	Bucket-Verteilung v1 $\rightarrow$ v3	133
5.17.	Override-Aktivierungen während v3 (alle Läufe)	134
5.18.	Llama-Vergleich lokal $\leftrightarrow$ Server (finale Werte)	136
5.19.	Bucket-Verteilung auf dem Gesamtbestand	137
5.20.	Alle rekonstruierbaren Konfigurationen – einheitliche Auswertung gegen finale GT (Golden 45)	138
5.21.	Tickets, die in 11 oder 12 der 12 Konfigurationen falsch klassifiziert wurden .	140
5.22.	Klassifikationsgüte der Frontier-Modelle auf Golden 45 (gegen menschliche Annotation)	141
5.23.	Wilson-Konfidenzintervalle auf Golden 45	143
5.24.	Wilson-Konfidenzintervalle auf Set 200	143
5.25.	McNemar-Tests für drei zentrale Konfigurationsvergleiche	144
5.26.	Out-of-Sample-Validierung auf 155 Tickets	146
5.27.	FN-Verteilung nach Bucket (Golden 45 + Set 200, 36 Tickets)	147
5.28.	Gate-Check-Läufe auf Golden-23 (alle Hard-Schwellen aus Tabelle 3.6) . . .	149
5.29.	Produktivlauf der RAG-Datenunit-Extraktion	150
5.30.	Quality-Flag-Verteilung im Produktivlauf	150
5.31.	Strukturerhaltung im Produktivlauf (6 060 Datenunits)	151
5.32.	Quantitative Wirkung von Fix 1 auf die Strukturerhaltung	152
5.33.	Aktivität des Verifiers über den Full-Run	152
5.34.	Eigenschaften des Zendesk-Stage-3-FAISS-Indexes	154
5.35.	Latenz der Retrieval-Stufen auf Mac M3	155
5.36.	Balance des RAG-Test-Sets ($n = 20$).	156
5.37.	Vorgeschlagene Interpretationsschwellen für RAGAS-Metriken.	158
5.38.	RAGAS-Scores der produktiven Systemkonfiguration (L3 = Llama 3.1 8B Q5_K_M, Reranker BGE-Reranker-v2-M3, Schwellenwert 0,40). Test-Set $n = 20$, Judge: Llama 3.1 8B Q4_K_M (lokal).	159
5.39.	Durchschnittliche Latenzzeiten der Pipeline-Stufen	162
7.1.	Im Rahmen dieser Bachelorarbeit verwendete KI-basierte Hilfsmittel.	179
A.1.	Übersicht der eingesetzten Technologien	180
A.2.	Eingesetzte LLM-Modelle nach Rolle	181
A.3.	Vollständige API-Endpunktübersicht	181
A.4.	Konfigurierbare Umgebungsvariablen	182
A.5.	Per-Ticket-Vergleichsmatrix Golden 45 (alle 12 Konfigurationen)	183
A.6.	Vergleichsmatrix: Menschliche Annotation vs. sieben Frontier-Modelle (Golden 45)	185
A.7.	Fünf repräsentative Frage-Antwort-Paare aus dem RAGAS-Test-Set.	194

Kurzzeichenverzeichnis

Kurzzeichen	Bedeutung
k	Anzahl der abgerufenen Kandidaten (Top-K)
d_k	Dimension der Key-Vektoren im Attention-Mechanismus
Q, K, V	Query-, Key- und Value-Matrizen im Transformer
C	Gewichteter Konfidenzwert
S_{rerank}	Normalisierter Rerank-Score
$S_{\text{separation}}$	Score-Separation zwischen Top-1 und Top-2
S_{llm}	LLM-Selbstkonfidenzbewertung
S_{coverage}	Abfrageterm-Abdeckungsrate
λ	MMR-Balanceparameter (Relevanz vs. Diversität)
T	Temperaturparameter der LLM-Generierung

Abkürzungsverzeichnis

ADF	Atlassian Document Format
AI	Artificial Intelligence
API	Application Programming Interface
BAAI	Beijing Academy of Artificial Intelligence
BGE	BAAI General Embeddings
BPE	Byte-Pair Encoding
CPU	Central Processing Unit
FAISS	Facebook AI Similarity Search
FN	False Negative
FP	False Positive
GBNF	GGML Backus-Naur Form (Grammatik-Format für constrained Generation)
GGUF	GGML Universal Format
GPT	Generative Pre-trained Transformer
GPU	Graphics Processing Unit
GT	Ground Truth
HC	Zendesk Help Center TKI (firmeneigene Help-Center-Wissensbasis unter https://help.tki-chemnitz.de , im Folgenden <i>HC</i>)
HTTP	Hypertext Transfer Protocol
IR	Information Retrieval
JSON	JavaScript Object Notation
JSONL	JSON Lines (zeilenweise JSON-Dokumente)

KI	Künstliche Intelligenz
LLM	Large Language Model
MMR	Maximal Marginal Relevance
NLP	Natural Language Processing
PDF	Portable Document Format
PII	Personally Identifiable Information
RAG	Retrieval-Augmented Generation
RAGAS	Retrieval-Augmented Generation Assessment
RAM	Random Access Memory
REST	Representational State Transfer
SC	Self-Consistency
SSE	Server-Sent Events
TF-IDF	Term Frequency–Inverse Document Frequency
TKI-Pr-KB	TKI-Produkt-Wissensbasis (aus internen Code-Repositories und dem Confluence-Wiki aggregierte Funktionsbeschreibung je Modul)
TN	True Negative
TP	True Positive
UI	User Interface
Zendesk	Zendesk Support TKI (das firmeneigene Zendesk-Ticketsystem; im Folgenden zur besseren Lesbarkeit kurz <i>Zendesk</i>)

Einleitung

Motivation

Der technische Kundensupport steht in der modernen Softwarebranche vor wachsenden Herausforderungen. Mit zunehmender Komplexität von Softwareprodukten steigen sowohl die Anzahl eingehender Support-Anfragen als auch die Anforderungen an die Qualität und Geschwindigkeit der Antworten. Support-Mitarbeiter müssen dabei auf eine Vielzahl verteilter Wissensquellen zugreifen – von Ticketsystemen über interne Wikis bis hin zu Slack-Kanälen und PDF-Dokumentationen – um passende Lösungen für Kundenprobleme zu finden.

Gleichzeitig hat die rasante Entwicklung im Bereich der Künstlichen Intelligenz, insbesondere bei Large Language Models (LLMs), neue Möglichkeiten eröffnet, wissensintensive Aufgaben zu automatisieren. Die Kombination von LLMs mit Retrieval-Augmented Generation (RAG) ermöglicht es, generative Sprachmodelle mit externem, aktuellem Wissen zu verknüpfen und so kontextbezogene, faktenbasierte Antworten zu generieren [29].

Für Unternehmen mit mehreren Softwareprodukten – wie im vorliegenden Fall die TKI-Produktfamilie mit NET Design, NET Engineering, NET Build und weiteren Lösungen – ist die effiziente Verwaltung und Bereitstellung von Support-Wissen von entscheidender Bedeutung. Die manuelle Suche durch Tausende von Tickets, Confluence-Seiten und Artikeln des Zendesk Help Center TKI (im Folgenden *HC*) ist zeitaufwändig und fehleranfällig. Ein KI-gestütztes System, das relevante Informationen automatisch abrufen und daraus Antwortvorschläge generiert, kann die Produktivität der Support-Mitarbeiter erheblich steigern.

In der aktuellen Support-Praxis werden bereits mehrere KI-Werkzeuge parallel eingesetzt: unternehmensweite Wissensassistenten wie Atlassian Rovo für die Recherche über Confluence und Jira [8] sowie geteilte Custom GPTs auf Basis von ChatGPT für wiederkehrende fachliche Anfragen [38, 39]. Jedes dieser Werkzeuge deckt jedoch nur einen Ausschnitt der für den Support relevanten Wissensquellen ab, erfordert einen manuellen Wechsel zwischen Benutzeroberflächen und verarbeitet Daten in der jeweiligen Anbieter-Cloud. Eine ausführliche Diskussion dieser Werkzeuge und ihrer Grenzen erfolgt in Kapitel 2; an dieser Stelle motiviert sie den im Folgenden entwickelten integrierten, lokal betriebenen Ansatz.

Zielsetzung

Ziel dieser Bachelorarbeit ist die Konzeption, Implementierung und Evaluation eines KI-gestützten Support-Agenten als Webanwendung. Das System soll folgende Kernfunktionalitäten bereitstellen:

- Automatisierte Extraktion und Normalisierung von Support-Daten aus vier produktiv eingebundenen Quellen (Zendesk, HC, Vitruvi und TKI-Produkte-Wissensbasis – im Folgenden *TKI-Pr-KB*); drei weitere unternehmensinterne Quellen (Confluence, Jira, Slack) wurden analysiert, aus zeitlichen Scope-Gründen jedoch nicht integriert
- Aufbau einer semantischen Suchstruktur mittels Vektoreinbettungen und FAISS-Indizierung
- Zweistufige Retrieval-Pipeline bestehend aus Bi-Encoder-Suche und Cross-Encoder-Reranking
- Generierung von Antwortvorschlägen durch ein lokal betriebenes LLM mit Streaming-Ausgabe
- Bereitstellung einer React-basierten Benutzeroberfläche mit konfigurierbaren Parametern
- Mehrstufige Konfidenzbewertung zur Qualitätseinschätzung der generierten Antworten

Das System richtet sich an Support-Mitarbeiter als Endnutzer und soll als Assistenzsystem – nicht als vollautomatischer Chatbot – fungieren. Die generierten Antworten dienen als Entwürfe, die von den Mitarbeitern geprüft und angepasst werden können.

Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert:

Kapitel 1 stellt die theoretischen Grundlagen vor, die für das Verständnis des entwickelten Systems notwendig sind. Dies umfasst die Bereiche Natural Language Processing, Large Language Models, Retrieval-Augmented Generation sowie die verwendeten Webtechnologien.

Kapitel 2 gibt einen Überblick über den aktuellen Stand der Technik im Bereich KI-gestützter Support-Automatisierung und grenzt den eigenen Ansatz von bestehenden Lösungen ab.

Kapitel 3 beschreibt das Gesamtkonzept und die Architektur des Systems, einschließlich der Anforderungsanalyse, der Datenverarbeitungspipeline und der Integration der verschiedenen Komponenten.

Kapitel 4 dokumentiert die technische Umsetzung des Backends, Frontends und der Datenextraktionspipelines im Detail.

Kapitel 5 präsentiert die Evaluationsergebnisse anhand realer Support-Anfragen und vergleicht verschiedene Konfigurationen des Systems.

Kapitel 6 fasst die Ergebnisse zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen.

Kapitel 1.

Grundlagen

Dieses Kapitel stellt die theoretischen und technologischen Grundlagen vor, auf denen das in dieser Arbeit entwickelte System aufbaut. Es behandelt die Verarbeitung natürlicher Sprache, die Theorie und Praxis des Information Retrieval, die Funktionsweise moderner Sprachmodelle einschließlich ihrer Decoding- und Self-Consistency-Verfahren, das Retrieval-Augmented-Generation-Paradigma, das zweistufige Retrieval mit Bi- und Cross-Encoder, die formalen Grundlagen der Klassifikationsbewertung sowie – knapp – die eingesetzten Webtechnologien. Jeder Abschnitt schließt mit einem expliziten *Bezug zur Arbeit*-Absatz, der die Verbindung zwischen Theorie und der konkreten Implementierung herstellt.

1.1. Natural Language Processing

Natural Language Processing (NLP) bezeichnet den Teilbereich der Künstlichen Intelligenz, der sich mit der maschinellen Verarbeitung und dem Verstehen menschlicher Sprache befasst. NLP-Systeme ermöglichen es Computern, Text zu analysieren, zu interpretieren und zu generieren [32].

1.1.1. Tokenisierung und Subword-Einheiten

Sprachmodelle arbeiten nicht auf Zeichenketten, sondern auf einer endlichen Menge diskreter Einheiten – den *Tokens*. Klassische Wort-Tokenisierer leiden an dem *Out-of-Vocabulary*-Problem: jedes nicht im Vokabular enthaltene Wort wird zu einem Spezial-Token `<unk>`. Moderne Modelle verwenden stattdessen *Subword-Tokenisierung*, die seltene Wörter in Teilstücke zerlegt und so eine vollständige Abdeckung des Eingabetexts gewährleistet.

Das in der Praxis dominante Verfahren ist *Byte-Pair Encoding* (BPE) [46]. BPE startet mit einem Vokabular aus Einzelzeichen und führt iterativ die häufigste Token-Folge im Korpus zu einem neuen Token zusammen, bis eine vorgegebene Vokabulargröße erreicht ist. Varianten umfassen *WordPiece* (Likelihood-basiert) und *SentencePiece* (Whitespace-unabhängig).

Bezug zur Arbeit. Alle in der vorliegenden Arbeit verwendeten Modelle nutzen Subword-Tokenisierer: das Embedding-Modell BGE-M3 verwendet den XLM-RoBERTa-Tokenizer (BPE), Llama 3.1 nutzt einen erweiterten BPE-Tokenizer (128k Vokabular), Qwen 2.5 ebenfalls einen BPE-basierten Tokenizer mit 152k Vokabular. Die Tokenanzahl bestimmt unmittelbar, wie viel Konversationskontext in das Kontextfenster passt – bei der LLM-basierten Qualitätsfilterung musste diese Begrenzung durch eine Head-and-Tail-Trunkierung des Ticket-Transkripts (siehe Abschnitt 4.2.2.4) berücksichtigt werden.

1.2. Klassisches und dichtes Information Retrieval

Information Retrieval (IR) ist die Disziplin, die sich mit dem Auffinden relevanter Dokumente in einer Sammlung anhand einer Anfrage befasst. Die zentrale Modellannahme aller IR-Verfahren ist, dass die Relevanz eines Dokuments zu einer Anfrage durch eine berechenbare Ähnlichkeitsfunktion approximiert werden kann.

1.2.1. Klassische Verfahren: TF-IDF und BM25

Klassische IR-Verfahren beruhen auf der Häufigkeit gemeinsamer Terme zwischen Anfrage und Dokument. Das *TF-IDF*-Maß gewichtet Terme mit ihrer Häufigkeit im Dokument (Term Frequency) und ihrer inversen Dokumenthäufigkeit im Korpus:

$$tfidf(t, d, D) = tf(t, d) \cdot \log \frac{|D|}{|\{d' \in D : t \in d'\}|}. \quad (1.1)$$

Eine probabilistisch fundierte Erweiterung ist *Okapi BM25* [44], die zusätzlich eine Längennormalisierung berücksichtigt und in Suchsystemen wie Lucene den De-facto-Standard darstellt:

$$BM25(q, d) = \sum_{t \in q} idf(t) \cdot \frac{tf(t, d) \cdot (k_1 + 1)}{tf(t, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{avgdl}\right)}. \quad (1.2)$$

Klassische Verfahren sind transparent und schnell, scheitern aber an semantischer Variation: Eine Anfrage „Lizenzproblem“ und ein Dokument zum „Aktivierungsfehler“ ohne gemeinsame Terme erzeugen einen BM25-Score von Null, obwohl beide dasselbe Anliegen betreffen.

1.2.2. Vektorräume, Kosinusähnlichkeit und dichtes Retrieval

Im *dichten Retrieval* (Dense Retrieval) werden Anfrage und Dokumente in denselben kontinuierlichen Vektorraum $\mathbb{R}^d$ eingebettet, sodass semantisch ähnliche Texte nahe beieinan-

der liegen. Als Ähnlichkeitsmaß dient die *Kosinusähnlichkeit*:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}. \quad (1.3)$$

Sind die Vektoren ℓ_2 -normalisiert ($\|\mathbf{u}\| = \|\mathbf{v}\| = 1$), reduziert sich die Kosinusähnlichkeit auf das innere Produkt $\mathbf{u} \cdot \mathbf{v}$, was die Suche in modernen Vektorindizes beschleunigt.

Dense-Retrieval-Modelle wie *Dense Passage Retrieval* (DPR) [27] werden mit *Contrastive Learning* trainiert: Positiv-Paare aus echten Anfrage-Dokument-Beziehungen werden im Vektorraum näher gezogen, während Negativbeispiele auseinandergedrückt werden. Die zugrunde liegende Verlustfunktion ist die *InfoNCE*-Loss [35]:

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp(\cos(\mathbf{q}, \mathbf{d}^+)/\tau)}{\exp(\cos(\mathbf{q}, \mathbf{d}^+)/\tau) + \sum_j \exp(\cos(\mathbf{q}, \mathbf{d}_j^-)/\tau)}, \quad (1.4)$$

mit Temperaturparameter τ und Negativbeispielen $\mathbf{d}_j^-$.

Bezug zur Arbeit. Die vorliegende Arbeit verwendet ausschließlich *dichtes Retrieval* mittels BGE-M3 (siehe Abschnitt 1.4.2), nicht BM25. Die Wahl folgt aus der Beobachtung, dass Support-Konversationen häufig Synonyme, Akronyme und Mehrsprachigkeit enthalten – klassische Term-Übereinstimmung würde dort systematisch versagen. Die InfoNCE-basierte Trainingsmethode des BGE-M3-Modells liefert die theoretische Grundlage dafür, dass die Embeddings semantische Ähnlichkeit korrekt im Vektorraum kodieren.

1.3. Large Language Models

Large Language Models (LLMs) sind neuronale Sprachmodelle mit Milliarden Parametern, die auf großen Textkorpora trainiert werden und die bedingte Wahrscheinlichkeitsverteilung $p(x_t | x_{<t})$ modellieren – die Wahrscheinlichkeit eines Tokens x_t gegeben den vorangehenden Kontext.

1.3.1. Transformer-Architektur

Die Grundlage moderner LLMs bildet die 2017 vorgestellte *Transformer*-Architektur [48]. Im Gegensatz zu rekurrenten Architekturen verarbeitet sie die Eingabesequenz parallel über den *Self-Attention*-Mechanismus.

Self-Attention berechnet für jedes Token gewichtete Beziehungen zu allen anderen Token mittels dreier gelernter Projektionen – Query (Q), Key (K) und Value (V):

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}. \quad (1.5)$$

Der Skalierungsfaktor $1/\sqrt{d_k}$ verhindert numerische Instabilitäten bei hohen Dimensionen. *Multi-Head Attention* berechnet diese Operation parallel in h unabhängigen Köpfen, sodass unterschiedliche Beziehungstypen erfasst werden können (etwa syntaktische in einem, semantische in einem anderen Kopf). Ein Transformer-Block kombiniert Multi-Head-Attention mit einer positionsweisen Feed-Forward-Schicht, jeweils ergänzt durch Residualverbindungen und Layer-Normalisierung.

1.3.2. Autoregressive Generierung

Decoder-only-Modelle wie GPT, Llama und Qwen verwenden *kausale Self-Attention* – jeder Token darf nur auf vorangehende Token zugreifen. Die Modelle generieren Text autoregressiv, indem sie iterativ die nächste Tokenwahrscheinlichkeit bedingt auf alle vorherigen Token bestimmen:

$$p(x_1, x_2, \dots, x_T) = \prod_{t=1}^T p_\theta(x_t \mid x_1, \dots, x_{t-1}). \quad (1.6)$$

Das Training optimiert die negative Log-Likelihood über große Korpora; nach diesem *Pre-Training* folgt typischerweise eine Phase aus Instruction Tuning und Reinforcement Learning from Human Feedback (RLHF) [41], die das Modell auf dialogorientierte Aufgaben spezialisiert. Die in dieser Arbeit verwendeten **-Instruct*-Varianten von Llama und Qwen sind durch eine solche Phase gegangen, weshalb sie System-Prompts und Few-Shot-Anweisungen zuverlässig befolgen.

1.3.3. Decoding-Strategien

Aus der vom Modell ausgegebenen Wahrscheinlichkeitsverteilung über das Vokabular muss zur Generierungszeit ein einzelner Token ausgewählt werden. Die wichtigsten Strategien sind:

- **Greedy decoding:** $x_t = \arg \max_v p_\theta(v \mid x_{<t})$. Deterministisch, aber neigt zu repetitiven Sequenzen.
- **Temperature sampling:** Die Logits werden vor der Softmax durch einen Temperaturparameter T skaliert, $p_T(v) \propto \exp(\ell_v/T)$. Für $T \rightarrow 0$ entartet die Verteilung zu Greedy; für $T = 1$ entspricht sie der Originalverteilung; höhere T erhöhen die Diversität.
- **Top- k - und Top- p -Sampling:** [22] schränken die Sampling-Verteilung auf die wahrscheinlichsten Token bzw. auf den kleinsten Tokenkern mit kumulativer Wahrscheinlichkeit $\geq p$ ein.

Bezug zur Arbeit. Die LLM-basierte Qualitätsfilterung der Zendesk-Tickets (Abschnitt 4.2.2.6) verwendet im produktiven Single-Pass-Modus *Greedy decoding* ($T = 0$), um eine deterministische und reproduzierbare Klassifikation zu erzielen. Im optionalen Self-Consistency-Modus (Abschnitt 1.3.5) wird stattdessen mit $T = 0,3$ Temperature-Sampling betrieben, sodass die N unabhängigen Aufrufe zu unterschiedlichen Vorhersagen führen können – nur dadurch ist die anschließende Mehrheitsabstimmung sinnvoll.

1.3.4. Few-Shot Learning und In-Context Learning

Eine zentrale Eigenschaft großer Sprachmodelle ist *In-Context Learning* [10]: Ohne Anpassung der Modellgewichte können sie eine neue Aufgabe ausschließlich anhand weniger Beispiele im Prompt erlernen. Bei *k-Shot Prompting* werden dem Modell k Eingabe-Ausgabe-Beispiele vorangestellt, gefolgt von einer ungelösten Eingabe; das Modell extrapoliert das gezeigte Muster.

Few-Shot-Prompting ist effektiv, wenn die Beispiele die gewünschte Aufgabenverteilung repräsentativ abdecken und konsistent formatiert sind. Die Auswahl der Beispiele beeinflusst die Modellleistung erheblich; das Modell tendiert dazu, das im Prompt gezeigte Format und Vokabular zu reproduzieren.

Bezug zur Arbeit. Der Klassifikator der Qualitätsfilterung (Abschnitt 3.3.1.7) verwendet *8-Shot Prompting* mit acht realen Tickets aus dem Entwicklungs-Set. Die Beispiele wurden so gewählt, dass jeder der sechs Outcome-Buckets (`solved`, `admin_only`, `escalated`, `abandoned`, `consultation`, `no_problem`) durch mindestens ein Beispiel repräsentiert ist, ergänzt um zwei synthetische How-to-Beispiele für die Bucket-Distinktion. Damit wird die Konsistenz des JSON-Ausgabeformats sichergestellt und das Modell lernt implizit, in welchen Fällen ein Ticket als `customer_confirmed = true` klassifiziert wird.

1.3.5. Self-Consistency

Self-Consistency [51] ist eine Technik, die die Robustheit von LLM-Ausgaben durch wiederholte Stichprobenziehung verbessert. Anstatt eine einzige Antwort des Modells zu nehmen, werden N unabhängige Aufrufe mit Temperature-Sampling ($T > 0$) ausgeführt, und die endgültige Antwort wird durch Mehrheitsabstimmung über die N Vorhersagen bestimmt:

$$\hat{y} = \arg \max_y \sum_{i=1}^N \mathbb{I}[y^{(i)} = y]. \quad (1.7)$$

Die theoretische Begründung folgt aus der Beobachtung, dass eine korrekte Antwort über mehrere unabhängige Sampling-Pfade häufiger erreicht wird, während Fehler typischerwei-

se idiosynkratisch verteilt sind. Self-Consistency ist besonders dann wirksam, wenn das Modell für die korrekte Antwort eine zwar niedrige, aber stabile Wahrscheinlichkeit zuweist und für inkorrekte Antworten verschiedene konkurrierende Modi.

Bezug zur Arbeit. In der Iterationsphase auf Llama 3.1 8B mit Q4_K_M-Quantisierung steigerte Self-Consistency mit $N = 3$ und Seeds $\{7, 11, 19, 23, 31\}$ die Accuracy auf dem Golden-45-Datensatz von rund 82 % (Single-Pass) auf 84,4 % (siehe Tabelle 5.4). Der lokale Spitzenwert von 88,9 % wurde anschließend ohne Self-Consistency erreicht, nachdem die Modellquantisierung auf Q5_K_M angehoben wurde (vgl. Abschnitt 5.2.4); auf Q5 lieferte der stabilere Single-Shot-Modus dieselbe oder bessere Genauigkeit ohne das dreifache Inferenzbudget. Für die finale Konfiguration mit Qwen 2.5 32B wird Self-Consistency ebenfalls nicht eingesetzt, da das größere Modell auch ohne Stichprobenmittelung eine ausreichende Konsistenz erreicht und der zusätzliche Faktor von N in der Inferenzzeit (~ 80 s je Ticket im Single-Pass) nicht vertretbar wäre.

1.3.6. Quantisierung

Quantisierung reduziert den Speicher- und Rechenbedarf eines Modells, indem die ursprünglich in 16-Bit-Gleitkommazahlen (FP16) gespeicherten Gewichte durch niedrigere Bitbreiten approximiert werden. Eine 4-Bit-Quantisierung verkleinert das Modell um den Faktor vier auf Kosten einer geringen Verschlechterung der Vorhersagequalität.

Das in dieser Arbeit verwendete GGUF-Format (Georgi Gerganov's Universal Format) [19] realisiert die Quantisierung über eine k -Means-Clusterung der Gewichtswerte je Block. Die Variante Q4_K_M („4-bit, K-quants, medium“) verwendet 4 Bit pro Gewicht plus eine zusätzliche, höhere Präzision für besonders sensitive Tensoren (Embedding-Tabelle, ausgewählte Attention-Matrizen). Der Mehraufwand für diese sensitiven Parameter steigert die Modellqualität erheblich gegenüber gleichmäßiger 4-Bit-Quantisierung, bei nur geringfügig höherem Speicherbedarf.

Eine empirische Charakterisierung der Größe-Qualitäts-Abwägung zeigt typischerweise: FP16 (Referenz) $\rightarrow$ Q5_K_M (5,7 GB für 8B-Parameter, sehr nahe an FP16) $\rightarrow$ Q4_K_M (4,9 GB, geringer Perplexitätsanstieg) $\rightarrow$ Q3_K_S (3,6 GB, sichtbarer Qualitätsverlust). Verwandte Verfahren wie GPTQ und QLoRA [14] erweitern die Quantisierung auf das Fine-Tuning quantisierter Modelle.

Bezug zur Arbeit. Die Wahl der Quantisierung folgt der jeweiligen Hardware-Situation: Auf dem produktiven Backend (16 GB RAM) wird Llama 3.1 8B in der Q4_K_M-Variante ausgeführt, da das Modell zusammen mit Embedding-Modell, Reranker und FastAPI-Server in den Arbeitsspeicher passen muss. In der Iterationsphase der Qualitätsfilte-

rung wurde stattdessen die hochwertigere Q5_K_M-Variante verwendet, weil dort nur das Klassifikator-Modell geladen wurde und die zusätzliche Modellqualität sich direkt in besseren Klassifikationsentscheidungen niederschlug. Auf der Server-Hardware (31 GB RAM) erlaubt schließlich Qwen 2.5 32B in Q4_K_M (20 GB als Split-GGUF) den produktiven Einsatz eines deutlich größeren Modells, das zuvor lokal nicht ausführbar gewesen wäre.

1.3.7. Grammar-constrainer Generierung mit GBNF

Trotz Anti-Halluzinations-Anweisungen im Prompt produzieren LLMs in der Praxis regelmäßig Ausgaben, die ein gefordertes Schema verletzen: fehlende Klammern, unbekannte Enum-Werte, falsche Datentypen, abgeschnittene Strings am Token-Budget-Ende. Ein nachgelagertes Reparieren mittels regulärer Ausdrücke oder JSON-Reparatur-Bibliotheken ist fehleranfällig und führt bei strukturell stark gestörten Ausgaben zu Datenverlust.

Grammar-constrained Generation adressiert dieses Problem an der Wurzel: Eine formale Grammatik wird dem Decoder als Constraint übergeben, und zu jedem Decodier-Schritt werden ausschließlich Tokens zugelassen, die zu einer gültigen Fortsetzung gemäß der Grammatik führen. Tokens, die zu einer Schema-Verletzung führen würden, erhalten Wahrscheinlichkeit 0 (Logit-Masking) noch vor dem Sampling. Damit *kann* das Modell rein technisch keine ungültige Ausgabe produzieren – syntaktische Korrektheit ist garantiert, nicht erhofft.

Die in dieser Arbeit eingesetzte LLM-Runtime `llama-cpp-python` [1] unterstützt diese Technik über das *GGML BNF*-Format (GBNF), eine kontextfreie Grammatik in BNF-ähnlicher Syntax. Eine GBNF-Datei besteht aus benannten Produktionsregeln; das folgende Beispiel definiert eine sehr einfache Grammatik, die genau ein JSON-Objekt mit einem Feld `label` akzeptiert, dessen Wert entweder `USEFUL` oder `USELESS` ist:

```
root ::= "{" ws "\"label\":" ws label ws "}"
label ::= "\"USEFUL\"" | "\"USELESS\""
ws ::= [ \t\n]*
```

Mehrere Felder, Arrays variabler Länge, optionale Werte und Enum-Listen lassen sich mit denselben Mitteln ausdrücken. Die Grammatik wird einmal beim Decoder-Initialisierung in eine deterministische Token-Filter-Maske kompiliert; der zusätzliche Rechenaufwand pro Decodier-Schritt ist vernachlässigbar.

Bezug zur Arbeit. Beide LLM-Stufen der Datenverarbeitungspipeline nutzen GBNF-constrainer Generation. In der *Qualitätsfilterung* (Abschnitt 3.3.1.3) erzwingt eine kurze Grammatik die Felder `outcome`, `customer_confirmed`, `rationale` und `label`, wobei `outcome` ausschließlich einen der sechs Bucket-Werte annehmen kann. In der *strukturierten Extraktion* (Abschnitt 3.3.1.11) werden drei separate Grammatiken eingesetzt – eine pro

Stage –, die unter anderem das exakte Schema des 6-Feld-Datenunits inklusive der 12 erlaubten Section-Kategorien festlegen. Dadurch entfällt jegliche manuelle JSON-Reparatur; Schema-Verstöße sind systembedingt unmöglich.

1.4. Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [29] ist ein Paradigma, das die Stärken von Information Retrieval mit generativen Sprachmodellen kombiniert. Anstatt sich ausschließlich auf das im Modell gespeicherte parametrische Wissen zu verlassen – das nur den Trainingszustand widerspiegelt und somit veraltet sein kann –, werden vor der Textgenerierung relevante Dokumente aus einer externen Wissensbasis abgerufen und dem Modell als Kontext bereitgestellt.

1.4.1. RAG-Pipeline

Eine RAG-Pipeline besteht aus drei aufeinanderfolgenden Stufen:

1. **Retriever:** Ruft zu einer Anfrage q die k ähnlichsten Dokumente $\{d_1, \dots, d_k\}$ aus einer indexierten Wissensbasis ab. Der Retriever kann klassisch (BM25), dicht (DPR-artig) oder hybrid sein.
2. **Kontextaufbereitung:** Filtert, dedupliziert und ordnet die Kandidatenmenge; bei Bedarf erfolgt ein *Reranking* durch ein zweites Modell.
3. **Generator:** Ein LLM erzeugt die Antwort, wobei die Eingabe aus der ursprünglichen Anfrage und den abgerufenen Dokumenten besteht: $y = \text{LLM}(q, d_1, \dots, d_k)$.

Der wesentliche Vorteil gegenüber einem reinen LLM-Ansatz liegt in der Reduktion von *Halluzinationen* – der Tendenz von Sprachmodellen, plausibel klingende, aber faktisch falsche Informationen zu generieren. Durch die Bereitstellung verifizierter Quellen kann das Modell seine Antworten auf reale Dokumente stützen; im Idealfall wird das Modell sogar zur Quellenangabe angehalten, sodass die Antwortqualität nachträglich überprüfbar bleibt.

1.4.2. Embedding-Modelle

Embedding-Modelle bilden die Grundlage des dichten Retrievers. Sentence-Transformers [43] sind eine Klasse neuronaler Netze, die ganze Sätze oder Absätze in dichte Vektoren fester Dimension überführen. Sie nutzen typischerweise einen vortrainierten Transformer-Encoder (etwa BERT [15]) und führen ein zusätzliches Pooling über die Token-Repräsentationen durch.

Das in dieser Arbeit eingesetzte *BGE-M3* (BAAI General Embedding – Multi-Linguality, Multi-Functionality, Multi-Granularity) [12] erzeugt 1024-dimensionale Vektoren und unterstützt nativ Mehrsprachigkeit, was für den deutsch-englischen Support-Kontext entscheidend ist. Das Modell wurde mit kontrastivem Lernziel auf einem Mix aus Web-, QA- und Suchanfragen-Daten trainiert und liefert Embeddings, die sowohl für dichtes Retrieval als auch für Reranking verwendbar sind. Die Vektoren werden ℓ_2 -normalisiert, sodass die Kosinusähnlichkeit direkt als inneres Produkt berechenbar ist.

1.4.3. Vektorindexierung mit FAISS

Zur Suche nach den k ähnlichsten Vektoren in einer großen Sammlung kommen *Vektorindizes* zum Einsatz. *FAISS* [26] unterstützt drei Hauptklassen:

- `IndexFlatIP` – exakte Suche, $\mathcal{O}(N)$ pro Anfrage. Garantiert das exakte Top- k -Ergebnis.
- `IndexIVFFlat` – Inverted File Index mit k -Means-Clusterung. Approximative Suche in $\mathcal{O}(\sqrt{N})$.
- `IndexHNSWFlat` – Hierarchical Navigable Small World Graph [31]. Sublineare Suche, sehr schnelle Anfragen.

Die Wahl zwischen exakter und approximativer Suche folgt einer pragmatischen Schwelle: Bei einigen tausend Dokumenten ist die exakte Suche im Sekundenbruchteil abgeschlossen; erst ab Hunderttausenden Dokumenten wird die approximative Suche mit ihren leichten Genauigkeitseinbußen attraktiv.

Bezug zur Arbeit. Das vorliegende System verwendet `IndexFlatIP` (Abschnitt 4.2.2.3), da die Wissensbasis aus einigen tausend Dokumenten besteht und die exakte Suche zu jeder Zeit garantiert wird. Approximative Indexstrukturen wie HNSW würden für diese Größenordnung keinen messbaren Latenzgewinn bringen und unnötige Komplexität in die Pipeline einführen.

1.5. Zweistufiges Retrieval und Reranking

Eine zentrale Entwurfsentscheidung der RAG-Pipeline ist die Aufteilung des Retrievals in zwei Stufen mit unterschiedlichen Modellklassen.

1.5.1. Bi-Encoder vs. Cross-Encoder

Ein *Bi-Encoder* kodiert Anfrage und Dokument unabhängig voneinander in zwei separate Vektoren $\mathbf{q} = f_\theta(q)$, $\mathbf{d} = f_\theta(d)$, deren Ähnlichkeit anschließend durch ein einfaches Skalarprodukt bestimmt wird. Der entscheidende Vorteil ist, dass die Dokumentvektoren *vorberechnet* und in einem Vektorindex gespeichert werden können – die Suche zur Laufzeit reduziert sich auf eine einzige Embedding-Operation für die Anfrage und einen schnellen Index-Lookup. Der Nachteil ist, dass keine direkte Interaktion zwischen Anfrage und Dokument während der Encodierung stattfindet; subtile semantische Beziehungen können dadurch verloren gehen.

Ein *Cross-Encoder* verarbeitet hingegen Anfrage und Dokument gemeinsam als ein zusammengesetztes Eingabepaar $[q; d]$. Der Self-Attention-Mechanismus erfasst direkt die Interaktion zwischen beiden, was eine deutlich präzisere Relevanzbewertung ermöglicht [34]. Die Kehrseite ist jedoch eine quadratische Skalierung in der Anzahl der zu bewertenden Paare – für eine Wissensbasis von N Dokumenten müsste der Cross-Encoder N -mal pro Anfrage ausgeführt werden, was zur Laufzeit nicht praktikabel ist.

Die in der Praxis dominierende Architektur ist daher ein *Reranking-Pipeline*: Der Bi-Encoder liefert eine Vorauswahl der k_1 (~ 50) ähnlichsten Dokumente, der Cross-Encoder rangiert diese auf die k_2 (~ 10) relevantesten um. Damit wird die Genauigkeit des Cross-Encoders mit der Skalierbarkeit des Bi-Encoders kombiniert.

1.5.2. Maximal Marginal Relevance

Reranking allein liefert keine garantierte Diversität: Die Top- k -Ergebnisse können semantisch sehr ähnliche Dokumente enthalten, die im Modellkontext kaum zusätzliche Information bereitstellen. *Maximal Marginal Relevance* (MMR) [11] adressiert dieses Problem durch eine iterative Auswahl, die Relevanz und Diversität abwägt:

$$MMR(d_i) = \lambda \cdot \text{sim}(q, d_i) - (1 - \lambda) \cdot \max_{d_j \in S} \text{sim}(d_i, d_j), \quad (1.8)$$

wobei S die bereits ausgewählten Dokumente und $\lambda \in [0, 1]$ den Trade-Off zwischen Relevanz ($\lambda \rightarrow 1$) und Diversität ($\lambda \rightarrow 0$) steuert.

Bezug zur Arbeit. Die Pipeline nutzt BGE-M3 als Bi-Encoder (Abschnitt 4.2.2.3) und BGE-Reranker-v2-M3 als Cross-Encoder mit nachgeschaltetem MMR ($\lambda = 0,7$). Der Wert $\lambda = 0,7$ priorisiert Relevanz, lässt aber 30 % Diversitätsanteil zu, sodass das LLM nicht mit nahezu identischen Dokumenten gefüttert wird. Der Score-Schwellenwert 0,2 filtert zusätzlich Treffer geringer Relevanz vor dem Generator.

1.6. Klassifikation und Evaluationsmetriken

Die in dieser Arbeit zentrale Qualitätsfilterung der Zendesk-Tickets ist eine binäre Klassifikationsaufgabe (USEFUL/USELESS). Ihre Bewertung erfolgt mit klassischen Klassifikationsmetriken, deren formale Definitionen hier eingeführt werden.

1.6.1. Konfusionsmatrix

Eine binäre Klassifikation kann für jeden Ticket genau einem von vier Ereignissen zugeordnet werden, die in einer *Konfusionsmatrix* aggregiert werden:

Tabelle 1.1.: Konfusionsmatrix für eine binäre Klassifikation

	Wahrheit: positiv	Wahrheit: negativ
Pred. positiv	TP (true positive)	FP (false positive)
Pred. negativ	FN (false negative)	TN (true negative)

In der vorliegenden Arbeit ist USEFUL die positive Klasse: Ein Ticket, das tatsächlich nützlich ist und vom Modell auch als USEFUL klassifiziert wird, ist ein TP; ein nützlicher, aber als USELESS verworfener Ticket ist ein FN; und so weiter.

1.6.2. Accuracy, Precision, Recall, F1

Aus der Konfusionsmatrix ergeben sich vier zentrale Metriken:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1.9)$$

$$Precision = \frac{TP}{TP + FP} \quad (1.10)$$

$$Recall = \frac{TP}{TP + FN} \quad (1.11)$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (1.12)$$

Accuracy misst den Gesamtanteil korrekter Vorhersagen und ist nur bei ausgewogenen Klassenverteilungen aussagekräftig. *Precision* (auch positiver Vorhersagewert) misst, welcher Anteil der als positiv klassifizierten Beispiele tatsächlich positiv ist, und entspricht der *Vertrauenswürdigkeit* der positiven Vorhersage. *Recall* (auch Sensitivität) misst, welcher Anteil der tatsächlich positiven Beispiele auch erkannt wurde, und entspricht der *Abdeckung*. Der F_1 -Score ist das harmonische Mittel aus Precision und Recall und wird als zusammenfassendes Qualitätsmaß verwendet, wenn eine einzelne Zahl benötigt wird.

Precision und Recall stehen in einem natürlichen Trade-Off: Ein konservativer Klassifikator klassifiziert nur sehr eindeutige Fälle als positiv und erreicht damit hohe Precision, aber niedrige Recall; ein liberaler Klassifikator akzeptiert auch unsichere Fälle und erreicht hohe Recall, aber niedrige Precision. Die geeignete Balance hängt von der Anwendung ab.

1.6.3. Annotation und Ground Truth

Die Bewertung jedes Klassifikators setzt einen *Ground-Truth-Datensatz* voraus – eine Sammlung von Beispielen mit bekanntem korrektem Label. In NLP-Aufgaben wird dieser typischerweise durch *manuelle Annotation* erstellt, bei der menschliche Annotatoren die Beispiele begutachten und klassifizieren.

Die Qualität der Annotation hängt von zwei Faktoren ab: der *Klarheit der Labeling-Richtlinien* und der *Konsistenz der Annotatoren*. Bei mehreren Annotatoren wird typischerweise das *Inter-Annotator Agreement* (z. B. Cohen's Kappa) erhoben, um die Verlässlichkeit der Annotationen quantitativ zu beurteilen. Bei einem einzelnen Annotator entfällt diese Möglichkeit, weshalb die Klarheit der Richtlinien und die Stabilität über die Annotationszeit hinweg umso wichtiger werden.

Bezug zur Arbeit. Der Annotationsdatensatz dieser Arbeit umfasst 200 Tickets, die durch den Autor mit einer eigens entwickelten Streamlit-Anwendung (`annotate_tickets.py`, Abschnitt 3.3.1.10) klassifiziert wurden. Die binäre Entscheidung zwischen `USEFUL` und `NOT_USEFUL` wurde durch eine zuvor festgelegte Definition „wiederverwendbares Wissen“ präzisiert; die zusätzlich gespeicherten Notizen dokumentieren die Entscheidungsbegründung. Da nur ein einzelner Annotator verfügbar war, ist eine Inter-Annotator-Agreement-Bewertung nicht möglich – dies ist eine bewusst akzeptierte Limitierung, die im Limitationen-Abschnitt 5.9.1 explizit als Bedrohung der internen Validität diskutiert wird.

In der Auswertung der Qualitätsfilterung (Abschnitt 5.2) wird *Precision* als primäre Optimierungsgröße gewählt: ein einzelner falsch-positiver Ticket beeinträchtigt potenziell sämtliche nachfolgenden Retrieval-Treffer, während falsch-negative Tickets durch eine spätere manuelle Sichtung jederzeit rehabilitiert werden können.

1.7. Webtechnologien

Die Benutzeroberfläche und der Backend-Service des Systems basieren auf einem aktuellen, etablierten Web-Stack. Da der Schwerpunkt dieser Arbeit auf den datenwissenschaftlichen Aspekten liegt, werden die verwendeten Technologien hier nur knapp eingeführt.

Das *Backend* ist als FastAPI-Anwendung implementiert. FastAPI ist ein Python-Webframework, das auf den ASGI-Standards (Asynchronous Server Gateway Interface)

aufsetzt, asynchrone Programmierung mittels `async/await` unterstützt, automatische Anfrage- und Antwort-Validierung über Pydantic-Modelle bereitstellt und eine OpenAPI-Dokumentation autogeneriert. Die asynchrone Architektur ist Voraussetzung für das Streaming der LLM-Antworten zur Laufzeit. Die Token-für-Token-Übertragung an den Client erfolgt über *Server-Sent Events* (SSE) – einen HTTP-basierten Standard für unidirektionale Echtzeitkommunikation, der gegenüber WebSockets schlankere Implementierung bei vergleichbarer Wirkung für den vorliegenden Anwendungsfall bietet.

Das *Frontend* ist eine Single-Page-Application in React (Version 19) mit TypeScript. React stellt komponentenbasierte UIs bereit; TypeScript ergänzt JavaScript um ein statisches Typsystem, das Schnittstellenfehler bereits zur Kompilierzeit erkennt. Als Build-Tool kommt *Vite* zum Einsatz, das im Vergleich zu klassischen Bundlern wie Webpack durch native ES-Modul-Unterstützung deutlich kürzere Startzeiten und schnelleres Hot-Module-Replacement bietet. Für die Komponenten-Bibliothek wird *shadcn/ui* auf Basis der *Radix-UI-Primitives* mit *Tailwind CSS* v4 als Utility-First-Styling-Layer eingesetzt; mehrsprachige Texte werden über *react-i18next* verwaltet (Deutsch als Standard, Englisch optional).

Bezug zur Arbeit. Die genannten Technologien werden in Kapitel 4 detailliert eingesetzt: Der FastAPI-Server in Abschnitt 4.2.9 stellt die Pipeline-Endpunkte bereit, die SSE-Implementierung erlaubt die progressive Token-Anzeige, und das React-Frontend (Abschnitt 4.3) bindet die Streaming-Antworten an die Benutzeroberfläche an.

Kapitel 2.

Stand der Technik

Dieses Kapitel ordnet den in dieser Arbeit entwickelten Ansatz in den aktuellen Stand kommerzieller und wissenschaftlicher Lösungen ein. Abschnitt 2.1 betrachtet Werkzeuge, die in der Support-Praxis bereits verbreitet eingesetzt werden – klassische Support-Plattformen mit integrierter KI, der unternehmensweite Wissensassistent Atlassian RoVo sowie geteilte Custom GPTs. Abschnitt 2.2 fasst frei verfügbare RAG-Frameworks zusammen, mit denen sich vergleichbare Systeme prinzipiell selbst aufbauen lassen. Abschnitt 2.3 stellt aktuelle Forschungsarbeiten vor, die über das Standard-RAG-Paradigma von Lewis et al. [29] hinausgehen – konkret Self-RAG, Corrective RAG, FLARE und HyDE – und ordnet den eigenen zweistufigen Retrieval-Ansatz gegenüber diesen Varianten ein. Abschnitt 2.4 fasst die Abgrenzung in zwei tabellarischen Vergleichen zusammen und benennt die offene Lücke, die durch die vorliegende Arbeit geschlossen wird.

2.1. Aktuelle Praxis im technischen Kundensupport

In Support-Organisationen werden KI-Funktionen heute auf drei strukturell unterschiedlichen Wegen genutzt: über die KI-Module der eingesetzten Ticketsystem-Anbieter, über horizontale Wissensassistenten wie Atlassian RoVo und über frei konfigurierbare Custom GPTs auf Basis von ChatGPT. Diese drei Schichten ergänzen sich in der Praxis, weisen aber jeweils eigene Grenzen auf, die im Folgenden mit konkreten, herstellerseitig dokumentierten Kennzahlen belegt werden.

2.1.1. Kommerzielle Support-Plattformen mit integrierter KI

Zendesk AI integriert generative Sprachmodelle direkt in die Zendesk-Suite. Das klassische, regelbasierte *Answer Bot* aus früheren Produktgenerationen erreichte nach Branchenangaben lediglich eine Ticket-Deflection von 2–3 %. Mit dem neueren *AI Agent Advanced* berichtet Zendesk im *CX Trends Report 2024*, dass KI-Chatbots zwischen 30 und 50 % der Tier-1-Tickets automatisch beantworten und in reifen Implementierungen bis zu 80 % erreichen können [54]. Das System ist als Teil der Zendesk-Suite konzipiert und un-

terstützt keinen On-Premise-Betrieb; die Verarbeitung erfolgt über Cloud-Modelle externer Anbieter.

Intercom Fin (in der aktuellen Version v2) basiert auf einem Cloud-LLM und ist eng in die Intercom-Plattform integriert. Intercom weist im hauseigenen Blog für Fin v2 eine durchschnittliche Resolution Rate von 67 % über mehr als 6.000 Kunden aus (Stand März 2026), bei einer Spannweite von 25 % bis über 80 % in Abhängigkeit von der Qualität des Help-Center-Inhalts [24]. Auch Fin verarbeitet Anfragen ausschließlich in der Intercom-Cloud und ist auf den Intercom-eigenen Konversations-Stack zugeschnitten.

Freshdesk Freddy AI kombiniert ein Self-Service-Modul, einen Agenten-Copiloten und ein Reporting-Modul auf Basis externer LLMs. Freshworks gibt in einer Pressemitteilung für die Early-Access-Phase ihrer agentischen Workflows eine durchschnittliche Deflection Rate von 65 % an, mit Einzelwerten bis 80 % [17]. Wie Zendesk und Intercom ist auch Freddy an die jeweilige Plattform gebunden.

Salesforce Einstein GPT for Service stellt KI-generierte Antwortvorschläge und Fallzusammenfassungen für Salesforce Service Cloud bereit. Die Verarbeitung erfolgt innerhalb des Salesforce *Trust Layer*, das Add-on wird mit etwa 50 USD pro Nutzer und Monat lizenziert; eine produktbezogene, öffentlich zugängliche Deflection- oder Resolution-Kennzahl wird von Salesforce nicht ausgewiesen [45].

Die Abbildungen 2.1.1 bis 2.1.4 zeigen beispielhafte Hersteller-Darstellungen der vier hier diskutierten Plattformen und verdeutlichen den jeweiligen UI-Kontext, in den die KI-Funktionen integriert sind.

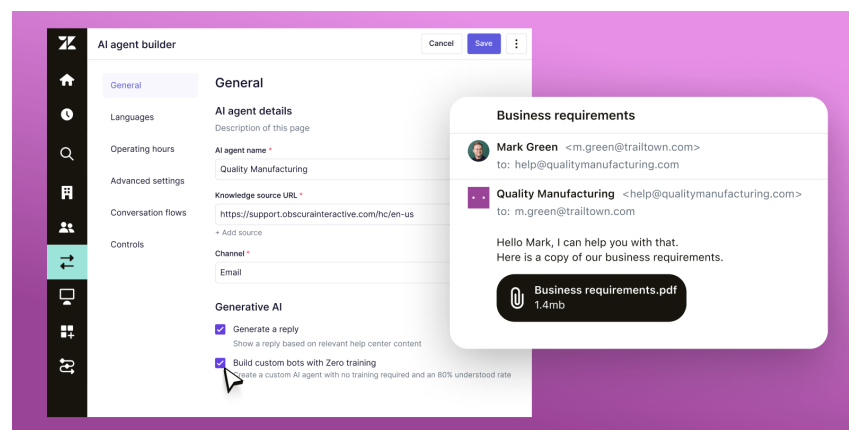


Abbildung 2.1.1.: Zendesk AI Agent in der Hersteller-Darstellung. Die KI-Funktion ist als integraler Bestandteil der Zendesk-Suite konzipiert [54].

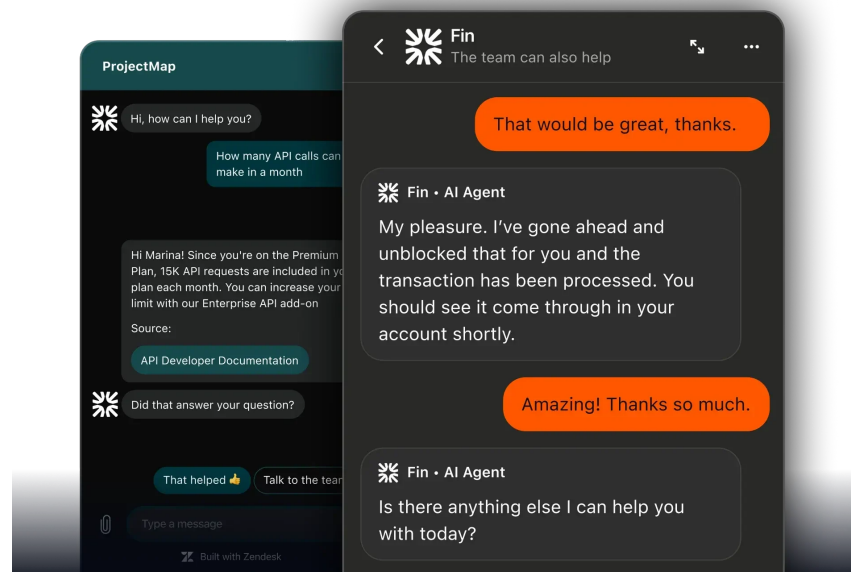


Abbildung 2.1.2.: Intercom Fin in der Live-Chat-Oberfläche. Fin v2 wird vom Hersteller mit einer durchschnittlichen Resolution Rate von 67 % ausgewiesen [24].

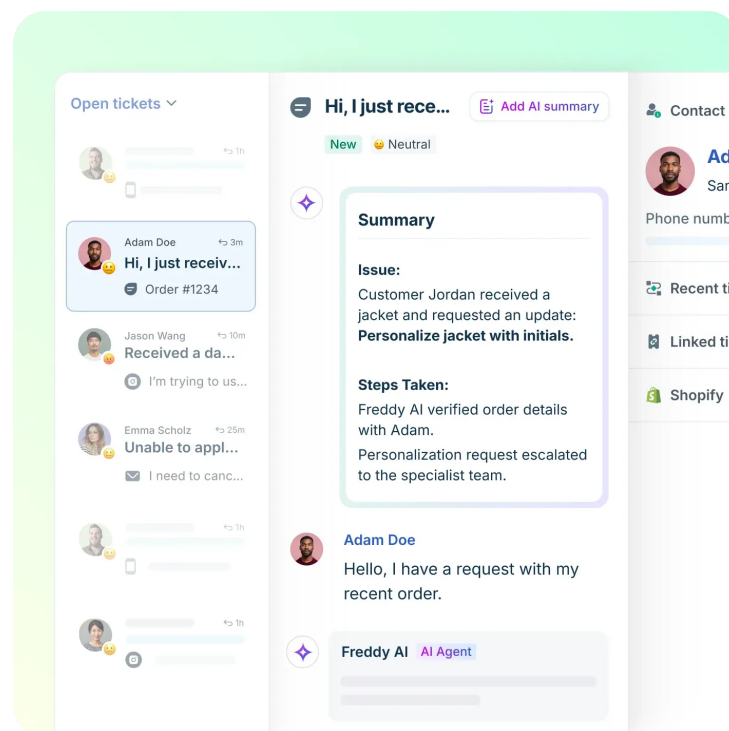


Abbildung 2.1.3.: Freshdesk Freddy AI in der Hersteller-Darstellung. Freshworks weist für den Early-Access-Stand eine Ø Deflection-Rate von 65 % aus [17].

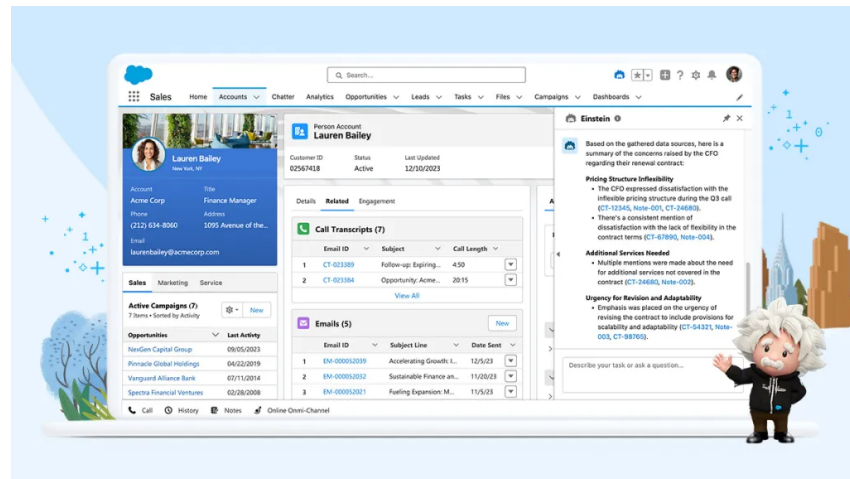


Abbildung 2.1.4.: Salesforce Einstein for Service. Die Verarbeitung erfolgt innerhalb des Salesforce *Trust Layer* als Add-on zur Service Cloud [45].

Diese vier Systeme teilen drei gemeinsame Eigenschaften, die ihre Anwendbarkeit auf den vorliegenden Anwendungskontext begrenzen. Erstens deckt jede Plattform jeweils nur die in ihrem eigenen Stack gehaltenen Wissensquellen ab – Tickets, Help-Center-Artikel, plattformeigene Konversationen –, während weitere Quellen wie Confluence, Jira, Slack oder PDF-Dokumentationen außerhalb des Indexes verbleiben. Zweitens werden die zugrunde liegenden Sprachmodelle ausschließlich als Cloud-Dienste betrieben, sodass Kundendaten den Vertrauensbereich des Unternehmens verlassen. Drittens skaliert das Lizenzmodell linear mit der Anzahl der Support-Mitarbeiter und nicht mit dem tatsächlichen Anfragevolumen.

2.1.2. Atlassian Roovo als unternehmensweiter Wissensassistent

Eine zweite, in vielen Support-Organisationen parallel eingesetzte Werkzeuggruppe sind horizontale Wissensassistenten, die nicht an ein Ticketsystem gebunden sind. Der aktuell verbreitetste Vertreter ist Atlassian Roovo, das laut Hersteller im Mai 2026 mehr als fünf Millionen monatlich aktive Nutzer aufweist [8]. Abbildung 2.1.5 zeigt die in den Atlassian-Stack eingebettete Benutzeroberfläche von Roovo.

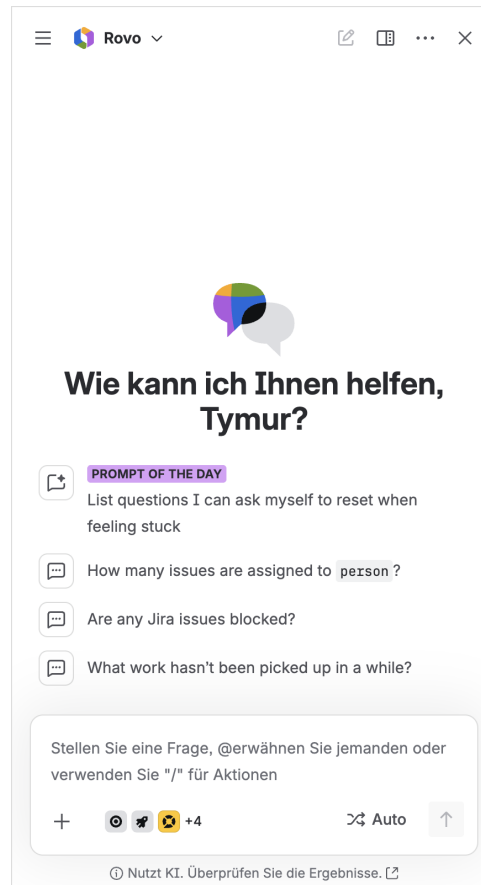


Abbildung 2.1.5.: Atlassian Rovo: integrierte Such- und Chat-Oberfläche innerhalb des Atlassian-Stacks. Rovo bündelt Suche über Confluence, Jira und weitere angebundene Quellen sowie einen dialogischen Assistenten in einer einheitlichen Oberfläche.

Rovo besteht aus drei Hauptkomponenten: *Rovo Search* bündelt die Suche über Confluence, Jira, Jira Service Management, Google Drive, Slack und weitere angebundene Quellen in einer einheitlichen Oberfläche; *Rovo Chat* stellt eine dialogische Schnittstelle bereit; *Rovo Agents* (inklusive *Rovo Studio*) erlauben die Definition spezialisierter Agenten mit eigenem Wissens- und Aktionsumfang [7, 8]. Architektonisch liegt allen drei Komponenten der *Teamwork Graph* zugrunde – ein graph-basierter Kontextlayer, der Personen, Seiten, Tickets, Dateien und Ereignisse mitsamt ihren Beziehungen modelliert und nicht nur als flacher Dokumentindex arbeitet. Anfragen liefern dadurch Ergebnisse, die nicht nur lexikalisch oder semantisch ähnlich sind, sondern auch die organisatorische Nähe zum Anfragenden (Team, Berechtigungen, Aktualität) berücksichtigen. Rovo respektiert dabei das Berechtigungssystem der Quellsysteme und liefert ausschließlich Inhalte zurück, die der anfragende Nutzer auch direkt einsehen dürfte.

Für den Einsatz im Support relevant sind drei dokumentierte Einschränkungen. Erstens deckt Rovo nicht-Atlassian-Ticketsysteme wie Zendesk oder Freshdesk nicht ab und unter-

stützt zudem aktuell keine *Custom Fields* in Jira [7]. Zweitens werden Anfragen an externe Modelle von OpenAI und Google weitergegeben; ein On-Premise- oder vollständig lokaler Betrieb ist nicht vorgesehen. Drittens basiert das Lizenzmodell auf einem Aufschlag von etwa 16 USD pro Nutzer und Monat zuzüglich eines Credit-Kontingents (25/70/150 Credits in den Plänen Standard/Premium/Enterprise; jede Rovo-Chat- oder Agent-Anfrage kostet 10 Credits, eine *Deep-Research*-Anfrage 100 Credits), wobei sämtliche zugewiesenen Nutzer abgerechnet werden – auch ohne aktive Nutzung [9]. Über produktbezogene, herstellerunabhängige Qualitätsmetriken wie Resolution Rate oder Faithfulness werden in der offiziellen Dokumentation lediglich Evaluationswerkzeuge beschrieben, jedoch keine veröffentlichten Vergleichszahlen genannt [6].

Damit deckt Rovo zwar einen großen Teil des im technischen Support relevanten internen Wissens ab (insbesondere Confluence und Jira), lässt aber den ticketbasierten Anteil aus dem Zendesk-Stack sowie produktspezifische PDF-Dokumentationen außerhalb der Atlassian-Cloud unberührt.

2.1.3. ChatGPT Custom GPTs als geteilte Wissensagenten

Als dritte Schicht hat sich in Support-Teams die Nutzung von *Custom GPTs* etabliert – konfigurierbarer ChatGPT-Instanzen, die mit einem festgelegten System-Prompt sowie hochgeladenen Wissensdateien betrieben werden [38]. In Enterprise- und Edu-Workspaces lassen sich Custom GPTs gezielt mit einzelnen Nutzern, gesamten Arbeitsbereichen oder über einen Freigabe-Link teilen, wobei feingranulare Rechte zwischen Bearbeiten und reiner Nutzung unterschieden werden [39]. Abbildung 2.1.6 zeigt einen anonymisierten Ausschnitt aus einem Unternehmens-Workspace, in dem mehrere fachspezifische Custom GPTs nebeneinander geteilt werden.

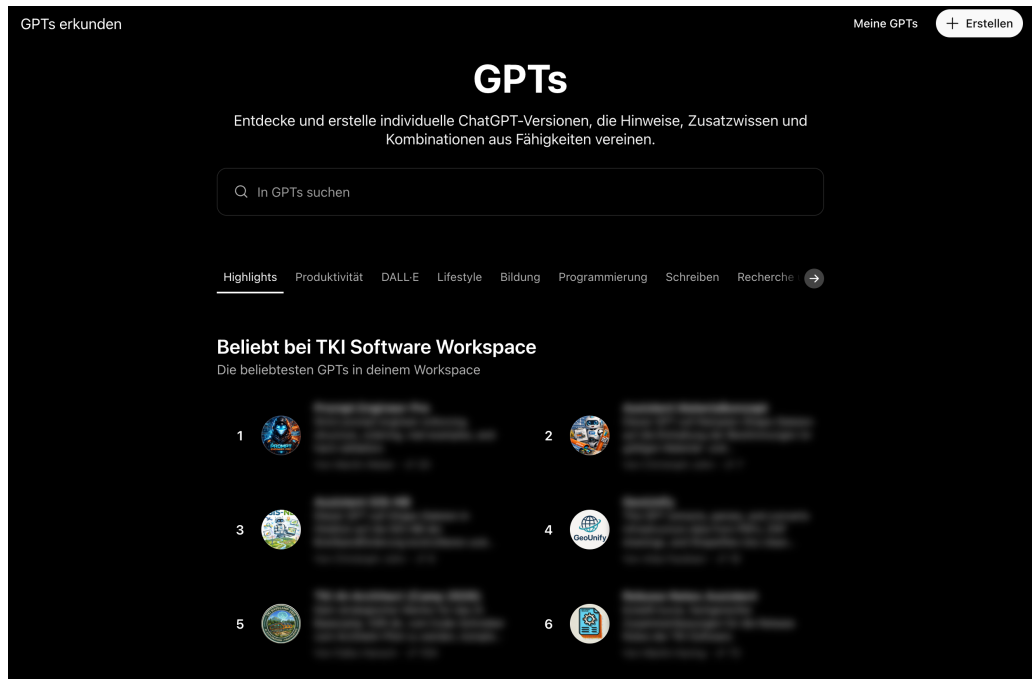


Abbildung 2.1.6.: Anonymisierter Ausschnitt aus einem Unternehmens-Workspace mit mehreren fachspezifischen Custom GPTs. Namen der Agenten und ihrer Autorinnen und Autoren wurden unkenntlich gemacht; die Darstellung dient ausschließlich zur Illustration des Sharing-Musters.

Im Support-Kontext entsteht so ein praxisrelevantes Muster: Eine erfahrene Support-Fachkraft kodiert ihr Erfahrungswissen in System-Prompt und Wissensdateien eines Custom GPT und verteilt den Zugang an weniger erfahrene Kolleginnen und Kollegen. Das implizite Erfahrungswissen wird damit explizit und überlebt personelle Fluktuation. Drei Einschränkungen verbleiben jedoch. Erstens werden alle hochgeladenen Dateien und Anfragen auf OpenAI-Servern verarbeitet, was bei kundenbezogenen oder vertraulichen Daten je nach Compliance-Anforderung problematisch ist. Zweitens ist die Wissensbasis eines Custom GPT statisch: Aktualisierungen erfordern manuelles erneutes Hochladen. Drittens fehlt eine quellenseitige Berechtigungsfilterung – ein Custom GPT zeigt allen freigegebenen Nutzern denselben Wissensumfang, unabhängig davon, ob diese in den Originalsystemen Zugriffsrechte hätten.

2.1.4. Zwischenfazit: Werkzeugfragmentierung als Kernproblem

Aus den drei vorangegangenen Unterabschnitten ergibt sich ein typisches Bild: Ein Support-Team mit gemischtem Werkzeugstack betreibt parallel ein KI-Modul im Ticketsystem (z. B. Zendesk AI für die Ticket-Triage), einen horizontalen Wissensassistenten wie Ro-vo (für interne Confluence- und Jira-Recherche) sowie mehrere geteilte Custom GPTs (für wiederkehrende fachliche Fragen). Daraus entstehen drei strukturelle Probleme: (a) die in

jedem Werkzeug separat indizierte Wissensbasis bleibt fragmentiert – keines der Werkzeuge sieht die volle Quellenmenge gleichzeitig; (b) Support-Mitarbeitende müssen zwischen heterogenen Oberflächen wechseln und vorab entscheiden, welches Werkzeug für eine Anfrage geeignet ist, was insbesondere bei weniger erfahrenen Kolleginnen und Kollegen zu Fehlentscheidungen führt; (c) die Lizenz- und Verarbeitungsmodelle der drei Schichten unterscheiden sich grundlegend, sodass eine einheitliche Datenschutz- und Kostenpolitik schwierig wird. Diese drei Probleme bilden den unmittelbaren Ausgangspunkt für den in dieser Arbeit verfolgten integrierten Ansatz.

2.2. Open-Source-Frameworks für RAG-Pipelines

Neben den geschlossenen kommerziellen Lösungen existiert eine wachsende Auswahl an Open-Source-Frameworks, mit denen sich RAG-Pipelines prinzipiell selbst aufbauen lassen. **LangChain** stellt ein allgemeines Orchestrierungsframework für LLM-Anwendungen bereit, das Komponenten wie Retriever, Prompt-Templates und Memory-Module bündelt, jedoch keine fertige Support-Anwendung darstellt [28]. **LlamaIndex** fokussiert auf Indizierung und Retrieval und bietet eine umfangreiche Konfigurationsbreite für unterschiedliche Vektor- und Hybrid-Index-Strategien [30]. **Haystack** der Firma deepset zielt explizit auf den produktiven Betrieb und liefert vorgefertigte Pipelines für Frage-Antwort-Systeme, lässt aber Integration in Ticketsysteme, Bedienoberfläche sowie Konfidenzbewertung als Aufgaben der jeweiligen Anwendung offen [13].

Diese Frameworks reduzieren die Implementierungstiefe für einzelne Komponenten erheblich, lösen aber nicht die für den Support-Kontext entscheidenden Aufgaben einer Multi-Quellen-Ingestion über sieben heterogene Systeme, einer mehrsignalgewichteten Konfidenzbewertung, eines Streaming-fähigen Antwortpfads und einer in den Support-Workflow eingebetteten Benutzeroberfläche. Diese Lücke schließt die vorliegende Arbeit durch eine integrierte Architektur, die auf etablierten Open-Source-Komponenten aufsetzt, jedoch nicht primär auf einem der genannten Frameworks beruht.

2.3. Aktuelle RAG-Forschung jenseits des Standardansatzes

Das ursprüngliche Retrieval-Augmented-Generation-Paradigma von Lewis et al. [29] koppelt eine Dense-Retrieval-Komponente mit einem generativen Sprachmodell in einem strikt sequentiellen *Retrieve–Generate*-Pfad. Nachfolgende Arbeiten haben dieses Grundmuster in vier Richtungen erweitert: Selbstreflexion über die Notwendigkeit eines Retrieval-Schritts, Bewertung und Korrektur der Retrieval-Qualität, aktives, fortlaufendes Retrieval während

der Generierung sowie die Verwendung hypothetischer Dokumente als Suchanfrage. Im Folgenden werden die vier prominentesten Vertreter dieser Richtungen knapp eingeführt, bevor die eigene Architektur gegenüber ihnen positioniert wird.

Self-RAG [5] erweitert das Sprachmodell um sogenannte *Reflection Tokens*, mit denen es selbst entscheidet, ob für die nächste Antwort ein Retrieval-Schritt erforderlich ist, ob die abgerufenen Passagen relevant sind und ob die generierte Antwort durch die Quellen ausreichend gestützt wird. Damit reduziert Self-RAG unnötige Retrieval-Aufrufe bei einfachen Anfragen und verbessert Faithfulness und Zitatgenauigkeit; der Preis ist ein spezialisiertes Instruction Tuning des Sprachmodells auf einem mit Reflection Tokens annotierten Datensatz, das nur für einzelne Modellfamilien öffentlich verfügbar ist.

Corrective RAG (CRAG) [53] ergänzt das Standard-Paradigma um einen leichtgewichtigen *Retrieval Evaluator*, der die abgerufenen Dokumente bewertet und einen Konfidenzgrad zurückgibt. Auf Basis dieses Konfidenzwerts wird entweder direkt generiert, eine Korrektur durch zusätzliche Web-Suche eingeleitet oder eine Mischstrategie gewählt. CRAG verbessert die Robustheit insbesondere bei wissensarmen oder irreführenden Treffern, setzt jedoch einen produktiv erreichbaren Web-Such-Endpunkt voraus.

FLARE [25] verfolgt einen *aktiven* Retrieval-Ansatz: Das Sprachmodell generiert Sätze schrittweise, beobachtet die Token-Wahrscheinlichkeiten und startet eine erneute Retrieval-Anfrage, sobald die Konfidenz für die nächste Phrase unter einen Schwellenwert fällt. Die durchschnittliche Häufigkeit liegt laut Originalarbeit bei 30–60 % aller Sätze. FLARE verbessert die Faktentreue insbesondere bei längeren Generierungen, erhöht jedoch durch die mehrfachen Retrieval-Aufrufe pro Antwort die Antwortlatenz spürbar.

HyDE [18] adressiert das Zero-Shot-Retrieval-Problem aus einer anderen Richtung. Statt die Nutzeranfrage direkt in den Embedding-Raum zu projizieren, lässt HyDE das Sprachmodell zunächst ein *hypothetisches* Dokument zur Anfrage generieren und nutzt dessen Einbettung als Suchvektor. Halluzinationen im hypothetischen Dokument werden dabei vom kontrastiv trainierten Encoder herausgefiltert, sodass die Bezugnahme letztlich am realen Korpus erfolgt. HyDE liefert insbesondere bei kurzen, vagen Anfragen Verbesserungen, erfordert jedoch einen zusätzlichen LLM-Aufruf vor jeder Suche.

Der in dieser Arbeit umgesetzte Ansatz nutzt einen klassischen zweistufigen Retrieve-Rerank-Pfad mit BGE-M3-Embeddings, BGE-Reranker-v2-M3 als Cross-Encoder und einer abschließenden Maximum-Marginal-Relevance-Diversifizierung (MMR). Auf die beschriebenen adaptiven Varianten wurde bewusst verzichtet. Die Entscheidung ist im Kontext des Anwendungsfalls aus vier Gründen begründbar. (a) Das System ist auf ein deterministisches Latenzbudget bei interaktiver Nutzung ausgelegt; aktive Mehrfach-Retrieval-Strategien wie FLARE oder Self-RAG erhöhen die Inferenzkosten linear mit der Anzahl der Retrieval-Iterationen. (b) Der Betrieb erfolgt vollständig offline auf unternehmenseigener Infrastruktur, ein Web-Fallback wie bei CRAG ist daher nicht verfügbar und auch nicht

erwünscht. (c) Die Wissensquellen sind kuratiert und thematisch homogen, sodass die durch HyDE generierten hypothetischen Dokumente gegenüber einer direkten Anfrage-Einbettung kaum messbaren Mehrwert versprechen. (d) Self-RAG erfordert ein spezialisiertes Fine-Tuning des Sprachmodells, das die Wartungs- und Reproduktionskosten der Pipeline für ein Bachelor-Projekt unverhältnismäßig erhöhen würde. Die adaptiven Varianten verbleiben damit als interessante Erweiterungsrichtung für zukünftige Arbeiten (vgl. Kapitel 6), während die vorliegende Arbeit auf einen reproduzierbaren, deterministischen und lokal lauffähigen Basis-Pfad fokussiert.

2.4. Abgrenzung und Positionierung

Die in den vorangegangenen Abschnitten beschriebenen Systeme und Forschungsansätze lassen sich anhand objektiver Merkmale gegenüber dem eigenen Ansatz einordnen. Tabelle 2.1 stellt zunächst die fünf wichtigsten kommerziellen bzw. quelloffenen Systeme gegenüber, Tabelle 2.2 ordnet anschließend die vier akademischen RAG-Varianten ein.

Im Gegensatz zu den in Tabelle 2.1 aufgeführten Vergleichssystemen vereinigt der eigene Ansatz vier produktiv eingebundene Wissensquellen (Zendesk-Tickets, HC, Vitruvi, TKI-Pr-KB) in einem einheitlichen Index und betreibt das Sprachmodell vollständig auf unternehmenseigener Hardware; drei weitere unternehmensinterne Quellen (Confluence, Jira, Slack) wurden analysiert, aus zeitlichen Scope-Gründen jedoch nicht in die produktive Pipeline aufgenommen. Gegenüber Custom GPTs und den Open-Source-Frameworks aus Abschnitt 2.2 bringt das System eine produktive Endnutzer-Oberfläche mit Streaming-Ausgabe sowie eine mehrsignalgewichtete Konfidenzbewertung mit, die in keinem der Vergleichssysteme öffentlich dokumentiert ist. Die in Spalte *Reported Benchmark* aufgeführten Werte stammen ausschließlich aus Herstellerquellen; eine unabhängige, vergleichende akademische Evaluation der genannten Plattformen liegt nach Recherchestand nicht vor, was die in Kapitel 5 durchgeführte RAGAS-basierte Eigenevaluation des entwickelten Systems zusätzlich rechtfertigt.

Tabelle 2.1.: Konkrete Support- und Wissens-Systeme im Vergleich zum eigenen Ansatz. Reported-Benchmark-Werte stammen aus Herstellerdokumentation und sind als Hersteller-Angaben zu lesen.

Merkmal	Zendesk AI	Intercom Fin v2	Freshdesk Freddy	Atlassian Rovo	ChatGPT GPT	Eigener Ansatz
LLM	Cloud	Cloud	Cloud	Cloud (OpenAI, Google)	Cloud (OpenAI)	lokal (Llama 3.1 8B, Q5)
Retrieval	proprietär	proprietär	proprietär	Teamwork Graph	RAG über Uploads	zweistufig + MMR
Quellen	nur Zendesk	nur Intercom	nur Freshworks	Confluence, Jira, Slack, Drive	Datei-Uploads	4 integriert (Zendesk, HC, Vitruvi, TKI-Pr-KB) + 3 analysiert
On-Premise	nein	nein	nein	nein	nein	ja
Datenfluss	Vendor-Cloud	Vendor + Anthropic	Vendor-Cloud	OpenAI / Google	OpenAI	vollständig lokal
Konfidenz	binär	binär	binär	nicht ausgewiesen	nicht ausgewiesen	4-Signal
Kostenmodell	pro Seat	pro Resolution	pro Seat	≈16 USD/User + Credits	Enterprise-Lizenz	Hardware-fix
Reported Benchmark	30–50 % Tier-1 ; bis 80 % [54]	Ø 67 % Resolution (3/2026) [24]	Ø 65 %, bis 80 % [17]	>5 Mio. MAU ; keine Qualitäts-KPI [6, 8]	n/a	RAGAS (Kap. 5)

Tabelle 2.2.: Erweiterte RAG-Varianten im Vergleich zum eigenen zweistufigen Retrieve-Rerank-Ansatz.

Variante	Schlüsselidee	Vorteil	Nachteil im vorliegenden Kontext	Übernommen?
RAG [29]	Retrieve → Generate	einfach, deterministisch	ohne Reranking anfällig für irrelevante Treffer	Basis-Pfad
Self-RAG [5]	Reflection Tokens; modellgetriebene Retrieve-Entscheidung	weniger unnötige Retrieval-Aufrufe; bessere Faithfulness	spezielles Fine-Tuning des LLM nötig	nein
CRAG [53]	Retrieval Evaluator + Web-Fallback	Robustheit bei irrelevanten Treffern	Web-Fallback unverträglich mit Offline-Betrieb	nein
FLARE [25]	aktives, satzweises Retrieval auf Basis von Token-Konfidenz	verbesserte Faithfulness bei langen Antworten	deutlich höhere Latenz durch Mehrfach-Retrieval	nein
HyDE [18]	hypothetisches Dokument als Embedding-Anfrage	bessere semantische Brücke bei vagen Anfragen	zusätzlicher LLM-Call vor jeder Suche	nein
Eigener Ansatz	zweistufig: BGE-M3 + Cross-Encoder + MMR; 4-Signal-Konfidenz	lokal, deterministisch, multi-quellenfähig	keine adaptive Komponente	—

Methodische Einschränkung der Hersteller-Kennzahlen. Die in Tabelle 2.1 aufgeführten quantitativen KPIs (Deflection-, Resolution- und MAU-Werte) stammen ausschließlich aus Hersteller-Quellen: Zendesk CX Trends Report, Intercom-Blog, Freshworks-Pressemitteilung sowie Atlassian-Dokumentation. Eine unabhängige, vergleichende akademische Evaluation der genannten Plattformen liegt nach Recherchestand nicht vor. Daraus folgen drei methodische Einschränkungen, die bei der Lektüre der Tabelle zu berücksichtigen sind. *Erstens* sind die berichteten Kennzahlen nicht direkt vergleichbar, da jeder Anbieter *Deflection* bzw. *Resolution Rate* unterschiedlich operationalisiert (Tier-1- vs. vollständige Eskalationsstufen, automatisierte Antwort vs. tatsächlich gelöstes Ticket, geöffnete vs. geschlossene Sitzungen) und auf jeweils eigenen, nicht veröffentlichten Stichprobengrundlagen misst. *Zweitens* dienen die Werte primär dem Marketing und sind daher tendenziell als obere Schranken einzustufen; die im Intercom-Material ausgewiesene Spannweite von 25 % bis über 80 % [24] verdeutlicht diesen Effekt exemplarisch. *Drittens* stehen die in der Tabelle versammelten Werte damit nicht für einen empirischen Genauigkeitsvergleich der Systeme bereit, sondern dokumentieren lediglich die jeweils öffentlich kommunizierte

Leistungserwartung der Anbieter.

Vor diesem Hintergrund stützt sich der Systemvergleich der vorliegenden Arbeit nicht auf die KPI-Spalte, sondern auf die in derselben Tabelle ausgewiesenen *strukturellen* Merkmale: LLM-Betriebsmodell (lokal vs. Cloud), Retrieval-Architektur, abgedeckte Wissensquellen, On-Premise-Fähigkeit, Datenfluss-Kontrolle, Konfidenz-Modell und Kostenstruktur. Diese Merkmale sind aus öffentlich zugänglicher Architektur- und Produktdokumentation überprüfbar und damit für einen wissenschaftlichen Vergleich tragfähig. Die quantitative Eigenevaluation des entwickelten Systems erfolgt anschließend in Kapitel 5 auf Basis des RAGAS-Frameworks an einem dokumentierten Testdatensatz; sie ist damit – im Unterschied zu den Hersteller-Angaben – methodisch reproduzierbar.

Zusammengefasst adressiert keine der untersuchten Systeme die spezifische Kombination aus (a) Multi-Quellen-Integration über sieben heterogene Wissensquellen, (b) vollständig lokalem LLM-Betrieb ohne Datenabfluss an Drittparteien, (c) differenzierter Mehrsignal-Konfidenzbewertung und (d) Integration in den bestehenden Arbeitsfluss der Support-Mitarbeitenden. Diese vierfache Lücke bildet den unmittelbaren Anknüpfungspunkt für das in den folgenden Kapiteln vorgestellte Konzept und seine Implementierung.

Kapitel 3.

Konzept und Architektur

Dieses Kapitel beschreibt das Gesamtkonzept und die Architektur des entwickelten KI-Support-Agenten. Ausgehend von den Anforderungen werden die Systemarchitektur, die Datenquellen, die Verarbeitungspipeline und die Integration der verschiedenen Komponenten erläutert.

3.1. Anforderungsanalyse

Die Anforderungen an das System wurden aus der Analyse des bestehenden Support-Workflows bei TKI abgeleitet. Sie lassen sich in funktionale und nicht-funktionale Anforderungen unterteilen.

3.1.1. Funktionale Anforderungen

- **FA-1: Semantische Suche** – Das System soll zu einer Support-Anfrage relevante Dokumente aus der Wissensbasis abrufen, wobei die Suche auf semantischer Ähnlichkeit basiert.
- **FA-2: Antwortgenerierung** – Auf Basis der gefundenen Dokumente soll ein Antwortvorschlag mittels LLM generiert werden.
- **FA-3: Produktfilterung** – Die Suche soll auf bestimmte Produkte (z. B. NET Design, NET Engineering) eingeschränkt werden können. Die Einschränkung erfolgt über einen API-Parameter; eine dedizierte UI-Komponente ist nicht erforderlich, sodass der Produktbezug in der Ergebnisliste lediglich als farbcodiertes Badge je Treffer dargestellt wird.
- **FA-4: Streaming-Ausgabe** – Die Antwort soll Token für Token in Echtzeit ausgegeben werden, um die wahrgenommene Antwortzeit zu verkürzen.
- **FA-5: Konfidenzbewertung** – Jede generierte Antwort soll mit einer Konfidenzeinschätzung versehen werden.

- **FA-6: Modellwechsel** – Es soll zur Laufzeit zwischen verschiedenen LLM-Modellen gewechselt werden können.
- **FA-7: Parametersteuerung** – Die Retrieval- und Generierungsparameter sollen über die Benutzeroberfläche konfigurierbar sein.
- **FA-8: Multi-Source-Datenintegration** – Daten aus den primär integrierten internen Quellen (Zendesk und HC), externen Wissensquellen zum Drittanbieter-Service Vitruvi (Developer-Portal, Customer Terms of Service) sowie aus internen Code-Repositories der TKI NET Suite und dem Confluence-Wiki aggregierte Funktionsbeschreibungen je Modul (TKI-Pr-KB) sollen in einer einheitlichen Suchstruktur zusammengeführt werden. Weitere unternehmensinterne Systeme (Confluence, Jira, Slack) wurden im Rahmen der Anforderungsanalyse betrachtet, aus zeitlichen Scope-Gründen jedoch nicht in die produktive Pipeline aufgenommen (vgl. Abschnitte 3.3.2, 3.3.3, 3.3.4).

3.1.2. Nicht-funktionale Anforderungen

- **NFA-1: Datenschutz** – Keine Übertragung von Kundendaten an externe Dienste. Die LLM-Inferenz erfolgt vollständig lokal.
- **NFA-2: Antwortzeit** – Die Gesamtlatenz der Pipeline (Suche + Reranking + Generierung) soll unter 15 Sekunden liegen.
- **NFA-3: Ressourceneffizienz** – Das System soll auf einem Server mit 16 GB RAM und 16 CPU-Kernen betreibbar sein.
- **NFA-4: Skalierbarkeit** – Die Architektur soll die gleichzeitige Bearbeitung mehrerer Anfragen unterstützen, ohne dass sich die Antwortqualität verschlechtert.
- **NFA-5: Wartbarkeit** – Der Code soll modular aufgebaut sein und die Hinzufügung neuer Datenquellen ohne tiefgreifende Änderungen ermöglichen.

3.2. Systemarchitektur

Die Gesamtarchitektur des Systems folgt einem Client-Server-Modell und besteht aus vier Hauptkomponenten: dem React-Frontend, dem FastAPI-Backend, der Datenverarbeitungspipeline und den externen Datenquellen. Abbildung 3.2.1 zeigt die Übersicht der Systemarchitektur.

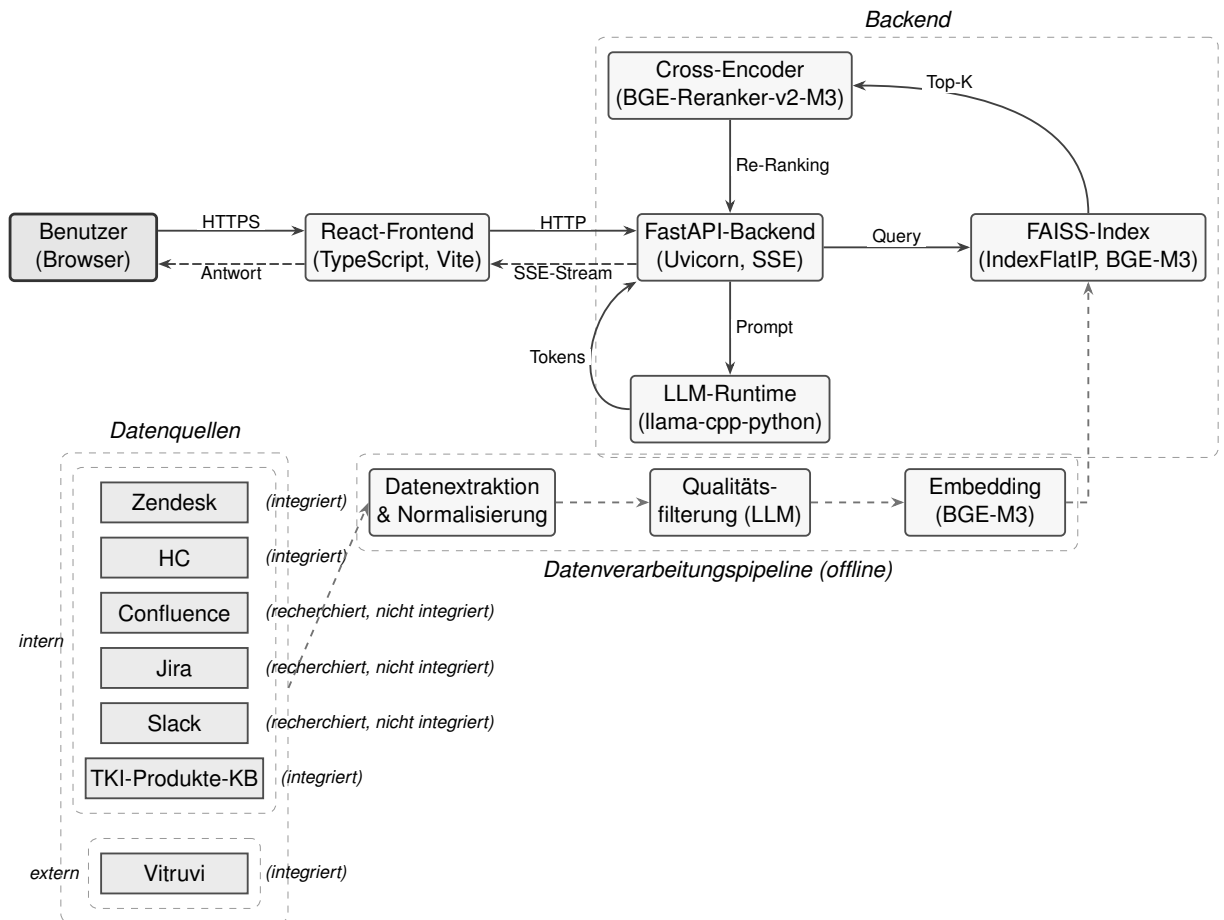


Abbildung 3.2.1.: Übersicht der Systemarchitektur. Der *Online-Anfrageweg* (oben) zeigt den Hin- und Rückweg eines Benutzeranfrage: durchgezogene Pfeile transportieren die Anfrage (Benutzer → Frontend → Backend → Retrieval/Reranking/LLM), eng-gestrichelte Pfeile den Draft-Antwortstrom (LLM → Backend → Frontend → Benutzer via SSE). Weit-gestrichelte Pfeile bilden den *Offline-Datenfluss* der Indizierungs-pipeline (Datenquellen → Extraktion → Qualitätsfilter → Embedding → FAISS-Index).

Das **Frontend** stellt die Benutzeroberfläche bereit und kommuniziert über HTTP/SSE mit dem Backend. Es ist als Single-Page-Application in React mit TypeScript implementiert und wird mit Vite gebaut.

Das **Backend** ist als FastAPI-Anwendung implementiert und stellt die folgenden Endpunkte bereit:

Die **Datenverarbeitungspipeline** wird offline ausgeführt und umfasst die Extraktion, Normalisierung, Einbettung und Indizierung der Daten. Sie erzeugt die Artefakte (Parquet-Datei, FAISS-Index, Vektordateien), die das Backend zur Laufzeit benötigt.

Tabelle 3.1.: API-Endpunkte des Backends

Endpunkt	Methode	Funktion
/api/search	POST	Semantische Suche mittels FAISS
/api/rerank	POST	Cross-Encoder-Reranking der Kandidaten
/api/draft	POST	LLM-basierte Antwortgenerierung
/api/pipeline	POST	Vollständige Pipeline (synchron)
/api/pipeline-stream	POST	Vollständige Pipeline mit SSE-Streaming
/api/models	GET	Liste verfügbarer LLM-Modelle
/api/models/switch	POST	Wechsel des aktiven LLM-Modells
/health	GET	Systemstatus und Modellverfügbarkeit

3.3. Datenquellen

Das System aggregiert Wissen aus drei Quellgruppen: zwei integrierte *interne* Quellen aus dem Unternehmens-Stack – Zendesk-Tickets (Abschnitt 3.3.1) als ticket-basierter Symptom-Lösung-Korpus und das Help Center (Abschnitt 3.3.5) als kuratierte Wissensbasis – externen Wissensquellen zum Drittanbieter-Service Vitruvi (Abschnitt 3.3.6) sowie der TKI-Pr-KB, einer aus internen Code-Repositories und dem Confluence-Wiki aggregierten Funktionsbeschreibung je Modul (Abschnitt 3.3.7). Drei weitere unternehmensinterne Quellen (Confluence, Jira, Slack) wurden analysiert, aber aus zeitlichen Scope-Gründen nicht in die produktive Pipeline aufgenommen; die Begründung erfolgt in den Abschnitten 3.3.2, 3.3.3 und 3.3.4. Die Trennung ist nicht nur organisatorisch, sondern auch technisch relevant: Externe Quellen erfordern eigene Crawling- und Authentifizierungsstrategien, sind außerhalb des Einflussbereichs des Unternehmens und können von Vitruvi jederzeit verändert werden; Zendesk-Tickets liefern Symptom-Lösung-Paare aus realem Support-Verlauf, das Help Center kuratierte Soll-Beschreibungen, während die TKI-Pr-KB die Soll-Funktionalität je Modul beschreibt und damit als komplementäre Schicht zu den Tickets dient.

3.3.1. Zendesk

Zendesk ist das primäre Ticketsystem des Unternehmens und stellt mit ca. 16 987 historischen Tickets den umfangreichsten Korpus innerhalb der Wissensbasis dar. Da nicht jeder Ticket-Verlauf einen für die Wissensbasis verwertbaren Beitrag liefert – viele Tickets enden in interner Provisionierung, abgebrochenen Threads oder Verweisen auf zukünftige Releases ohne Workaround –, wurde für Zendesk ein zweistufiger Extraktionsprozess konzipiert: (i) die rohe API-basierte Extraktion und Normalisierung der Tickets, (ii) eine LLM-gestützte Qualitätsfilterung, die jeden Ticket auf Eignung als Wissensbaustein prüft. Dieser zweite Schritt ist der zentrale Beitrag der vorliegenden Arbeit für die Datenqualität und wird in den Abschnitten 3.3.1.3 bis 3.3.1.10 ausführlich behandelt.

3.3.1.1. Datenextraktion und API-Anbindung

Die Extraktion erfolgt über die *Zendesk Search Export API* (Endpunkt `/api/v2/search/export.json`), die im Gegensatz zur klassischen Search API auch archivierte Tickets liefert und über cursorbasierte Paginierung (Header `links.next`) eine vollständige Iteration über den gesamten Bestand ermöglicht. Die Kommentare werden separat über `/api/v2/tickets/{id}/comments.json` abgerufen, wobei sowohl öffentliche als auch interne Notizen erfasst werden, die spätere Filterung auf öffentliche Kommentare aber bereits in dieser Stufe erfolgt.

Die Authentifizierung nutzt Basic-Auth mit dem Token-Schema `{email}/token` und API-Token, geladen aus einer `.env`-Datei. Bei Rate-Limit-Antworten (HTTP 429) wird der `Retry-After-Header` respektiert und exponentielles Backoff angewandt. Ein Checkpoint-Mechanismus serialisiert nach jeweils 100 verarbeiteten Tickets den Zwischenstand in `zendesk_full_download_checkpoint.json`, sodass eine Unterbrechung die wiederaufnahmesichere Fortsetzung ermöglicht.

Die Rohausgabe ist ein einzelnes JSON-Dokument (`zendesk_full_download.json`, ca. 810 MB) mit einem Array von Ticket-Objekten, deren Struktur dem Zendesk-Schema entspricht und um das Feld `comments` ergänzt wird.

3.3.1.2. Datennormalisierung

Die Rohdaten werden anschließend in eine einheitliche, suchtaugliche Form überführt. Die Normalisierung umfasst vier Aspekte:

- **PII-Maskierung:** E-Mail-Adressen und Telefonnummern werden mittels regulärer Ausdrücke (`[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\b(?:\d{3,4}|\d{7,10})`) durch Platzhalter (`<EMAIL>`, `<PHONE>`) ersetzt. Damit bleiben die Tickets nutzbar, ohne dass personenbezogene Daten in den Embedding-Index gelangen.
- **HTML- und Boilerplate-Entfernung:** Anhang-URLs des HC, Markdown-Bildverweise, deutsche und englische Mail-Disclaimer („Diese E-Mail enthält vertrauliche Informationen...“), Outlook-Warnhinweise und Standardsignaturen („Mit freundlichen Grüßen“, „Best regards“) werden mittels einer Reihe von Mustererkennern entfernt.
- **Rollen-Tagging:** Jeder Kommentar wird durch Vergleich von `author_id` mit `requester_id` eindeutig einer der Rollen `[CUSTOMER]` oder `[SUPPORT]` zugeordnet und der Body mit dem entsprechenden Präfix versehen.
- **Produktnormalisierung:** Über eine kanonische Liste der zehn TKI-Produkte werden Schreibvarianten auf eine einheitliche Form abgebildet (z. B. `NET Design`, `NET En-`

gineering, NET Build, NET Cloud, NET Scan). Die Erkennung erfolgt über einen Suchmuster-basierten Mehrfach-Match in Betreff und Body.

Das Ergebnis liegt als Apache-Parquet-Datei (zendesk/stage3_rag/data_support/processed/zendesk_parquet/normalized.parquet) mit den Feldern `ticket_id`, `title`, `text`, `created_at`, `updated_at`, `status`, `priority`, `tags`, `url`, `product`, `module`, `technology` vor.

3.3.1.3. Qualitätsfilterung – Motivation

Eine erste Sichtung der ca. 16 987 Tickets ergab, dass nur ein Bruchteil der Konversationen tatsächlich *wiederverwendbares Wissen* enthält. Eine manuelle Annotation von 200 Tickets durch den Autor (siehe Abschnitt 3.3.1.10) führte zu einer Verteilung von 62 nützlichen (31 %) und 138 nicht nützlichen (69 %) Tickets. Die Aufnahme aller Tickets in die Wissensbasis würde den Embedding-Raum mit Rauschen belegen, in dem die Retrieval-Stufe mit höherer Wahrscheinlichkeit irrelevante Treffer liefert – z. B. abgebrochene Threads ohne Lösung, reine Provisionierungsanfragen oder Verweise auf eine zukünftige Release ohne Workaround.

Daraus folgte die Anforderung an einen *automatisierten Qualitätsfilter*, der jeden Ticket vor der Indizierung in eine binäre Verwertungsklasse einsortiert: USEFUL (wird in die Wissensbasis aufgenommen) oder USELESS (wird verworfen). Die Komplexität der Aufgabe liegt darin, dass die Entscheidung nicht aus oberflächlichen Metadaten (Status `solved`, Tag-Liste, Antwortzeit) ableitbar ist, sondern eine semantische Beurteilung des Konversationsverlaufs erfordert: ob am Ende des Threads eine *wiederverwendbare technische Erkenntnis* steht – ein konkreter Workaround, ein UI-Pfad, eine Versionsangabe, eine generalisierbare Regel.

Da die manuelle Klassifikation des gesamten Bestands nicht skaliert, wurde die Aufgabe an ein lokales Large Language Model übergeben und durch eine Kombination aus deterministischen Vorfiltern, Few-Shot-Klassifikation und nachgelagerten regelbasierten *Confidence-Overrides* robust gemacht. Die endgültige Architektur ist in Abbildung 3.3.1 dargestellt.

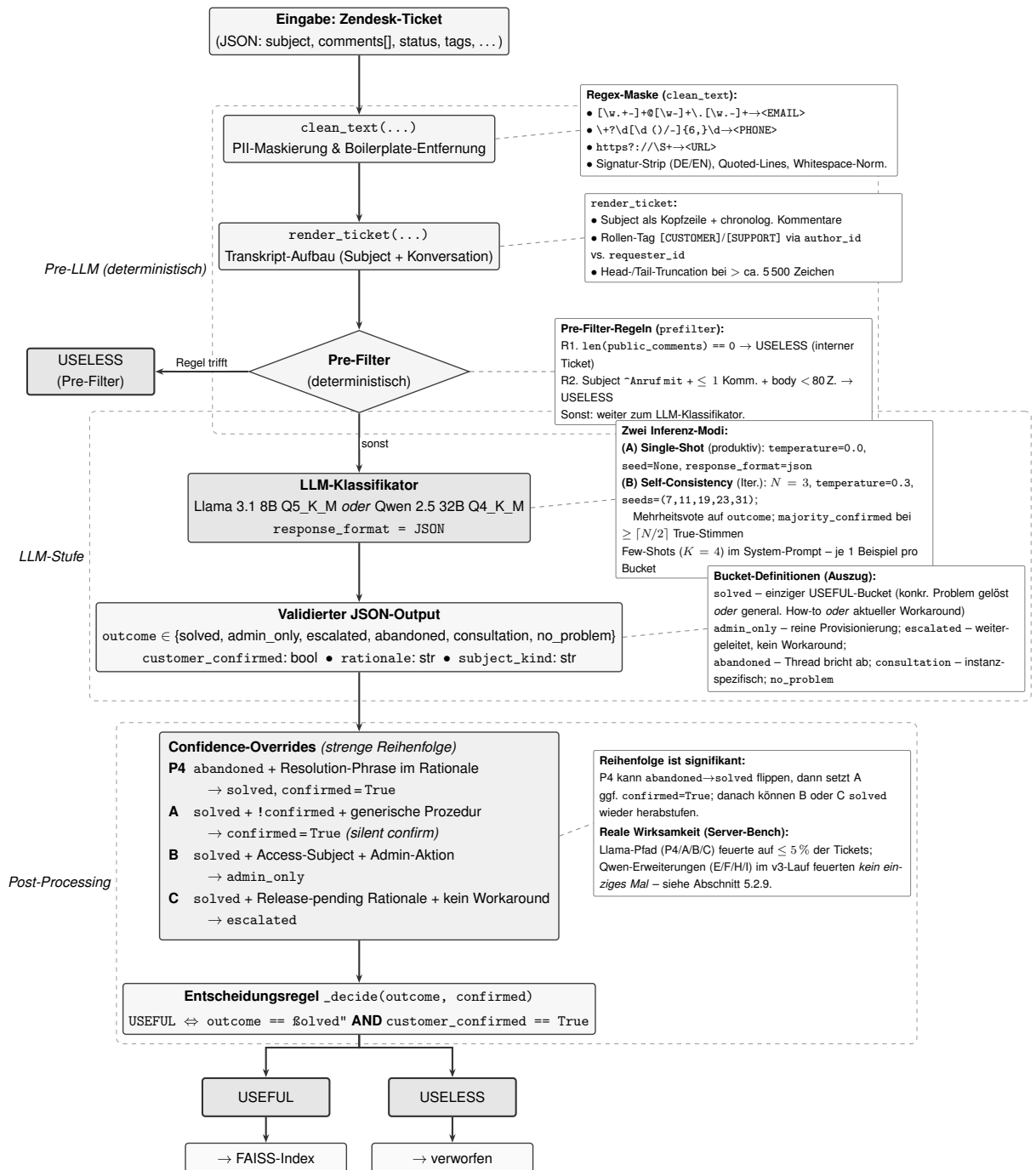


Abbildung 3.3.1.: Architektur der Qualitätsfilter-Pipeline auf Sub-Schritt-Ebene. Hauptfluss in der linken Spalte (vertikal): `clean_text` → `render_ticket` → `Pre-Filter` → `LLM-Klassifikator` (Single-Shot oder Self-Consistency $N = 3$) → `Confidence-Overrides P4/A/B/C` → `Entscheidungsregel outcome == solved AND customer_confirmed`. Die rechte Spalte annotiert jede Stufe mit den zugehörigen Parametern und Implementierungsmustern (Regex-Muster, Pre-Filter-Bedingungen, Inferenz-Parameter beider Modi, Bucket-Semantik, Reihenfolge-Effekte der Overrides). USEFUL-Tickets gehen in den FAISS-Index, USELESS-Tickets werden verworfen.

3.3.1.4. Bucket-Schema

Anstatt direkt eine binäre USEFUL/USELESS-Vorhersage vom LLM zu verlangen, wird der Ausgang einer Konversation zunächst in eines von sechs feingranularen *Outcome-Buckets* klassifiziert. Diese Indirektion verbessert sowohl die Konsistenz der Modellausgabe (das LLM trifft präzisere Entscheidungen, wenn es konkrete Kategorien wählen darf) als auch die Erklärbarkeit nachgelagerter Korrekturen. Tabelle 3.2 zeigt die sechs Buckets mit jeweiliger Definition.

Tabelle 3.2.: Outcome-Buckets der Qualitätsfilterung

Bucket	Definition
solved	Der Thread endet mit <i>wiederverwendbarem Wissen</i> : (a) konkretes technisches Problem mit bestätigter Lösung, (b) How-to-Frage mit generalisierbarer Antwort (UI-Pfad, Menü, Versionsangabe), oder (c) Workaround vorhanden (auch wenn ein Future-Release-Fix zusätzlich erwähnt wird). <i>Einziger USEFUL-Bucket</i> .
admin_only	Reine Provisionierungs- oder organisatorische Aktion ohne dahinterliegendes Software-Problem (Zugang, Lizenz, Passwort-Reset, VPN).
escalated	Reales Problem an Entwicklung weitergeleitet, kein aktueller Workaround. Ausnahme: bei explizit benannter Fix-Version wird zu solved reklassifiziert.
abandoned	Reales Problem, aber Thread endet ohne Auflösung (Agent fragt nach Details, kundenseitige Reaktion bleibt aus).
consultation	Agent prüft <i>kundenspezifische</i> Daten/Konfiguration und gibt nicht-generalisierbare Hinweise. Bei klassengültiger Regel wird zu solved reklassifiziert.
no_problem	Anrufnotiz, interner Thread, vage Einzelzeile, Smalltalk – kein technischer Inhalt.

Zusätzlich zum Bucket erzeugt das Modell ein boolesches Feld `customer_confirmed`, das angibt, ob der Kunde die Lösung explizit bestätigt hat. Eine ausdrückliche Bestätigung ist nicht zwingend erforderlich – ein klarer prozedurale Antwort des Agenten ohne nachfolgenden Widerspruch des Kunden gilt ebenfalls als stillschweigende Bestätigung („silent confirm“). Diese Lockerung ist erforderlich, da Kunden in der Praxis oft nicht antworten, wenn ein Problem gelöst ist.

3.3.1.5. USEFUL/USELESS-Entscheidungsregel

Aus den beiden Modellausgaben wird die binäre Verwertungsklasse deterministisch abgeleitet:

$$label = \begin{cases} \text{USEFUL} & \text{wenn } outcome = \text{ solved } \wedge customer_confirmed = \text{ true}, \\ \text{USELESS} & \text{sonst.} \end{cases} \quad (3.1)$$

Diese Regel ist absichtlich konservativ: nur ein bestätigter `solved`-Bucket landet im Index. Die Konsequenz ist eine hohe Präzision (siehe Abschnitt 5.2) bei moderater Recall, was für eine Wissensbasis-Vorfilterung das gewünschte Verhalten darstellt – besser ein verworfener nützlicher Ticket als ein verrauschter Index.

3.3.1.6. Pre-Filter-Regeln

Vor dem LLM-Aufruf werden zwei deterministische Regeln angewendet, die strukturelle Trivialfälle ohne Inferenzkosten ausfiltern:

1. *Keine öffentlichen Kommentare:* Tickets ohne öffentliche Kommunikation werden direkt als `USELESS` klassifiziert (Bucket `no_problem`, Begründung „internal-only ticket“).
2. *Anrufnotiz-Erkennung:* Tickets mit Betreff der Form „Anruf mit ...“, höchstens einem Kommentar und einem Body von weniger als 80 Zeichen werden als interne Telefonnotiz klassifiziert.

Auf dem 200-Ticket-Set greifen diese beiden Regeln auf 16 Tickets (8 %) und erreichen dabei eine Präzision von 100 % – alle 16 sind tatsächlich `USELESS`.

3.3.1.7. LLM-Klassifikator und Few-Shot-Prompt

Tickets, die den Pre-Filter passieren, werden an einen LLM-Klassifikator übergeben. Der Prompt besteht aus einem System-Prompt (ca. 650 Tokens), der die Rolle („technischer Support-Klassifikator für GIS-/CAD-/Glasfaser-Software“), die sechs Bucket-Definitionen und das exakte JSON-Ausgabeformat festlegt, gefolgt von 8 Few-Shot-Beispielen aus realen Dev-Set-Tickets, die das gesamte Bucket-Spektrum abdecken. Der Ticket-Verlauf wird in das User-Template eingebettet und die Antwort des Modells als JSON mit den Feldern `outcome`, `customer_confirmed`, `rationale` und `subject_kind` zurückgegeben.

Die Inferenz erfolgt mit `temperature=0.0` (Greedy-Sampling) und `n_ctx=8192`, sodass auch lange Threads vollständig in das Kontextfenster passen. Lange Konversationen werden vor dem Embedding über eine Head-and-Tail-Strategie (max. 5500 Zeichen) gekürzt, wobei der Anfang und das Ende des Threads – in denen Problemstellung bzw. Auflösung typischerweise stehen – erhalten bleiben.

3.3.1.8. Confidence-Overrides

Die Modellvorhersagen werden anschließend durch eine Reihe regelbasierter *Overrides* korrigiert. Jeder Override prüft ein bestimmtes Muster im *rationale*-Feld der Modellausgabe und nimmt unter zusätzlichen Bedingungen eine Promotion oder Demotion des Buckets vor. Diese Schicht kompensiert systematische Fehler des Modells, ohne den Prompt selbst zu modifizieren – wodurch die Modellausgabe interpretierbar bleibt und Korrekturen versionierbar werden. Tabelle 3.3 fasst die im Produktivsystem verwendeten Overrides zusammen.

Tabelle 3.3.: Confidence-Overrides der Qualitätsfilterung

Override	Bewegung	Auslöser
P4	abandoned → solved	Rationale beschreibt kundenseitige Auflösung trotz Bucket abandoned
A	solved + False → confirmed	Generische Prozedur erkennbar („generic how-to“, „step-by-step procedure“, US/UK-Schreibvarianten)
B	solved → admin_only	Zugangs-/Login-Betreff plus reine Admin-Aktion in Rationale
C	solved → escalated	Rationale verweist auf zukünftige Release ohne aktuellen Workaround
E	escalated → solved	Konkrete Fix-Version benannt (z. B. NET 14.2.2) oder „fix scheduled for [Monat]“
F	consultation+True → solved	Klassen-gültige Regel erkennbar („setting X to Y“, „X vs Y“, Statusfilter)
H	solved + False → confirmed	Agent liefert benannten Installer/Version („provides the new installer“)
I	admin_only+True → solved	Klassen-gültige Technik in Rationale trotz admin_only-Bucket

Die Overrides P4, A, B, C wurden ursprünglich für Llama 3.1 8B kalibriert; E, F, H, I wurden zusätzlich für Qwen 2.5 32B entwickelt, dessen Rationales in amerikanischer Schreibweise und mit anderem stilistischen Bias formuliert sind. Die Reihenfolge der Override-Anwendung ist relevant: P4 → A → H → B → C → E → F → I.

3.3.1.9. Self-Consistency

Als optionaler Modus steht eine Self-Consistency-Variante zur Verfügung, die jedem Ticket N unabhängige LLM-Aufrufe widmet (Standard $N = 3$, mit deterministischen Seeds {7, 11, 19, 23, 31} und Temperatur 0,3). Aus den N Vorhersagen wird der

Mehrheits-Bucket bestimmt, und das Feld `customer_confirmed` setzt eine strikte Mehrheit ($\lceil N/2 \rceil$ Stimmen) voraus. Self-Consistency wurde während der Iterationsphase auf Llama 3.1 8B mit Q4_K_M-Quantisierung eingesetzt und steigerte dort die Golden-45-Accuracy von rund 82 % auf 84,4 %; der spätere lokale Spitzenwert von 88,9 % wurde nach dem Wechsel auf die Q5_K_M-Quantisierung im Single-Shot-Modus erreicht, so dass Self-Consistency dort entbehrlich wurde (siehe Abschnitt 5.2). Im Produktivbetrieb mit Qwen 2.5 32B wird derzeit der Single-Pass-Modus verwendet, da das größere Modell auch ohne Stichprobenmittelung eine ausreichend hohe Konsistenz aufweist.

3.3.1.10. Benchmark-Methodik und Evaluationsdatensätze

Zur Beurteilung der Qualitätsfilterung wurden zwei Annotationsdatensätze aufgebaut, die in dieser Arbeit als *Golden 45* und *Set 200* bezeichnet werden.

Golden 45 – ein Entwicklungsdatensatz aus 45 Tickets (23 USEFUL, 22 USELESS), der für schnelle Iteration und Feinkalibrierung der Overrides genutzt wird. Aufgrund seiner Größe ist die Laufzeit eines Benchmarklaufs auf Server-Hardware kurz genug, um mehrere Konfigurationen pro Stunde zu vergleichen.

Set 200 – ein größerer Validierungsdatensatz aus 200 Tickets (62 USEFUL, 138 USELESS), zufällig aus dem Gesamtbestand stratifiziert nach Produkt und Status gezogen.

Annotationsprozess und Streamlit-Werkzeug. Um die binäre Klassifikation (USEFUL/NOT_USEFUL) effizient und reproduzierbar durchzuführen, wurde ein dediziertes Web-Werkzeug auf Basis von Streamlit [47] entwickelt (`annotate_tickets.py`, ca. 350 Zeilen Python). Die Anwendung lädt die 200 vorverarbeiteten Tickets aus der Datei `annotation_dataset_200_full.json` und führt den Annotator nacheinander durch sie hindurch; nach jeder Markierung wird die Datei `annotations_200.json` mit `label`, optionalen `notes`, `timestamp` und `annotator`-Feld atomar gespeichert. Der Autor las jeden der 200 Ticketverläufe vollständig und entschied auf Basis der zuvor festgelegten Definition „wiederverwendbares Wissen“, ob das Ticket einen für die Wissensbasis verwertbaren Beitrag enthält.

Die Oberfläche der Anwendung gliedert sich in drei Bereiche, die in den Abbildungen 3.3.2 bis 3.3.4 dokumentiert sind.

Der *Konversationsbereich* (Abbildung 3.3.3) stellt den vollständigen Ticketverlauf nach der in Abschnitt 3.3.1.2 beschriebenen PII-Maskierung dar. Jede Nachricht ist farblich nach Rolle und Sichtbarkeit kodiert:

- **Blau** – Kundennachrichten (Rolle `[CUSTOMER]`, abgeleitet aus `author_id == requester_id`).

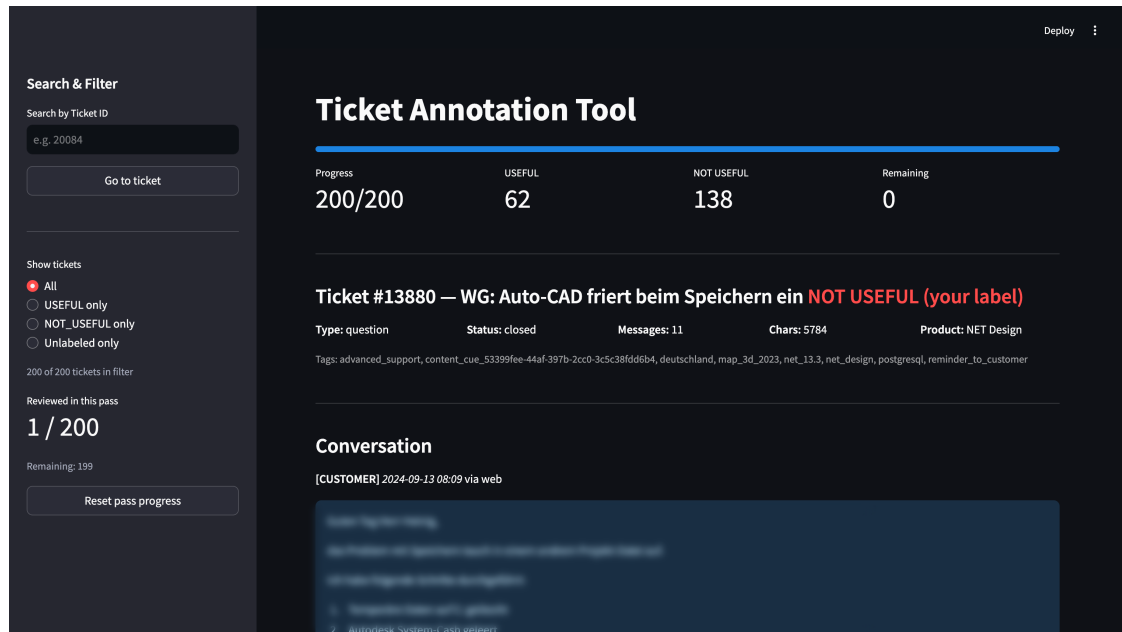


Abbildung 3.3.2.: Streamlit-Annotationsanwendung. Der Kopfbereich zeigt den Titel des aktuell betrachteten Tickets (Nummer + Betreffzeile) sowie die globale Annotationsstatistik: bisherige Fortschrittsanzeige, Anzahl bereits als USEFUL bzw. NOT USEFUL markierter Tickets und verbleibende Anzahl. Die Seitenleiste links bietet eine Volltextsuche nach Ticket-ID sowie Filteroptionen (Alle / USEFUL / NOT USEFUL / Unlabeled).

- **Grün** – öffentliche Support-Antworten (Rolle [SUPPORT], für den Kunden sichtbar).
- **Gelb** – interne Notizen des Support-Teams (Rolle [SUPPORT], `public == false`), die der Kunde nicht zu Gesicht bekommt.

Diese Farbkodierung beschleunigt die Annotation erheblich, weil der Autor auf einen Blick erkennen kann, welche Teile des Threads für den Kunden bestimmt waren und welche nur intern abgesprochen wurden.

Am Ende jedes Tickets befinden sich die beiden zentralen Annotationsbuttons (Abbildung 3.3.4). Ein Klick speichert das gewählte Label, optionale Notizen und einen ISO-Zeitstempel atomar in der Annotationsdatei und navigiert automatisch zum nächsten noch nicht annotierten Ticket der gewählten Filtermenge. Die anschließende Navigationsleiste (*Previous / Skip / Next*) erlaubt die manuelle Rückkehr zu vorherigen Tickets, etwa für Konsistenzprüfungen.

Alle 200 Annotationen sind zu 100 % durch den Autor verifiziert. Da nur ein einzelner Annotator verfügbar war, ist eine Inter-Annotator-Agreement-Bewertung (z. B. Cohen's κ) nicht möglich – dies ist eine bewusst akzeptierte Limitierung der Arbeit, die im Limitationen-Abschnitt 5.9.1 explizit als Bedrohung der internen Validität diskutiert wird. Zur externen Validierung der Annotation wurde zusätzlich eine unabhängige Klassifikation des

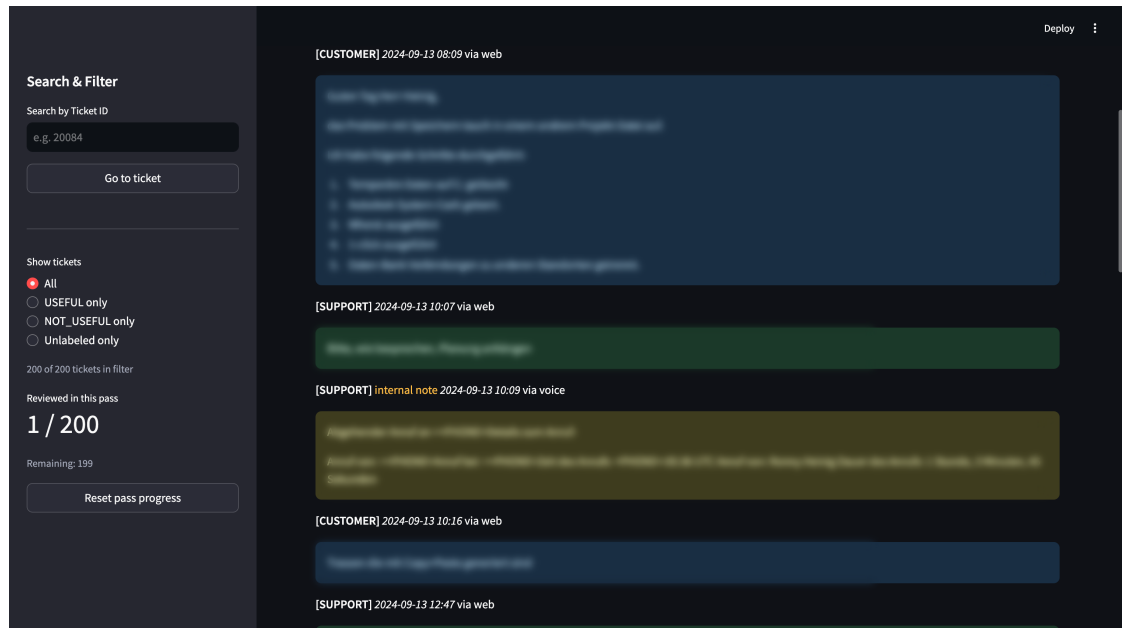


Abbildung 3.3.3.: Konversationsansicht der Annotationsanwendung. Kundennachrichten ([CUSTOMER]) sind blau hinterlegt, öffentliche Support-Antworten grün, interne Support-Notizen („internal note“) gelb. Jede Nachricht trägt einen Header mit Rolle, Anzahl Anhängen, Zeitstempel und Übertragungskanal. Die Reihenfolge ist strikt chronologisch; Standardsignaturen, Begrüßungsfloskeln und Disclaimer wurden bereits durch `clean_body(...)` entfernt, E-Mail-Adressen und Telefonnummern durch `<EMAIL>/<PHONE>` maskiert.

Golden-45-Datensatzes durch sieben kommerzielle Frontier-LLMs durchgeführt (siehe Abschnitt 5.2.12).

Auswertung. Die Auswertung erfolgt mit den klassischen Klassifikationsmetriken Accuracy, Precision, Recall und F1, ergänzt um eine Confusion-Matrix und eine *Bucket-Verteilung* zur Diagnose systematischer Fehlklassifikationsmuster. Die Wahl von Precision als primärer Optimierungsgröße folgt aus der Anforderung an die Wissensbasis: ein einzelner falsch-positiver Ticket beeinträchtigt die Qualität aller späteren Retrieval-Treffer, während falsch-negative Tickets lediglich nicht in den Index gelangen, aber jederzeit nachträglich rehabilitiert werden können.

3.3.1.11. Strukturierte Extraktion zu RAG-Datenunits

Nach Abschluss der Qualitätsfilterung (Abschnitt 3.3.1.3) liegen ca. 5 400 als USEFUL markierte Tickets bereit, deren Inhalte in den FAISS-Index der Wissensbasis aufgenommen werden sollen. Das rohe Ticket-JSON ist für eine semantische Suche jedoch ungeeignet: Es enthält Begrüßungsfloskeln, mehrere Konversationsstränge, redundante Wiederholun-

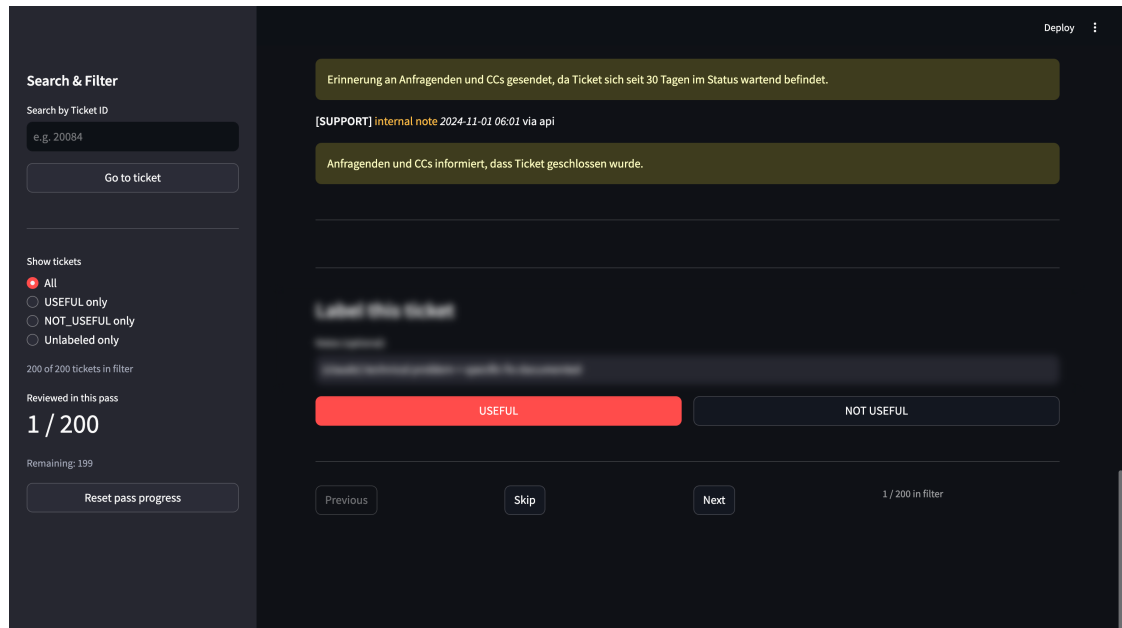


Abbildung 3.3.4.: Annotationsbuttons. Am Ende der Konversationsansicht stehen zwei farblich hervorgehobene Buttons **USEFUL** und **NOT USEFUL** bereit; darüber ein optionales Eingabefeld für Notizen, die zur Entscheidungsbegründung dienen können. Ein Klick speichert die Annotation in `annotations_200.json` und navigiert zum nächsten Ticket der aktiven Filtermenge.

gen und keine maschinenlesbare Trennung von Problem, Diagnose und Lösung. Die zweite Stufe der Verarbeitungspipeline überführt daher jedes **USEFUL**-Ticket in eine oder mehrere strukturierte *Datenunits* – JSON-Objekte mit einheitlichem Feldschema, ein Eintrag pro semantisch eigenständigem Thema des Tickets.

Iterationen v2/v3/v4 (Vorgeschichte). Bevor die aktuell produktive Architektur entstand, wurden drei Vorgängerversionen untersucht: v2 mit explizit getrennten Feldern (`customer_request`, `support_action`, `final_outcome`, `evidence_spans`) und Heuristik-basierter Vorab-Extraktion von Status und Version; v3 mit topic-basierter Zerlegung in Subtopics und narrativem 3-Block-Format; v4 mit Pydantic-erzwungener Schema-Ausgabe über die Bibliothek *Instructor*. Aus diesen Versuchen wurde abgeleitet, dass (i) Topic-Chunking notwendig ist, weil multi-thematische Tickets sonst in einer einzigen, generischen Beschreibung resultieren, (ii) deklarative Schema-Enforcement am LLM-Aufruf entscheidend ist, weil Post-Processing-Reparatur mit ca. 3 % Schema-Verletzungen unzuverlässig blieb, und (iii) eine Halluzinations-Verifikation als separate Stufe nötig ist, weil das Modell trotz Anti-Halluzinations-Anweisungen im Prompt regelmäßig Versionsnummern und Dateinamen erfindet.

Scope der Evaluation. Die in Abschnitt 5.3 dokumentierte quantitative Bewertung der strukturierten Extraktion bezieht sich ausschließlich auf die im Folgenden beschriebene produktive Architektur. Die hier skizzierten Vorgängerversionen v2/v3/v4 sind nicht Gegenstand einer eigenständigen quantitativen Auswertung: Sie wurden im Iterationsverlauf jeweils vollständig durch die nachfolgende Architektur ersetzt und sind im Code-Stand der Arbeit nicht mehr separat lauffähig vorhanden; ihre Funktion in dieser Arbeit liegt in der dokumentierten Begründung der Architektur-Entscheidungen der Produktivversion – Topic-Chunking, deklaratives Schema-Enforcement und separate Halluzinations-Verifikation –, nicht in einem direkten Genauigkeitsvergleich gegen sie. Eine empirische Aufarbeitung der drei Vorgängerversionen wäre methodisch nur durch eine Code-Reanimation der v2/v3/v4-Varianten möglich und verbleibt als Erweiterungsrichtung für zukünftige Arbeiten (vgl. Kapitel 6).

Aktuelle Architektur – drei sequenzielle LLM-Aufrufe. Die produktive Pipeline (`zendesk/stage2_extraction/`, ca. 3 400 Zeilen Python) realisiert die strukturierte Extraktion durch *drei aufeinanderfolgende LLM-Aufrufe pro Chunk* – alle drei mit Grammar-constrained Generation (GNBF-Grammatiken von `llama.cpp` [1], Abschnitt 1.3.7). Eingesetzt wird genau ein lokales LLM (Llama 3.1 8B Q5_K_M), das einmal bei Pipeline-Start geladen wird und alle drei Aufruftypen bedient (`raw_complete` für Stage 1, `chat` für Stage 2 und 3). Ein vollständiger Durchlauf über die 5 411 USEFUL-Tickets benötigt auf dem reinen CPU-Server WEBSRV38 (16-Core, 31 GB RAM, kein GPU) erwartungsgemäß mehrere Tage; die im Produktivlauf tatsächlich gemessene Wall-Time beträgt $\approx 9,7$ Tage (vgl. Abschnitt 5.3.2). Abbildung 3.3.5 zeigt den vollständigen Datenfluss; die folgenden Abschnitte beschreiben jede Stufe im Detail.

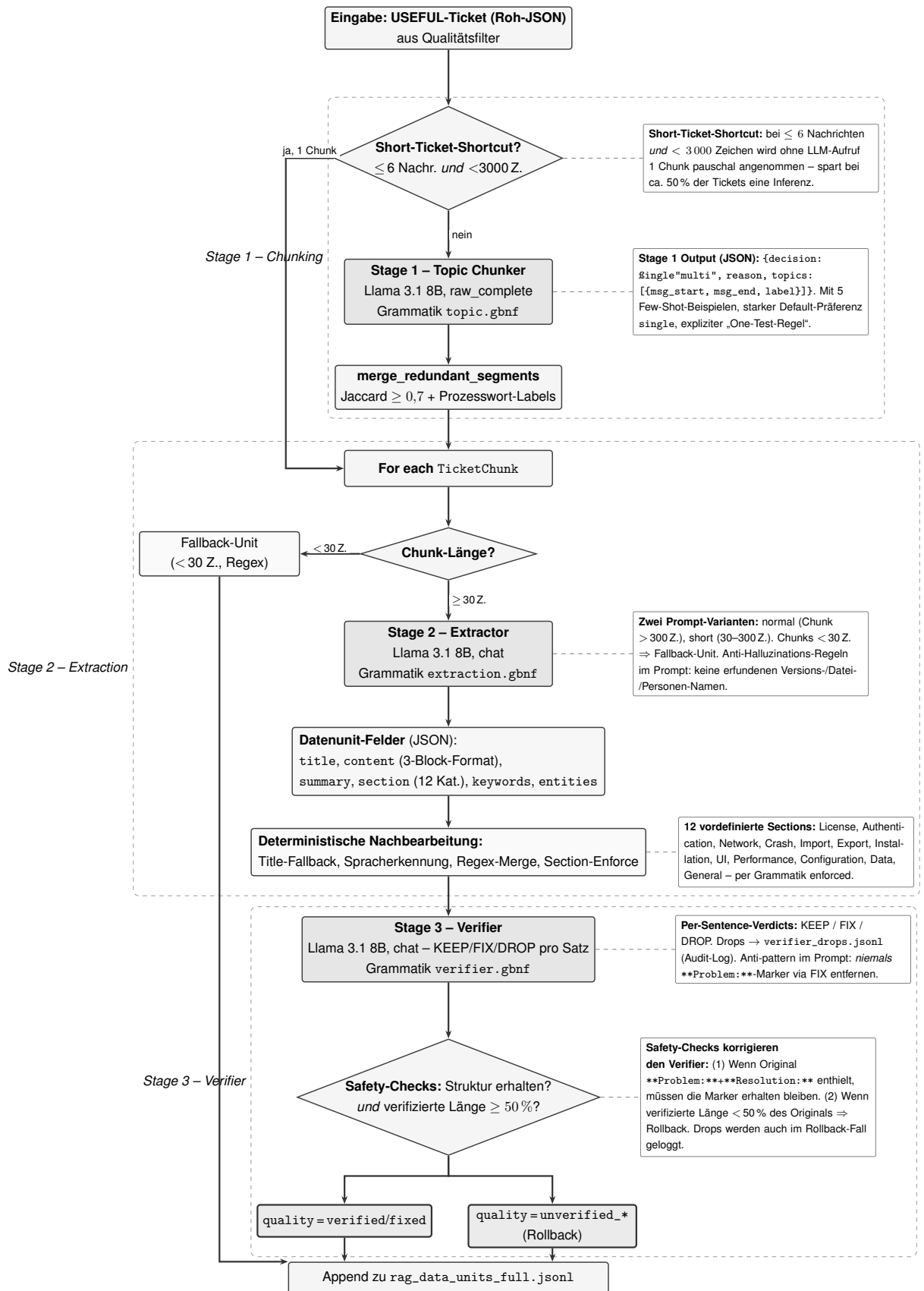


Abbildung 3.3.5.: Architektur der RAG-Datenunit-Extraktionspipeline. Drei sequenzielle, grammar-constrainer LLM-Aufrufe (Topic Chunker, Extractor, Verifier), jeweils mit eigener GBNF-Grammatik; zwei abschließende Safety-Checks (Struktur-Erhaltung und Content-Shrinkage) schützen den Output vor zu aggressiven Verifier-Eingriffen. Die rechte Spalte annotiert jede Stufe mit Implementierungs- und Parameterdetails.

Stage 1 – Topic Chunking. Die erste Stufe (`zendesk/stage2_extraction/src/topic_chunker.py`) entscheidet, ob ein Ticket *ein* zusammenhängendes Thema behandelt oder in mehrere semantisch eigenständige Subthemen zerfallen muss. Sie beginnt mit einem deterministischen *Short-Ticket-Shortcut*: Tickets mit höchstens 6 Nachrichten und insgesamt unter 3000 Zeichen werden ohne LLM-Aufruf direkt als ein einzelner Chunk behandelt. Für alle übrigen Tickets erfolgt ein grammar-constrainer LLM-Aufruf (Grammatik `zendesk/stage2_extraction/src/grammars/topic.gbnf`), der die durchnummerierte Konversation und fünf Few-Shot-Beispiele für typische Mehrfach-Topic-Muster (Bug → Fix, Multi-Symptom-mit-einer-Ursache, Lieferung+Installationsproblem, mehrere Hypothesen, genuin multi-thematisch) enthält. Der System-Prompt enthält eine starke Präferenz für *single* und die explizite „One-Test-Regel“: würde eine einzelne Lösung für alle Nachrichten gelten? Wenn ja, dann *single*. Das Modell antwortet mit:

```
{
  "decision": "single" | "multi",
  "reason": "kurze Begründung",
  "topics": [{"msg_start": 1, "msg_end": 5, "label": "Topic-Label"}]
}
```

Bei *multi* folgt eine deterministische Nachbearbeitung `merge_redundant_segments`, die zwei benachbarte Segmente wieder verschmilzt, wenn ihre Token-Jaccard-Ähnlichkeit $\geq 0,7$ ist oder wenn das Label ein Prozessbegriff („meeting“, „follow-up“) anstelle eines technischen Topics ist. Diese Merge-Stufe kompensiert das beobachtete leichte Over-Splitting des Modells. Resultat: eine Liste von `TicketChunk`-Objekten mit jeweiligem Nachrichtenbereich und optionalem Topic-Label.

Stage 2 – Strukturierte Extraktion. Die zweite Stufe (`zendesk/stage2_extraction/src/extractor.py`) wandelt jeden Chunk in ein vollständiges Datenunit-JSON um. Zwei Prompt-Varianten werden je nach Chunk-Länge gewählt: *normaler Prompt* bei Chunks über 300 Zeichen, ein verkürzter *Short-Prompt* bei 30–300 Zeichen, der trotz Kürze die obligatorische 3-Block-Struktur (**Problem**:...**Resolution**:...) erzwingt. Chunks unter 30 Zeichen überspringen die LLM-Stufe komplett und erzeugen ein Fallback-Unit aus Regex-Signalen.

Ein einzelner grammar-constrainer LLM-Aufruf (Grammatik `zendesk/stage2_extraction/src/grammars/extraction.gbnf`) liefert ein JSON-Objekt mit den sechs Pflichtfeldern:

Tabelle 3.4.: Felder eines RAG-Datenunits (Stage-2-Output)

Feld	Inhalt
title	5–12 Wörter, problem-spezifisch (kein generischer „Issue with X“)
content	Vollständige Beschreibung, drei Blöcke: **Problem:** , **Investigation:** , **Resolution:** , in der Originalsprache der Konversation
summary	1–2-Satz-Überblick
section	Eine von 12 vordefinierten Kategorien (License, Authentication, Network, Crash, Import, Export, Installation, UI, Performance, Configuration, Data, General), durch die Grammatik enforced
keywords	3–6 technische Stichwörter aus dem Rohtext
entities	3–10 benannte technische Entitäten (Produkte, Tabellennamen, Befehle, Versionen); ausdrücklich <i>nicht</i> Projekt-, Kundenfirmen- oder Personennamen

Die Grammatik schließt syntaktische Schema-Verletzungen am Decodier-Schritt aus – das Modell *kann* keinen ungültigen JSON-Wert oder kein Section-Label außerhalb der 12 erlaubten Werte emittieren. Die Anti-Halluzinations-Anweisungen im Prompt (keine erfundenen Versionsnummern, Dateinamen, Personen; bei mehreren Hypothesen ohne bestätigten Fix dies explizit benennen) bleiben weiche Constraint und werden in Stage 3 verifiziert.

Nach der LLM-Antwort durchläuft das Unit eine deterministische Nachbearbeitung: *Title-Fallback* bei generischen Titeln oder Titellängen unter 10 Zeichen (Verwendung des Zendesk-Subject als Ersatz), *Spracherkennung* via deutscher/englischer Stoppwort-Counts (ca. 99% Genauigkeit auf diesem Korpus, ohne LLM), *Regex-Signal-Merge* für Produkte (z. B. „NET Engineering“, „AutoCAD“) in die Keywords und *Section-Enforcement* als letztes Sicherheitsnetz (Fallback auf General bei unzulässigem Wert – die Grammatik sollte dies bereits verhindern).

Stage 3 – Verifier (Halluzinationsprüfung). Die dritte Stufe (zendesk/stage2_extraction/src/verifier.py) prüft, ob jede Behauptung im content-Feld durch die rohe Konversation gedeckt ist. Ein grammar-constrainer LLM-Aufruf (Grammatik zendesk/stage2_extraction/src/grammars/verifier.gbnf) erhält die durchnummerierte Konversation sowie den content-Text und liefert ein Array mit Urteilen pro Satz:

```
[
  {"verdict": "KEEP", "sentence": "...", "fixed": ""},
  {"verdict": "FIX", "sentence": "...", "fixed": "Korrektur"},
  {"verdict": "DROP", "sentence": "...", "fixed": ""}
]
```

Die Pipeline rekonstruiert den content entlang der Urteile: KEEP-Sätze unverändert, FIX-Sätze durch die korrigierte Variante ersetzt, DROP-Sätze entfernt. Alle gelöschten oder korrigierten Sätze werden in einer separaten Datei `verifier_drops.jsonl` mitsamt Begründung protokolliert, um spätere Audits zu ermöglichen.

Im Anschluss greifen zwei *Safety-Checks*, die als kritisches Korrektiv des Verifiers wirken. Sie sind das Ergebnis eines konkreten Iterationszyklus: eine erste Stage-2-Iteration am 2026-05-18 wurde nach 18 Stunden bzw. 9,8 % des Korpus per Stichprobeninspektion gestoppt und zeigte bei Drop-Buckets ≥ 3 einen Strukturverlust von ca. 71 %, insgesamt ca. 59 % verlorene Marker. Die nachfolgend hinzugefügten Safety-Checks senken den erwarteten Strukturverlust auf $< 10\%$:

1. **Struktur-Erhaltung:** Wenn der ursprüngliche content die Marker `**Problem:**` und `**Resolution:**` oder `**Investigation:**` enthielt, müssen sie im verifizierten Output erhalten sein – sonst wird die Verifier-Version verworfen, die Extractor-Version beibehalten und `quality=unverified_structure_lost` gesetzt.
2. **Content-Shrinkage:** Wenn die verifizierte Version unter 50 % der Länge des Originals liegt, wurde der Verifier vermutlich zu aggressiv – die Verifier-Version wird verworfen, die Extractor-Version beibehalten, `quality=unverified_too_aggressive`.

Auch bei Rollback werden die ursprünglichen Drop-Vorschläge weiterhin protokolliert; so bleibt ein vollständiger Audit-Trail.

Quality-Flag-System. Jedes Unit trägt im `metadata.quality`-Feld einen von sechs Werten (Tabelle 3.5). Die nachgelagerte Retrieval-Stufe kann diese Werte als Gewichtung in die Relevanz-Score-Berechnung einbringen, um Units niedrigerer Vertrauenswürdigkeit zu de-prioritisieren, ohne sie vollständig auszuschließen.

Tabelle 3.5.: Quality-Flag-Werte des RAG-Datenunit-Schemas

Wert	Bedeutung	Retrieval-Gewicht
verified	Verifier hat 0 DROPs/FIXes produziert; Extractor-Output ohne erkennbare Halluzinationen.	1,00
fixed	Verifier hat ≥ 1 Änderung vorgenommen; beide Safety-Checks bestanden, Struktur erhalten.	0,90
unverified_structure_lost	Verifier wollte Problem-/Resolution-Marker entfernen; Rollback auf Extractor-Output.	0,85
unverified_too_aggressive	Verifier wollte mehr als 50 % des Inhalts löschen; Rollback auf Extractor-Output.	0,85
unverified	Verifier-Aufruf selbst fehlgeschlagen (selten); Extractor-Output unverändert.	0,70
fallback	Chunk unter 30 Zeichen, keine LLM-Stufe durchlaufen; Unit aus Regex-Signalen + Rohtext gebaut.	0,40

Quality-Gate vor Full-Run. Vor jedem Full-Run über die ca. 5400 USEFUL-Tickets läuft die Pipeline zuerst auf einem fixierten Datensatz aus 23 hand-kuratierten Golden-Tickets (`zendesk/stage2_extraction/data/test_golden_useful.json`). Anschließend liest `zendesk/stage2_extraction/src/snapshot.py` das produzierte JSONL sowie die hand-erstellte Referenzdatei `zendesk/stage2_extraction/data/golden_data_units.json` (25 Referenz-Units) und berechnet sieben deterministische Metriken. Das Skript `zendesk/stage2_extraction/src/gate_check.py` prüft diese gegen feste Schwellen (Tabelle 3.6):

Tabelle 3.6.: Quality-Gate-Schwellen vor dem Full-Run

Metrik	Schwelle	Bedeutung
units	≥ 23	Mindestens ein Unit pro Golden-Ticket
errors	≤ 1	Maximal ein Pipeline-Fehler im Lauf
fallback_rate	$\leq 10\%$	Anteil fallback-Units (Chunks zu klein)
section_match	$\geq 0,55$	Übereinstimmung der section-Klassifikation mit Referenz
split_recall	$\geq 0,50$	Multi-Topic-Tickets müssen erkannt werden
entity_recall_avg	$\geq 0,30$	Wiederfindung benannter Entitäten (nur Advisory)
general_rate	$\leq 10\%$	Anteil Units mit Fallback-Section General

Sind alle Hard-Schwellen bestanden, startet der Wrapper `run_test_then_full.sh` automatisch den Full-Run; bei Verletzung einer Schwelle endet der Wrapper, und der Operator muss die Regression untersuchen. Dieses Gate ist das Sicherheitsnetz, das verhindert, dass tagelange Server-Compute auf einer Regression verschwendet wird.

Resume-Safety. Beide Läufe (Gate auf 23 und Full-Run auf 5 411) schreiben ihre Ausgabe inkrementell als JSONL mit `flush` nach jeder Zeile. Bei einem Server-Reboot oder einem `kill`-Signal genügt ein erneuter Start desselben Kommandos – die Funktion `load_done_ids()` liest die existierende JSONL-Datei und sammelt alle Ticket-IDs, deren Unit gültigen Inhalt von mindestens 30 Zeichen aufweist; alles als Fehler markierte wird nicht als erledigt gezählt und beim erneuten Lauf neu verarbeitet. So überlebt die mehrtägige Verarbeitung auch instabile Infrastruktur ohne Verlust mehrerer Tage Compute.

3.3.2. Confluence – Analyse und bewusste Nicht-Implementierung

Korpus-Eigenschaften. Das unternehmensinterne Confluence dient als zentrales Wiki und enthält technische Dokumentationen, Anleitungen sowie Prozessbeschreibungen, die in einer hierarchischen Ordnerstruktur (Spaces → Pages → Sub-Pages) organisiert sind. Inhaltlich überschneidet sich der Korpus partiell mit dem in Abschnitt 3.3.5 beschriebenen Help Center, dessen redaktionell verfasste Artikel die für den Support relevanten Soll-Zustände bereits in kuratierter Form abdecken.

Zugriffsanalyse. Eine Sondierung der Confluence REST API (`/wiki/rest/api/content`) bestätigte die technische Erschließbarkeit: die Endpunkte liefern paginierte Seitenlisten samt Titel, HTML-Body, Labels, Versionsmetadaten und einem Ancestors-Array, aus dem sich die hierarchische Ordnerstruktur rekonstruieren lässt. Eine vollständige

Integration hätte zusätzlich ein HTML-zu-Text-Parsing, eine Behandlung von Confluence-spezifischen Makros (z. B. Code-Blöcke, Tabellen, eingebundene Diagramme) sowie eine Inkrementelle Synchronisation mit dem zentralen Index erfordert.

Resultierende Entscheidung. Auf Grundlage dieser Analyse wurde der Aufwand einer Integration für die vorliegende Arbeit als unverhältnismäßig hoch eingestuft, insbesondere im Verhältnis zum erwarteten Mehrwert über den bereits abgedeckten Korpus aus Zendesk-Tickets, HC-Artikeln und der TKI-Pr-KB. Confluence verbleibt damit als prioritäre Erweiterungsrichtung für zukünftige Arbeiten (vgl. Kapitel 6).

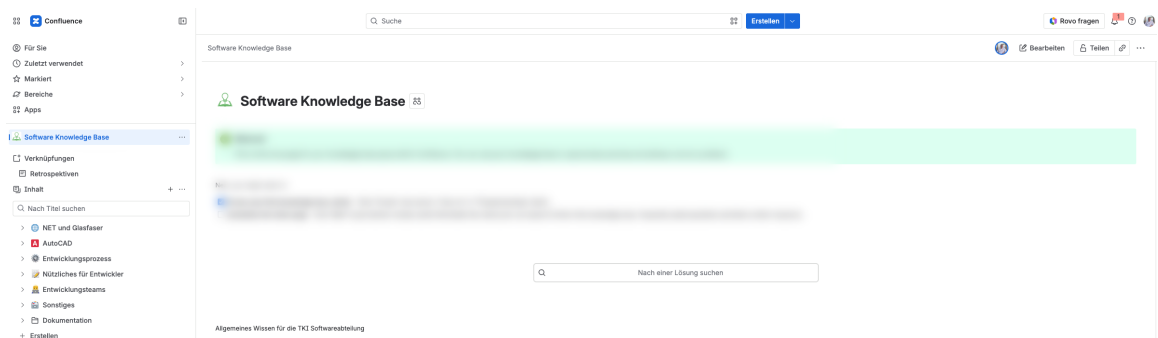


Abbildung 3.3.6.: Confluence *Software Knowledge Base* der TKI-Softwareabteilung in der Sondierungsansicht aus dem persönlichen Account des Verfassers. Die linke Seitenleiste zeigt die hierarchische Strukturierung des Wikis nach Themenbereichen (NET und Glasfaser, AutoCAD, Entwicklungsprozess, Dokumentation u. a.). Inhaltliche Felder sind aus Vertraulichkeitsgründen ausgegraut.

3.3.3. Jira – Analyse und bewusste Nicht-Implementierung

Korpus-Eigenschaften. Jira enthält technische Issues, Bug-Reports und Feature-Requests aus dem internen Entwicklungs- und Support-Prozess. Im Unterschied zu Zendesk-Tickets, die kundeninitiierte Anfragen mit Symptom-Lösung-Charakter abbilden, dokumentieren Jira-Issues primär den Implementierungs- und Bug-Tracking-Zustand der Produkte und sind damit für die Support-Praxis nur indirekt relevant.

Strukturanalyse: Atlassian Document Format (ADF). Eine Sondierung der Jira REST API ergab, dass Beschreibungen und Kommentare nicht als reiner Text, sondern im Atlassian Document Format (ADF) gespeichert sind – einer verschachtelten JSON-Struktur, deren Knotentypen (Paragraphen, Listen, Code-Blöcke, Erwähnungen, eingebettete Bilder) eine eigene rekursive Parser-Implementierung erfordern. Eine prototypische Extraktion zeigte, dass der reine Textinhalt vergleichsweise einfach extrahierbar ist, jedoch semantische

Strukturinformationen (etwa Code-vs-Prosa-Unterscheidung) ohne weitere Logik verloren gehen.

Resultierende Entscheidung. Aufgrund der begrenzten Support-Relevanz des Korpus und des nichttrivialen Parsing-Aufwands wurde Jira im Rahmen dieser Arbeit nicht in die produktive Pipeline aufgenommen. Eine zukünftige Integration ist im Ausblick (Kapitel 6) skizziert.

	Typ	Schlüssel	Zusammenfassung	Status	Kommentare	Zugewiesene Person	Fälligkeitsdatum	Priorität	Stichwörter	Erstellt	Aktualisiert	Autor
☐	Support	SUPPORT-1	...	FEHLENDE	1 Kommentar hinz.			Normal	Kundenanpassung	20. Feb. 2019	9. Feb. 2022	
☐	Support	SUPPORT-2	...	FEHLENDE	1 Kommentar hinz.			Normal		29. Apr. 2019	5. Juni 2019	
☐	Support	SUPPORT-3	...	FEHLENDE	4 Kommentare			Normal		30. Apr. 2019	28. Juni 2019	
☐	Support	SUPPORT-4	...	FEHLENDE	1 Kommentar			Höchste		3. Mai 2019	14. Aug. 2019	
☐	Support	SUPPORT-5	...	FEHLENDE	3 Kommentare			Höchste	+ Stichwort hinzufügen	8. Mai 2019	28. Juni 2019	
☐	Support	SUPPORT-6	...	FEHLENDE	1 Kommentar hinz.		24. Mai 2019	Normal		15. Mai 2019	14. Aug. 2019	
☐	Support	SUPPORT-7	...	FEHLENDE	6 Kommentare			Normal		23. Mai 2019	6. Sept. 2019	
☐	Support	SUPPORT-8	...	FEHLENDE	5 Kommentare			Hoch	SASAG	5. Juni 2019	13. Dez. 2019	
☐	Support	SUPPORT-9	...	FEHLENDE	3 Kommentare			Normal		7. Juni 2019	14. Aug. 2019	
☐	Support	SUPPORT-10	...	FEHLENDE	9 Kommentare			Hoch	lufemin	13. Juni 2019	10. Juli 2019	
☐	Support	SUPPORT-11	...	FEHLENDE	1 Kommentar			Normal	Aling	18. Juni 2019	14. Aug. 2019	
☐	Support	SUPPORT-12	...	FEHLENDE	6 Kommentare			Normal		19. Juni 2019	3. Juli 2019	
☐	Support	SUPPORT-13	...	FEHLENDE	3 Kommentare			Normal		19. Juni 2019	28. Aug. 2019	
☐	Support	SUPPORT-14	...	FEHLENDE	3 Kommentare			Normal		19. Juni 2019	14. Aug. 2019	
☐	Support	SUPPORT-15	...	FEHLENDE	1 Kommentar			Hoch		20. Juni 2019	14. Aug. 2019	
☐	Support	SUPPORT-16	...	FEHLENDE	7 Kommentare			Normal		20. Juni 2019	14. Apr. 2020	
☐	Support	SUPPORT-17	...	FEHLENDE	4 Kommentare			Hoch	Migration	20. Juni 2019	14. Aug. 2019	

Abbildung 3.3.7.: Jira-Projekt *Support* der TKI in der tabellarischen Issue-Übersicht aus dem persönlichen Account des Verfassers. Sichtbar sind die für die Strukturanalyse relevanten Spalten (Typ, Schlüssel, Zusammenfassung, Status, Kommentare, Erst- und Aktualisierungsdatum). Inhaltliche Felder sind aus Vertraulichkeitsgründen ausgegraut.

3.3.4. Slack – Analyse und bewusste Nicht-Implementierung

Korpus-Eigenschaften. Slack-Kanäle enthalten informelle Support- und Entwicklungskommunikation, die mitunter wertvolles Kontextwissen aus Ad-hoc-Abstimmungen abbildet, das in formellen Ticketsystemen nicht erfasst ist. Im Gegensatz zu Zendesk-Tickets und HC-Artikeln fehlt jedoch eine kuratierte Struktur: Nachrichten sind chronologisch verkettet, häufig in mehrere Threads zerstreut und ohne explizite Zuordnung zu Symptom oder Lösung.

Zugriffsanalyse. Eine Sondierung der Slack Web API über das offizielle `slack_sdk` bestätigte die Erschließbarkeit von Kanal-Nachrichten und Thread-Antworten via Bot-Token mit den erforderlichen OAuth-Scopes. Die Nachrichten umfassen Text, Autor, Zeitstempel, Thread-Zugehörigkeit, Reaktionen und Dateianhänge.

Resultierende Entscheidung. Die niedrige Signal-Dichte des informellen Korpus – ohne explizite Symptom-Lösung-Paare – in Kombination mit dem zeitlichen Rahmen der Bachelorarbeit führte zu der Entscheidung, Slack nicht in die produktive Pipeline aufzunehmen. Eine sinnvolle Integration würde zusätzlich eine Filter- bzw. Klassifikationslogik erfordern, um relevante Konversationen von alltäglicher Kommunikation zu trennen; dieser Vorverarbeitungsschritt wird im Ausblick (Kapitel 6) als eigenständige Erweiterungsrichtung skizziert.

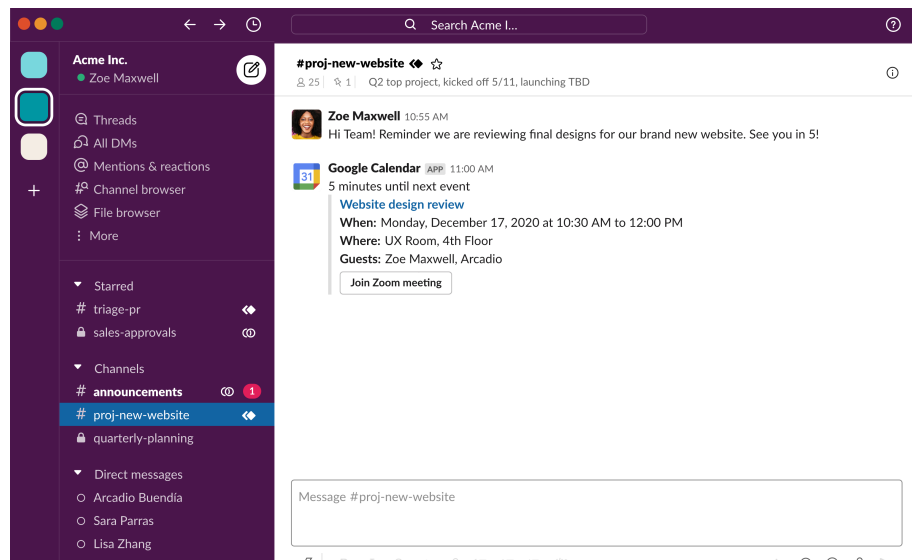


Abbildung 3.3.8.: Beispielhafte Darstellung eines Slack-Workspaces mit Kanalliste (Channels), Direktnachrichten und einem ausgewählten Kanal mit Threads und eingebetteten App-Karten. Abbildung aus dem offiziellen Slack-Produktmaterial; eigene Workspace-Screenshots wurden aus Vertraulichkeitsgründen nicht in die Arbeit aufgenommen.

3.3.5. Zendesk Help Center TKI (HC)

Das firmeneigene HC unter <https://help.tki-chemnitz.de> ist die öffentliche Wissensdatenbank des Unternehmens und ergänzt die Ticket-Quellen aus Abschnitt 3.3.1 um *kuratierte*, vom Support-Team selbst verfasste Anleitungen, Release Notes und Vorlagendokumente. Anders als bei Tickets, die ad-hoc-Lösungen aus realer Kommunikation enthalten, sind HC-Artikel redaktionell verfasst und sollen einen Soll-Zustand für den Anwender beschreiben – damit decken sie typische Anfragen zu Bedienoberfläche, Konfiguration und Versionsänderungen ab, ohne dass diese erneut im Ticket-Kanal erscheinen müssen.

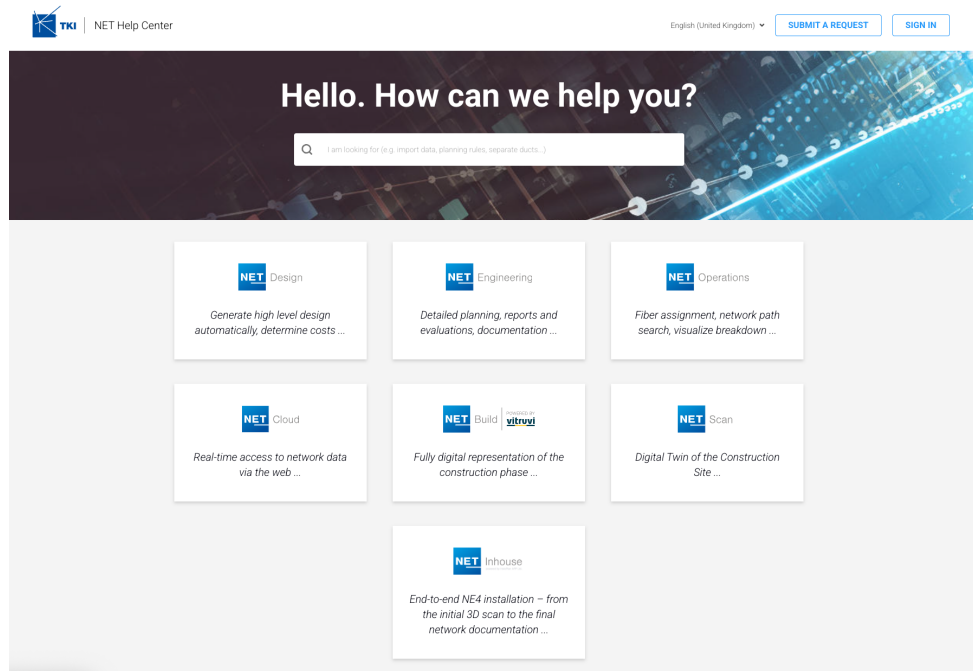


Abbildung 3.3.9.: Öffentlich zugängliche Einstiegsseite des TKI Help Centers unter <https://help.tki-chemnitz.de>. Die Kacheln strukturieren die Wissensbasis nach den NET-Produktmodulen (NET Design, NET Engineering, NET Operations, NET Cloud, NET Build, NET Scan, NET Inhouse) und bilden damit dieselbe Modul-Taxonomie ab, die auch in der TKI-Pr-KB (Abschnitt 3.3.7) verwendet wird.

Korpus-Überblick. Eine vollständige Sondierung der HC-API zum Zeitpunkt der letzten Indexierung ergab 878 Artikel in 196 Sektionen, die unter elf Top-Level-Kategorien gegliedert sind. Tabelle 3.7 zeigt die Verteilung über die fünf größten Kategorien – zwei davon („NET Build“ und „NET Engineering“) zusammen knapp 60 % des Korpus, ergänzt um die Release-Notes-Kategorie, in der jede Produktversion ein eigener Artikel mit hierarchischer Unterstruktur ist.

Kategorie	Artikel	Chunks (nach Pipeline)
NET Build	275	1.267
NET Engineering	240	1.379
NET Design	127	365
Release Notes & Downloads	78	2.632
Allgemeine Grundlagen	41	195
übrige sechs Kategorien	117	600
Summe	878	6.438

Tabelle 3.7.: HC-Korpus nach Kategorien zum Zeitpunkt der letzten Indexierung. Die Spalte *Chunks* berücksichtigt sämtliche drei Subtypen (Artikel-HTML, PDF-Anhänge, Office-Vorlagen).

Drei Subtypen, ein Schema. Jeder Artikel kann beliebig viele Anhänge tragen, die im HC-Modell als eigenständige Dateien geliefert werden. Die letzte Sondierung zählte 8.015 Anhangs-Referenzen, deren MIME-Typen sich grob in drei Klassen zerfallen: 7.883 Bilder (PNG/JPEG), 120 PDFs (überwiegend Release-Notes-Zusammenfassungen, Quick-Start-Guides und Datenblätter) und 4 Office-Dokumente (DOCX-Vorlagen für Bauprotokolle plus ein einzelnes Legacy-DOC). Daraus folgen drei zu indexierende Inhaltsklassen mit jeweils eigener Extraktionsstrategie – `hc_article` (HTML-Artikelkörper), `hc_pdf` (PDF-Anhänge) und `hc_docx` (Office-Vorlagen) – die in Abschnitt 4.2.6 detailliert beschrieben sind. Alle drei Klassen münden in das gleiche unified-schema-Format wie Zendesk-Units (Abschnitt 3.3.1) und Vitruvi-Chunks (Abschnitt 3.3.6), sodass sie ohne weitere Konvertierung in den gemeinsamen Index einfließen.

Kategorie als Produkt-Label. Die elf Top-Level-Kategorien des HC korrespondieren in sieben Fällen direkt mit einem Produkt der TKI NET Suite („NET Engineering“, „NET Build“, „NET Design“, „NET Cloud“, „NET Scan“, „NET Inhouse“, „NET Operations“). Für diese Fälle wird der Kategorie-Name eins-zu-eins in die `product`-Spalte des unified-Schemas übernommen, sodass der API-seitige Produkt-Filter (vgl. Anforderung FA-3) HC-Chunks ebenso adressierbar macht wie Zendesk-Tickets oder TKI-Produkt-Dokumente. Die übrigen vier Kategorien (*Release Notes & Downloads*, *Administration*, *Allgemeine Grundlagen*, *Häufige Fragen*, *Tipps & Tricks*) sind inhaltlich produktübergreifend und behalten daher absichtlich `product = null`, um irreführende Filterung zu vermeiden. Die Backfill-Regel selbst lebt als reine Lookup-Tabelle in `zendesk_help_center/src/ingest/base.py::product_for_category()` und wird von allen drei Chunkern aufgerufen, was die Konvention zwischen den Subtypen synchron hält. In der vorliegenden Indexierung erhalten so

3.367 der 6.438 HC-Chunks (52 %) ein verlässliches Produkt-Etikett.

Bewusst aus dieser Iteration ausgeschlossen: Bild-Annotationen. Die 7.883 PNG- und JPEG-Anhänge – überwiegend UI-Screenshots, Konfigurations-Dialoge und Netzdiagramme – werden in der vorliegenden Arbeit *nicht* extrahiert. Eine Auswertung des HTML-`alt`-Attributs ergab, dass dieses in 96,7 % der Fälle nur autogenerierte Dateinamen (etwa `mceclip0.png`) enthält und damit für das Retrieval wertlos ist. Die methodisch korrekte Lösung wäre eine Vision-Language-Model-basierte Bildbeschreibung, die für jedes Bild eine ein- bis zweisätzliche Caption erzeugt und diese vor dem Chunking in den Artikel-Körper einfügt. Da diese Erweiterung eine eigene Validierungsschleife (Caption-Qualität, Latenz, Modellauswahl) erfordert und mehrere Stunden Inferenz auf der vorhandenen Hardware nach sich zieht, ist sie als Folgearbeit dokumentiert und in der Architektur (Abschnitt 4.2.6) als drop-in-erweiterbarer Schritt vor dem HTML-Chunker vorgesehen.

3.3.6. Externe Wissensquellen: Vitruvi

3.3.6.1. Motivation und Quellen-Übersicht

Neben den firmeninternen Quellen ist die Support-Praxis des betrachteten Unternehmens auf das Wissen über den Drittanbieter-Service Vitruvi angewiesen, da dieser eng in die hauseigenen Produkte integriert ist. Eine vom Support-Team durchgeführte Erhebung ergab, dass ein nicht zu vernachlässigender Anteil eingehender Anfragen Grundlagenwissen zu Vitruvi voraussetzt – typischerweise zu API-Verhalten, Datenmodell oder Vertragsrahmen. Da dieses Wissen außerhalb des firmeneigenen Confluence und Zendesk liegt, wurden drei externe Quellen identifiziert und in das Retrieval-System integriert.

1. **Vitruvi Developer-Portal** (<https://developers.vitruvi.cc>) [49]: öffentlich zugängliche Entwicklerdokumentation, intern aufgeteilt in einen Konzept-Bereich (`/docs`) und eine OpenAPI-Referenz (`/reference`).
2. **Vitruvi Customer Terms of Service (PDF v4.0)**: vom Support-Team bereitgestelltes Vertragsdokument mit 37 Seiten, das die rechtlichen und funktionalen Rahmenbedingungen des Services definiert.
3. **Vitruvi Help Center (Zendesk Knowledge Base)** (<https://vitruvi.zendesk.com/hc/en-us>) [50]: geschlossene, authentifizierungspflichtige Wissensdatenbank mit Schulungsartikeln und Videos. Diese Quelle wurde analysiert, in Abschnitt 3.3.6.4 dokumentiert, jedoch im Rahmen dieser Arbeit nicht produktiv eingebunden.

3.3.6.2. Vitruvi Developer-Portal

Das Vitruvi Developer-Portal wird auf der Plattform *ReadMe.io* [42] gehostet und unterteilt sich in zwei Sektoren, die unterschiedliche Inhalte und damit unterschiedliche Chunking-Strategien erfordern. Abbildung 3.3.10 zeigt die Navigations-Aufteilung der Startseite, in der die Hauptbereiche *Documentation* und *API Reference* klar voneinander getrennt sind.

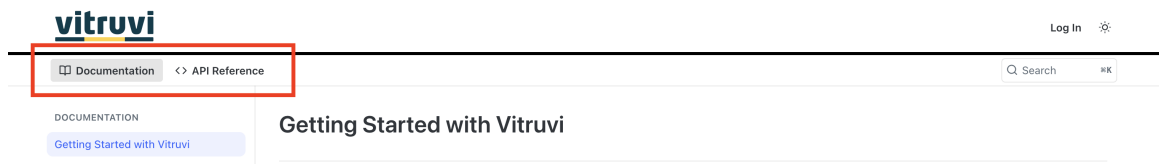


Abbildung 3.3.10.: Aufteilung des Vitruvi Developer-Portals (developers.vitruvi.cc) in die beiden Sektoren *Documentation* (Konzept-Guides, in dieser Arbeit als `/docs` referenziert) und *API Reference* (OpenAPI-Endpoints, `/reference`). Die rote Markierung verdeutlicht die strukturelle Zweiteilung, die für die Wahl unterschiedlicher Chunking-Strategien ausschlaggebend ist.

Konzept-Guides (`/docs`). Dieser Sektor enthält konzeptionelle Anleitungen, Tutorials und Übersichtsartikel in Markdown-Format. Die Inhalte sind in Hierarchie aus Hauptseite und Unterseiten organisiert; einzelne Artikel enthalten Code-Beispiele in fenced Code-Blöcken. Eine Besonderheit der Plattform ist, dass die offizielle `sitemap.xml` lediglich vier der insgesamt 25 Guides auflistet; die übrigen Seiten sind nur über die HTML-Navigationsleiste auffindbar. Aus diesem Grund kombiniert das System drei Discovery-Mechanismen: (i) Sitemap-Parsing, (ii) Auswertung der inoffiziellen `/llms.txt`-Datei und (iii) als Fallback eine regex-basierte Extraktion aller `href`-Attribute aus der HTML-Sidebar. Das Chunking erfolgt mit dem *HybridChunker* der Docling-Bibliothek [23], der Markdown-Heading-aware segmentiert, Sibling-Sections unter gleichem Heading mergt (`merge_peers=True`) und ein Token-Limit von 512 Tokens je Chunk respektiert. Code-Blöcke bleiben dabei stets als zusammenhängende Einheit erhalten.

API-Referenz (`/reference`). Dieser Sektor besteht aus 1.059 Endpoint-Beschreibungen, die als OpenAPI 3.0.3-konforme JSON-Spezifikationen [40] in fenced Code-Blöcken eingebettet sind. Eine direkte Indexierung des Roh-JSON wäre für das Retrieval ungeeignet, da typische Nutzeranfragen wie „How do I create an AI agent log?“ lexikalisch und semantisch nicht zu einer Schlüssel-Wert-Repräsentation passen. Stattdessen wird jeder Endpoint vor der Embedding-Erzeugung in ein natürlichsprachliches Profil überführt (*Natural Language Rendering*), das HTTP-Methode, Pfad, Operations-ID, eine textuelle Beschreibung sowie eine erläuterte Auflistung von Pfad-, Query- und Body-Parametern, Response-Codes und Auth-Schemata enthält. Pro Endpoint entsteht so ein in sich geschlossener Chunk.

3.3.6.3. Vitruvi Customer Terms of Service (PDF)

Die vom Support-Team bereitgestellte Vertragsversion v4.0 ist ein 37-seitiges juristisches Dokument mit ausgeprägter dreistufiger Klausel-Hierarchie: nummerierte Top-Klauseln (z. B. „5 SOFTWARE SUPPORT“), Sub-Klauseln („5.7“) und alphabetisch aufgezählte Unterpunkte („(a)“, „(b)“). Eine generische Chunking-Strategie würde diese Hierarchie verletzen und Klauseln willkürlich trennen, was bei juristischen Texten zu inhaltlich unvollständigen Treffern führt. Abbildung 3.3.11 zeigt die Titelseite des Dokuments.

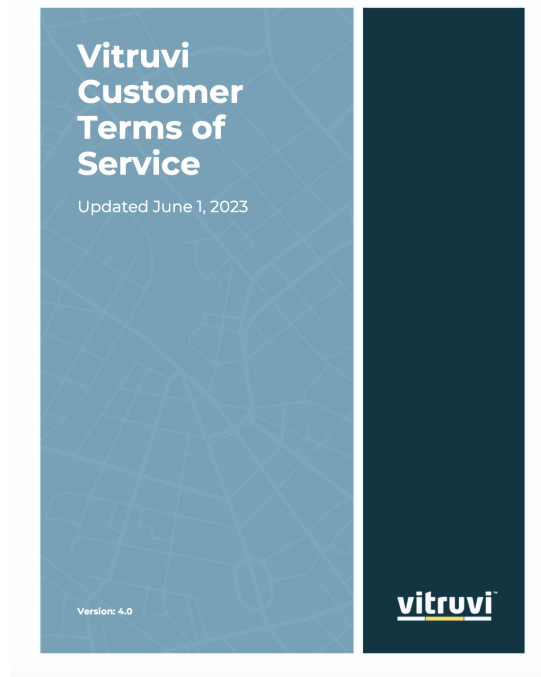


Abbildung 3.3.11.: Titelseite der vom Support-Team bereitgestellten Vitruvi Customer Terms of Service v4.0. Das 37-seitige Dokument bildet die Grundlage für rechtliche und funktionale Rahmenbedingungen des Vitruvi-Services und wird mit einem eigens entwickelten Clause-Aware-Chunker verarbeitet.

Aus diesem Grund wurde ein eigener *Clause-Aware-Chunker* implementiert, der die Hierarchie über drei reguläre Ausdrücke (`^\\d{1,2}\\s+[A-Z]{3,}`, `^\\d+(\\.\\d+)+`, `^\\(\\w\\)`) detektiert und die Pack-Strategie an den Klausel-Grenzen ausrichtet: Innerhalb des 512-Token-Limits werden möglichst viele Sub-Klauseln zu einem Chunk zusammengefasst, der jedoch nie über eine Top-Klausel-Grenze hinweg packt. Definitionen aus Abschnitt 1 werden als eigenständige Chunks gespeichert, und jeder Chunk erhält ein Breadcrumb-Präfix der Form „Section 5 > 5.7“ sowie Metadaten zu Seitenbereich und Klausel-Typ (*definition* / *clause* / *intro*). Zusätzlich extrahiert Docling Tabellen aus dem PDF und hängt sie als GitHub-Flavored-Markdown an die entsprechenden Chunks an.

3.3.6.4. Vitruvi Help Center – Analyse und bewusste Nicht-Implementierung

Das Vitruvi Help Center stellt nach Aussage des Support-Teams die inhaltsreichste Wissensquelle dar, da es Schulungsartikel, Screenshots und Lernvideos zentral bündelt. Eine Integration wurde geprüft; die Analyse führte aufgrund der dabei aufgedeckten Aufwandssstruktur jedoch zu einer expliziten Verschiebung in den Ausblick (Kapitel 6). Die Entscheidung wird im Folgenden begründet, weil sie methodisch ebenso relevant ist wie die implementierten Quellen.

Abbildung 3.3.12 zeigt die Startseite des Vitruvi Help Center nach erfolgter Authentifizierung. Inhaltlich relevant ist hierbei nur die rot markierte Sektion *Knowledge Base*, in der die fachlichen Schulungsartikel zusammengefasst sind. Die direkt darunter sichtbare Sektion *API Documentation* ist lediglich ein Verweis auf das in Abschnitt 3.3.6.2 beschriebene Developer-Portal und wird daher nicht erneut als eigenständige Quelle behandelt.

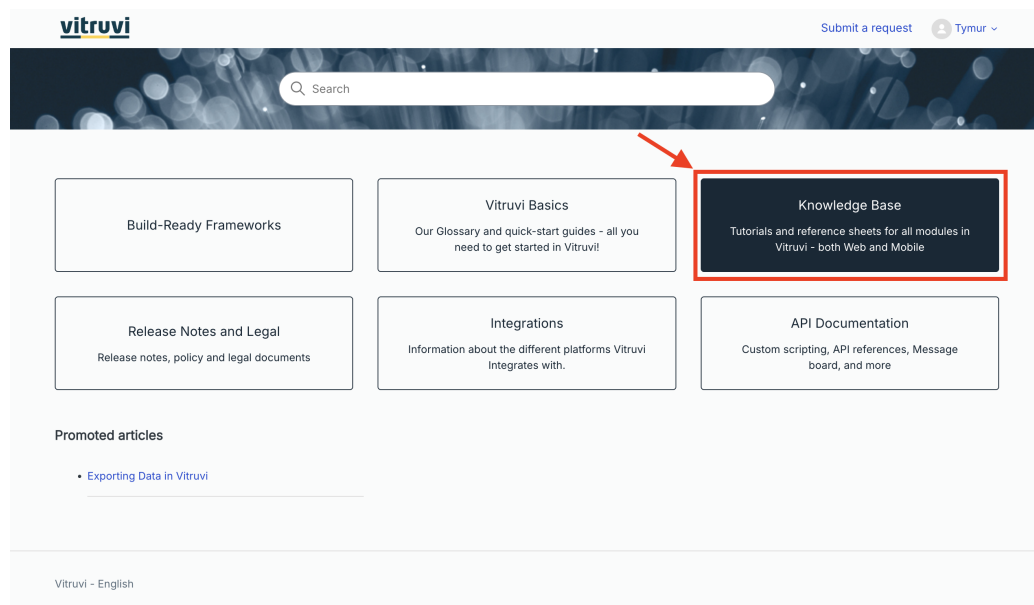


Abbildung 3.3.12.: Startseite des Vitruvi Help Center (vitruvi.zendesk.com) nach erfolgreicher Authentifizierung. Inhaltlich relevant für die vorliegende Analyse ist die rot markierte Sektion *Knowledge Base*; die Sektion *API Documentation* verweist auf das bereits verarbeitete Developer-Portal.

Authentifizierung. Eine Live-Sondierung der API-Endpunkte von vitruvi.zendesk.com ergab, dass das Help Center weder anonym (HTTP 403 bereits auf der Startseite) noch über klassische API-Tokens des bestehenden Zendesk-Accounts (HTTP 401) zugänglich ist. Lediglich *End-User Password Basic Auth* mit den persönlichen Zugangsdaten eines berechtigten Nutzers liefert eine erfolgreiche Authentifizierung (HTTP 200). Diese Authentifizierungsform ist fragil (bricht bei Passwortwechsel) und sicherheitstechnisch ungünstiger als ein API-Token. Die langfristig saubere Lösung – Anforderung eines API-Tokens vom

Instanz-Eigentümer Vitruvi – liegt außerhalb des Einflussbereichs des Bachelorprojekts.

Korpus-Eigenschaften. Eine Bestandsaufnahme über alle KB-Artikel ergab folgendes Bild: 158 publizierte Artikel in 11 Sektionen, ausschließlich englischsprachig, gesamt ca. 574 KB reiner Textinhalt nach HTML-Stripping (entspricht ≈ 144.000 Tokens). Bemerkenswert ist, dass 85 % der Artikel mindestens ein eingebettetes Bild enthalten – insgesamt 1.146 Bilder mit einer extrapolierten Gesamtgröße von rund 260 MB – sowie 6 Vimeo-Videos in 6 Artikeln. Tabelle 3.8 fasst die Sektions-Verteilung zusammen.

Tabelle 3.8.: Verteilung der 158 KB-Artikel über die 11 Sektionen des Vitruvi Help Centers (Messstand: 28. Mai 2026).

Sektion	Artikel	Bilder	Text (KB)
Work Management	29	198	102
Field Reporting	29	270	100
Engineering	28	201	86
General	23	133	89
Administration	18	125	104
Network Operator Configuration	12	96	58
Invoicing	8	57	18
Workflow Overviews	4	0	15
Tickets	4	27	12
A55 Process (UK only)	2	35	5
Materials	1	4	2
Summe	158	1.146	591

Eine visuelle Verifikation der in Tabelle 3.8 aufgeführten Sektionen erfolgt anhand der tatsächlichen Help-Center-Oberfläche (Abbildung 3.3.13). Die elf rot markierten Sektions-Boxen entsprechen exakt den elf Einträgen der Tabelle, was die korrekte vollständige Erfassung des Korpus während der API-Sondierung bestätigt.

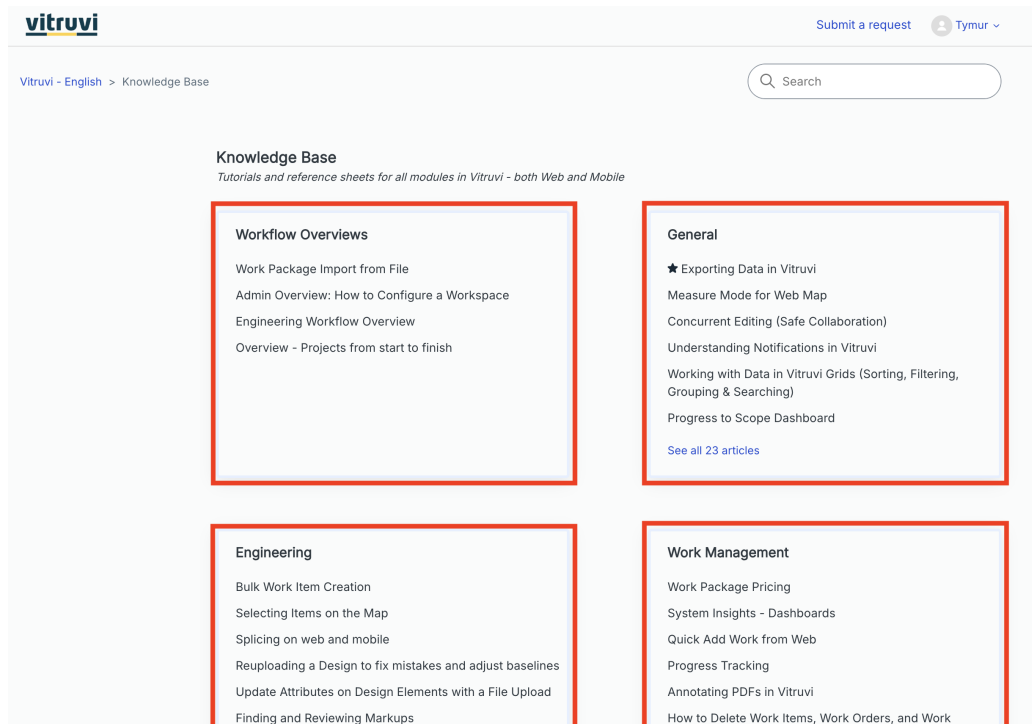


Abbildung 3.3.13.: Sektions-Übersicht der Vitruvi Knowledge Base. Die rot markierten Bereiche entsprechen den elf Sektionen aus Tabelle 3.8 und bestätigen die Vollständigkeit der zugrunde liegenden API-Sondierung.

Bild-Strategie als kritischer Entscheidungspunkt. Eine stichprobenartige Inspektion zeigte, dass die 1.146 Bilder fast ausschließlich UI-Screenshots mit klar lesbaren Feldnamen, Menüpfaden und Schaltflächen sind – nicht Diagramme oder Schemata. Damit ergibt sich eine ungewöhnliche Trade-off-Struktur: Ein generisches Vision-Language-Modell (VLM) liefert für UI-Screenshots typischerweise schwächere Ergebnisse als reines OCR, da der relevante Signalwert in präzisiertem Text und nicht in visueller Semantik liegt. Tabelle 3.9 stellt die untersuchten Strategien gegenüber.

Tabelle 3.9.: Untersuchte Strategien zur Verarbeitung der 1.146 in den Help-Center-Artikeln eingebetteten Bilder, mit Zeitabschätzung für die einmalige Verarbeitung des Korpus.

Strategie	Indizierter Inhalt aus Bildern	Zeitaufwand	Einsatzbedingung
Bilder verwerfen	nichts	0	wenn Fließtext Workflows ausreichend beschreibt
OCR (Tesseract / EasyOCR)	sichtbarer Text (Feldnamen, Labels)	15–25 min	wenn Screenshots Signale tragen, die im Fließtext fehlen
Lokales VLM (SmoIVLM-256M)	generische Bildbeschreibung	30–50 min	marginal – kleine VLMS verfehlen spezifische UI-Labels
Lokales VLM (Qwen2-VL-2B)	reichhaltige Beschreibung	2–3 h	nur bei Diagramm-dominierten Korpora
Remote VLM (z. B. GPT-4o mini)	beste Beschreibung	ca. 10 min	nur bei akzeptierter Datenweitergabe

Resultierende Entscheidung. Auf Grundlage dieser Analyse wurde der Aufwand einer Integration für die vorliegende Arbeit als unverhältnismäßig hoch eingestuft. Eine reine Text-Ingestion wäre in unter einer Minute möglich, würde aber zentrale Signale aus UI-Screenshots verlieren; eine OCR-Anreicherung wäre methodisch ergiebig, hätte aber den Rahmen der Bachelorarbeit zeitlich überspannt, insbesondere im Verhältnis zum erwarteten Mehrwert über den bereits abgedeckten internen Korpus von ca. 16.987 Tickets. Eine zukünftige Erweiterung wird im Ausblick (Kapitel 6) entlang der bereits ausgearbeiteten Architektur skizziert: ein dedizierter Crawler unter `src/ingest/zendesk/` mit Passwort-Basic-Auth, Bild-Strippen vor dem Docling-Aufruf und einer schrittweisen OCR-Anreicherung bei messbaren Retrieval-Lücken.

3.3.6.5. Validierung der Vitruvi-Pipeline (Demo-Test)

Zur Verifikation, dass die drei implementierten Quellen tatsächlich aus dem gemeinsamen Index erreichbar sind und die quelltyp-spezifische Zitations-Logik korrekt funktioniert, wurde im Demo-Notebook `vitruvi/notebooks/99_demo_rag.ipynb` ein Smoke-Test mit drei repräsentativen Fragen ausgeführt – jeweils eine pro Quelltyp. Die zugrunde liegende Konfiguration entspricht der späteren Produktivnutzung: *Llama 3.1 8B Instruct* in der Quantisierung Q5_K_M, BGE-M3-Embedder auf Apple-Silicon-MPS, `top_k=5`, Temperatur 0,1. Der System-Prompt instruiert das Modell, ausschließlich aus dem Kontext zu antworten und jede Aussage mit einem Zitations-Tag zu belegen; das Tag-Format wird je nach `source_type` unterschiedlich aufgebaut:

- pdf – [§ X.Y ToS p.N-M]
- web_guide – [Vitruvi Docs > ... > Heading]
- openapi_endpoint – [METHOD /api/v1/...]

Die folgenden drei Beispiele dokumentieren je Retrieval-Treffer (Rang, Score, Quelltyp, Zitations-Tag), die generierte Antwort sowie die manuelle Verifikation an der Originalquelle.

Frage 1 – PDF Customer ToS. Die Frage „What are Improvements according to the Vitruvi Terms of Service?“ ist gezielt auf eine Definition aus dem Vertragsdokument formuliert. Die Pipeline benötigte 15,42 s, der Prompt umfasste 728 Tokens und die Antwort 65 Tokens.

Retrieved (top-5):

```
#1 [0.719] pdf    [Sec 1.23 ToS p.6-6]
#2 [0.666] pdf    [Sec 5.20 ToS p.25-25]
#3 [0.664] pdf    [Sec 7.1  ToS p.26-26]
#4 [0.640] pdf    [Sec 7.4  ToS p.27-27]
#5 [0.633] pdf    [Sec 7.3  ToS p.26-27]
```

Answer:

```
According to the Vitruvi Customer Terms of Service, "Improvements"
means any and all improvements, derivative works, variations,
updates, modifications, enhancements or adaptations on, to or of
the Technology, the Company Intellectual Property, or the
Company's Confidential Information [Sec 1.23 ToS p.6-6].
```

Alle fünf Retrieval-Treffer stammen erwartungsgemäß aus dem PDF-Korpus. Die manuelle Verifikation in Abbildung 3.3.14 bestätigt, dass die zitierte Klausel 1.23 auf Seite 6 des Originaldokuments tatsächlich die wiedergegebene Definition enthält. Sowohl die Klausel-Nummer als auch die Seitenangabe sind korrekt; die Antwort ist eine wörtliche Wiedergabe der zitierten Definition.

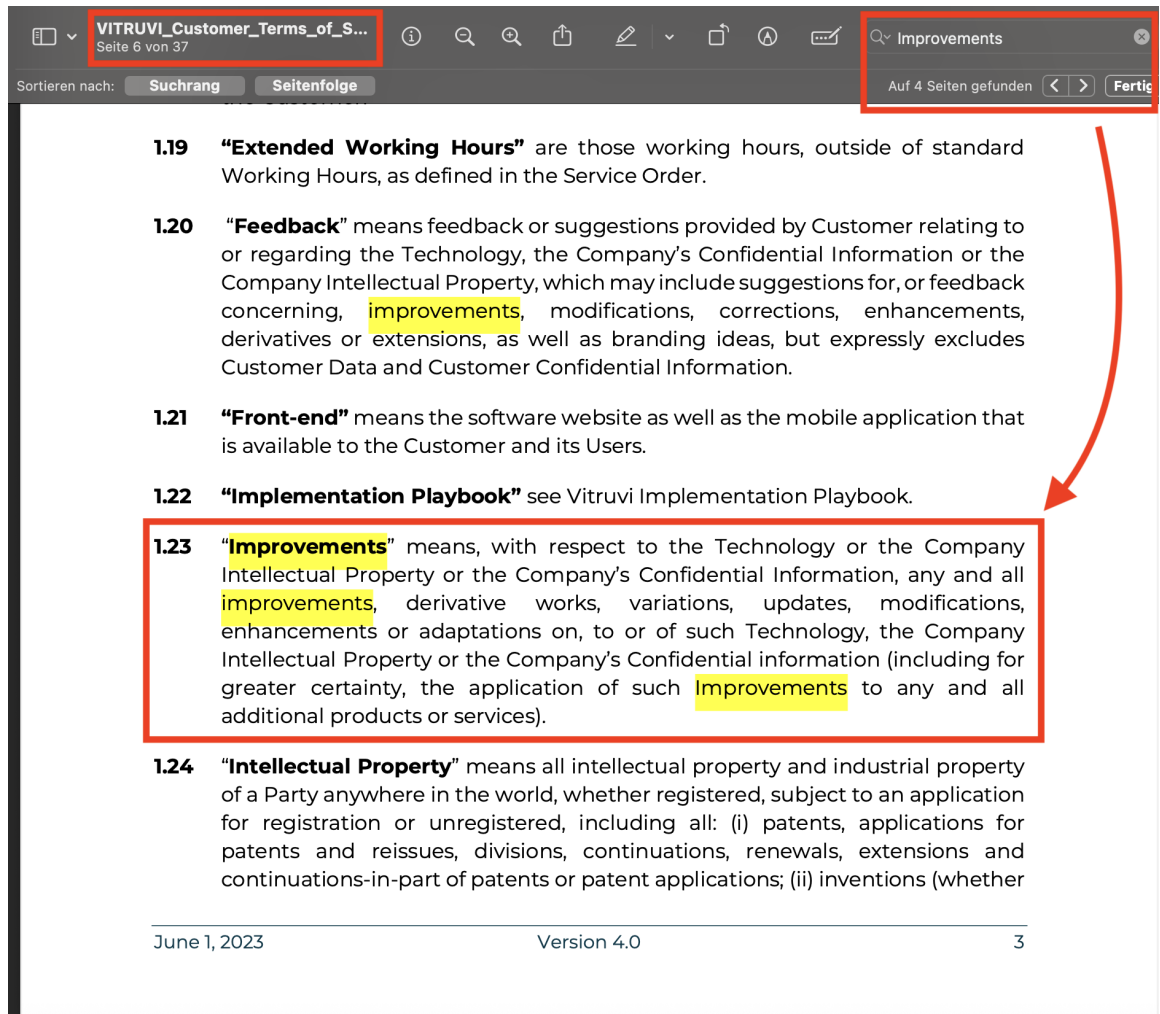


Abbildung 3.3.14.: Manuelle Verifikation der Antwort zu Frage 1 im Vitruvi Customer Terms of Service (PDF v4.0). Die rote Markierung zeigt die Original-Klausel 1.23 auf Seite 6, deren Wortlaut die generierte Antwort wieder-gibt.

Frage 2 – Developer-Guides. Die Frage „How do I authenticate to the API using JWT?“ ist auf eine konzeptionelle Erklärung aus dem /docs-Sektor zugeschnitten, deckt aber auch verwandte API-Endpoints ab. Die Pipeline benötigte 21,65s bei 676 Prompt- und 238 Antwort-Tokens.

Retrieved (top-5):

#1 [0.744]	web_guide	[Vitruvi Docs > Authentication > 1. JSON Web Token (JWT)]
#2 [0.694]	openapi_endpoint	[POST /auth/token/obtain]
#3 [0.660]	web_guide	[Vitruvi Docs > Authentication > 1. JSON Web Token (JWT) > Obtaining a new JWT token]
#4 [0.650]	openapi_endpoint	[POST /api/v1/mobile/devices/]
#5 [0.647]	openapi_endpoint	[POST /auth/token/verify]

Answer (gekuerzt):

To authenticate to the API using JWT, you should pass the token on the Authorization header with JWT on front: "Authorization: JWT"

<Token>" [Vitruvi Docs > Authentication > 1. JSON Web Token (JWT)].
 To obtain a new JWT token, the client needs to send a POST with
 email and password to /auth/token/obtain [POST /auth/token/obtain]
 ... Some API endpoints (e.g. [POST /api/v1/mobile/devices/]) also
 support JWT authentication as indicated by the jwtAuth tag.

Die Treffer-Liste mischt erwartungsgemäß den Konzept-Guide (Rang 1 und 3) mit zugehörigen API-Endpoints (Rang 2, 4, 5). Die Antwort kombiniert die konzeptuelle Erklärung aus dem Guide mit den konkreten Endpoint-Verweisen, jeweils mit korrektem Zitations-Tag. Abbildung 3.3.15 zeigt den Originalartikel im Developer-Portal mit der rot markierten Passage zur Header-Syntax Authorization: JWT <Token>; der vom Modell wiedergegebene Wortlaut stimmt mit dem Original-Quelltext überein.

The screenshot shows the Vitruvi Docs website. The left sidebar contains a navigation menu with sections like DOCUMENTATION, GETTING STARTED WITH VITRUVI, INTRODUCTION TO WEB APIS, and GETTING STARTED WITH VITRUVI APIS. The 'Authentication' link under 'GETTING STARTED WITH VITRUVI APIS' is highlighted. The main content area is titled '1. JSON Web Token (JWT)' and contains the following text:

Vitruvi currently supports 3 different auth methods.

1. JSON Web Token (JWT)

When it's possible, the client using Vitruvi APIs should use JWT tokens for authentication. This token is passed on each request on the `Authorization` header with `JWT` on front:

`Authorization: JWT <Token>`

Obtaining a new JWT token

To retrieve a new JWT token, the client needs to send a POST with email and password to:

`/auth/token/obtain`

receiving the token with user information

Refresh Token

To refresh the JWT token, the clients needs to send a POST with the refresh token to:

`/auth/token/refresh`

receiving an updated token if the token provided is still within the refresh window.

2. Access Token

Access token are encrypted tokens that are created by the [support team](#) for special use cases, they don't expire unless being revoked by the support team.

Useful for service integrations that don't have capabilities to get a new token on each session. They are passed on each request with the `Authorization` header with `Bearer` on front:

`Authorization: Bearer <Token>`

3. Basic Authentication

Although is not recommended for API usage, all APIs support basic authentication, using email and password. This is particular useful for integrations where tokens are not supported.

Updated 8 months ago

On the right side of the page, there is a 'Copy Page' button and a table of contents with the following items:

- 1. JSON Web Token (JWT)
 - Obtaining a new JWT token
 - Refresh Token
- 2. Access Token
- 3. Basic Authentication

Abbildung 3.3.15.: Manuelle Verifikation der Antwort zu Frage 2 im Developer-Guide Vitruvi Docs > Authentication > JSON Web Token (JWT). Die rote Markierung zeigt die JWT-Header-Syntax, deren Wortlaut die generierte Antwort wiedergibt.

Frage 3 – OpenAPI-Referenz. Die Frage „How do I export accounts payable items via the API?“ adressiert einen spezifischen API-Endpoint; die Pipeline benötigte 22,18 s bei 1.134 Prompt- und 199 Antwort-Tokens.

Retrieved (top-5):

#1 [0.721]	openapi_endpoint	[GET /api/v1/reporting/accounts_payable_items_export/]
#2 [0.673]	openapi_endpoint	[GET /api/v1/reporting/accounts_receivable_items_export/]
#3 [0.616]	openapi_endpoint	[GET /api/v1/reporting/invoices_export/]

```
#4 [0.616] web_guide      [Vitruvi Docs > Available APIs > Choosing the Right API > Reporting APIs >
Export APIs]
#5 [0.612] openapi_endpoint [POST /api/v1/invoicing/deliverable_payable_item_templates/batch_delete/]
```

Answer (gekuerzt):

To export accounts payable items via the API, you can use the [GET /api/v1/reporting/accounts_payable_items_export/] endpoint. This endpoint allows you to export Vitruvi data for use in system integrations, client backups, and reporting. You can customize the data returned by specifying query parameters such as fields, filter_expression, limit, offset, ordering, and search. Note that this endpoint requires authentication using basicAuth, jwtAuth, or tokenAuth.

Der Top-Treffer ist der exakt gesuchte Export-Endpoint, gefolgt von zwei thematisch verwandten Export-Endpoints (Forderungen, Rechnungen) sowie dem zugehörigen Konzept-Guide auf Rang 4. Die Antwort beschreibt korrekt den HTTP-Verb, den vollständigen Pfad, die unterstützten Query-Parameter und die Authentifizierungs-Schemata; Abbildung 3.3.16 zeigt den Original-Endpoint in der API-Referenz, dessen Eigenschaften die Antwort wiedergibt.

The screenshot shows the Vitruvi API Reference interface. On the left is a sidebar with a navigation menu including 'Qualifications', 'Quantity descriptions', 'Reporting', and various export endpoints. The 'Export Accounts Payable Items' endpoint is highlighted. The main content area displays the endpoint details for 'GET https://(workspace).api.vitruvi.cc/api/v1/reporting/accounts_payable_items_export/'. It includes a description: 'Export Vitruvi data for use in system integrations, client backups and reporting.' Below this is a 'Recent Requests' table showing no requests. The 'Query Params' section lists parameters: 'fields' (string), 'filter_expression' (string), and 'limit' (integer, default 1000). On the right, there are sections for 'LANGUAGE' (Shell, Node, Ruby, PHP, Python), 'CREDENTIALS' (Basic auth with username and password fields), 'URL' (Base URL and full endpoint URL), and a 'cURL Request' block with a curl command. At the bottom right, the 'RESPONSE' section shows an example response: 'application/json' with a status of '200'.

Abbildung 3.3.16.: Manuelle Verifikation der Antwort zu Frage 3 in der Vitruvi API-Referenz GET /api/v1/reporting/accounts_payable_items_export/. Die rote Markierung zeigt die Endpoint-Definition mit Pfad, Query-Parametern und Authentifizierungs-Schemata, die die generierte Antwort wiedergibt.

Beobachtungen aus dem Demo-Test. Die drei Beispiele decken alle drei implementierten Quelltypen ab und bestätigen drei zentrale Design-Eigenschaften der Vitruvi-Pipeline: (a) das Retrieval routet semantisch passende Treffer zum jeweils erwarteten Quelltyp;

(b) die quelltyp-spezifische Zitations-Logik erzeugt für jeden Quelltyp ein konsistentes, manuell nachvollziehbares Tag-Format; (c) die manuelle Verifikation an den Originalquellen bestätigt für alle drei Fragen die semantische Korrektheit der Antworten samt korrekter Seitennummern bzw. URL-Pfade. Dieser Smoke-Test ersetzt keine systematische Evaluation; eine quantitative Bewertung mit dem RAGAS-Framework erfolgt in Kapitel 5.

3.3.6.6. Zusammenfassung der externen Quellen

Tabelle 3.10 fasst die Eigenschaften der drei externen Quellen kompakt zusammen. Die drei implementierten Quellen tragen 1.291 Chunks zum gesamten Korpus bei und decken die Vitruvi-bezogenen Wissenslücken im Support ab, soweit sie textuell repräsentiert sind. Der in Abschnitt 3.3.6.5 dokumentierte Demo-Test bestätigt, dass alle drei Quelltypen aus dem gemeinsamen Index erreichbar sind und die quelltyp-spezifische Zitations-Logik korrekt funktioniert.

3.3.7. TKI-Produkte-Wissensbasis (TKI-Pr-KB)

Die TKI-Pr-KB fasst die Soll-Funktionalität der TKI NET Suite auf Modul-Ebene zusammen: Sie besteht aus acht strukturierten Markdown-Dokumenten, eines je Modul (NET Engineering, NET Cloud, NET Design, NET Build, NET Scan, NET Operations, NET Inhouse sowie der PGP-PostgreSQL-Provider), insgesamt rund 244 KB deutschsprachiger Text. Inhaltlich beschreibt jedes Dokument den Funktionsumfang, das Lizenzmodell, die Release-Historie und die Abhängigkeiten zu den übrigen Modulen.

Herkunft der Inhalte. Anders als die ticket-basierten Quellen, die reale Support-Verläufe abbilden, sind die Modul-Dokumente *aus internen Code-Repositories der TKI NET Suite und dem Confluence-Wiki aggregierte* Funktionsbeschreibungen. Die primären Quellen jedes Dokuments sind in dessen YAML-Frontmatter unter dem Schlüssel `sources` deklariert; Beispiel aus `tki_products_kb/data/source/pgp.md`:

```
sources:
  - Code-Basis ('TKI.PGP.*'-Projekte aus net-main; 20 Projekte)
  - Confluence PGP-Space (Release Notes 4.x - 5.2.1)
  - help.tki-chemnitz.de (Sektion PostgreSQL-Provider PGP)
  - Jira PGP-Projekt (laufende Tickets, Bug-Pattern)
```

Diese Deklaration ist Teil jeder Datei und damit unmittelbar aus der Datenbasis ablesbar; sie macht das Dokument auf seine konkreten Code-Pfade und Confluence-Spaces zurückverfolgbar.

Komplementarität zu den anderen Quellen. Die TKI-Pr-KB ergänzt die beiden anderen Quellgruppen entlang einer klaren funktionalen Trennlinie:

Tabelle 3.10.: Übersicht der externen Vitruvi-Wissensquellen mit ihren jeweiligen Discovery-, Chunking- und Mengen-Eigenschaften.

Quelle	Format	Discovery	Chunker	Chunks	Status
Developer Portal /docs	HTML / Markdown	Sitemap + llms.txt + Sidebar-Fallback	Docling HybridChunker (heading-aware)	127	implementiert
Developer Portal /reference	OpenAPI 3.0.3 JSON	Sitemap	NL-Rendering pro Endpoint	1.059	implementiert
Customer Terms of Service v4.0	PDF (37 Seiten)	manuell bereitgestellt	Clause-Aware-Regex + Docling-Tabellen	105	implementiert
Vitruvi Help Center	HTML + Bilder + Vimeo	HC-API	(geplant: Docling HTML + OCR)	≈600	analysiert, Ausblick
Summe implementiert				1.291	

- *Ticket-basierte Quellen* (Zendesk, Confluence, Jira, Slack, HC) liefern *Symptom-Lösung-Paare* aus realem Support-Verlauf: konkrete Workarounds, beobachtete Fehlermeldungen, gefixte Bugs. Was hier fehlt, ist die kanonische Beschreibung, wofür ein Modul überhaupt gebaut wurde.
- *Externe Vitruvi-Quellen* (PDF-ToS, Web-Guides, OpenAPI-Referenz) liefern Vertragstexte und API-Spezifikationen eines Drittanbieter-Services.
- Die *Produkt-Wissensbasis* schließt diese Lücke, indem sie pro Modul beschreibt, welche Funktionen es bereitstellt, in welchen Code-Projekten die Implementierung liegt und wie es zu den übrigen Modulen related ist. Damit wird das Retrieval auch in Fällen reproduzierbar, in denen kein Ticket-Verlauf existiert, weil die Frage auf die Soll-Funktionalität („Was kann NET Engineering?“, „Wann nutze ich NET Inhouse statt NET Scan?“) und nicht auf eine konkrete Fehlersituation zielt.

Schema und Volumen. Aus den acht Dokumenten erzeugt die in Abschnitt 4.2.5 beschriebene Pipeline 329 Chunks mit gleicher Embedding-Dimensionalität und Schema wie die übrigen Quellen, sodass sie ohne Sonderbehandlung in den gemeinsamen FAISS-Index (siehe 3.4) einfließen.

3.4. Datenverarbeitungspipeline

Die Offline-Datenverarbeitungspipeline transformiert die Rohdaten der verschiedenen Quellen in eine einheitliche, suchbare Struktur. Der Prozess umfasst folgende Schritte:

1. **Extraktion:** Abruf der Rohdaten aus den jeweiligen APIs (siehe Abschnitt 3.3).
2. **Normalisierung:** Vereinheitlichung der Datenformate. Dies umfasst:
 - Bereinigung von HTML-Tags und Formatierungen
 - Maskierung personenbezogener Daten (E-Mails, Telefonnummern) durch Platzhalter (<EMAIL>, <PHONE>)
 - Entfernung von Signaturen, Disclaimern und Bildverweisen
 - Extraktion und Normalisierung des Produktnamens mittels kanonischer Produktliste
3. **Zusammenführung:** Alle normalisierten Daten werden in einer gemeinsamen Parquet-Datei (`combined.parquet`) zusammengeführt. Jeder Datensatz enthält die Felder: `ticket_id`, `text`, `title`, `product`, `source`, `status`, `priority`.

4. **Chunking:** Lange Texte werden quelltyp-spezifisch in überlappende Abschnitte aufgeteilt. Zendesk-Tickets nutzen einen zeichenbasierten Splitter (2500 Zeichen, 200 Zeichen Überlappung), HC, TKI-Pr-KB und die externen Vitruvi-Quellen arbeiten token-basiert (500–512 Tokens je Chunk). Die konkreten Parameter und Begründungen sind in den jeweiligen Implementierungsabschnitten dokumentiert. Allen Strategien gemeinsam ist, dass die resultierenden Chunks innerhalb des Kontextfensters des Embedding-Modells liegen und an Chunk-Grenzen entweder Überlappung oder semantische Schnittpunkte den Informationsverlust minimieren.
5. **Embedding:** Die Chunks werden durch das BGE-M3-Modell in 1024-dimensionale Vektoren transformiert und als NumPy-Arrays gespeichert.
6. **Indizierung:** Die Vektoren werden in einem FAISS-IndexFlatIP-Index zusammengeführt. Zusätzlich wird ein Mapping von Vektorindex auf Ticket-ID gespeichert, um die Zuordnung zur Laufzeit zu ermöglichen.

Die Reproduzierbarkeit der Pipelines wird durch versionierte Konfigurationen (`config.py` je Pipeline) sowie commit-genaue Git-Referenzen der jeweiligen Iterationen sichergestellt. Eine spezialisierte Snapshot-Utility für die Zendesk-Stage-2-Iterationsphase (`zendesk/stage2_extraction/src/snapshot.py`) erfasst zusätzlich Git-Hash, Dirty-State und Run-Zeitstempel pro Lauf.

3.5. Semantische Suche

Die semantische Suche bildet die erste Stufe der RAG-Pipeline. Der Ablauf ist wie folgt:

1. Die Benutzeranfrage wird durch das BGE-M3-Modell in einen 1024-dimensionalen Vektor eingebettet.
2. FAISS führt eine Nearest-Neighbor-Suche durch und gibt die k ähnlichsten Vektoren zurück (Standard: $k = 50$).
3. Über das Vektorindex-Ticket-ID-Mapping werden die zugehörigen Metadaten aus der Parquet-Datei abgerufen.
4. Für HC-Artikel wird der passende Chunk mittels Sliding-Window-Rekonstruktion extrahiert.

Die Kandidaten werden anschließend der Reranking-Stufe übergeben. Der Cross-Encoder bewertet jedes Anfrage-Dokument-Paar und erzeugt einen Relevanz-Score. Anschließend wird mittels Maximal Marginal Relevance (MMR) eine diversitätsoptimierte Aus-

wahl der Top- k Ergebnisse getroffen, wobei der Parameter $\lambda = 0.7$ die Balance zwischen Relevanz und Diversität steuert.

Ein Score-Schwellenwert von 0,40 filtert Ergebnisse mit zu geringer Relevanz vor der Weiterleitung an das LLM.

3.6. LLM-Integration und Antwortgenerierung

Die Antwortgenerierung erfolgt durch ein lokal betriebenes LLM, das über die Bibliothek `llama-cpp-python` geladen wird. Die Konfiguration umfasst:

- **Kontextfenster:** 4096 Token
- **CPU-Threads:** 8 (angepasst an die Serverhardware)
- **GPU-Schichten:** 0 (reiner CPU-Betrieb für Konsistenz)
- **Temperatur:** 0.3 (Standard; konfigurierbar über die UI)

Der Prompt wird im Llama 3.1 Chat-Format aufgebaut und enthält:

1. Einen System-Prompt, der die Rolle als technischer Support-Assistent für TKI-Produkte definiert
2. Die Instruktion, Antworten ausschließlich auf den bereitgestellten Dokumenten zu basieren
3. Maximal 7 Dokumente mit ihren Scores, Quellen und Titeln
4. Eine XML-basierte Ausgabestruktur (`<answer>`, `<steps>`, `<confidence>`)

Die Antwort wird Token für Token generiert und über SSE an das Frontend gestreamt. Nach Abschluss der Generierung wird die XML-Struktur geparkt, um die einzelnen Bestandteile (Antworttext, Schritte, LLM-Konfidenz) zu extrahieren.

3.7. Konfidenzbewertung

Die Konfidenzbewertung kombiniert vier Signale zu einem gewichteten Gesamtwert:

$$C = 0,30 \cdot S_{\text{rerank}} + 0,20 \cdot S_{\text{separation}} + 0,35 \cdot S_{\text{llm}} + 0,15 \cdot S_{\text{coverage}} \quad (3.2)$$

- S_{rerank} – Normalisierter Rerank-Score des bestplatzierten Dokuments

- $S_{\text{separation}}$ – Differenz zwischen dem besten und zweitbesten Rerank-Score (misst die Eindeutigkeit des Ergebnisses)
- S_{llm} – Selbsteinschätzung des LLM (aus dem <confidence>-Tag der Antwort)
- S_{coverage} – Anteil der Anfrageterme, die in den Top-3-Dokumenten vorkommen

Basierend auf dem Gesamtwert C wird ein qualitatives Label zugewiesen:

- $C \geq 0,75$: **high** – hohe Konfidenz, Entwurf wahrscheinlich verwendbar
- $0,40 \leq C < 0,75$: **medium** – mittlere Konfidenz, Prüfung empfohlen
- $0,25 \leq C < 0,40$: **low** – niedrige Konfidenz, Fallback-Warnung wird angezeigt
- $C < 0,25$: **hard reject** – der Entwurf wird verworfen und durch eine Standard-Meldung ersetzt

Kapitel 4.

Implementierung

Dieses Kapitel dokumentiert die technische Umsetzung des Systems. Die Implementierung ist modular aufgebaut und gliedert sich in Backend, Frontend und die Datenextraktionspipelines.

4.1. Projektstruktur

Das Gesamtprojekt ist in mehrere eigenständige Module organisiert:

```
ai-agent-for-support/
  react_app/
    backend/
      app.py          # FastAPI-Einstiegspunkt
      src/
        config.py     # Zentrale Konfiguration
        queue_manager.py # Concurrency-Steuerung
      api/
        schemas.py    # Pydantic-Datenmodelle
        routes/
          search.py    # Such-Endpunkt
          rerank.py    # Reranking-Endpunkt
          pipeline_stream.py # SSE-Streaming-Pipeline
          pipeline_utils.py # Resource-Link-Extraktion
        embedding/
          search.py    # FAISS-Suche
          reranker.py  # Cross-Encoder
      rag/
        answer_drafter.py # LLM-Integration
        confidence.py    # Konfidenzbewertung
    frontend/
      src/
        App.tsx        # Hauptkomponente
      api/
        client.ts       # API-Client
        types.ts        # TypeScript-Typen
```

```
components/          # UI-Komponenten
constants/           # SourceKey, Tag-Styles
hooks/               # React Hooks
utils/               # Source-Ref- und Datums-Formatter
zendesk/             # Zendesk-Tickets (Stages 0--3)
zendesk_help_center/ # HC-Extraktion (HTML/PDF/Office)
vitruvi/             # Vitruvi-Quellen (/docs, /reference, ToS)
tki_products_kb/     # TKI-Produkte-Wissensbasis
```

Listing 4.1: Projektstruktur (vereinfacht)

4.2. Backend

Das Backend ist als FastAPI-Anwendung implementiert und läuft auf einem Uvicorn-Server. Bei der Initialisierung werden alle ML-Modelle (Embedding-Modell, Reranker, LLM, FAISS-Index) vorab in den Speicher geladen, um die Latenz der ersten Anfrage zu vermeiden.

4.2.1. Zentrale Konfiguration

Die Konfiguration ist in der Datei `config.py` zentralisiert und verwendet typisierte Dataclass-Strukturen:

```
@dataclass
class FaissConfig:
    index_path: Path
    vectors_path: Path
    top_k: int = 50

@dataclass
class RerankerConfig:
    model_name: str = "BAAI/bge-reranker-v2-m3"
    candidate_k: int = 50
    top_k: int = 10
    mmr_lambda: float = 0.7
    score_threshold: float = 0.40
```

Listing 4.2: Konfigurationsbeispiel (vereinfacht)

Die Pfadauflösung erfolgt relativ zum Projektstamm, wobei das LLM-Modellverzeichnis über die Umgebungsvariable `LLM_MODEL` konfiguriert werden kann.

4.2.2. Zendesk-Pipeline

Die Zendesk-Pipeline ist die umfangreichste Datenpipeline des Systems und setzt sich aus drei Schichten zusammen: (i) rohe API-Extraktion, (ii) Normalisierung und Embedding, (iii) LLM-gestützte Qualitätsfilterung. Schichten (i) und (ii) entsprechen dem Standard-RAG-Vorgehen, während Schicht (iii) den eigenständigen Beitrag dieser Arbeit darstellt und in den folgenden Abschnitten ausführlich behandelt wird.

4.2.2.1. API-Extraktion

Die rohe Extraktion ist in `zendesk/stage0_acquisition/fetch_zendesk.py` implementiert. Sie nutzt die Zendesk *Search Export API* und ruft die Kommentare jedes Tickets über einen separaten Endpunkt nach.

Die Extraktion läuft in zwei Phasen: eine cursorbasierte Hauptschleife ruft zunächst alle Tickets ohne Kommentare über die Search-Export-API ab (`links.next` als Folge-Link, `meta.has_more` als Abbruchkriterium); eine zweite Schleife reichert anschließend jedes Ticket über den Kommentar-Endpunkt an und schreibt nach je 100 angereicherten Tickets einen Checkpoint, der bei einem Abbruch eine Wiederaufnahme ermöglicht. Listing 4.3 zeigt diese Kernlogik; die vollständige Funktion mit HTTP-Retry- und Rate-Limit-Behandlung ist in Anhang A.7.1 dokumentiert.

```
# Phase 1: Cursor-Schleife uebergibt alle Tickets (ohne Kommentare)
while url:
    data = requests.get(url, auth=auth, params=params).json()
    tickets.extend(data.get("results", []))
    if data.get("meta", {}).get("has_more"):
        url = data.get("links", {}).get("next")
        params = {} # Cursor liegt im Next-Link
    else:
        url = None

# Phase 2: Anreicherung mit Kommentaren + Checkpoint je 100 Tickets
for ticket in tickets:
    if ticket.get("comments"):
        continue
    ticket["comments"] = fetch_comments_for_ticket(ticket["id"], ...)
    processed += 1
    if processed % 100 == 0:
        save_checkpoint(checkpoint_file, tickets)
```

Listing 4.3: Kern der Ticket-Extraktion: Cursor-Paginierung und 100-Ticket-Checkpoint (Schlüsselausschnitt; vollständige Funktion s. Anhang A.7.1)

Die Authentifizierung folgt dem Zendesk-Token-Schema ({email}/token, API-Token aus .env). Die Paginierung erfolgt cursorbasiert über `links.next`; Rate-Limit-Antworten (HTTP 429) werden mit dem `Retry-After-Header` behandelt. Alle 100 verarbeiteten Tickets schreibt die Funktion einen Checkpoint, der nach Abbruch eine Wiederaufnahme erlaubt. Das Resultat ist ein einzelnes JSON-Dokument (`zendesk_full_download.json`, ca. 810 MB, ca. 16 987 Tickets) mit dem unveränderten Zendesk-Schema, ergänzt um das Feld `comments`.

4.2.2.2. Normalisierung

Die Normalisierung in `zendesk/src/preprocess/normalize_zendesk.py` überführt die Roh-JSON-Daten in eine Apache-Parquet-Datei mit einheitlichem Schema. Sie umfasst PII-Maskierung, HTML- und Boilerplate-Entfernung, Rollen-Tagging und Produktnormalisierung.

```
EMAIL_RE = re.compile(r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}")
PHONE_RE = re.compile(r"\b\+?\d[\d\s() .-]{7,}\b")

def _mask_pii(text: str) -> str:
    text = EMAIL_RE.sub("<EMAIL>", text)
    text = PHONE_RE.sub("<PHONE>", text)
    return text
```

Listing 4.4: PII-Maskierung

Die Rollenzuordnung erfolgt durch Vergleich von `author_id` und `requester_id` pro Kommentar, anschließend wird der Body mit `[CUSTOMER]` oder `[SUPPORT]` präfixiert. Die Produktnormalisierung erfolgt anhand einer kanonischen Liste der zehn TKI-Produkte und ist innerhalb der Zendesk-Pipeline implementiert:

```
from shared.product_utils import (normalize_product,
                                  normalize_module,
                                  normalize_technology)

PRODUCT_PATTERN = re.compile(
    r"\b(NET Design|NET Engineering|NET Build|NET Cloud|"
    r"NET Scan|NET Operations|NET Inhouse|NET REST|"
    r"NET Field Survey|Dynamics Sales|TNIM|Comsof Fiber)\b")
```

Listing 4.5: Produktnormalisierung

Das Ergebnis ist eine Parquet-Datei mit den Spalten `ticket_id`, `title`, `text`, `created_at`, `updated_at`, `status`, `priority`, `tags`, `url`, `product`, `module`, `technology`.

4.2.2.3. Embedding und FAISS-Index für Zendesk

Die normalisierten Texte werden in `zendesk/src/embedding/embedder.py` mit *BGE-M3* (1024-dimensional) eingebettet. Das Chunking erfolgt semantisch an Satz-, Frage- und Absatzgrenzen mit einer Zielgröße von 2500 Zeichen (etwa 600 Tokens) und einer Überlappung von 200 Zeichen. Anschließend baut `build_faiss.py` den `IndexFlatIP`-Index auf:

```
def build_index(vectors_path, index_path):
    vecs = np.load(vectors_path).astype("float32")
    index = faiss.IndexFlatIP(vecs.shape[1])
    index.add(vecs)
    faiss.write_index(index, str(index_path))
```

Listing 4.6: FAISS-Index-Erstellung für Zendesk-Vektoren

Die Artefakte (`vectors.npy`, `ids.npy`, `faiss.index`) werden im Verzeichnis `zendesk/stage3_rag/data_support/embeddings/` abgelegt und vom Backend zur Laufzeit gemeinsam mit den Vektoren der übrigen Quellen geladen.

4.2.2.4. Qualitätsfilter – Pipeline-Architektur

Die Qualitätsfilterung lebt im Unterprojekt `zendesk/stage1_filter/`. Die zentrale Datei ist `pipeline.py` (Llama-3.1-8B-kalibriert) mit dem Fork `pipeline_qwen.py`, der Qwen-2.5-32B-spezifische Erweiterungen enthält. Die Pipeline-Stufen sind:

1. `clean_text(raw)` – Entfernung von Signaturen, Quoting, Boilerplate, Whitespace-Normalisierung.
2. `render_ticket(ticket)` – Konstruktion eines Konversations-Transkripts (Subject + nummerierte [customer]/[agent]-Nachrichten); Head-und-Tail-Trunkierung bei Threads über 5500 Zeichen.
3. `prefilter(ticket)` – deterministische Vorfilterregeln (Abschnitt 4.2.2.5).
4. `classify(ticket)` bzw. `classify_sc(ticket, n)` – LLM-Klassifikation mit Few-Shot-Prompt; Self-Consistency-Variante optional (Abschnitt 4.2.2.8).
5. `apply_overrides(prediction)` – Reihenfolge $P4 \rightarrow A \rightarrow H \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow I$; jeder Override prüft Muster in `rationale` und nimmt ggf. Promotion/Demotion vor.
6. `_decide(outcome, customer_confirmed)` – finale binäre Klassifikation gemäß Gleichung 3.1.

Die Eingabe an die Pipeline ist ein Roh-Ticket-Dict aus dem Zendesk-Export; die Ausgabe ist ein Prediction-Datensatz mit den Feldern `label` (USEFUL/USELESS), `outcome` (Bucket), `customer_confirmed`, `rationale`, `subject_kind`, `applied_overrides` und (im SC-Modus) `votes`. Diese Struktur wird ticketweise als JSONL in das `output/-`-Verzeichnis geschrieben, sodass jeder Benchmarklauf inspizierbar und reproduzierbar bleibt.

4.2.2.5. Pre-Filter

Die deterministischen Vorfilterregeln sind als reine Python-Funktionen ohne externe Abhängigkeiten implementiert. Sie werfen zwei Klassen offenkundig nicht verwertbarer Tickets: Tickets ohne öffentliche Kommentare (interne Provisionierungstickets) und sehr kurze Telefonnotiz-Tickets mit Betreff „Anruf mit ...“. Listing 4.7 zeigt das Kontrollflussmuster der Regelkaskade; die vollständigen Regeln mit jeweiligem `rationale`-Text sind in Anhang A.7.2 dokumentiert.

```
def prefilter(ticket: dict) -> Prediction | None:
    public = [c for c in ticket["comments"] if c.get("public")]
    if not public:                                # Regel 1
        return Prediction("USELESS", source="prefilter", ...)
    if _is_short_call_note(ticket, public):        # Regel 2
        return Prediction("USELESS", source="prefilter", ...)
    return None                                   # an LLM
weiterreichen
```

Listing 4.7: Kontrollflussmuster der Pre-Filter-Regelkaskade (Schlüsselausschnitt; vollständige Regeln s. Anhang A.7.2)

Auf dem 200-Datensatz greifen diese Regeln auf 16 Tickets (8%) und liefern dort 100% Präzision – jeder vom Pre-Filter erkannte Ticket ist tatsächlich USELESS.

4.2.2.6. LLM-Klassifikator

Der LLM-Aufruf erfolgt über `llama-cpp-python`. Für Qwen 2.5 32B Q4_K_M wird das Chat-Format `chatml` mit einem Kontextfenster von 8192 Tokens auf 16 CPU-Threads ohne GPU-Offload betrieben, mit festem Seed 7 für Reproduzierbarkeit. Listing 4.8 zeigt die zentralen Parameter; die vollständige Initialisierung inklusive der Boilerplate-Optionen ist in Anhang A.7.5 dokumentiert.

```
llm = Llama(
    model_path=str(MODEL_PATH),    # Qwen 2.5 32B Q4_K_M (split GGUF)
    n_ctx=8192,
    n_threads=16,
    chat_format="chatml",          # Qwen-2.x-spezifisch
    seed=7,
```

)

Listing 4.8: LLM-Initialisierung für Qwen 2.5 32B (Schlüsselausschnitt; vollständige Initialisierung s. Anhang A.7.5)

Der System-Prompt umfasst ca. 650 Tokens und enthält die sechs Bucket-Definitionen, das exakte JSON-Ausgabeschema und die Regeln zur Bestimmung von `customer_confirmed`. Die acht Few-Shot-Beispiele stammen ausschließlich aus echten Tickets des Entwicklungs-Datensatzes (Tickets #4848, #6189, #13880, #19092, #19508, #12462 sowie zwei synthetische How-to-Beispiele) und decken alle sechs Buckets ab.

Das User-Template enthält den Konversations-Auszug zwischen `<ticket>`-Tags:

```
Classify this Zendesk ticket.

<ticket>
{transcript}
</ticket>

Return only the JSON object.
```

Listing 4.9: User-Template

4.2.2.7. Confidence-Overrides

Jeder Override ist als Funktion implementiert, die eine Prediction entgegennimmt und entweder unverändert oder modifiziert zurückgibt; die angewandten Overrides werden im Feld `applied_overrides` mitprotokolliert. Beispielhaft Override P4, der kundenseitige Auflösungen aus dem `abandoned`-Bucket nach `solved` hochstuft. Die Erkennungslogik basiert auf einem regulären Ausdruck, der typische Formulierungen wie „customer confirmed it“, „customer fixed the issue“ oder „after agent suggestion customer confirms“ abdeckt; Listing 4.10 zeigt das Kontrollflussmuster, die vollständige Regex und Funktion sind in Anhang A.7.3 dokumentiert.

```
def override_p4(p: Prediction) -> Prediction:
    if p.outcome == "abandoned" \
        and _RESOLVED_IN_RATIONALE.search(p.rationale or ""):
        p.outcome = "solved" # Bucket-Promotion
        p.customer_confirmed = True
        p.applied_overrides.append("P4")
    return p
```

Listing 4.10: Kontrollfluss Override P4 (Schlüsselausschnitt; vollständige Regex s. Anhang A.7.3)

Die Override E (Qwen-spezifisch) wandelt einen `escalated`-Bucket in `solved` um, wenn das Modell eine konkrete Fix-Version benannt hat (z. B. „NET 14.2.2“, „Converter 2.1“, „fix

scheduled for December“), aber kein `_TRULY_UNRESOLVED`-Marker („no ETA“, „customer left“) das Gegenteil signalisiert. Override `F` entsprechend für `consultation+confirmed`, sofern eine klassen-gültige Regel im Rationale erkennbar ist („setting X to Y“, „X vs Y“, „import via Excel“). Die vollständigen Definitionen der acht Overrides sind im Anhang aufgeführt.

4.2.2.8. Self-Consistency

Die Self-Consistency-Variante `classify_sc()` führt N unabhängige LLM-Aufrufe mit unterschiedlichen Seeds und einer Sampling-Temperatur von 0,3 durch und aggregiert die Ergebnisse durch eine zweistufige Mehrheitsentscheidung: das Outcome-Bucket per Mehrheits-Voting und die Bestätigung des Kunden per einfacher Mehrheit. Listing 4.11 zeigt diesen Aggregationskern; die vollständige Funktion einschließlich der Rückgabe der Stimmenverteilung ist in Anhang A.7.4 dokumentiert.

```
samples = [_one_pass(ticket, seed=s, temperature=0.3)
            for s in seeds[:n]]
votes = Counter(s.outcome for s in samples)
winning, _ = votes.most_common(1)[0]
majority_confirmed = sum(s.customer_confirmed
                          for s in samples) >= (n // 2 + 1)
return Prediction(label=_decide(winning, majority_confirmed),
                  outcome=winning, ...)
```

Listing 4.11: Aggregationskern der Self-Consistency-Variante (Schlüsselausschnitt; vollständige Funktion s. Anhang A.7.4)

Während der Iterationsphase auf Llama 3.1 8B mit Q4_K_M-Quantisierung steigerte Self-Consistency mit $N = 3$ die Accuracy auf Golden 45 von rund 82% (Single-Pass) auf 84,4%. Der lokale Spitzenwert von 88,9% wurde anschließend ohne Self-Consistency erreicht, nachdem die Modellquantisierung auf Q5_K_M angehoben wurde (vgl. Abschnitt 5.2.4); der stabilere Single-Shot-Modus mit Q5 lieferte dieselbe oder bessere Genauigkeit ohne dreifaches Inferenzbudget. Im Produktivbetrieb mit Qwen 2.5 32B kommt ebenfalls der Single-Pass-Modus zum Einsatz, da die Inferenzzeit je Ticket bei ca. 80 Sekunden liegt und Self-Consistency den Durchsatz um den Faktor N reduzieren würde.

4.2.2.9. Strukturierte Extraktion – Vorgängerversionen v2/v3/v4

Den Qualitätsfilter ergänzten zunächst drei Extraktions-Iterationen, die als Vorgänger der heutigen Produktivpipeline (Abschnitt 4.2.2.10) dienten. Diese Vorgängerversionen sind im Code-Stand der Arbeit nicht mehr als eigenständige Unterprojekte vorhanden – sie wurden im Iterationsverlauf jeweils vollständig durch die nachfolgende Architektur ersetzt; ihre Skiz-

zierung dient ausschließlich der Dokumentation der Designentscheidungen, aus denen die heutige Produktivpipeline hervorgegangen ist:

- **v2 – klassische JSON-Extraktion.** Ein einzelnes JSON-Objekt pro Ticket mit den Feldern `problem_summary`, `problem_category`, `affected_component`, `affected_version`, `root_cause`, `resolution_status`, `how_resolved`, `severity`, `error_messages`, `key_details`, `customer_request`, `support_action`, `final_outcome`, `evidence_spans`.
- **v3 – Topic-Chunking als eigene Pipeline-Stufe.** Ein dedizierter Topic-Chunker entschied je nach Länge des Tickets zwischen Pass-Through (kurz, ≤ 6 Nachrichten und $\leq 3\,000$ Zeichen) und LLM-gestützter Subtopic-Zerlegung. Pro Subtopic wurde eine narrativ formatierte Wissens-Einheit (**`**Problem:**`** ... **`**Investigation:**`** ... **`**Resolution:**`**) erzeugt. Diese Topic-Chunking-Logik wurde in die heutige Produktivpipeline übernommen (vgl. `zendesk/stage2_extraction/src/topic_chunker.py`).
- **v4 – Schema-Enforcement via Instructor.** Pilot mit der Bibliothek *Instructor*: Pydantic-erzwungenes Schema direkt im LLM-Aufruf, ohne manuelles Repair-Postprocessing.

Aus diesen drei Iterationen wurden die zentralen Lehren der heutigen Produktivpipeline gewonnen: (i) Topic-Chunking muss vor der Extraktion stattfinden (v3-Insight); (ii) Schema-Enforcement gehört in den LLM-Aufruf selbst, nicht ins Post-Processing (v4-Insight); (iii) eine separate Halluzinations-Verifikation ist nötig, weil rein prompt-basierte Anti-Halluzinations-Anweisungen nicht hinreichend sind.

4.2.2.10. RAG-Extraktionspipeline – aktuelle Produktivversion

Die aktuelle Produktivpipeline liegt im Unterprojekt `zendesk/stage2_extraction/` (ca. 3 400 Zeilen Python) und realisiert die in Abschnitt 3.3.1.11 konzeptionell beschriebene drei-stufige Architektur (Topic-Chunker → Extractor → Verifier). Die Modul-Aufteilung folgt einer strikten Single-Responsibility-Heuristik:

Tabelle 4.1.: Modul-Aufteilung der RAG-Extraktionspipeline

Datei	Funktion
<code>zendesk/stage2_extraction/src/pipeline.py</code>	Orchestrator + CLI; iteriert über USEFUL-Tickets, ruft Stages der Reihe nach auf, schreibt JSONL inkrementell mit Resume-Fähigkeit.
<code>zendesk/stage2_extraction/src/topic_chunker.py</code>	Stage 1 – Single/Multi-Entscheidung mit Short-Ticket-Shortcut, LLM-Aufruf via <code>raw_complete</code> , deterministischer Merge-Schritt.
<code>zendesk/stage2_extraction/src/extractor.py</code>	Stage 2 – Grammar-constrainer Aufruf zur Erzeugung des 6-Feld-Datenunits, normale vs. short Prompt-Variante, Nachbearbeitung.
<code>zendesk/stage2_extraction/src/verifier.py</code>	Stage 3 – Per-Sentence-KEEP/FIX/DROP-Aufruf, Verdict-Anwendung, Safety-Checks (Struktur-Erhaltung, Content-Shrinkage).
<code>zendesk/stage2_extraction/src/llm_client.py</code>	Wrapper um <code>llama-cpp-python</code> ; einmaliges Modellladen, Methoden <code>chat()</code> (für Stages 2/3) und <code>raw_complete()</code> (für Stage 1 mit eingebetteten Llama-3-Chat-Tokens).
<code>zendesk/stage2_extraction/src/grammars/topic.gbnf</code>	GBNF-Grammatik für Stage-1-Output (decision/reason/topics).
<code>zendesk/stage2_extraction/src/grammars/extraction.gbnf</code>	GBNF-Grammatik für Stage-2-Output (6 Felder, 12 Section-Enum).
<code>zendesk/stage2_extraction/src/grammars/verifier.gbnf</code>	GBNF-Grammatik für Stage-3-Output (Array von KEEP/FIX/DROP-Verdicts).
<code>zendesk/stage2_extraction/src/gate_check.py</code>	Pass/Fail-Auswertung der 7 Quality-Gate-Schwellen vor

Server-Workflow. Der Deployment-Prozess ist in `deploy_to_server.sh` skriptiert: `rsync` von `src/`, `data/` und `runs/` nach `/home/websrv38/data_processing/zendesk_chunking/` auf WEBSRV38. Der Run wird manuell in einer `screen`-Session gestartet, damit er einen SSH-Disconnect überlebt:

```
ssh websrv38 "screen -d -m -S chunking bash -c '
source ~/data_processing/.venv/bin/activate
cd ~/data_processing/zendesk_chunking
python src/pipeline.py \
--tickets data/useful_5411_tickets.json \
--output output/rag_data_units_full.jsonl \
--drops output/verifier_drops.jsonl \
--ctx 16384 --threads 16 --chat-format llama-3
,"
```

Vor dem Start wird der Produktiv-Service `support-backend.service` via `systemctl stop` angehalten, um die ca. 6 GB RAM für das Q5-Modell freizugeben (das Modell selbst benötigt ca. 5,5 GB, der KV-Cache mit `n_ctx=16384` zusätzlich ca. 3–4 GB). Während des Laufs kann mit `tail -f output/pipeline_*.log` der Fortschritt verfolgt werden. Resultate werden mit `scp` zurückgeholt und über `zendesk/stage2_extraction/src/snapshot.py` in das Verzeichnis `runs/<datum>_<name>/` mit `config.json`, `results.jsonl`, `metrics.json`, `per_ticket.json` und `notes.md` archiviert. Die Datei `runs/REGISTRY.md` dient als chronologischer Index aller Experimente – pro Zeile ein Lauf mit den wichtigsten Metrik-Werten.

4.2.2.11. Benchmark-Tooling

Drei Benchmark-Runner kapseln die Auswertung der Filter-Pipeline:

- `benchmark_qwen32.py` – Baseline-Runner für `pipeline.py` auf Qwen 2.5 32B; lädt das Split-GGUF, iteriert sequentiell über Golden 45 und Set 200, schreibt Ergebnisse inkrementell als JSONL mit Resume-Fähigkeit.
- `benchmark_qwen32_v2.py` – Runner für die v2-Override-Erweiterungen (`pipeline_qwen.py`); aufgrund der erhöhten Inferenzzeit pro Ticket (~80 s) ist der Set-200-Lauf nicht unter Standardbedingungen abschließbar.
- `benchmark_qwen32_v3.py` und `benchmark_qwen32_v3_golden_only.py` – Runner für die v3-Variante mit auf realen Rationales rekalierten Overrides; der Golden-only-Runner ermöglicht schnelle Iteration bei der Override-Entwicklung.

Während der Benchmark-Läufe wird der Produktiv-Service `support-backend.service` auf dem Server WEBSRV38 temporär gestoppt, um die ca. 22 GB RAM für die LLM-Inferenz freizugeben; nach Abschluss des Laufs wird der Service automatisch wieder gestartet.

4.2.3. PDF-Verarbeitung (shared)

Die PDF-Verarbeitung stellt keine eigenständige Datenquelle dar, sondern eine wiederverwendete Hilfs-Pipeline, die von zwei integrierten Quellen genutzt wird: der Vitruvi-Pipeline für die *Customer Terms of Service* (Abschnitt 4.2.4) sowie der HC-Pipeline für PDF-Anhänge der Help-Center-Artikel (Abschnitt 4.2.6.3). Sie erfolgt in zwei Phasen. Zunächst konvertiert der `pdf_converter` PDF-Dateien mittels Docling in strukturierte Dokumente:

```
def _create_converter():
    pipeline_options = PdfPipelineOptions()
    pipeline_options.do_generate_page_images = True
    pipeline_options.generate_picture_images = True
    pipeline_options.images_scale = 1.5
    return DocumentConverter(
        pipeline_options=pipeline_options,
        device="mps" # Apple Silicon
    )
```

Listing 4.12: PDF-Konvertierung mit Docling

Anschließend werden die konvertierten Dokumente durch den `HybridChunker` in semantisch kohärente Abschnitte aufgeteilt. Der Chunker respektiert die Abschnittsgrenzen der Originaldokumente und nutzt den BGE-M3-Tokenizer zur Kontrolle der Chunk-Größe:

```
def create_chunker():
    return HybridChunker(
        tokenizer="BAAI/bge-m3",
        max_tokens=512
    )
```

Listing 4.13: Semantisches Chunking

4.2.4. Externe Vitruvi-Pipeline

Während die bisher beschriebenen Pipelines firmeninterne Quellen verarbeiten, erfordert die Anbindung externer Vitruvi-Quellen (vgl. Abschnitt 3.3.6) eine eigene Codebasis unter `vitruvi/src/ingest/`. Die Implementierung verfolgt einen modularen Ansatz: Jede Quelle besitzt einen eigenen Crawler bzw. Konverter und einen eigenen Chunker, und alle münden in eine einheitliche Tabellenstruktur, aus der ein gemeinsamer Index gebildet wird.

4.2.4.1. Vertragsmodell: REQUIRED_COLUMNS

In `vitruvi/src/ingest/base.py` ist das Schema jedes Chunks als Klassenkonstante hinterlegt. Jeder Chunker muss diese Pflichtspalten füllen, bevor das Ergebnis als Parquet-Datei gespeichert wird:

```

REQUIRED_COLUMNS = [
    "chunk_id",          # eindeutige ID, z.B. "vitruvi-tos-v4_chunk_0042"
    "source_id",         # "vitruvi-tos-v4" | "vitruvi-docs-guides" | "
    vitruvi-api-v1"
    "source_type",       # "pdf" | "web_guide" | "openapi_endpoint"
    "source_url",        # Web-URL (None bei PDF)
    "fetched_at",        # ISO-8601 UTC (None bei statischen Quellen)
    "content_hash",      # sha256(text_raw) -> inkrementelle Rebuilds
    "text",              # Embedding-Input: Breadcrumb + Body
    "text_raw",          # nur Body ohne Breadcrumb
    "token_count",       # bge-m3 Tokenizer
    "char_count",
]

```

Listing 4.14: Verbindliches Chunk-Schema für alle externen Pipelines

Die zentrale Modul-registry überprüft die Konformität bei jedem Lauf und konkateniert die per-Quelle-Parquets zu `vitruvi/data/processed/chunks.parquet`, die anschließend als Eingabe für den Embedder dient. Damit lässt sich eine neue Quelle hinzufügen, ohne den Embedding- oder Index-Code anzufassen.

4.2.4.2. Web-Crawler für das Developer-Portal

Der Crawler in `vitruvi/src/ingest/web/crawler.py` besteht aus drei Discovery-Strategien, die als Union ausgeführt werden, um die in Abschnitt 3.3.6.2 beschriebene Lückenhaftigkeit der `sitemap.xml` der ReadMe.io-Plattform auszugleichen. Listing 4.15 zeigt das prinzipielle Vorgehen.

```

def discover_slugs(base_url: str) -> set[str]:
    slugs: set[str] = set()
    # 1) Official sitemap (XML)
    slugs.update(parse_sitemap(f"{base_url}/sitemap.xml"))
    # 2) llms.txt (Markdown list, often more complete)
    slugs.update(parse_llms_txt(f"{base_url}/llms.txt"))
    # 3) Fallback: HTML sidebar
    slugs.update(parse_html_sidebar(f"{base_url}/docs"))
    return slugs

```

Listing 4.15: Drei-fache Discovery-Strategie für das Developer-Portal

Für die Inkrementalität pflegt der Crawler eine Manifest-Datei (`vitruvi/data/raw/web/manifest.json`) mit einem Eintrag pro gecrawlter Seite, der `slug`, `etag`, `last_fetched`, `content_hash` und `http_status` speichert. Bei jedem Folge-Lauf wird der ETag-Wert als If-None-Match-Header übertragen, sodass unveränderte Seiten vom Server mit HTTP 304 quittiert werden und nicht erneut verarbeitet werden müssen.

Zur Schonung des Vitruvi-Servers wird ein Token-Bucket-Rate-Limiter (Default: 5 Requests pro Sekunde) eingesetzt; die `robots.txt` wird beim Crawl-Start gelesen und respektiert. Bei HTTP-429- oder 5xx-Antworten erfolgt ein dreifacher Retry mit exponentiellem Backoff (1 s, 2 s, 4 s), und der `Retry-After-Header` wird beachtet. Speicheroperationen verwenden Atomic-Writes über eine `.tmp`-Datei und anschließendes `replace()`, um Crash-Sicherheit zu gewährleisten.

4.2.4.3. Chunker für PDF, Web-Guides und OpenAPI

Die drei externen Quellen erfordern strukturell unterschiedliche Chunking-Strategien. Tabelle 4.2 fasst die wesentlichen Parameter zusammen.

Tabelle 4.2.: Vergleich der drei Chunker-Implementierungen für die externen Vitruvi-Quellen.

Eigenschaft	<code>clause_chunker.py</code> (PDF)	<code>guide_chunker.py</code> (/docs)	<code>openapi_chunker.py</code> (/reference)
Strategie	Clause-Hierarchie via Regex	Heading-aware Markdown-Split	NL-Rendering pro Endpoint
Max. Tokens	512 (Klausel-respektierend)	512 (<code>merge_peers=True</code>)	1 Endpoint = 1 Chunk
Quell-Metadaten	<code>section_no</code> , <code>page_start/end</code> , <code>chunk_type</code>	<code>slug</code> , <code>headings</code> , <code>chunk_type</code>	<code>http_method</code> , <code>http_path</code> , <code>operation_id</code> , <code>auth_schemes</code>
Besonderheit	Tabellen werden als GFM-Markdown angehängt	Code-Blöcke bleiben atomar	<code>\$ref</code> -Auflösung in OpenAPI-Schemata
Ergebnis	105 Chunks, Ø 210 Tokens	127 Chunks, Ø 229 Tokens	1.059 Chunks, Ø 198 Tokens

Clause-Chunker für die ToS. Die in `vitruvi/src/ingest/pdf/clause_chunker.py` implementierte Pack-Logik erkennt zunächst die drei Hierarchie-Ebenen über reguläre Ausdrücke und führt anschließend einen klausel-respektierenden Packlauf durch:

```
TOP_CLAUSE = re.compile(r"^(\\d{1,2})\\s+([A-Z][A-Z\\s]{2,})$")
SUB_CLAUSE = re.compile(r"^(\\d+(?:\\.\\d+)+)\\s+(.+)")
LETTERED   = re.compile(r"^(\\w)\\s+(.+)")

def classify_line(line: str) -> tuple[str, ...]:
    if m := TOP_CLAUSE.match(line):
        return ("top",) + m.groups()
    if m := SUB_CLAUSE.match(line):
        return ("sub",) + m.groups()
    if m := LETTERED.match(line):
```

```

    return ("letter",) + m.groups()
    return ("body", line)

```

Listing 4.16: Hierarchie-Erkennung im Clause-Chunker (vereinfacht)

Guide-Chunker für Docling. Für die Markdown-Guides aus `/docs` kommt der `HybridChunker` der `Docling`-Bibliothek zum Einsatz, wie er bereits in der internen PDF-Pipeline (vgl. Abschnitt 4.2.3) verwendet wird. Der Aufruf ist identisch konfiguriert (`max_tokens=512`, `merge_peers=True`); der Unterschied liegt nur in der Eingabeform (Markdown statt PDF).

NL-Rendering für die OpenAPI-Referenz. `vitruvi/src/ingest/web/openapi_chunker.py` extrahiert aus jeder `/reference`-Seite den eingebetteten OpenAPI-JSON-Block, löst `$ref`-Verweise einmalig auf und erzeugt aus jedem Operation-Objekt einen natürlichsprachlichen Chunk. Listing 4.17 zeigt den prinzipiellen Aufbau eines gerenderten Chunks.

```

POST /api/v1/agent-logs (operation: createAgentLog)
Description: Creates a new AI agent log entry...
Authentication: jwtAuth or tokenAuth.
Path parameters: none.
Query parameters: none.
Request body (required): JSON with fields
  - agentId (string, required)
  - eventType (enum: started|stopped|failed)
  - payload (object, optional)
Responses:
  - 201 Created -> AgentLog object
  - 401 Unauthorized
  - 422 Validation error
Tags: agent-logs

```

Listing 4.17: Beispiel-Output des NL-Rendering pro Endpoint

Diese Repräsentation ermöglicht es dem BGE-M3-Embedder, semantische Nähe zwischen Nutzeranfragen wie „wie erstelle ich einen Agent-Log-Eintrag?“ und dem zugehörigen Endpoint herzustellen – eine Eigenschaft, die bei einer rohen JSON-Indexierung verloren ginge.

4.2.4.4. Registry und FAISS-Integration

Nach Abschluss der drei Chunker liegen drei per-Quelle-Parquets vor (`chunks_tos.parquet`, `chunks_guides.parquet`, `chunks_openapi.parquet`). Die Funktion

`rebuild_canonical()` in `vitruvi/src/ingest/registry.py` konkateniert diese in eine einheitliche Datei (`chunks.parquet`, 1.291 Zeilen), prüft auf duplikate `chunk_id`-Werte und sortiert nach `source_id`.

Diese kanonische Datei wird vom Embedder eingelesen, mit BGE-M3 in 1.024-dimensionale Vektoren überführt und in einen `IndexFlatIP-FAISS`-Index geschrieben. Da die Vektoren ℓ_2 -normalisiert werden, entspricht das innere Produkt der Kosinus-Ähnlichkeit. Eine Partitionierung des Indexes nach Quelle ist nicht vorgesehen; stattdessen erfolgt eine optionale Quellfilterung bei Bedarf zur Laufzeit über die `source_id`-Spalte des begleitenden DataFrames.

4.2.5. TKI-Pr-KB-Pipeline

Die in Abschnitt 3.3.7 eingeführte TKI-Pr-KB benötigt – anders als die Vitruvi-Pipeline – weder Crawler noch Authentifizierung: Die acht Modul-Dokumente sind statische Markdown-Dateien unter `tki_products_kb/data/source/` und liegen damit als kanonische Eingabe bereits in repository-eingecheckter Form vor. Die Pipeline besteht aus zwei Schritten – Chunking und Embedding – und ist als zwei sequentiell ausführbare Jupyter-Notebooks unter `tki_products_kb/notebooks/` implementiert.

4.2.5.1. Frontmatter-Parser

Jedes Modul-Dokument beginnt mit einem YAML-Block zwischen `---`-Trennern, in dem das Modul, die analysierten Code-Pfade und Confluence-Spaces, die zugehörigen Lizenzen und die Querverweise zu anderen Modulen deklariert sind. `tki_products_kb/src/frontmatter.py` schneidet diesen Block ab und parst ihn mit PyYAML. Da das frei verfasste YAML in einer Datei nicht streng konform ist (z. B. enthält `net-inhouse.md` die Zeile `- "NET Inhouse- TKI-Markennamen...`, die PyYAML als Mapping-Fehler ablehnt), wurde ein *Regex-Fallback* ergänzt: Schlägt das vollständige Parsing fehl, werden zumindest die flachen Skalarwerte (`module`, `language`, `analysis_date`, `scope`) per Zeilen-Regex extrahiert. Damit kann das kritische Pflichtfeld `module` immer aufgelöst werden, ohne dass ein einzelner fehlerhafter Listeneintrag das gesamte Dokument verwirft.

4.2.5.2. Hierarchischer Chunker

`tki_products_kb/src/md_chunker.py` läuft anschließend über den Markdown-Körper und nutzt die natürliche Heading-Hierarchie (`#`, `##`, `###`) als primären Splitting-Hinweis. Jeder Section wird ein `section_path`-Array aus den umgebenden Heading-Titeln zugeordnet (etwa `["Lizenzen", "Standard Edition"]`), das später als Breadcrumb in das

Embedding-Text-Präfix sowie in das UI-Label einfließt. Die Größenkontrolle erfolgt tokenbasiert über `tiktoken` mit Ziel von 500 Token pro Chunk und 80 Token Overlap zwischen aufeinanderfolgenden Chunks; übersteigt eine einzelne Section dieses Ziel, wird sie in mehrere Schiebefenster über Satzgrenzen aufgeteilt. Sätze, die für sich allein das Maximum überschreiten (Code-Blöcke ohne natürliche Satzenden), werden mit einem Hard-Slice auf Token-IDs fallbacking versorgt, sodass kein Chunk das harte Maximum überschreitet. Sehr kleine Sub-Chunks (<80 Token) werden mit ihrem unmittelbaren Vorgänger unter demselben Parent-Heading verschmolzen, um Rauschen aus Einzeiler-Subsections zu vermeiden. Der Output ist eine flache Liste von Chunk-Datenklassen mit `section_path`, `heading_level`, `text` und `token_count`.

4.2.5.3. Notebook-Pipeline

Das Notebook `tiki_products_kb/notebooks/01_chunk.ipynb` kombiniert die beiden vorherigen Bausteine und schreibt das Ergebnis direkt in der *einheitlichen* Spalten-Schema-Form, die der nachgelagerte Merge erwartet. Damit entfällt – im Gegensatz zur Vitruvi-Pipeline mit ihrer dreistufigen Quelle-zu-Schema-Konvertierung – jede Mapping-Funktion in der späteren Index-Konsolidierung. Konkret enthält jede ausgegebene Zeile die Felder `ticket_id` (Format `tiki-<filename-stem>_chunk_<NNNN>`), `source = tiki_products_kb`, `source_subtype = tiki_module_doc`, `title = <module>: <last_heading>`, `text` (Chunk-Body mit vorangestelltem Heading), `embedding_text` (das gleiche, jedoch mit Breadcrumb-Präfix `<module> > <section_path>`), `product` (Modul aus der Frontmatter), `language` (i.d.R. `de`), `quality = verified`, `quality_weight = 1,0` sowie `metadata_json` (vollständige Frontmatter plus `section_path`, `chunk_index`, `chunk_count`).

Der breadcrumb-präfixierte `embedding_text` ist bewusst so geformt, dass das Cross-Encoder-Reranking den Modul-Kontext beim Scoring sieht („NET Engineering > Lizenzen > Standard Edition“); ohne dieses Präfix bevorzugt das Reranking-Modell systematisch längere Sections gegenüber knappen Definitions-Chunks, weil die Modul-Identität sonst nur im UI-Titel auftaucht.

Das zweite Notebook (`02_embed.ipynb`) lädt das in der ersten Stufe erzeugte `chunks.parquet`, kodiert die `embedding_text`-Spalte mit BGE-M3 (Batch-Größe 8 auf Apple-Silicon-MPS) und prüft anschließend, dass alle Vektoren auf $\|v\|_2 \approx 1$ normalisiert sind. Output sind zwei NumPy-Arrays unter `tiki_products_kb/data/embeddings/`: `vectors.npy` mit Form `(329, 1024)` und `chunk_ids.npy`, das die zugehörige Reihenfolge der `ticket_id`-Werte trägt. Diese beiden Dateien sowie die `chunks.parquet` sind die Eingaben, die der zentrale Merge-Schritt im React-App-Verzeichnis als dritte Quelle einliest (vgl. Abschnitt 4.2.7).

4.2.6. HC-Pipeline

Die in Abschnitt 3.3.5 eingeführte HC-Wissensbasis erfordert – ähnlich der Vitruvi-Pipeline – eine eigene Codebasis, weil sie über eine Web-API erschlossen wird und drei strukturell unterschiedliche Inhaltsklassen verarbeitet. Die Implementierung folgt der Vitruvi-Konvention (Abschnitt 4.2.4): pro Inhaltsklasse ein eigenes Chunker-Modul unter `zendesk_help_center/src/ingest/`, alle mit einer gemeinsamen Spalten-Vertragsdefinition in `ingest/base.py` und einem zentralen `rebuild_canonical()` in `ingest/registry.py`, der die per-Klassen-Parquets schemavalidiert konkateniert. Die operative Steuerung erfolgt über sechs nummerierte Jupyter-Notebooks unter `zendesk_help_center/notebooks/`, denen jeweils ein gleichwertiges Python-Skript unter `zendesk_help_center/scripts/` zur Seite gestellt ist (siehe Abschnitt 4.2.6.5).

4.2.6.1. API-Anbindung und Download

Der HC-Zugriff erfolgt vollständig über die offizielle HC-API (`/api/v2/help_center/de/`), nicht über Web-Scraping. `zendesk_help_center/src/ingest/api/client.py` kapselt eine generische paginierte HTTP-Schicht: jede Listen-Endpoint-Antwort enthält einen `next_page`-Cursor, dem der Iterator folgt; HTTP-429-Antworten werden über den `Retry-After-Header` und einen exponentiellen Backoff in `tenacity` robust abgefangen. `api/articles.py` ruft in fester Reihenfolge `/categories.json`, `/sections.json`, `/articles.json` sowie pro Artikel `/articles/{id}/attachments.json` ab und schreibt die Rohdaten als separate JSON-Dateien unter `data/raw/`. Der binäre Downloader in `api/attachments.py` arbeitet *resumable*: jede Datei, die mit der erwarteten Bytezahl bereits auf der Festplatte vorliegt, wird übersprungen; einzelne Fehlschläge werden in `data/raw/attachments_download_stats.json` protokolliert und blockieren den Gesamtlauf nicht. Bild-Anhänge sind in einer expliziten *Skip-Liste* der MIME-Typen aufgeführt und werden in der vorliegenden Iteration nicht geladen (vgl. die Begründung in Abschnitt 3.3.5).

4.2.6.2. HTML-Chunker für Artikelkörper

`zendesk_help_center/src/ingest/html/article_chunker.py` überführt den HTML-Körper jedes Artikels zunächst nach Markdown mittels `markdownify` mit den Parametern `heading_style=ÄTX`, `bullets=""` und `strip=[img]`. Die Library reproduziert GFM-Tabellen, verschachtelte Listen, fenced und inline Code sowie die Heading-Hierarchie verlustfrei – eine Eigenschaft, die der vorherige reguläre-Ausdruck-basierte Stripper nicht erfüllte und die in der separat dokumentierten A/B-Auswertung (`zendesk_help_center/research/html_conversion_ab.md`) auf einer Stichprobe von 30 Artikeln quantitativ belegt ist. Auf dem produzierten Markdown läuft anschließend ein geteilter, Header-basierter

Splitter: er erkennt `###/###/####` als Sektionsgrenzen, pflegt einen `section_path`-Stack über die umgebenden Heading-Titel und respektiert ein Token-Budget von 500 Tokens je Chunk mit 80 Token Overlap. Übersteigt eine Sektion das Budget, wird sie satz-granular mit Sliding-Window aufgeteilt; einzelne Sätze, die selbst über dem Limit liegen (z. B. eingebettete Code-Blöcke ohne Satzgrenzen), werden über eine Token-ID-Hartabtrennung versorgt, sodass kein Chunk das harte Maximum überschreitet. Sub-Chunks unter 80 Token werden mit ihrem unmittelbaren Vorgänger unter demselben Parent-Heading verschmolzen. Der gleiche Splitter wird auch von den PDF- und Office-Chunkern wiederverwendet (Abschnitte 4.2.6.3 und 4.2.6.4), was strukturelle Eigenschaften – Token-Verteilung, `section_path`-Form – subtypübergreifend identisch hält.

4.2.6.3. PDF-Chunker mit Docling

`zendesk_help_center/src/ingest/pdf/pdf_chunker.py` verwendet – analog zur Vitruvi-PDF-Pipeline (`vitruvi/src/ingest/pdf/pdf_converter.py`) – Docling als PDF-zu-Markdown-Konverter. Die `PdfPipelineOptions` sind identisch konfiguriert: MPS-Beschleunigung, vier Threads, `do_picture_description=False` und `generate_picture_images=False`, da Bild-Annotationen in dieser Iteration weiterhin außerhalb des Scope liegen. Vorteile gegenüber einer rohen PyMuPDF-Textauszug-Variante sind in der Praxis vor allem: (i) die strukturwahre Extraktion mehrspaltiger und mehrzeilig-zusammengeführter Tabellen aus den Release-Notes-PDFs, deren `TC_PL_*`-Schema-Beschreibungen sonst zu unleserlichen Zeilen verschmelzen; (ii) ein eingebauter OCR-Pfad für die zwei Customer-Export-PDFs („E.discom-Export“, „TANNIS-Export“), der ohne separates Tesseract-Setup auskommt und in der Messung etwa um den Faktor zwei schneller als die PyMuPDF/Tesseract-Kette terminiert. Der konvertierte Markdown-String wird anschließend an denselben Splitter aus Abschnitt 4.2.6.2 übergeben.

4.2.6.4. Office-Chunker für DOCX und Legacy-DOC

`zendesk_help_center/src/ingest/office/doc_chunker.py` parst `.docx`-Dateien über `python-docx` auf Absatz-Ebene und promotiert Word-Stile *Heading 1* und *Title* zu `##`-Markdown-Headings sowie *Heading N* für $N \geq 2$ zu `###`. Tabellen werden als vollständige GFM-Tabellen ausgegeben, einschließlich der Separator-Zeile `| --- |`, sodass der gemeinsame Splitter sie als Tabellen-Struktur erkennt. Für das einzige Legacy-DOC-Dokument im Korpus (eine Bauleiter-Vorlage) wird auf das CLI-Werkzeug `antiword` zurückgegriffen, falls dieses im PATH installiert ist; andernfalls wird die Datei mit einer expliziten Warnung im Notebook-Summary übersprungen, sodass kein stillschweigender Datenverlust eintritt.

4.2.6.5. Merge, Embedding und Laufzeit

Das Notebook 06_merge_and_embed.ipynb und sein äquivalentes Skript `scripts/run_merge_and_embed.py` rufen zuerst `rebuild_canonical()` auf, das die drei per-Klassen-Parquets schemavalidiert konkateniert und ein *deterministisch sortiertes* `chunks.parquet` schreibt; anschließend kodiert `src/embedding/embedder.py` die `embedding_text`-Spalte mit BGE-M3 auf MPS und persistiert `vectors.npy` und `chunk_ids.npy` unter `data/embeddings/`. Für die beiden langlaufenden Stufen (PDF-Konvertierung über 120 Dateien sowie das BGE-M3-Encoding über mehrere tausend Chunks) wurde `jupyter nbconvert` in der Praxis als unzuverlässig identifiziert – es signalisierte mehrfach einen erfolgreichen Exit-Code, ohne die produzierten Zellen-Outputs zu persistieren. Daher steht zu jedem dieser beiden Schritte ein gleichwertiges Standalone-Skript (`scripts/run_chunk_pdfs.py` bzw. `scripts/run_merge_and_embed.py`) bereit, das alle 20 Dokumente einen Parquet-Snapshot schreibt und sich damit auch bei einem Abbruch der Sitzung resumebar verhält. Die produzierten Artefakte – `chunks.parquet` mit 6.438 Zeilen sowie `vectors.npy` mit Form (6438, 1024) – sind die Eingaben, die der zentrale Merge-Schritt im React-App-Verzeichnis als *vierte* Quelle einliest (vgl. Abschnitt 4.2.7).

4.2.7. FAISS-Index und Embedding-Pipeline

Der Aufbau des FAISS-Index erfolgt offline in drei Schritten:

```
def build_index(vectors_path, index_path):
    vecs = np.load(vectors_path).astype("float32")
    index = faiss.IndexFlatIP(vecs.shape[1])
    index.add(vecs)
    faiss.write_index(index, str(index_path))
    return index
```

Listing 4.18: FAISS-Index-Erstellung

Die Embedding-Erzeugung verwendet das BGE-M3-Modell mit L2-Normalisierung:

```
model = SentenceTransformer("BAAI/bge-m3",
                           device="mps")
vectors = model.encode(texts,
                       normalize_embeddings=True,
                       batch_size=1,
                       show_progress_bar=True)
```

Listing 4.19: Embedding-Erzeugung

Zur Laufzeit werden alle Artefakte (FAISS-Index, Vektor-ID-Mapping, DataFrame) mittels `functools.lru_cache` gecacht, um wiederholtes Laden zu vermeiden.

4.2.8. Cross-Encoder Reranking

Der Cross-Encoder (`reranker.py`) bewertet Anfrage-Dokument-Paare und erzeugt Relevanz-Scores. Die Implementierung nutzt den BGE-Reranker-v2-M3 ohne Aktivierungsfunktion, da die rohen Logits als Score verwendet werden:

```
reranker = CrossEncoder("BAAI/bge-reranker-v2-m3",
                        automodel_args={
                            "torch_dtype": "auto"
                        },
                        trust_remote_code=True)
pairs = [(query, build_rerank_text(doc))
          for doc in candidates]
scores = reranker.predict(pairs)
```

Listing 4.20: Cross-Encoder-Scoring

Die Maximal Marginal Relevance (MMR) Selektion verhindert redundante Ergebnisse durch eine Abwägung zwischen Relevanz und Diversität:

```
def _mmr_select(candidates, texts, top_k, lam=0.7):
    selected = []
    remaining = list(range(len(candidates)))
    while len(selected) < top_k and remaining:
        best_idx = max(remaining,
                      key=lambda i:
                          lam * relevance[i]
                          - (1-lam) * max_similarity(i, selected))
        selected.append(best_idx)
        remaining.remove(best_idx)
    return [candidates[i] for i in selected]
```

Listing 4.21: MMR-Selektion (vereinfacht)

4.2.9. FastAPI-Server und Streaming-Pipeline

Der Streaming-Endpunkt `/api/pipeline-stream` ist das zentrale Element der Echtzeitkommunikation. Die Implementierung nutzt einen asynchronen Generator, der die drei Pipeline-Stufen (semantische Suche, Reranking, Antwortgenerierung) sequentiell durchläuft und nach jeder Stufe ein Server-Sent Event sendet. CPU-intensive Operationen (FAISS-Suche, Reranking, LLM-Inferenz) werden über `asyncio.to_thread()` in separate Threads ausgelagert, um den Event-Loop nicht zu blockieren. Den Kern der Token-Streaming-Stufe – die für die wahrgenommene Reaktivität entscheidende dritte Phase – zeigt Listing 4.22; die vollständige Endpunkt-Definition mit allen drei Stufen ist in Anhang A.7.6 dokumentiert.

```
# Stage 3: Draft Generation (Token-Streaming)
for token in draft_answer_stream(req.query, reranked):
    if token["type"] == "token":
        yield {"event": "draft_token",
              "data": token["token"]}
    elif token["type"] == "final":
        yield {"event": "complete",
              "data": json.dumps(result)}
```

Listing 4.22: Token-Streaming der Antwortgenerierung (Schlüsselausschnitt; vollständige Pipeline s. Anhang A.7.6)

4.2.10. Queue-Management

Ein Semaphore-basiertes Queue-Management (`queue_manager.py`) begrenzt die Anzahl gleichzeitiger Pipeline-Ausführungen auf maximal 2, mit einer Warteschlange für bis zu 10 weitere Anfragen. Dies verhindert eine Überlastung des Servers, da das LLM allein ca. 5–6 GB RAM belegt.

4.3. Antwortpfad und Benutzerführung

Dieser Abschnitt beschreibt das Frontend nicht aus der Perspektive seiner visuellen Gestaltung, sondern aus der Perspektive des Antwortpfads – also dessen, was vom Klick des Support-Mitarbeitenden bis zur überprüfbar Antwort im Browser passiert. Die Darstellung folgt der Reihenfolge, in der ein neuer Nutzer das System erlebt: zunächst die leere Arbeitsfläche beim ersten Aufruf, dann die Konfigurationsmöglichkeiten, anschließend eine beispielhafte Antwort und schließlich die Live-Phasen, die während der Beantwortung sichtbar werden. Die nachgelagerten Unterabschnitte zur Quellenrückführung, Editierbarkeit und Diagnose vertiefen Aspekte, die in der Antwortdarstellung bereits implizit sichtbar geworden sind.

4.3.1. Erste Ansicht: Arbeitsbereich beim Aufruf

Beim Aufruf der Webanwendung erscheint die in Abbildung 4.3.1 gezeigte Arbeitsfläche. Die Fläche gliedert sich in vier Zonen, die jeweils ein fest umrissene Funktion tragen.

Die **Kopfzeile** trägt das Markenlogo, einen Umschalter zwischen Deutsch und Englisch und eine Schaltfläche, die das Einstellungs-Panel öffnet. In der **Seitenleiste** stehen drei vordefinierte Profile zur schnellen Wahl der Retrieval-Tiefe (*Fast*, *Balanced*, *Deep*) sowie der bisherige Anfragenverlauf der aktuellen Sitzung; die Profile sind ein Angebot an den Nutzer, keine Verpflichtung – jeder Parameter lässt sich in den Einstellungen frei justieren,

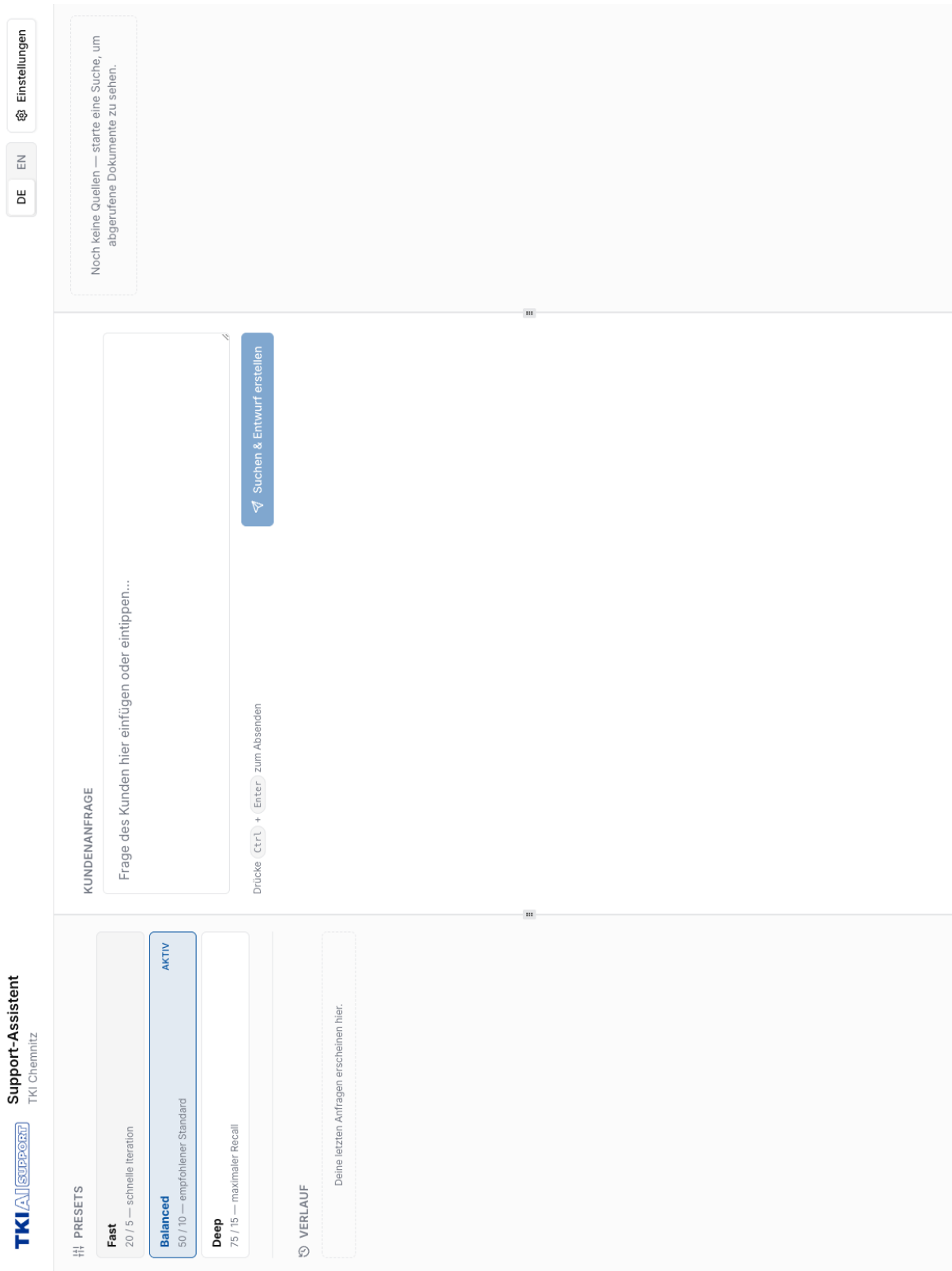


Abbildung 4.3.1.: Arbeitsbereich beim ersten Aufruf der Webanwendung. Oben die Kopfzeile mit Logo, Sprachwahl und Schaltfläche *Einstellungen*, links die Seitenleiste mit Profilen (*Fast, Balanced, Deep*) und Anfragenverlauf, in der Mitte das Eingabefeld für die Kundenanfrage und rechts die zunächst leere Quellenleiste mit dem Platzhalter *Noch keine Quellen – starte eine Suche*.

was Abschnitt 4.3.2 im Detail beschreibt. Die **Mittelspalte** enthält das Eingabefeld für die Kundenanfrage und ist nach dem Absenden der Anfrage zugleich Anzeigeort für Phasenstatus, Antwortentwurf und optionale Diagnose. Die **rechte Quellenleiste** bleibt vor der ersten Anfrage leer; sobald eine Anfrage abgeschickt wird, listet sie die retrieveden und rerangierten Dokumente in zwei Tabs (*Rerangiert*, *Alle*).

Technisch ist die Anwendung als React-19-Single-Page-Application in TypeScript implementiert und wird mit Vite 7 gebaut. Die Zustandsverwaltung erfolgt ausschließlich lokal über `useState`- und `useRef`-Hooks; persistente Werte (Einstellungen, Anfragenverlauf, Panelbreiten, Sprachpräferenz) werden im `localStorage` des Browsers abgelegt. Die Kommunikation mit dem Backend erfolgt über zwei Endpunkte: `GET /api/models` liefert die verfügbaren LLM-Varianten, `POST /api/pipeline-stream` liefert das Server-Sent-Events-Antwortprotokoll.

4.3.2. Einstellungen und Personalisierung

Die drei Profile in der Seitenleiste decken den häufigsten Fall ab, in dem der Nutzer einen vordefinierten Kompromiss aus Antwortqualität und Latenz wählt. Wer eine Anfrage gezielter steuern will, öffnet über die Kopfzeile das Einstellungs-Panel (Abbildung 4.3.2); dort lässt sich jeder Parameter der Pipeline einzeln justieren, ohne an die Profile gebunden zu sein.

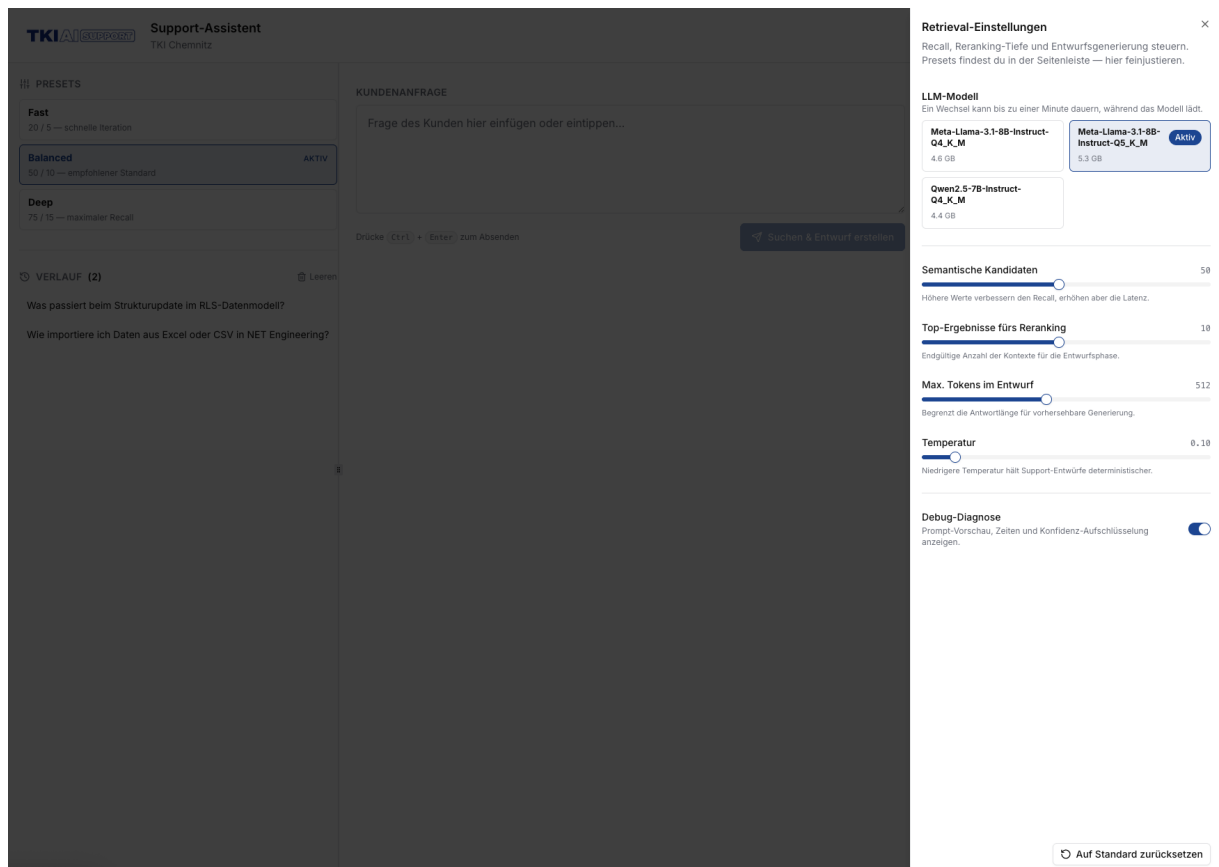


Abbildung 4.3.2.: Einstellungs-Panel mit Modellwahl und individuell justierbaren Retrieval-Parametern. Im oberen Bereich stehen die im Backend geladenen GGUF-Modelle zur Auswahl; darunter regulieren vier Schieberegler die Anzahl semantischer Kandidaten, die Reranking-Tiefe, die maximale Antwortlänge und die Temperatur des LLM-Samplings; ein Schalter aktiviert die Debug-Diagnose unterhalb der Antwort.

Das Panel ist in zwei Bereiche gegliedert. Oben steht die **Modellwahl**: jedes lokal verfügbare GGUF-Modell erscheint als Schaltfläche mit Namen und Speicherbedarf; der Wechsel wird durch einen POST an `/api/models/switch` ausgelöst und ist ohne Service-Neustart möglich, dauert je nach Modellgröße jedoch bis zu 60 Sekunden. Darunter folgen die vier zentralen **Retrieval- und Generierungsparameter**:

Die drei Profile in der Seitenleiste sind Voreinstellungen genau dieser vier Werte: *Fast* verwendet engere Treffermengen für eine schnellere Reaktion, *Balanced* entspricht den Standardwerten der Tabelle, *Deep* weitet die semantische und Reranking-Tiefe aus. Eine erweiterte Retrieval-Tiefe übergibt dem Modell mehr Kontextstellen und kann dadurch zusätzliche relevante Aspekte einer Frage erfassen; sie kann jedoch zugleich thematisch ähnliche, aber für die konkrete Frage nicht antwortrelevante Treffer einschließen und auf diese Weise die Klarheit der Antwort verwässern. Aus diesem Grund ist die Wahl des Profils nicht im Sinne einer Qualitätsabstufung zu verstehen, sondern als Abwägung zwischen

Tabelle 4.3.: Individuell justierbare Parameter im Einstellungs-Panel

Parameter	Bereich	Standard	Beschreibung
Semantische Kandidaten	5–100	50	Anzahl der Bi-Encoder-Treffer, die in das Reranking eingehen
Top-Ergebnisse Reranking	1–20	10	Endgültige Anzahl der Kontexte für die Entwurfsphase
Max. Tokens im Entwurf	128–1024	512	Begrenzt die Antwortlänge für vorhersehbare Generierung
Temperatur	0,0–1,0	0,3	Streuung der LLM-Sampling-Verteilung

Recall und Schärfe für den jeweiligen Anwendungsfall. Im Profilfall *Balanced* ist gleichzeitig die für die meisten Anfragen geeignete Voreinstellung gewählt; die übrigen Profile sind als Werkzeug für gezielte Iterationen vorgesehen, ohne dass der Nutzer dafür die Bedeutung der einzelnen Parameter kennen muss.

Eine Schaltfläche unten im Panel setzt sämtliche Werte auf die Standards aus Tabelle 4.3 zurück. Die Einstellungen werden im `localStorage` persistiert; ein Browser-Neustart erhält die letzte Konfiguration.

4.3.3. Beispielhafte Antwort und Ergebnisdarstellung

Sobald eine Anfrage abgeschickt ist, fügt die Mittelspalte die Phasenanzeige und nach Abschluss der Pipeline den Antwortentwurf ein; gleichzeitig füllt sich die rechte Quellenleiste mit den ausgewählten Treffern. Abbildung 4.3.3 zeigt diese Anordnung anhand der Anfrage „Wie importiere ich Daten aus Excel oder CSV in NET Engineering?“.

Die Antwort-Karte in der Mittelspalte enthält vier wiederkehrende Elemente, die in jedem späteren Beispiel in derselben Anordnung erscheinen. **Erstens** steht im Kopf der Karte die Konfidenzanzeige: ein Zahlenwert in der Form *Konfidenz: 0,67 (medium)* sowie ein farbig getönter Fortschrittsbalken; die Farbgebung folgt der vom Backend gelieferten Stufe und reicht von Grün (*high*) über Gelb (*medium*) bis Rot (*low*). **Zweitens** folgt der eigentliche Antwortentwurf als Fließtext mit eingebetteten *Zitations-Chips*: jeder Chip in einer Quellfarbe ist mit dem konkreten Quellchunk verknüpft und macht so jede Aussage des Modells für den Nutzer in einem Schritt überprüfbar (Abschnitt 4.3.5 beschreibt das Verfahren im Detail). **Drittens** listet ein abtrennbarer Bereich die vom Modell vorgeschlagenen *Lösungsschritte*; bei rein definitorischen Fragen darf das Modell diesen Bereich leer lassen. **Viertens** bieten Schaltflächen oben in der Karte die Möglichkeit, den Antworttext zu bearbeiten oder in die Zwischenablage zu kopieren (Abschnitt 4.3.6).

Oberhalb der Antwort werden außerdem die aus den Treffern extrahierten *Ressourcen-Links* angezeigt, sofern die zugehörigen Quellen eine externe URL führen. Damit lassen

TKI AI SUPPORT

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

VERLAUF (1)

Leeren

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

Drücke **Ctrl** + **Enter** zum Absenden

Abgeschlossen in **56.7s** — Semantische Suche 3.7s → Reranking 18.7s → Entwurfsgenerierung 34.3s

WEITERFÜHRENDE LINKS

<https://help.tki-chemnitz.de/hc/de/articles/23375378552498-Release-Notes-NET-14-0-3>

<https://help.tki-chemnitz.de/hc/de/articles/360014728199-Daten-importieren>

<https://help.tki-chemnitz.de/hc/de/articles/360018911759-Daten-importieren-und-exportieren>

<https://help.tki-chemnitz.de/hc/de/articles/2580485547730-Import-Excel>

<https://help.tki-chemnitz.de/hc/de/articles/4404696136082-Daten-mit-Excel-exportieren-und-importieren>

KUNDENANFRAGE

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

Suchen & Entwurf erstellen

Antwortentwurf

high

Bearbeiten

In Zwischenablage kopieren

Konfidenz: 0.87 (high)

87%

Um Daten aus Excel oder CSV in NET Engineering zu importieren, können Sie den allgemeinen Import verwenden, der ab Version NET 14.0.3 auch Excel- und CSV-Dateien unterstützt. **HC-RELEASE NOTES NET 14.0.3: OBJECT OBJECT** Sie können die Daten im Excel- oder CSV-Format nutzen und die Spaltenbenennung muss den Steuerelementen der Ziel-Objektklasse entsprechen.

NET ENGINEERING > NET ENGINEERING > 8. IMPORT > EXCEL-IMPORT (DETAIL) Wenn Sie Excel-Dateien importieren möchten, müssen Sie die Spalten **'GEOM.X'** und **'GEOM.Y'** in Excel hinzufügen, um die Geometrie zu importieren.

NET ENGINEERING > NET ENGINEERING > 8. IMPORT > EXCEL-IMPORT (DETAIL)

Lösungsschritte

1. Öffnen Sie den allgemeinen Import in NET Engineering.

2. Wählen Sie den Quelldatentyp als Excel oder CSV.

3. Laden Sie die Excel- oder CSV-Datei hoch.

4. Konfigurieren Sie die Spaltenbenennung und die Geometrie.

5. Starten Sie den Import-Prozess.

Alle (50)

Relangiert (10)

TKI Docs

Help Center

#1

HELP CENTER

Release Notes NET 14.0.3: **Import**

99%

HC - Release Notes & Downloads > Release Notes und Hinweise > Release NET 14.0.3 im Detail > NET Engineering > **Import**

#2

TKI DOCS

NET Engineering: Daten importieren und exportieren

99%

NET Engineering > NET Engineering > Erste Schritte (Section der Hilfesite) > Daten importieren und exportieren

#3

HELP CENTER

NET Design

Daten importieren: Import eines Comsof Fiber Workspace

99%

HC - NET Design > Planung vorbereiten > Import eines Comsof Fiber Workspace

#4

HELP CENTER

Release Notes NET 14.0.3: Zusammenfassung

99%

HC - Release Notes & Downloads > Release Notes und Hinweise > Zusammenfassung

#5

HELP CENTER

NET Engineering

Daten importieren und exportieren: Importieren/Exportieren über das Formular

98%

HC - NET Engineering > Erste Schritte > Importieren/Exportieren über das Formular

Abbildung 4.3.3.: Antwort auf eine deutschsprachige HC-Anfrage. Sichtbar sind in der Mittelspalte die abgeschlossene Phasenleiste mit den Latenzen je Stufe, die ausgewählten Ressourcen-Links der retrieveden Dokumente, der Antwortentwurf mit Konfidenzanzeige (0,67 medium) und inline eingebetteten Zitations-Chips sowie die Lösungsschritte. Rechts die Quellenleiste mit zehn rerangierten HC-Treffern.

sich primäre Verweise auch ohne den Umweg über das Diagnose-Panel oder die Quellenleiste in einem Klick öffnen.

4.3.4. Live-Phasen während der Anfrage

Vor dem in Abschnitt 4.3.3 gezeigten Endzustand durchläuft jede Anfrage die im Backend vorgesehenen Phasen sichtbar in der Mittelspalte. Abbildung 4.3.4 zeigt einen Ausschnitt aus dieser Phasenleiste während einer laufenden Anfrage.



Abbildung 4.3.4.: Live-Phasenleiste während einer Anfrage. Von links nach rechts: abgeschlossene semantische Suche mit 50 Kandidaten in 3,4 Sekunden, abgeschlossenes Reranking mit 10 Treffern in 12,8 Sekunden und die aktive Entwurfsgenerierung mit laufendem Sekundenzähler (2,6 Sekunden).

Jede Stufe der Pipeline wird durch ein eigenes Element dargestellt: ein Marker für *abgeschlossen*, *aktiv* oder *ausstehend*, die Stufenbezeichnung sowie ergänzende Werte (Anzahl gefundener Dokumente, Latenz der Stufe). Der Hook `usePipelineStream` kapselt die Kommunikation mit dem Streaming-Endpunkt und stellt einen explizit modellierten Phasenautomaten bereit:

```
type PipelineStage =
  | 'idle'           // keine aktive Anfrage
  | 'queued'        // Backend-Semaphore noch nicht freigegeben
  | 'searching'     // Bi-Encoder-Retrieval laeuft
  | 'reranking'     // Cross-Encoder + MMR
  | 'drafting'      // LLM generiert Token
  | 'done'          // Antwort abgeschlossen
  | 'error';        // Fehler aufgetreten
```

Listing 4.23: Phasen des Antwortpfads im Frontend

Jede vom Backend gesendete Nachricht (`search_done`, `rerank_done`, `draft_started`, `draft_token`, `complete`, `error`) löst eine gezielte Zustandsänderung aus. Während der Phase `drafting` werden die einzelnen Token vom Modell akkumuliert und sofort gerendert; der wahrgenommene Antwortbeginn liegt damit deutlich vor der eigentlichen Antwortlänge. Nach abgeschlossenem Reranking sind die Quellen in der rechten Leiste bereits sichtbar und das erste Antwort-Token erscheint binnen weniger Sekunden. Eine eingehende Diskussion der Wahl zwischen Mittelwert und 95. Perzentil als Berichtsgröße sowie die zugehörigen Latenz-Messwerte folgen im Ergebniskapitel (Abschnitt 5.6).

4.3.5. Quellenrückführung über Zitationsanker

Ein zentrales Designziel des Systems ist, dass jede Aussage im Antwortentwurf in einem konkreten Quellchunk verankert ist und der Nutzer diese Verankerung in einem Schritt überprüfen kann. Das System realisiert diese Eigenschaft über eine durchgehende *Zitations-Tag-Kette*, die im Backend beginnt und im Frontend als interaktiver Anker endet.

Tag-Erzeugung im Backend. Pro retrievedem Chunk berechnet die Funktion `build_citation_tag` einen semantischen Tag, dessen Form vom Quelltyp abhängt (Tabelle 4.4). Der Tag wird dem Chunk im Prompt als Kopfzeile vorangestellt; der Systemprompt instruiert das Modell, jeden Aussagesatz mit dem zugehörigen Tag verbatim (einschließlich der eckigen Klammern) zu zitieren. Damit zitiert das Modell selbsterklärende Quellen wie `[POST /auth/token/obtain]` statt anonymer Indizes wie `DOC 1`.

Tabelle 4.4.: Semantische Zitations-Tags je Quelltyp

Quelltyp	Tag-Format
Vitruvi-OpenAPI	<code>[METHOD /api/v1/.../]</code>
Vitruvi-Web-Guide	<code>[Vitruvi Docs > Authentication > 1. JWT]</code>
Vitruvi-ToS-PDF	<code>[§1.23 ToS p.6]</code>
Zendesk-Ticket	<code>[Ticket #18038]</code>
HC	<code>[HC: Artikel-Titel / Abschnitt]</code>
TKI-Produkte-KB	<code>[NET Engineering > GIS Export]</code>

Der Tag wird zugleich als Feld `citation_tag` an die `RetrievedDoc`-Antwort angehängt und damit auch im Frontend zur Verfügung gestellt.

Tag-Erkennung und Chip-Rendering im Frontend. Vor dem Markdown-Render lässt die Hilfsfunktion `linkifyDocRefs` den Antworttext durch und ersetzt jedes Vorkommen eines bekannten `citation_tag` durch einen Markdown-Link mit pseudo-URL der Form `#docref-<ticket_id>`. Der eigene `<a>`-Renderer der `ReactMarkdown`-Komponente erkennt dieses URL-Schema und rendert anstelle eines normalen Ankers eine `DocChip`-Komponente: einen kompakten farbigen Chip in der Quellfarbe, dessen Inhalt der Tag-Text ohne Klammern ist. Die Sortierung der Tags nach Länge garantiert, dass längere Tags zuerst gematcht werden – ein kürzerer Tag wie `[POST /auth]` kann nicht den Treffer des längeren `[POST /auth/token/obtain]` kapern. Ein zweiter Durchlauf greift defensiv ein, falls das Modell die eckigen Klammern weggelassen hat: für jeden Tag, dessen Klammer-Form nicht gefunden wurde, wird der Inhalt ohne Klammern gesucht; ein *Shadowing-Schutz* verhindert dabei, dass ein Subset-Tag innerhalb des bereits verlinkten Bereichs eines längeren

Tags erneut gematcht wird. Die korrespondierenden Korrektheitsfälle sind als 12 Unit-Tests in `linkifyDocRefs.test.ts` dokumentiert (vgl. Abschnitt 4.4.4).

Interaktion: Vorschau und Sprung zur Quelle. Bewegt der Nutzer den Mauszeiger über einen Chip, erscheint eine Vorschau-Karte mit Titel und einem Auszug von ca. 220 Zeichen aus dem zugehörigen Chunk. Ein Klick auf den Chip löst kein externes Navigieren aus, sondern emittiert ein Custom-Event auf dem `docrefBus`; die Komponente `SourcesPanelContent` hört auf dieses Event, wechselt gegebenenfalls auf den Tab, in dem das Dokument liegt, scrollt zur betreffenden `SourceCard` und markiert diese für zwei Sekunden mit einem farbigen Rahmen. Damit erreicht der Nutzer die Quelle, ohne die Antwort zu verlassen.

4.3.6. Editierbarkeit und Verantwortungskette

Das System versteht sich nicht als endkundenfähiger Chatbot, sondern als Assistenzwerkzeug für Support-Mitarbeitende. Der erzeugte Entwurf ist nicht das finale Kommunikationsartefakt; die letzte fachliche Verantwortung bleibt beim Support-Mitarbeiter, der den Vorschlag prüft, gegebenenfalls anpasst und dann erst in das Ticketsystem übernimmt. Die UI unterstützt diese Kette explizit:

Lese-Ansicht als Standard. Im Lese-Modus rendert die `DraftResponse`-Komponente den Antworttext über `react-markdown` samt `remark-gfm` und `Code-Block-Highlighting`. Die Zitations-Chips bleiben in dieser Ansicht aktiv und klickbar.

Bearbeiten-Schalter. Ein Klick auf *Bearbeiten* ersetzt die Lese-Ansicht durch eine `textarea` mit dem Roh-Markdown des Drafts und der Lösungsschritte. Eine Zurücksetzen-Schaltfläche stellt den ursprünglichen Modell-Output wieder her.

Kopier-Schaltfläche. Eine separate Schaltfläche kopiert den gegebenenfalls bearbeiteten Volltext (Antwort plus Lösungsschritte, getrennt durch eine Leerzeile) in die Zwischenablage; eine Toast-Bestätigung quittiert den Vorgang.

4.3.7. Diagnose-Panel

Für die Entwicklung und gezielte Fehleranalyse stellt die UI eine Diagnose-Sicht bereit, die nur bei aktivierter Debug-Option erscheint und unterhalb der Antwort als eigene Karte gerendert wird. Innerhalb eines Tab-Containers werden in einem Tab die vier Signale der Konfidenzaufschlüsselung sowie die Phasenlaufzeiten in Millisekunden tabelliert, im zweiten Tab der vollständige an das LLM gesendete Prompt im Klartext angezeigt. Damit lässt sich für jede einzelne Anfrage prüfen, welche Quellen in welchem Format an das Modell

gegangen sind – ein Kontrollwerkzeug, das während der iterativen Verfeinerung des Antwortpfads (Abschnitt 4.4) unentbehrlich war. Abbildungen 4.4.2 und 4.4.6 im folgenden Abschnitt zeigen das Diagnose-Panel im geöffneten Zustand für je eine Anfrage mit mittlerer und mit hoher Konfidenz.

4.3.8. Produktfilterung

Die Produktfilterung ist als reine API-Funktion realisiert: Der Streaming-Endpoint akzeptiert ein optionales Feld `products` im Request-Body, das eine Liste kanonischer Produktnamen enthält. Bei aktiviertem Filter vergrößert die FAISS-Suche die Kandidatenmenge um den Faktor 10 und filtert anschließend die Ergebnisse anhand der gewünschten Produkte. Eine dedizierte UI-Komponente für die manuelle Auswahl existiert in der aktuellen Fassung nicht; der Produktbezug wird je Treffer als kleines Etikett in der Quellenkarte dargestellt. Programmatische Aufrufer können das Feld weiterhin direkt setzen.

4.4. Funktionale Korrektheit des Endprodukts

Während Abschnitt 4.3 beschreibt, wie der Antwortpfad strukturell aufgebaut ist, dokumentiert der vorliegende Abschnitt vier Aspekte seiner Korrektheit: die iterative Verfeinerung mit ihren empirisch ausgelösten Designentscheidungen, exemplarische End-zu-End-Anfragen, die Latenz-Charakteristik unter realer Last und die Abdeckung durch Komponententests.

4.4.1. Iterative Verfeinerung des Antwortpfads

Die produktive Antwortpipeline ist nicht in einem Wurf entstanden, sondern in fünf eigenständig motivierten Korrekturen, die jeweils auf einer konkreten Fehlbeobachtung im laufenden System beruhen. Jede Korrektur wird im Folgenden mit ihrer auslösenden Beobachtung, der zugrunde liegenden Ursache und ihrem Effekt dokumentiert.

1. Sigmoid-Aktivierung am Cross-Encoder-Output. Der Cross-Encoder *BGE-Reranker-v2-M3* wurde initial mit `default_activation_function=None` instanziiert, sodass dessen Ausgabe *Roh-Logits* im Bereich von ca. -3 bis $+5$ waren. Der konfigurierbare Schwellenwert `RERANK_SCORE_THRESHOLD` mit dem Standardwert `0,6` wurde im Filterschritt direkt mit diesen Logits verglichen, obwohl der Kommentar in der Konfigurationsdatei den Wert als „60 % Cross-Encoder-Konfidenz“ bezeichnete. Beobachtet wurde dieser Mangel bei deutschsprachig formulierten Anfragen über Datenbank-Tabellennamen (z. B. `TC_SP_MANHOLE`): obwohl die FAISS-Suche im Top-1-Treffer den exakt passenden HC-Artikel auswarf, lieferte das Reranking *null* überlebende Ergebnisse, der Entwurf

erhielt keinen Kontext und das System antwortete mit einer *Not-Covered*-Meldung. Die Behebung umfasst zwei Zeilen Codeänderung: Wrappen des Cross-Encoder-Outputs in `torch.nn.Sigmoid()`, womit die Ausgabe in $[0, 1]$ liegt und der Schwellenwert die im Kommentar gemeinte Semantik tatsächlich annimmt.

2. Absenkung des Reranker-Schwellenwerts. Selbst nach der Sigmoid-Aktivierung lag die Score-Verteilung deutschsprachiger Antwort-Chunks bei Code-style Anfragen typischerweise zwischen 0,45 und 0,55. Eine Anhebung des Schwellenwerts auf 0,6 verworfen einen relevanten Teil dieser Chunks unnötig. Eine empirische Anpassung über drei Iterationen (Standard 0,6 $\rightarrow$ 0,45 $\rightarrow$ 0,4) zeigte, dass der Wert 0,4 einen guten Kompromiss zwischen Rauschunterdrückung und Recall darstellt; tiefer eingestellt erschienen wiederholt thematisch nahe, aber nicht antwortrelevante Tickets in der Auswahl.

3. Refaktorisierung des Systemprompts. Die augenfälligste Korrektur betraf nicht die Retrieval-, sondern die LLM-Stufe. Der ursprüngliche Systemprompt enthielt zwei Formulierungen, die in Kombination zu Selbstwiderspruchs-Antworten führten: erstens den eng gefassten Identitätssatz „You are a technical support assistant for TKI software products (NET Design, NET Engineering, NET Build, Infrastructure Administrator)“, der das Modell auf einen Quellbereich verengt, der den Vitruvi-Anteil nicht abdeckt; zweitens die wiederholte Hedging-Aufforderung „If documents lack sufficient information, state this explicitly“. Bei strukturierten OpenAPI-Chunks, die für die Anfrage „How do I export accounts payable items via the API?“ mit einem 99%-Reranker-Score ausgewählt wurden, antwortete das Modell mit dem für eine Bachelorarbeit untypisch selbstwidersprüchlichen Satz „Unfortunately, the provided documents do not contain information on how to export accounts payable items via the API. However, they do provide information on how to export accounts payable items using a GET request to the /api/v1/reporting/accounts_payable_items_export/endpoint“. Eine vergleichende Ausführung über das in `vitruvi/notebooks/99_demo_rag.ipynb` dokumentierte Forschungs-Notebook – mit identischem Stack, aber neutralerem Prompt – lieferte für dieselbe Anfrage eine klare, vollständige Antwort.

Der überarbeitete Systemprompt verzichtet auf die produktspezifische Eingrenzung, ersetzt die Hedging-Aufforderung durch eine knappe Drei-Regel-Form und definiert die Zitationsinstruktion über das ebenfalls neu eingeführte Tag-System (Abschnitt 4.3.5). Die `<STEPS>`-Klausel wurde von verpflichtend auf optional umgestellt, sodass das Modell bei rein definitorischen Anfragen den Block leer lassen darf, statt halluzinierte Folgeschritte zu erfinden.

4. Semantische Zitations-Tags statt anonymer Indizes. Im ursprünglichen Prompt wurden Chunks mit Headern der Form `[DOC 1 | score=0,99 | source=vitruvi]` eingelei-

tet, und die Zitations-Instruktion lautete „Cite sources as (DOC N)“. Das führte zu mechanischen Verweisen wie „... as described in DOC 1 and DOC 2“, die im UI nicht klickbar gemacht werden konnten, weil die Indizes keinen semantischen Bezug zur Quelle tragen. Mit der Einführung der semantischen Tags (vgl. Tabelle 4.4) zitiert das Modell selbsterklärende Anker wie [POST /auth/token/obtain], die zugleich vom Frontend in interaktive Chips übersetzt werden.

5. Erweiterter Dokumentenkontext. Der Parameter LLM_MAX_DOCS, der die Zahl der dem Modell übergebenen Chunks begrenzt, wurde von 5 auf 7 angehoben. Bei komplexen Mehr-Aspekt-Anfragen (z. B. „Wie funktioniert der Vitruvi Mobile Quick Start?“) führte der engere Standardwert dazu, dass auch nach Threshold-Senkung relevante Chunks außerhalb der Top-5 lagen und damit dem Modell unbekannt blieben.

Wirkung der Iterationen im Gesamtbild. Die fünf Korrekturen sind nicht voneinander unabhängig, sondern verstärken sich gegenseitig: Sigmoid und Schwellenwert sorgen dafür, dass thematisch nahe Chunks überhaupt das Reranking überstehen; die Tag-Architektur stellt sicher, dass diese Chunks im Antworttext zitiert und vom Nutzer überprüft werden können; der überarbeitete Prompt verhindert, dass das Modell trotz vorliegender Quellen ausweichend antwortet. Die Wirkung dieser Iterationen lässt sich an den im folgenden Abschnitt 4.4.2 dokumentierten Anfragen ablesen: dieselben strukturellen Quellen (Vitruvi-OpenAPI-Endpunkte, deutschsprachige HC-Artikel, TKI-Pr-KB), die unter dem ursprünglichen Prompt zu *Not-Covered*- oder Selbstwidersprüchlichen Antworten führten, liefern unter der überarbeiteten Pipeline vollständige, mit klickbaren Zitations-Chips ausgestattete Entwürfe.

4.4.2. Exemplarische Anfragen und Antworten

Zur Überprüfung der End-zu-End-Korrektheit wurde ein heterogenes Set von sieben Anfragen geprüft, das die in der produktiven Wissensbasis vertretenen Quellen (HC, TKI-Pr-KB, Vitruvi), beide Sprachen (Deutsch, Englisch) und mehrere Anfragetypen (kurze definitorische, sachlich-prozedurale, mehrschrittige Workflow-Beschreibungen) abdeckt. Tabelle 4.5 fasst die Anfragen mit der gemessenen Konfidenzbewertung, dem aktivierten Profil und einer qualitativen Bewertung der Antwort zusammen. Die zugehörigen Antwortansichten finden sich in den Abbildungen 4.4.1–4.4.6; die erste der Anfragen ist bereits in Abbildung 4.3.3 aus Abschnitt 4.3.3 gezeigt und dient deshalb in der Tabelle ohne erneute Abbildung als Referenz.

Sechs der sieben Anfragen werden korrekt beantwortet; eine Anfrage (Nr. 3) liefert nur eine teilweise Beschreibung des nachgefragten Sachverhalts. In diesem Fall ist metho-

Tabelle 4.5.: Exemplarische Anfragen mit gemessener Konfidenzbewertung, aktivem Profil zum Zeitpunkt der Anfrage und qualitativer Bewertung. Die Bewertung folgt der dreistufigen Skala *korrekt* (Antwort vollständig und faktisch richtig), *teilweise* (richtige Quelle, aber unvollständige Aussage) und *nicht abgedeckt* (Quelle nicht im Korpus oder unzureichendes Retrieval).

#	Anfrage	Sprache	Konfidenz	Profil	Bewertung
1	Wie importiere ich Daten aus Excel oder CSV in NET Engineering?	DE	0,67 (medium)	Balanced	korrekt
2	Wie erfasse ich Arbeitszeiten in NET Build?	DE	0,67 (medium)	Fast	korrekt
3	Was passiert beim Strukturupdate im RLS-Datenmodell?	DE	0,65 (medium)	Fast	teilweise
4	Describe the complete workflow for performing a BW export from NET Design.	EN	0,79 (high)	Fast	korrekt
5	Wie funktioniert der Vitruvi Mobile Quick Start?	DE	0,69 (medium)	Fast	korrekt
6	Welche Benutzereinstellungen gibt es in NETCloud?	DE	0,52 (medium)	Fast	korrekt
7	What are improvements?	EN	0,92 (high)	Fast	korrekt

disch besonders aufschlussreich, dass das System die Unvollständigkeit nicht verschleiert, sondern die Konfidenzbewertung von 0,65 als *medium* an die Anzeige weitergibt – für die Support-Mitarbeiterin ist damit unmittelbar erkennbar, dass der Entwurf einer zweiten Prüfung bedarf. Die hohe Konfidenzbewertung von 0,92 in Anfrage 7 reflektiert dagegen eine exakte definitorische Antwort aus den Vitruvi Customer Terms of Service, deren Wortlaut sich im Antwortentwurf wiederfindet.

Bezug zu den Profilen. Die in Tabelle 4.5 ausgewiesenen Profile dokumentieren, dass die Anfragen überwiegend mit dem schlanken Profil *Fast* bewertet wurden, das eine kürzere Reranking-Liste und kompaktere Antwortfenster verwendet. Eine Wiederholung derselben Anfragen mit *Balanced* oder *Deep* liefert in der Regel mehr abgerufene Quellen, führt aber wie in Abschnitt 4.3.2 ausgeführt nicht zwangsläufig zu einem qualitativ besseren Antwortentwurf; die Profilwahl ist eine Abwägung zwischen Recall und Antwortschärfe, nicht eine Qualitätsabstufung.

4.4.3. Latenz und Reaktivität

Die im produktiven Betrieb auf dem Server-CPU gemessenen Phasen-Latenzen variieren mit Anfragelänge, Modellgröße und Treffermenge. Tabelle 4.6 fasst die typischen Werte für die Server-Konfiguration (Llama 3.1 8B Q5_K_M, 16-Core-CPU) zusammen; eine er-

TKI AI SUPPORT

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

AKTIV

VERLAUF (3)

Leeren

Wie erfasse ich Arbeitszeiten in NET Build?

Was passiert beim Strukturupdate im RLS-Datenmodell?

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

KUNDENANFRAGE

Wie erfasse ich Arbeitszeiten in NET Build?

Drucke **Ctrl** + **Enter** zum Absenden

Suchen & Entwurf erstellen

Antwortentwurf

Bearbeiten

In Zwischenablage kopieren

Konfidenz: 0.87 (high)

87%

Um Arbeitszeiten in NET Build zu erfassen, navigieren Sie zur Seite „Arbeitszeiten“ im Modul „Digitale Bauakte“ und klicken Sie auf „Hinzufügen“ in der oberen rechten Ecke des Fensters, um einen neuen Stundenzettel zu erstellen. Wählen Sie das erforderliche Work Package und die Work Order aus den Dropdown-Listen aus, um die zugehörigen Work Items anzuzeigen. Wählen Sie das erforderliche Work Item oder die erforderlichen Work Items aus der Liste aus und klicken Sie dann auf „>“, um sie zur Auswahl des Berichts hinzuzufügen.

HC-ARBEITSZEITEN EINGEBEN: ERSTELLEN EINES STUNDENZETTEL

#docref-hc-hc-hc-article-20784393993234_chunk_0001

Lösungsschritte

1. Navigieren Sie zur Seite „Arbeitszeiten“ im Modul „Digitale Bauakte“.

2. Klicken Sie auf „Hinzufügen“ in der oberen rechten Ecke des Fensters.

3. Wählen Sie das erforderliche Work Package und die Work Order aus den Dropdown-Listen aus.

4. Wählen Sie das erforderliche Work Item oder die erforderlichen Work Items aus der Liste aus.

5. Klicken Sie auf „>“, um sie zur Auswahl des Berichts hinzuzufügen.

DIAGNOSTICS

Konfidenz-Auflösung / Phasen-Zeiten

Prompt-Vorschau

DE EN Einstellungen

Rerangiert (5) Alle (20)

Help Center

#1

Arbeitszeiten eingeben

97%

HC: NET Build > Arbeitszeiten

#2

Arbeitszeiten eingeben: Erstellen eines Stundenzetels

93%

HC: NET Build > Arbeitszeiten > Erstellen eines Stundenzetels

#3

Termine in NET Build einplanen

92%

HC: NET Build > Allgemein

#4

Arbeitszeiten eingeben: Erstellen eines Stundenzetels

98%

HC: NET Build > Arbeitszeiten > Erstellen eines Stundenzetels

#5

Termine in NET Build einplanen: Flexible 7-Tage-Planung

86%

HC: NET Build > Allgemein > Statusdaten > Flexible 7-Tage-Planung

Abbildung 4.4.1.: Antwort auf „Wie erfasse ich Arbeitszeiten in NET Build?“. Der Entwurf fasst die Schritte zur Erfassung in einem mehrteiligen Verfahren zusammen und verlinkt jeden Schritt mit dem zugehörigen HC-Artikel. Konfidenz 0,67 (medium), Profil Fast.

TKI AI ^{SUPPORT}

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

VERLAUF (3)

Was passiert beim Strukturupdate im RLS-Datenmodell?

Wie erfasse ich Arbeitszeiten in NET Build?

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

KUNDENANFRAGE

Was passiert beim Strukturupdate im RLS-Datenmodell?

Drücke **Ctrl** + **Enter** zum Absenden

Suchen & Entwurf erstellen

Antwortentwurf

medium

Bearbeiten

In Zwischenablage kopieren

Beim Strukturupdate im RLS-Datenmodell werden die datenbankseitigen Projekt Schemata, Strukturen und Daten verändert **[HC: STRUKTURUPDATE]**. Die genauen Änderungen sind nicht explizit im bereitgestellten Dokument beschrieben, aber es wird erwähnt, dass bei der Erweiterung des Datenmodells um Tabellen und Formulare zur Verwaltung von LWL-ONTs neue Funktionen hinzugefügt werden
[HC: RELEASE NOTES NET 15.0 / RELEASE NOTES NET 15.0.0 PDF / ALLGEMEIN].

Konfidenz: 0.65 (medium)

65%

DIAGNOSTICS

Konfidenz-Aufschlüsselung / Phasen-Zeiten

Prompt-Vorschau

KONFIDENZ-AUFSCHÜSSELUNG

rerank

separation

coverage

llm

1.00

0.53

0.25

0.60

Ersteinrichtung: Datenmodell-Erweiterung aktivieren

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren > Datenmodell-Erweiterung aktivieren

85%

1-Klick-Wartungen: Nachträgliches Strukturupdate

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren > Nachträgliches Strukturupdate

94%

GlasfaserPlus RLS DM Erweiterung: Datenmodell und Strukturupdate '9117.7.21058'

Rerank

HC - Release Notes & Downloads > RLS Datenmodell-Erweiterung für GlasfaserPlus > Datenmodell und Strukturupdate '9117.7.21058'

80%

Release Notes NET 15.0 / Release Notes NET 15.0.0.pdf / Allgemein

Rerank

HC - PDF - Release Notes NET 15.0.0.pdf

74%

Strukturupdate

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren

99%

DE

EN

Einstellungen

Rerangiert (5)

Alle (20)

Help Center

#1

HELP CENTER

Strukturupdate

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren

99%

#2

HELP CENTER

1-Klick-Wartungen: Nachträgliches Strukturupdate

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren > Nachträgliches Strukturupdate

94%

#3

HELP CENTER

Ersteinrichtung: Datenmodell-Erweiterung aktivieren

Rerank

HC - Release Notes & Downloads > Dokumentation für Administratoren > Datenmodell-Erweiterung aktivieren

85%

#4

HELP CENTER

GlasfaserPlus RLS DM Erweiterung: Datenmodell und Strukturupdate '9117.7.21058'

Rerank

HC - Release Notes & Downloads > RLS Datenmodell-Erweiterung für GlasfaserPlus > Datenmodell und Strukturupdate '9117.7.21058'

80%

#5

HELP CENTER

Release Notes NET 15.0 / Release Notes NET 15.0.0.pdf / Allgemein

Rerank

HC - PDF - Release Notes NET 15.0.0.pdf

74%

Abbildung 4.4.2.: Antwort auf „Was passiert beim Strukturupdate im RLS-Datenmodell?“. Die Konfidenz (0,65, *medium*) signalisiert die Unvollständigkeit der vorliegenden Beschreibung. Das im unteren Teil eingeblendete Diagnose-Panel zeigt die Aufschlüsselung der vier Konfidenzsignale.

TKI

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

VERLAUF (6)

Leeren

Describe the complete workflow for performing a BW export from NET Design.

What attribute automatically generates sequential IDs?

How is the Ba_bez attribute determined if multiple naming options exist?

Was passiert beim Strukturupdate im RLS-Datenmodell?

Wie erfasse ich Arbeitszellen in NET Build?

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

KUNDENANFRAGE

Describe the complete workflow for performing a BW export from NET Design.

Drücke **Ctrl** + **Enter** zum Absenden

Suchen & Entwurf erstellen

Abgeschlossen in 48.0s — Semantische Suche 3.1s → Reranking 8.2s → Entwurfsgenerierung 36.7s

WEITERFÜHRENDE LINKS

<https://help.tki-chemnitz.de/hc/de/articles/440603752522-Allgemeines-zum-Export-nach-GIS-Nebenbestimmungen-fr%3A%C3%BCher-BMW-Export>

Antwortentwurf

high

Bearbeiten

In Zwischenablage kopieren

To perform a BW export from NET Design, follow these steps:

1. Open the project in the Infrastructure Administrator [Einrichtung BW im NET Design].
2. Activate the "TKI NET Design BMW/BW-Datenmodell" [Einrichtung BW im NET Design].
3. Go to the Ribbon-Menü "NET Design" > Exportieren [Aufruf des BW-Exports im NET Design].
4. Select BMW/BW and choose a directory for the export [Aufruf des BW-Exports im NET Design].
5. The Layer Fläche will be filled from the BMW Markterkundungsgebiet [Layer Fläche].
6. The attributes for the Layer Fläche must be filled by the user, including ID, Dat_Erheb, Dat_Ende, Erg_Erheb, Erg_Koax, Erg_Kupfer, and Erg_Sonstige [Layer Fläche].
7. The export settings can be configured in the Export-Fenster [Optionen im Export-Fenster].
8. The export can be started by clicking on Erstelle [Aufruf des BW-Exports im NET Design].

DIAGNOSTICS

Konfidenz-Aufschlüsselung / Phasen-Zeiten

Prompt-Vorschau

KONFIDENZ-AUFSCHLÜSSELUNG

rerank generation

1.00

0.79

Alle (20)

Rerangiert (5)

Help Center

Alle

#1

HELP CENTER

NET Engineering

Algemeines zum Export nach GIS-Nebenbestimmungen (früher BMW Export) / Anleitung_BMW_BW.pdf / Wichtiger Hinweis

Rerank

74%

HC - PDF - Anleitung_BMW_BW.pdf

#2

HELP CENTER

NET Engineering

Algemeines zum Export nach GIS-Nebenbestimmungen (früher BMW Export) / Anleitung_BMW_BW.pdf / Layer Fläche

Rerank

68%

HC - PDF - Anleitung_BMW_BW.pdf

#3

HELP CENTER

NET Engineering

Algemeines zum Export nach GIS-Nebenbestimmungen (früher BMW Export) / Anleitung_BMW_BW.pdf / Aufruf des BW-Exports im NET Design

Rerank

69%

HC - PDF - Anleitung_BMW_BW.pdf

#4

HELP CENTER

NET Engineering

Algemeines zum Export nach GIS-Nebenbestimmungen (früher BMW Export) / Anleitung_BMW_BW.pdf / Version 1.1

Rerank

61%

HC - PDF - Anleitung_BMW_BW.pdf

#5

HELP CENTER

NET Engineering

Algemeines zum Export nach GIS-Nebenbestimmungen (früher BMW Export) / Anleitung_BMW_BW.pdf / Wichtiger Hinweis

Rerank

54%

HC - PDF - Anleitung_BMW_BW.pdf

Abbildung 4.4.3.: Antwort auf „Describe the complete workflow for performing a BW export from NET Design“. Der Entwurf fasst den kompletten Ablauf in einer nummerierten Schrittfolge zusammen und verknüpft jeden Schritt mit dem zugehörigen HC-Artikel. Konfidenz 0,79 (high), Profil Fast.

TKI A

SUPPORT

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

VERLAUF (7)

Leeren

Wie funktioniert der Vitruvi Mobile Quick Start?

Describe the complete workflow for performing a BW export from NET Design.

What attribute automatically generates sequential IDs?

How is the Ba_bez attribute determined if multiple naming options exist?

Was passiert beim Strukturupdate im RLS-Datenmodell?

Wie erfasse ich Arbeitszeiten in NET Build?

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

KUNDENANFRAGE

Wie funktioniert der Vitruvi Mobile Quick Start?

Drücke **Ctrl** + **Enter** zum Absenden

Suchen & Entwurf erstellen

Abgeschlossen in **45.5s** — Semantische Suche 239ms → Reranking 12.1s → Entwurfsgenerierung 33.1s

WEITERFÜHRENDE LINKS

<https://help.tki-chemnitz.de/hc/de/articles/4415731424914-erste-schritte-mit-der-app>

Antwortentwurf

high

Bearbeiten

In Zwischenablage kopieren

Konfidenz: 0.81 (high)

61%

Der Vitruvi Mobile Quick Start ist ein Leitfaden, der die ersten Schritte mit der App erklärt. [HC: Erste Schritte mit der App / Vitruvi Mobile Quick Start Guide - June 2024.pdf] Um den Quick Start zu nutzen, müssen Sie zunächst die App öffnen und sich anmelden. [HC: Erste Schritte mit der App / Spanish- Vitruvi Mobile Quick Start Guide- July 2024.pdf] Anschließend können Sie die verschiedenen Funktionen der App erkunden, wie zum Beispiel die Arbeitselemente, die Arbeitsaufträge und die Arbeitsmaterialien. [HC: Erste Schritte mit der App / Vitruvi Mobile Quick Start Guide - June 2024.pdf] Der Quick Start enthält auch Beispiele und Anleitungen, um Ihnen bei der Verwendung der App zu helfen. [HC: Erste Schritte mit der App / Vitruvi Mobile Quick Start Guide - June 2024.pdf]

Lösungsschritte

- Öffnen Sie die App und melden Sie sich an.
- Erkunden Sie die verschiedenen Funktionen der App.
- Lesen Sie den Quick Start, um Beispiele und Anleitungen zu erhalten.

DIAGNOSTICS

Konfidenz-Auflösung / Phasen-Zeiten

Prompt-Vorschau

KONFIDENZ-AUFSCHLÜSSELUNG

rerank

1.00

DE

EN

Einstellungen

Rerangiert (5)

Alle (20)

Vitruvi

Help Center

#1

HELP CENTER

NET Build

Erste Schritte mit der App / Vitruvi Mobile Quick Start Guide - June 2024.pdf / CONNECT TO: university Login -TR U-TR vity code 2206 SLHH Due: -& James Kirk Station: - - Add to Report Work Tickets yournamehere@example.com Workspace: university Login Forgot Password

FAISS

61%

HC - PDF - Vitruvi Mobile Quick Start Guide - June 2024.pdf

#2

HELP CENTER

NET Build

Erste Schritte mit der App / Vitruvi Mobile Quick Start Guide - June 2024.pdf / Submitted by James Kirk EWOR20240606-JK Submitted by James Kirk EWOR20231031-LM Submitt EWOR Submitt EWOR20240403-5R Submitted by Steve Rogers EWOR20231025-LM Submitted by Leonard Mc... 2024-06-06 & DETAILS - WORK ITEMS (1) FIB - -Station: - -Pull Fiber Cable 123 Due Date: 2024-06-10 - DETAILS WORK PACKAGE - (required) WORK ORDER * (required) DESCRIPTION NOTES -- WORK ITEMS (1) FIB - -Station: ----Reject Pull Fiber Cable 123 Due Date: 2024-06-10 Pull Fiber Cable 123 Due Date: 2024-06-10 Pull Fiber Cable 123 ACTIVITY CODE FIB DESCRIPTION -- DUE DATE 2024-06-10 STATUS Not completed COMMENTS --\ LABOR & MATERIA Template: Pull Fiber Cabl V LABOR (ft) Act:* WORK ORDER * (required) 05 UG Construction-Construction FSA001-Was... DESCRIPTION NOTES --WORK ITEMS (1) FIB 0 - -Station: - - p Reject Tap here to reject. Approve

FAISS

61%

HC - PDF - Vitruvi Mobile Quick Start Guide - June 2024.pdf

#3

VITRUVI

ToS 1.30 "Mobile Application"

FAISS

59%

ToS §1-30 "Mobile Application" - p.7

Abbildung 4.4.4.: Antwort auf „Wie funktioniert der Vitruvi Mobile Quick Start?“. Der Entwurf greift sowohl auf Vitruvi-Web-Guides als auch auf HC-Artikel zurück. Konfidenz 0,69 (medium), Profil Fast.

TKI A

Support-Assistent

TKI Chemnitz

PRESETS

Fast

20 / 5 — schnelle Iteration

Balanced

50 / 10 — empfohlener Standard

Deep

75 / 15 — maximaler Recall

VERLAUF (8)

Leeren

Welche Benutzereinstellungen gibt es in NETCloud?

Wie funktioniert der Vitruvi Mobile Quick Start?

Describe the complete workflow for performing a BW export from NET Design.

What attribute automatically generates sequential IDs?

How is the Ba_bez attribute determined if multiple naming options exist?

Was passiert beim Strukturupdate im RLS-Datenmodell?

Wie erfasse ich Arbeitszeiten in NET Build?

Wie importiere ich Daten aus Excel oder CSV in NET Engineering?

KUNDENANFRAGE

Welche Benutzereinstellungen gibt es in NETCloud?

Drücke **Ctrl + Enter** zum Absenden

Suchen & Entwurf erstellen

Abgeschlossen in 49.5s — Semantische Suche 5.5s → Reranking 2.6s → Entwurfsgenerierung 41.5s

WETTERFÜHRENDE LINKS

<https://help.tki-chemnitz.de/help/articles/440954926834-Allgemeines>

<https://help.tki-chemnitz.de/help/articles/4409682409342-Nutzer-verwalten>

<https://help.tki-chemnitz.de/help/articles/22423876967570-Berechtigungssystem-ACL>

Antwortentwurf

medium

Bearbeiten

In Zwischenablage kopieren

Konfidenz: 0,52 (medium)

52%

In NETCloud gibt es die Benutzereinstellungen, die im rechten oberen Bereich des Webportals über das Dropdown-Menü erreichbar sind. Hier können Sie Ihr Benutzerkonto öffnen und Ihre Daten ändern oder ergänzen, einschließlich der Aktivierung der Zwei-Faktor-Authentifizierung. **TKI-ALLGEMEINES: BENUTZEREINSTELLUNGEN**. Darüber hinaus gibt es globale Einstellungen, die unter der Obernmenüleiste (globale Einstellungen) erreichbar sind, wie z.B. die Sprache, Admin-Portal und Benutzereinstellungen. **TKI-NET CLOUD > NET CLOUD > FUNKTIONSKATALOG > ZUGRIFF UND LOGIN > HAUPTNAVIGATION & UI-LAYOUT**. Außerdem gibt es globale Einstellungen, die unter dem Menüpunkt "F: Globaleinstellungen" erreichbar sind, wie z.B. Marketplace-Einstellungen, GeoServer-Integrationen und MapTiler-Konfigurationen. **TKI-NET CLOUD > NET CLOUD > FUNKTIONSKATALOG > ADMIN- UND KONFIGURATIONSFUNKTIONEN > F: GLOBALEINSTELLUNGEN**. Die Rollenvergabe ist ein zentrales Element zur Steuerung der Benutzerrechte und der Gewährleistung der Systemstabilität. Es gibt verschiedene Rollen, wie z.B. Registered, NET Cloud Read Only User, NET Cloud Office & Field User und NET Cloud Field Photo App User. **TKI-NUTZER VERWALTEN: ROLLEN VERGEBEN**. Das Berechtigungssystem in NET Cloud basiert auf Gruppen und ermöglicht es, bestimmte Berechtigungen und Einschränkungen innerhalb eines Projekts zu definieren. **TKI-BERECHTIGUNGSSYSTEM-ACL**.

DIAGNOSTICS

Konfidenz-Aufschlüsselung / Phasen-Zeiten

Prompt-Vorschau

DE

EN

Einstellungen

Rerangiert (5)

Alle (20)

TKI Docs

Help Center

Al

NET Cloud

#1

Algemeines: Benutzereinstellungen

99%

HC - NET Cloud > Allgemeines und Kartenansicht > Benutzereinstellungen

#2

TKI Docs

NET Cloud

NET Cloud: Hauptnavigation & UI-Layout

94%

NET Cloud > NET Cloud > Funktionskatalog > Zugriff und Login > Hauptnavigation & UI-Layout

#3

TKI Docs

NET Cloud

NET Cloud: F. Globaleinstellungen

94%

NET Cloud > NET Cloud > Funktionskatalog > Admin- und Konfigurationsfunktionen > F. Globaleinstellungen

#4

HELP CENTER

NET Cloud

Nutzer verwalten: Rollen vergeben

89%

HC - NET Cloud > NET Cloud Administration > Rollen vergeben

#5

HELP CENTER

NET Cloud

Berechtigungssystem "ACL"

85%

HC - NET Cloud > NET Cloud Administration

Abbildung 4.4.5.: Antwort auf „Welche Benutzereinstellungen gibt es in NETCloud?“. Der Entwurf listet die in der TKI-Pr-KB dokumentierten Einstellungsbereiche auf. Konfidenz 0,52 (medium), Profil Fast.

gänzende Latenz-Tabelle für die Apple-M3-Iterationsumgebung mit Metal-Beschleunigung sowie die Mittelwert-vs-p95-Diskussion finden sich in Abschnitt 5.6.

Tabelle 4.6.: Typische Phasen-Latenzen unter Server-Konfiguration

Phase	Latenz	Beschreibung
Semantische Suche	80–200 ms	FAISS-IndexFlatIP über 14 118 Vektoren
Reranking	3–12 s	Cross-Encoder + MMR über Top-50
Entwurfsgenerierung	18–40 s	Llama-Streaming, modellabhängig
Gesamt (wahrnehmbar)	25–50 s	End-zu-End vom Klick zum vollständigen Draft

Aus Nutzersicht ist die für die NFA-2-Konformität entscheidende Größe nicht die Gesamtdauer, sondern die *Time-to-First-Information*: Nach Abschluss der Reranking-Phase sind die Quellen bereits sichtbar und das erste Antwort-Token erscheint binnen weniger Sekunden. Damit liegt der wahrgenommene Antwortbeginn typischerweise zwischen vier und vierzehn Sekunden, was die NFA-2-Vorgabe von 15 Sekunden in der Mehrzahl der Fälle einhält. Eine eingehende Diskussion der gewählten Berichtsgröße (Mittelwert mit Bereich gegenüber p95) sowie die zugehörigen Latenz-Messwerte folgen im Ergebniskapitel (Abschnitt 5.6).

4.4.4. Komponententests und Validierungsinfrastruktur

Frontend: 63 Unit-Tests. Das Frontend verfügt über 63 Unit-Tests, organisiert in zehn Testdateien und ausgeführt mit *Vitest* und der *React Testing Library*. Die Tests gliedern sich in vier Schichten:

API-Schicht (4 Tests). SSE-Parser-Tests in `client.test.ts` prüfen die korrekte Sequenz-Erkennung mehrerer Ereignisse, die Aufteilung eines Ereignisses über TCP-Paketgrenzen, das Stream-Ende ohne abschließenden Zeilenumbruch sowie die Fehlerbehandlung bei nicht-2xx-Antworten.

Hooks (10 Tests). Der Hook `useQueryHistory` wird auf Persistenz im `localStorage`, Duplizierung von Wiederholungen, Begrenzung auf 20 Einträge und defensive Behandlung defekter JSON-Inhalte geprüft.

Utilities (12 Tests). Die Hilfsfunktion `linkifyDocRefs` kapselt die in Abschnitt 4.3.5 beschriebene Ersetzungslogik. Die Tests umfassen das Standard-Vorkommen eines einzelnen Tags, mehrfache Erwähnungen, Longest-Match (Schutz gegen versehentliches Verkürzen längerer Tags durch enthaltene kürzere), Fallback-Verhalten bei vom Modell weggelassenen Klammern und den Shadowing-Schutz bei verschachtelten Tags.

Komponenten (37 Tests). Die sieben Hauptkomponenten (`DraftResponse`, `SourcesPanelContent`, `DiagnosticsPanel`, `PipelineProgress`, `QueryInput`, `ErrorBoundary`, `ResourceLinks`) decken bedingtes Rendering, Benutzerinteraktion (Bearbeiten, Kopieren, Zurücksetzen, Tab-Wechsel, Quellenfilter), die Konfidenz-Anzeige sowie das Streaming-Verhalten ab.

Die Test-Suite wird vor jedem Build automatisch ausgeführt; ein nicht bestandener Test bricht den Build und damit auch die Deployment-Pipeline ab. End-zu-End-Tests (Playwright, Cypress) sind nicht implementiert; ihr Mehrwert wäre für die spezifische Streaming-Charakteristik der Pipeline begrenzt, da das Backend zur Laufzeit reproduzierbar gemockt werden müsste, um deterministische Browser-Tests zu erlauben.

Backend: Validierungsinfrastruktur. Für die Python-Backend-Komponenten wurde keine pytest-basierte Unit-Test-Suite im klassischen Sinn aufgebaut. Die Korrektheits- und Regressionsprüfung verteilt sich stattdessen auf vier ineinandergreifende Ebenen, deren Ergebnisse jeweils in den fachlich zuständigen Kapiteln ausgewertet werden:

1. **Qualitätsfilter-Benchmarks** (`zendesk/stage1_filter/eval/`). 13 Skripte führen den in Abschnitt 5.2.5 und Folgeabschnitten dokumentierten Vergleich der Klassifikatorkonfigurationen durch – die Modellvarianten (Llama 3.1 8B in Q4/Q5, Qwen 2.5 32B in den drei Iterationen v1/v2/v3 sowie der Full-Corpus-Lauf), die vier regelbasierten Baselines B1–B4 (`benchmark_baselines.py`) und die Out-of-Sample-Auswertung auf 155 Tickets (`compute_oos_metrics.py`). Die Wilson-Konfidenzintervalle und McNemar-Tests werden zentral durch `compute_statistics.py` berechnet (vgl. Abschnitt 5.2.13).
2. **Quellen-spezifische Smoke- und Demo-Skripte.** Jede der drei Retrieval-Quellen mit eigenständiger Indexierungs-Pipeline besitzt einen reproduzierbaren Sanity-Check für die End-zu-End-Erschließbarkeit aus dem gebauten Index:
 - `zendesk/stage3_rag/src/units/smoke_test.py` sendet sieben kanonische Anfragen (deutsch und englisch, gemischte Themenbereiche) gegen den gebauten Units-Index und gibt die Top- K -Treffer samt Quality-Score-Spalte aus.
 - `zendesk_help_center/notebooks/99_demo_rag.ipynb` prüft kategorienbezogen, ob deutschsprachige Support-Anfragen aus den NET-Produktmodulen (NET Build, NET Engineering, NET Design) sowie die Release-Notes-Kategorie in den jeweils erwarteten Top-3-Treffern landen.
 - `vitruvi/notebooks/99_demo_rag.ipynb` ist der in Abschnitt 3.3.6.5 dokumentierte Demo-Test: drei repräsentative Anfragen über die drei Vitruvi-Quellentypen

(PDF, Web-Guide, OpenAPI) mit Llama-Antwortgenerierung und Prüfung der quelltyp-spezifischen Zitations-Logik.

3. **Merge-Stage-Integritätsprüfungen** (`react_app/notebooks/01_merge_faiss.ipynb`). Vor der Fusion der vier Quellen zum gemeinsamen Index prüft das Notebook über `assert`-Anweisungen drei strukturelle Invarianten der eingehenden Vektorblöcke: einheitliche Embedding-Dimension (1.024), einheitlicher Datentyp (`float32`) und ℓ_2 -Normalisierung. Anschließend folgt eine *Sanity-Validierung mit acht Testanfragen* (zwei pro Quellgruppe plus eine indirekte Übereinstimmungs-Probe), die die korrekte Routing-Charakteristik des fusionierten Index belegt. Die produzierten Artefakte (`combined.parquet`, `faiss.index`, `merge_metadata.json`) tragen den Git-Commit-Hash, unter dem sie erzeugt wurden.
4. **End-zu-End-RAGAS-Evaluation (Kapitel 5)**. Die übergeordnete Korrektheitsprüfung des RAG-Systems erfolgt schließlich über das RAGAS-Framework: die Metriken Faithfulness, Answer Relevance sowie Context Recall und Context Precision bewerten das Verhalten des gesamten Antwortpfads (Retrieval, Reranking, LLM-Generierung) gegen einen dokumentierten Testdatensatz.

Diese Aufteilung ist sachlich angemessen: Die Backend-Pipeline besteht zu großen Teilen aus Modell-Inferenz (BGE-M3, BGE-Reranker-v2-M3, Llama 3.1 8B), deren Verhalten nicht durch isolierte Funktionstests, sondern nur retrieval- und antwortempirisch sinnvoll bewertbar ist. Klassische Unit-Tests würden hier entweder schwergewichtige Modell-Artefakte laden oder so umfänglich mocken, dass der Erkenntnisgewinn gegenüber den oben genannten Validierungsebenen marginal bliebe.

4.5. Pro-Pipeline-Hilfsfunktionen

Wiederkehrende Utility-Funktionen (Logger, Datei-I/O, kanonische Produktliste) sind nicht in einem zentralen `shared/-`Modul, sondern jeweils innerhalb der einzelnen Pipeline-Module organisiert (`react_app/backend/src/utils/`, `vitruvi/src/utils/`, `zendesk_help_center/src/utils/`, `zendesk/stage3_rag/src/utils/`). Diese Aufteilung war Folge der iterativen Entwicklung der Pipelines; eine Konsolidierung in ein gemeinsames `Top-Level-shared/-`Modul ist eine sinnvolle Erweiterungsrichtung.

Kanonische Produktliste. Die zehn TKI-Module (NET Design, NET Engineering, NET Build, NET Cloud, NET Scan, NET Operations, NET Inhouse, NET REST, NET Field Survey, Dynamics Sales) bilden die Grundlage der Produktnormalisierung in allen Datenextraktionspipelines. Die Liste ist über die Pipelines hinweg konsistent gepflegt; die Normalisierung erfolgt jeweils per Substring-Matching mit Fallback-Logik.

Snapshot-Utility für Stage-2-Iterationen. Die Iterationsphase der Zendesk-Stage-2-Extraktion verwendet eine Snapshot-Utility (`zendesk/stage2_extraction/src/snapshot.py`), die pro Lauf Git-Hash, Dirty-State, Lauf-Zeitstempel und reproduzierbare Metriken in einem versionierten `runs/<YYYY-MM-DD>_<run_name>/`-Verzeichnis ablegt. Diese Utility ist Stage-2-spezifisch und nicht auf andere Pipelines verallgemeinert.

Kapitel 5.

Ergebnisse und Evaluation

Dieses Kapitel beschreibt die Evaluation des entwickelten Systems. Zunächst werden der Testdatensatz und die Evaluationsmethodik vorgestellt. Anschließend werden die Ergebnisse der semantischen Suche, der Antwortgenerierung und des Modellvergleichs präsentiert und diskutiert.

5.1. Testdatensatz und Methodik

Für die Evaluation wurde ein Testdatensatz aus realen Zendesk-Support-Anfragen erstellt (`zendesk_test_set/`). Der Datensatz enthält repräsentative Anfragen zu verschiedenen TKI-Produkten und deckt unterschiedliche Schwierigkeitsgrade ab – von einfachen Konfigurationsfragen bis hin zu komplexen technischen Problemen, die Wissen aus mehreren Quellen erfordern.

Die Evaluation erfolgt auf mehreren Ebenen:

1. **Retrieval-Qualität:** Bewertung der semantischen Suche und des Reranking anhand der Relevanz der abgerufenen Dokumente.
2. **Generierungsqualität:** Bewertung der LLM-generierten Antworten anhand des RAGAS-Frameworks.
3. **Systemperformance:** Messung der Latenzzeiten der einzelnen Pipeline-Stufen.
4. **Modellvergleich:** Vergleich verschiedener LLM-Konfigurationen hinsichtlich Antwortqualität und Geschwindigkeit.

5.1.1. RAGAS-Metriken

Das RAGAS-Framework [16] (Retrieval-Augmented Generation Assessment) definiert folgende Metriken:

- **Answer Relevance:** Misst, ob die generierte Antwort die Frage tatsächlich beantwortet. Ein hoher Wert bedeutet, dass die Antwort inhaltlich auf die Anfrage eingeht.

- **Faithfulness:** Misst, ob die generierte Antwort nur Informationen enthält, die in den bereitgestellten Dokumenten vorhanden sind. Ein hoher Wert zeigt geringe Halluzination an.
- **Context Recall:** Misst, ob die abgerufenen Dokumente alle Informationen enthalten, die für eine vollständige Antwort benötigt werden.
- **Context Precision:** Misst, ob die abgerufenen Dokumente relevant sind und kein „Rauschen“ enthalten.

5.2. Qualitätsfilterung der Zendesk-Tickets

Dieser Abschnitt evaluiert die in Abschnitt 3.3.1.3 konzipierte und in Abschnitt 4.2.2.4 implementierte Qualitätsfilterung der Zendesk-Tickets. Die Auswertung umfasst die beiden Annotationsdatensätze Golden 45 und Set 200, eine Iterationsanalyse über die Entwicklungsphase, einen Modellvergleich zwischen Llama 3.1 8B und Qwen 2.5 32B sowie eine Fehleranalyse mit Bucket-Verteilung.

5.2.1. Datensätze und Test-Set-Hierarchie

Die Evaluation erfolgt auf drei aufeinander aufbauenden Datensätzen, die eine schrittweise Skalierung von der Prompt- und Override-Entwicklung bis zur produktiven Anwendung auf dem Gesamtbestand ermöglichen:

- **Golden 45** – 45 hand-annotierte Tickets, ausbalanciert (23 USEFUL / 22 USELESS). Wird für die schnelle Iteration der Prompts und Confidence-Overrides verwendet, da ein vollständiger Lauf auf Server-Hardware in unter einer Stunde abgeschlossen ist.
- **Set 200** – 200 hand-annotierte Tickets mit produktionsnaher Klassenverteilung (62 USEFUL / 138 USELESS, 31 % useful). *Set 200 ist eine echte Obermenge von Golden 45* ($45 \subset 200$): die 45 Entwicklungstickets sind im Validierungsdatensatz enthalten und werden um 155 zusätzliche, stratifiziert nach Produkt und Status gezogene Tickets erweitert. Die Überlappung wird bewusst akzeptiert, da das Hauptziel von Set 200 die Generalisierung auf bisher nicht berücksichtigte Beispiele ist; eine Bias-Korrektur erfolgt durch die zusätzlichen 155 Tickets.
- **Full Corpus** – 16 987 Tickets ohne menschliche Ground-Truth. Wird ausschließlich zur Beurteilung der praktischen Anwendbarkeit eingesetzt: Verteilung der Vorhersagen, Latenz, Robustheit der Pipeline.

Die manuelle Annotation der Golden 45 und Set 200 erfolgte über eine eigens entwickelte Streamlit-Anwendung (`annotate_tickets.py`), die jeden Ticket nach PII-Maskierung als chronologische Konversation anzeigt und eine binäre Eingabe (USEFUL/NOT_USEFUL) mit optionalen Notizen entgegennimmt. Tabelle 5.1 fasst die Eckdaten zusammen.

Tabelle 5.1.: Datensätze für die Evaluation der Qualitätsfilterung

Datensatz	Tickets	USEFUL	USELESS	Verwendung
Golden 45	45	23 (51 %)	22 (49 %)	Iteration, Prompt- und Override-Kalibrierung
Set 200	200	62 (31 %)	138 (69 %)	Validierung; enthält die Golden 45 als Teilmenge
Full Corpus	16 987	–	–	Anwendung auf den Gesamtbestand (keine Ground Truth)

Tabelle 5.2.: Hardware-Konfigurationen während der Experimente

Umgebung	Beschreibung
<i>Lokal</i>	Apple MacBook (M3, Metal-beschleunigt). Verwendet für die initiale Forschungsphase und Prompt-Engineering mit Llama 3.1 8B Q5_K_M.
<i>Server</i>	WEBSRV38, Ubuntu 24.04, 16-Core CPU, 31 GB RAM, kein GPU. Hostet das Produktivsystem; aufgrund seiner größeren RAM-Ausstattung Voraussetzung für den Betrieb von Qwen 2.5 32B (ca. 20 GB) und für die Verarbeitung des Gesamtbestands.

Die Verteilung im Set 200 (31 % USEFUL) entspricht dem Anteil verwertbarer Tickets im Gesamtbestand und unterscheidet sich bewusst vom annähernd ausgewogenen Golden 45-Datensatz. Beide annotierten Sätze sind zu 100 % menschlich verifiziert.

Statistische Vorbemerkung zur Datensatzgröße. Mit $n = 45$ Tickets entspricht auf Golden 45 ein einzelner Klassifikationsfehler bereits 2,22 Prozentpunkten Accuracy. Differenzen zwischen Konfigurationen, die diese Größenordnung unterschreiten, sind statistisch nicht von zufälligen Einzelentscheidungen unterscheidbar; jede Aussage zur Modellqualität, die ausschließlich auf Golden 45 beruht, ist daher mit einem entsprechenden Unsicherheitsbereich behaftet. Auf Set 200 entspricht ein einzelner Fehler $1/200 = 0,5$ Prozentpunkten und liefert damit eine etwa viermal feinere Auflösung. Diese Arbeit nutzt Golden 45 deshalb primär als *Iterations- und Kalibrierungsdatsatz* und Set 200 als den *ausagekräftigeren Validierungsdatsatz*; eine vollständige Robustheitsanalyse mit Wilson-Konfidenzintervallen und McNemar-Tests für die zentralen Konfigurationsvergleiche erfolgt in Abschnitt 5.2.13.

Out-of-Sample-Anteil von Set 200. Da Golden 45 als echte Teilmenge in Set 200 enthalten ist und Prompts sowie Confidence-Overrides anhand von Golden 45 kalibriert wurden, ist die Auswertung auf den vollen 200 Tickets potenziell zugunsten der dort enthaltenen 45 Kalibrierungstickets verzerrt. Eine statistisch saubere Out-of-Sample-Validierung umfasst daher nur die verbleibenden $200 - 45 = 155$ Tickets, die ausschließlich in Set 200 enthalten sind. Auf diesem Out-of-Sample-Anteil ist die Klassenverteilung mit 25,2 % USEFUL (39 von 155) etwas stärker zur Klasse USELESS verschoben als auf dem vollen Set 200 (31 % USEFUL); die 45 entfernten Tickets enthalten überproportional viele USEFUL-Tickets (23 von 45, 51 %). Eine separate Auswertung der 155 Out-of-Sample-Tickets erfolgt in Abschnitt 5.2.13 und quantifiziert den Einfluss dieser Überlappung auf die berichteten Set-200-Resultate.

5.2.2. Vergleich mit regelbasierten Baselines

Eine isolierte Auswertung der LLM-basierten Pipeline beantwortet nicht die für die Aufwandsrechtfertigung zentrale Frage, wie viel der LLM-Ansatz gegenüber einfacheren, regelbasierten Klassifikatoren tatsächlich beiträgt. Aus diesem Grund werden in diesem Abschnitt vier regelbasierte Baselines auf denselben Datensätzen Golden 45 und Set 200 evaluiert und den im Verlauf der Arbeit eingesetzten LLM-Konfigurationen gegenübergestellt. Die Implementierung der Baselines liegt in `benchmark_baselines.py` und `benchmark_prefilter.py`; alle Metriken wurden mit derselben Konfusionsmatrix-Logik wie in den LLM-Benchmarks berechnet (vgl. `benchmark_qwen32_v3_golden_only.py`, Z. 184–198).

Vier Baselines werden untersucht. **B1 (Majority)** klassifiziert jeden Ticket als USELESS und entspricht damit dem informationsfreien Mehrheits-Prädiktor; auf Set 200 mit produktiver Klassenverteilung markiert sie den niedrigsten denkbaren Schwellenwert. **B2 (Status-only)** klassifiziert jeden Ticket im resolved-Endzustand der Zendesk-Suite (`status ∈ {closed, solved}`) als USEFUL. **B3 (Status + ≥ 3 öffentliche Kommentare)** ist genau die vom Gutachter im Feedback explizit benannte Heuristik¹: Tickets gelten als USEFUL, wenn sie im resolved-Endzustand vorliegen und mindestens drei öffentliche Kommentare aufweisen. **B4 (5-stufiger Pre-Filter)** verwendet eine im Verlauf der Arbeit entwickelte regelbasierte Pipeline mit sechs Hard-Filter-Regeln, 25+ Features, fünf gewichteten Score-Dimensionen (resolution, technical, conversation, substance, reusability), fünf Override-Regeln und perzentilbasierten Tier-Schwellen (A/B/C/D); die binäre Klassifikation erfolgt über die Abbildung „A oder B $\rightarrow$ USEFUL, sonst USELESS“. Die Pre-Filter-Pipeline wurde nicht-iterativ entwickelt und ist in `notebooks/ticket_quality_prefilter.ipynb` dokumentiert; sie stellt damit die rechenstärkste rein regelbasierte Vergleichsbasis dar, die im Rahmen dieser Arbeit überhaupt verfügbar ist.

¹Gutachterfeedback Prof. Espig, 13. Mai 2026, Abschnitt B.3.

Den Baselines gegenübergestellt werden drei produktive LLM-Konfigurationen aus dem Iterationsverlauf (Abschnitt 5.2.3): **L1 Qwen 2.5 32B v1** (Server, Llama-tunierte Pipeline ohne Override-Verfeinerung), **L2 Qwen 2.5 32B v3** (Server, real-kalibrierte Overrides) und **L3 Llama 3.1 8B Q5_K_M** (Server, finale produktive Konfiguration). Tabelle 5.3 fasst die Ergebnisse zusammen.

Tabelle 5.3.: Vergleich regelbasierter Baselines (B1–B4) mit drei LLM-Konfigurationen (L1–L3) auf den Datensätzen Golden 45 und Set 200. Alle Werte in Prozent; die Spalte *F1* bezieht sich auf die Klasse USEFUL.

Klassifikator	Golden 45				Set 200			
	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
B1 Majority (alle USELESS)	48,9	0,0	0,0	0,0	69,0	0,0	0,0	0,0
B2 Status-only	53,3	52,3	100,0	68,7	33,0	31,4	98,4	47,7
B3 Status + ≥ 3 öff. Kommentare	62,2	58,3	91,3	71,2	61,0	43,9	93,5	59,8
B4 5-stufiger Pre-Filter (rule-based)	51,1	52,9	39,1	45,0	68,5	49,2	48,4	48,8
L1 Qwen 2.5 32B v1 (Server)	77,8	100,0	56,5	72,2	84,5	87,8	58,1	69,9
L2 Qwen 2.5 32B v3 (Server)	86,7	94,7	78,3	85,7	85,0	82,0	66,1	73,2
L3 Llama 3.1 8B Q5 (Server, produktiv)	91,1	95,2	87,0	90,9	87,0	81,0	75,8	78,3

Vier Beobachtungen sind methodisch relevant.

Erstens, die Majority-Baseline (B1) verdeutlicht den Klassenverteilungs-Effekt. Auf Set 200 mit 31 % USEFUL erreicht der reine Mehrheits-Prädiktor bereits 69,0 % Accuracy, allerdings bei F1 von null. Diese Beobachtung verbietet eine Bewertung der Klassifikatoren ausschließlich anhand der Accuracy, insbesondere auf dem produktionsnahen Set 200; F1 ist hier die aussagekräftigere Vergleichsgröße.

Zweitens zeigen die metadatenbasierten Baselines B2/B3 die Grenze einfacher Heuristiken. B2 mit ausschließlicher Status-Prüfung erreicht zwar nahezu vollständigen Recall (98–100 %), kollabiert aber auf Set 200 zu einer Accuracy von 33,0 % und einer Precision von 31,4 %, weil die überwältigende Mehrheit der Tickets im resolved-Endzustand archiviert wird. Die vom Gutachter vorgeschlagene Erweiterung B3 (zusätzliches Kommentarzahl-Kriterium) verbessert die Precision auf Set 200 deutlich (43,9 % statt 31,4 %) bei vergleichbarem Recall (93,5 %) und stellt damit die beste *tatsächlich-trivial* aufgebaute Baseline dar: F1 71,2 % auf Golden 45 und 59,8 % auf Set 200. Diese Werte definieren den Schwellenwert, den ein LLM-basierter Ansatz inhaltlich überschreiten muss, um seine zusätzlichen Inferenzkosten rechtfertigen.

Drittens demonstriert B4 die obere Grenze des rein regelbasierten Vorgehens. Die 5-stufige Pre-Filter-Pipeline mit 25+ Features, fünf gewichteten Score-Dimensionen und mehreren Override-Regeln erreicht keine Verbesserung gegenüber der trivialen B3-Heuristik: 51,1 % Accuracy auf Golden 45 (F1 45,0 %) und 68,5 % auf Set 200 (F1 48,8 %).

Der Recall bleibt mit 39,1 % bzw. 48,4 % deutlich hinter B3 zurück. Diese Beobachtung ist nicht-trivial: die Pipeline wurde mit dem Ziel entworfen, die obere Genauigkeitsgrenze regelbasierter Klassifikation auszuloten. Ihre Unterlegenheit gegenüber B3 zeigt, dass die binäre USEFUL/USELESS-Entscheidung keine reine Feature-Engineering-Aufgabe ist: was „wiederverwendbares Wissen“ ausmacht, lässt sich nicht zuverlässig über zähl- und regex-basierte Heuristiken bestimmen, sondern erfordert eine semantische Beurteilung des Konversationsverlaufs.

Viertens quantifiziert der Abstand zur produktiven LLM-Konfiguration den eigentlichen Mehrwert des LLM-Ansatzes. Gegenüber der besten regelbasierten Baseline (B3) hebt L3 (Llama 8B Q5 Server) die Accuracy auf Golden 45 von 62,2 % auf 91,1 % (+28,9 Prozentpunkte), den F1-Wert von 71,2 % auf 90,9 % (+19,7 Prozentpunkte) und den Recall von 91,3 % auf 87,0 % bei gleichzeitiger Steigerung der Precision von 58,3 % auf 95,2 %. Auf Set 200 verbessert sich die Accuracy von 61,0 % auf 87,0 % (+26,0 Prozentpunkte). Bereits die erste produktive LLM-Konfiguration L1 (Qwen 32B v1, 77,8 % G45) überschreitet B3 spürbar und erreicht 100 % Precision auf Golden 45; jede weitere LLM-Iteration vergrößert diesen Vorsprung. Damit ist der LLM-bedingte Genauigkeitsgewinn nicht inkrementell, sondern qualitativ: er verschiebt die Aufgabe von der recall-orientierten Vorauswahl in den Bereich produktiver Klassifikationsgüte.

Die Kostenseite dieser Verbesserung wird in Abschnitt 5.2.6 dokumentiert: die LLM-Konfigurationen benötigen je nach Quantisierung ca. 12–65 Sekunden Inferenzzeit pro Ticket bei 6–22 GB RAM-Bedarf, während die regelbasierten Baselines unter einer Millisekunde pro Ticket auskommen. Die Entscheidung zwischen B3 und L3 ist damit primär eine Frage des akzeptierten Qualitätsniveaus: ein +26-Prozentpunkte-Genauigkeitsgewinn rechtfertigt die zusätzliche Hardware- und Latenzkosten für einen Anwendungsfall, in dem ungeeignete Tickets dauerhaft den Retrieval-Index verschlechtern würden.

5.2.3. Iterationsverlauf

Die Entwicklung der Qualitätsfilterung verlief über drei Phasen: eine frühe Prompt-Findungsphase mit einer noch nicht finalisierten Ground-Truth-Annotation (April 2026), eine konzentrierte Iterationsphase auf Llama 3.1 8B Anfang Mai 2026 sowie eine anschließende Server-Phase mit Qwen 2.5 32B und der finalen Rückkehr zu Llama 3.1 8B Q5_K_M. Tabelle 5.4 dokumentiert sämtliche reproduzierbaren Iterationen chronologisch. Die Spalte *Acc. G45 (rep.)* gibt die im Notebook gespeicherte Accuracy zum Zeitpunkt des Commits wieder; die Spalte *final GT* zeigt die gegen die am 2026-05-05 finalisierte Ground-Truth-Annotation neu berechnete Accuracy.²

²Zwischen Commit 5d81d59 (Mo, 2026-05-04 21:52) und 44750c8 (Di, 2026-05-05 11:17) wurde das Label des Tickets #2135 (*PDF report changes*) von USEFUL zu USELESS korrigiert. Dies erklärt die geringen Abweichungen zwischen der zum Commit-Zeitpunkt gemessenen und der gegen die finale Annotation

Tabelle 5.4.: Iterationsverlauf der Qualitätsfilterung (alle reproduzierbaren Commits)

Datum	Commit	Modell	Quant.	Modus / Konfiguration	rep.	final GT
<i>Phase 1 – Frühe Prompt-Findung (vor finaler Annotation)</i>						
2026-04-21	37cd5d9	Llama 3.1 8B	Q4_K_M	Two-Pass-Prompt (nur Set 200)	77,5%*	–
2026-04-22	78c3f2e	Llama 3.1 8B	Q4_K_M	Two-Pass, frühe Heuristiken	35,6%	–
2026-04-22	bb8c513	Llama 3.1 8B	Q4_K_M	Two-Pass + Hard-Rules	42,2%	–
2026-04-22	a73f781	Llama 3.1 8B	Q4_K_M	Two-Pass, gelockerte Regeln	77,8%	–
2026-04-23	24ffb23	Llama 3.1 8B	Q4_K_M	Two-Pass, Zwischenstand	66,7%	–
2026-04-23	ea2299d	Llama 3.1 8B	Q4_K_M	Two-Pass, verfeinerte Regeln	73,3%	–
2026-04-24	b85f80a	Llama 3.1 8B	Q4_K_M	Two-Pass + 2A/2B-Rechecks	82,2%	–
<i>Phase 2 – Llama-Iteration auf finaler Annotation (lokal, Apple M3 Metal)</i>						
2026-05-04	a8317a8	Llama 3.1 8B	Q4_K_M	v8-Prompt, Borderline-Pass	71,1%	–
2026-05-04	2898268	Qwen 2.5 7B	Q4_K_M	v8-Prompt, Pilot-Test	62,2%	62,2%
2026-05-04	5d81d59	Llama 3.1 8B	Q4_K_M	Neue pipeline.py, Single-Shot $T = 0,0$	75,6%	77,8%
2026-05-05	dda7a86	Llama 3.1 8B	Q4_K_M	Pipeline-Refactoring, Single-Shot	77,8%	80,0%
2026-05-05	4a0fef1	Llama 3.1 8B	Q4_K_M	Override-Erweiterung A/B/C	80,0% [†]	82,2%
2026-05-05	44750c8	Llama 3.1 8B	Q4_K_M	8-Few-Shot Single-Pass	75,6% [†]	75,6%
2026-05-05	7d45b0d	Llama 3.1 8B	Q4_K_M	Self-Consistency $N = 3$, $T = 0,3$	84,4%	84,4%
2026-05-05	623917d	Llama 3.1 8B	Q5_K_M	Single-Shot $T = 0,0$, Quant.-Upgrade	86,7%	86,7%
2026-05-05	4560fa1	Llama 3.1 8B	Q5_K_M	Single-Shot $T = 0,0$ – lokaler Spitzenwert	88,9%	88,9%
<i>Phase 3 – Server-Phase (WEBSRV38, 16-Core CPU, 31 GB RAM, kein GPU)</i>						
2026-05-07	e75bfb3	Qwen 2.5 32B	Q4_K_M	v1: Llama-tuned Pipeline (Baseline)	77,8%	77,8%
2026-05-07	(v2)	Qwen 2.5 32B	Q4_K_M	v2: synth.-kalibrierte Overrides E/F	77,8%	77,8%
2026-05-07	923bd7a	Qwen 2.5 32B	Q4_K_M	v3: real-kalibrierte Overrides + Prompt-Nudges	86,7%	86,7%
2026-05-09	(Server)	Llama 3.1 8B	Q5_K_M	Single-Shot $T = 0,0$, CPU llama.cpp – produktiv	91,1%	91,1%

* Lauf nur auf Set 200, kein Golden-45-Ergebnis verfügbar. † Notebook-Output zum Commit-Zeitpunkt; die Commit-Nachrichten nannten 82 % bzw. 84,4 %, da die Pipeline nach den Code-Änderungen nicht erneut ausgeführt wurde.

Die ersten Wochen entsprachen einer Prompt-Findungsphase mit einer noch nicht finalisierten Ground-Truth-Annotation. Die effektive Iterationsphase begann mit Commit 5d81d59 (Mo, 2026-05-04 21:52), nach Einführung einer modularen Pipeline mit deterministischem Pre-Filter, Klassifikator und nachgelagerten Confidence-Overrides. Über sieben aufeinanderfolgende Iterationen wurde die Accuracy von 75,6 % auf 88,9 % gesteigert. Zwei Hebel waren wesentlich: zum einen die Einführung eines Self-Consistency-Verfahrens mit $N = 3$ und niedriger Temperatur (Commit 7d45b0d, +6,7 Prozentpunkte gegenüber dem vorhergehenden Single-Pass), zum anderen der Wechsel der Modellquantisierung von Q4_K_M zu Q5_K_M (Commit 623917d, der trotz Verzichts auf Self-Consistency eine Genauigkeit von 86,7 % erzielte; eine isolierte Auswertung dieses Quantisierungseffekts gibt Abschnitt 5.2.4 wieder).

Anschließend erfolgte der Übergang auf Qwen 2.5 32B Q4_K_M mit Inferenz auf einem dedizierten Server (WEBSRV38, 16-Core-CPU, 31 GB RAM, kein GPU). Eine produktive Konfiguration war erforderlich, da das quantisierte Split-GGUF allein ca. 20 GB Speicher belegt und der Produktiv-Service `support-backend.service` während der Benchmark-Läufe gestoppt werden musste. Drei Iterationen wurden durchgeführt (v1/v2/v3, siehe Abschnitt 5.2.5 und 5.2.8). Die anschließende Anwendung auf den Gesamtbestand mit Qwen 32B wurde wegen einer Hochrechnung von ca. 9 Tagen Laufzeit nach 1 253 von 16 987 Tickets abgebrochen; stattdessen wurde – wie in Abschnitt 5.2.10 begründet – zur lokalen Konfiguration Llama 3.1 8B Q5_K_M zurückgekehrt, die auf Server-Hardware ein Resultat von 91,1 % auf Golden 45 erreicht hat.

Reproduzierbarkeit der Iterationshistorie

Sämtliche Werte in der Spalte *Acc. G45 (final GT)* wurden gegen die finale Annotationsdatei `golden_45_ground_truth.json` (Stand 2026-05-05) neu berechnet. Für die Llama-Commits der Phase 2 standen die ursprünglich gespeicherten JSONL-Vorhersagedateien nicht in jedem Fall zur Verfügung; die Vorhersagen wurden daher aus der jeweils im Jupyter-Notebook `filter_tickets.ipynb` gespeicherten *Mistakes-Tabelle* (Ausgabe der Auswertungszelle) rekonstruiert. Für jedes Commit c enthält das Notebook eine Tabelle mit allen falsch klassifizierten Tickets aus Golden 45; alle übrigen Tickets sind per Definition korrekt klassifiziert, sodass aus diesen beiden Mengen die vollständige Vorhersage rekonstruierbar ist. Diese Vorhersagen wurden anschließend gegen die finale Ground-Truth-Annotation evaluiert. Die Server-Iterationen (Qwen 32B v1/v2/v3, Llama 8B Q5

neu berechneten Accuracy für Iterationen vor diesem Zeitpunkt.

Server) verfügen über vollständige JSONL-Vorhersagedateien (SERVER_qwen32_*.jsonl, golden_45_llama8b_q5_results.jsonl), die für die Berechnung direkt verwendet wurden. Die vollständige Per-Ticket-Vergleichsmatrix für zwölf rekonstruierbare Konfigurationen ist in Anhang A.4 dokumentiert.

5.2.4. Quantisierungseffekt Q4_K_M $\rightarrow$ Q5_K_M

Innerhalb der lokalen Iterationsphase (Phase 2, Tabelle 5.4) wurde ein direkter Vergleich zwischen den Quantisierungsstufen Q4_K_M und Q5_K_M von Meta Llama 3.1 8B-Instruct möglich, da die unmittelbar aufeinanderfolgenden Commits 7d45b0d, 623917d und 4560fa1 identische Pipeline-Logik bei verschiedener Quantisierung verwendeten. Tabelle 5.5 dokumentiert die isolierte Wirkung dieses Quantisierungsupdates.

Tabelle 5.5.: Isolierter Quantisierungseffekt Q4_K_M $\rightarrow$ Q5_K_M auf Golden 45

Commit	Quant.	Modus / Konfiguration	Acc.	Prec.	Rec.	F1
7d45b0d	Q4_K_M	SC $N = 3$, $T = 0,3$, Mehrheits-Voting	84,4 %	90,0 %	78,3 %	83,7 %
623917d	Q5_K_M	Single-Shot $T = 0,0$, ohne SC	86,7 %	90,5 %	82,6 %	86,4 %
4560fa1	Q5_K_M	Single-Shot $T = 0,0$, Override-Verfeinerung	88,9 %	90,9 %	87,0 %	88,9 %

Der Übergang von Q4_K_M (Mehrheits-Voting über drei Inferenzen) zu Q5_K_M (Single-Shot) ergibt eine Genauigkeitssteigerung um +2,3 Prozentpunkte (84,4 % $\rightarrow$ 86,7 %), obwohl gleichzeitig auf das aufwändige Self-Consistency-Verfahren verzichtet wurde. Eine geringfügige nachgelagerte Override-Verfeinerung im Commit 4560fa1 ergänzte weitere +2,2 Prozentpunkte und führte zum lokalen Spitzenwert von 88,9 % (40 von 45). Es ist zu beachten, dass +2,3 Prozentpunkte auf Golden 45 exakt einem einzelnen Ticket entsprechen; eine McNemar-basierte Robustheitsprüfung dieser Differenz erfolgt in Abschnitt 5.2.13 und ergibt $p = 1,00$, sodass die Genauigkeitsdifferenz zwischen Q4 und Q5 *nicht* als statistisch signifikant interpretiert werden darf. Die Wahl zugunsten von Q5 wird daher nicht durch eine signifikante Genauigkeitsverbesserung, sondern durch die im Folgenden diskutierten Latenz- und Wartbarkeitsvorteile begründet. Drei Beobachtungen sind methodisch relevant:

- **Quantisierungsverlust ist nicht vernachlässigbar.** Q4_K_M reduziert die Modellgewichte gegenüber Q5_K_M um ca. 17 % Speicherbedarf (4,9 GB vs. 5,7 GB), bezahlt diesen Speichervorteil jedoch in der konkreten Klassifikationsaufgabe mit einem messbaren Genauigkeitsverlust. Für die hier untersuchte binäre Klassifikation reichte Q5_K_M aus, um den Self-Consistency-Mehrwert vollständig zu kompensieren.

- **Self-Consistency wird durch Quantisierungsupgrade obsolet.** In der Q4-Konfiguration brachte SC mit $N = 3$ einen Recall-Gewinn, der bei Q5 nicht mehr erforderlich war: das stabilere Modell traf in Single-Shot dieselben oder bessere Entscheidungen.
- **Latenzgewinn beachtlich.** Single-Shot Q5 auf Apple M3 liefert ca. 11,7 s pro Ticket, SC $N = 3$ Q4 benötigt ca. $3 \times$ dieser Zeit (~ 35 s/Ticket). Auf dem Server-CPU-Setup (~ 20 s pro Ticket für Single-Shot Q5) wird dieser Latenzvorteil zur entscheidenden Größe für die Verarbeitung des Gesamtbestands.

Votum-Verteilung der Self-Consistency. Um die Aussagekraft des Self-Consistency-Verfahrens quantitativ einordnen zu können, wurde die Q4-mit-SC-Konfiguration nachträglich auf Golden 45 mit denselben Seeds $\{7, 11, 19\}$ und Temperatur 0,3 reproduziert; das Reproduktionsskript liegt unter `zendesk/research/stage1/benchmarks/benchmark_llama8b_q4_sc_golden45.py`. Tabelle 5.6 fasst die Verteilung der drei Einzelvoten je Ticket zusammen.

Tabelle 5.6.: Votum-Verteilung der Self-Consistency ($N = 3$, Llama 8B Q4_K_M) auf Golden 45.

Klasse	Tickets	Anteil	Accuracy
Einstimmig (3:0)	40	88,9 %	90,0 % (36/40)
Mehrheitsvotum (2:1)	5	11,1 %	80,0 % (4/5)
Drei-Wege-Aufteilung (1:1:1)	0	0,0 %	—
Gesamt	45	100 %	88,9 % (40/45)

In 88,9 % der Tickets stimmen alle drei Einzelläufe überein, in 11,1 % der Tickets bringt das Mehrheitsvotum 2:1 eine Entscheidung herbei; eine vollständige Aufteilung in drei verschiedene outcome-Bucket tritt nicht auf, was bei einer binären Endzielklassifikation und sechs möglichen outcome-Buckets (solved, admin_only, escalated, abandoned, consultation, no_problem) plausibel ist. Diese Verteilung belegt empirisch den im vorherigen Punkt formulierten Befund „Self-Consistency wird durch Quantisierungsupgrade obsolet“: in fast neun von zehn Tickets liefert ein einzelner Inferenzlauf bereits dieselbe Entscheidung wie der Mehrheitsbeschluss aus drei Läufen, sodass der dreifache Inferenzaufwand keinen zusätzlichen Informationsgewinn erbringt.

Aufschlussreich ist die Aufschlüsselung der Accuracy nach Votum-Klasse: einstimmige Voten sind mit 90,0 % Accuracy klar genauer als Mehrheitsvoten (80,0 %). Self-Consistency identifiziert damit zwar systematisch jene Tickets, an denen das Modell unsicher operiert (Grenzfälle mit semantisch nah beieinanderliegenden outcome-Bucket), erhöht ihre Klassifikationsgüte aber nur partiell. Vier von 40 einstimmigen Tickets sind dabei *einstimmig*

falsch klassifiziert – ein einstimmiges Votum gewährleistet also nicht die Richtigkeit, sondern lediglich die Stabilität der Modellentscheidung. Die Beobachtung stützt das in dieser Arbeit getroffene Designargument, ein stabileres Q5-Single-Shot-Modell dem aufwändigeren Q4-mit-Self-Consistency-Verfahren vorzuziehen, anstatt eine SC-Stufe auf Q5 aufzusatteln, die im überwiegenden Fall ohne Wirkung bliebe.

Dieser Befund motiviert die in Abschnitt 5.2.10 dokumentierte produktive Konfiguration: Llama 3.1 8B Q5_K_M Single-Shot anstelle der zunächst entwickelten Q4-mit-SC-Konfiguration.

5.2.5. Baseline – Qwen 2.5 32B

Der erste produktive Benchmark des Qwen-32B-Klassifikators wurde am 2026-05-07 um 11:34 Uhr abgeschlossen (Commit e75bf63). Tabelle 5.7 zeigt die Konfusionsmatrizen auf beiden Datensätzen.

Tabelle 5.7.: Baseline-Ergebnisse Qwen 2.5 32B Q4_K_M (Single-Pass, $T = 0, 0$)

Datensatz	TP	FP	TN	FN	Acc.	Prec.	Rec.
Golden 45	13	0	22	10	77,8 %	100,0 %	56,5 %
Set 200	36	5	133	26	84,5 %	87,8 %	58,1 %

Auffällig ist die *vollständige Abwesenheit falsch-positiver Ergebnisse* auf Golden 45: kein einziger der vom Modell als *USEFUL* klassifizierten Tickets ist in Wahrheit *USELESS*. Diese Beobachtung muss jedoch mit Blick auf die Stichprobengröße eingeordnet werden: Die ausgewiesenen 100 % Precision basieren auf lediglich 13 als *USEFUL* klassifizierten Tickets. Das zugehörige Wilson-95 %-Konfidenzintervall reicht von 77,2 % bis 100 % (vgl. Tabelle 5.23 in Abschnitt 5.2.13); bereits ein einziger falsch-positiver Treffer in einer erweiterten Stichprobe würde die Precision auf 92,9 % (12/13) bzw. 95,5 % bei 22 negativen Beispielen absenken. Der Befund ist damit als *Tendenz zur konservativen Klassifikation* zu lesen, nicht als beweisbar perfekte Precision. Die Bucket-Verteilung des Modells (Tabelle 5.8) zeigt eine konservative Tendenz: *admin_only* ist auf Set 200 der häufigste Bucket, was mit der tatsächlichen Verteilung interner Provisionierungstickets korreliert.

Die Abdeckung der Pre-Filter-Regeln auf Set 200 beträgt 16 Tickets (8 %) bei einer Präzision von 100 %; die LLM-Stufe verarbeitet entsprechend 184 Tickets.

5.2.6. Modellvergleich

Tabelle 5.9 stellt die drei eingesetzten Konfigurationen gegenüber: die lokale Forschungs-konfiguration mit Llama 3.1 8B Q5_K_M (Apple Silicon, Metal), die finalen Qwen-2.5-32B-

Tabelle 5.8.: Bucket-Verteilung der Modellvorhersagen (Qwen 2.5 32B Baseline)

Bucket	Golden 45	Set 200
solved	16	46
admin_only	10	52
consultation	7	22
escalated	6	17
abandoned	5	33
no_problem	1	30
Σ (LLM)	45	184
Pre-Filter (deterministisch)	0	16

Iterationen v1 und v3 auf Server-CPU sowie die zurückgekehrte Llama-Konfiguration auf Server-CPU.

Vier Befunde fallen ins Auge. Erstens ist Llama 3.1 8B Q5_K_M auf Apple Silicon ungefähr fünf- bis sechsmal schneller als Qwen 2.5 32B auf 16-Core-CPU bei vergleichbarer oder besserer Accuracy auf Golden 45 (88,9% vs. 86,7%). Zweitens schließt sich die Genauigkeitslücke auf dem größeren Set 200 – Qwen v3 erreicht 85,0%, Llama 8B Q5 auf Server-Hardware erreicht 87,0%. Drittens ergibt sich aus der Latenz pro Ticket eine kritische Schranke für die Anwendung auf den Gesamtbestand: bei 16 987 Tickets entspricht 50 s/Ticket einer Hochrechnung von ca. 9 Tagen Server-Laufzeit, was praktisch nicht vertretbar ist und den in Abschnitt 5.2.10 dokumentierten Wechsel zurück zu Llama 3.1 8B motiviert. Viertens überträgt sich der Genauigkeitsvorsprung von Llama 8B Q5 nicht nur, sondern verstärkt sich auf der Server-Plattform sogar leicht: 91,1 % auf Golden 45 (+2,2 Prozentpunkte gegenüber dem lokalen Lauf) und 87,0 % auf Set 200 (+2,0 Prozentpunkte gegenüber dem besten Qwen-Lauf). Eine ticketweise Analyse zeigt, dass der Server-Lauf den falsch-positiven Fall #16836 (*TKI cable design*) korrigiert; gemeinsame Fehler verbleiben auf den drei Tickets #4413, #8240 und #18717.

5.2.7. v2-Versuch: synthetisch kalibrierte Overrides

Im Anschluss an die Baseline wurde der Versuch unternommen, den Recall durch zwei neue Overrides zu erhöhen: Override E (*escalated* → *solved* bei benannter Fix-Version) und Override F (*consultation+confirmed* → *solved* bei generalisierbarer Regel). Die Regex-Muster wurden anhand synthetisch erzeugter Beispieltex te entwickelt. Tabelle 5.10 zeigt das Ergebnis auf Golden 45.

Die Accuracy blieb identisch, doch die Verschiebungen sind aufschlussreich: Override E und F *griffen auf keinem einzigen der 45 Tickets*; das Feld `applied_overrides` blieb in allen Datensätzen leer. Die zwei Vorhersage-Wechsel zwischen v1 und v2 entstammen

Tabelle 5.9.: Modellvergleich – alle eingesetzten Konfigurationen

	Llama 3.1 8B (lokal Mac)	Qwen 2.5 32B (Server, v1)	Qwen 2.5 32B (Server, v3)	Llama 3.1 8B (Server)
Quantisierung	Q5_K_M	Q4_K_M Split-GGUF	Q4_K_M Split-GGUF	Q5_K_M
Modus	Single-Shot $T = 0,0 + \text{Overrides P4/A/B/C}$	Single-Pass, Llama-tuned	Single-Pass + Prompt-Nudges	Single-Shot $T = 0,0 + \text{P4/A/B/C}$
Acc. Golden 45	88,9 % (40/45)	77,8 % (35/45)	86,7 % (39/45)	91,1 % (41/45)
Acc. Set 200	nicht durchgeführt	84,5 % (169/200)	85,0 % (170/200)	87,0 % (174/200)
Acc. Full Corpus	nicht durchgeführt	nicht durchgeführt	abgebrochen	laufend (ca. 5 300 USEFUL erwartet)
Precision G45	90,9 %	100,0 %	94,7 %	95,2 %
Recall G45	87,0 %	56,5 %	78,3 %	87,0 %
F1 G45	88,9 %	72,2 %	85,7 %	90,9 %
Latenz/Ticket	~ 11,7 s	50–65 s	50–65 s	ca. 20 s (CPU)
RAM-Bedarf	ca. 6 GB	ca. 22 GB	ca. 22 GB	ca. 6 GB
ETA Full Corpus 17k	–	–	ca. 9 Tage	ca. 3–4 Tage (laufend)

ausschließlich den Anpassungen am SYSTEM_PROMPT (Tabelle 5.11).

Auf Set 200 wurde der Lauf bei 11 von 200 Tickets abgebrochen, da die Inferenzzeit pro Ticket bei ca. 80 Sekunden lag und das vollständige Ergebnis erst nach mehreren Stunden zu erwarten gewesen wäre.

Ursachenanalyse

Eine systematische Auswertung der Misserfolge ergab, dass die Regex-Muster gegen *synthetische* Beispieltexte kalibriert wurden, die im Vorfeld verfasst wurden. Die echten Rationales, die Qwen zur Laufzeit produziert, weichen jedoch in Phrasing, Reihenfolge und Wortwahl von diesen Vorlagen ab und matchen die Regex-Muster nicht.

Beispiele:

- Override E suchte nach exakten Versionsangaben („NET 14.2.2“, „Converter 2.1“). Reale Qwen-Rationales formulieren stattdessen „forwarded to product team with no current workaround“ oder „fix scheduled for December“ – ohne explizite Versionsnummer.
- Override F enthielt einen Anti-Pattern (_INSTANCE_TIED), der bei „tied to attached files“ oder „tied to their specific project data“ blockierte. Qwen verwendet die Phrase

Tabelle 5.10.: Vergleich Baseline (v1) vs. Versuch v2 auf Golden 45

	v1 Baseline	v2-Versuch	Δ
Accuracy	77,8 %	77,8 %	0,0 %
TP	13	14	+1
FP	0	1	+1
TN	22	21	-1
FN	10	9	-1
Precision	100,0 %	93,3 %	-6,7 PP
Recall	56,5 %	60,9 %	+4,4 PP
F1	72,2 %	73,7 %	+1,5 PP

Tabelle 5.11.: Tickets mit unterschiedlicher v1/v2-Klassifikation

Ticket	Wahrheit	v1 → v2	Ursache
#1196 (DXF Einmessung)	USEFUL	USELESS → USEFUL (korrekt)	Prompt-Nudge (FN gerettet)
#1946 (Ticket Sortierung)	USELESS	USELESS → USEFUL (Fehler)	Prompt-Nudge (neuer FP)

„tied to“ jedoch in nahezu jedem consultation-Rationale, auch wenn das eigentliche Verfahren generalisierbar ist. Dies blockierte gerade die Fälle, die Override F hätte rehabilitieren sollen.

- Der Prompt-Nudge im SYSTEM_PROMPT verschob die Bucket-Verteilung von escalated (6 in v1 → 4 in v2), wodurch Tickets, die durch Override E hätten korrigiert werden sollen, schon im LLM-Schritt anders klassifiziert wurden und die Override-Stufe nicht erreichten.

5.2.8. v3: realdaten-kalibrierte Overrides und Ergebnisse

Aus dem v2-Versuch wurde abgeleitet, dass die Override-Kalibrierung auf realen Modellausgaben erfolgen muss, nicht auf synthetischen Beispieltextrn. Für die v3-Iteration wurden die 10 falsch-negativen Tickets aus der Baseline-Datei `SERVER_qwen32_golden_45_results.jsonl` sowie 26 weitere falsch-negative Tickets aus dem Set-200-Lauf systematisch ausgewertet. Die Auswertung ergab eine Aufteilung in drei Gruppen (Tabelle 5.12).

Aus dieser Analyse wurden vier umgeschriebene Overrides und zwei Prompt-Nudges abgeleitet, die ausschließlich auf den realen Phrasings der Gruppe A basieren:

Tabelle 5.12.: Falsch-negative Tickets der Baseline – Gruppierung nach Rückgewinnbarkeit

Gruppe	Anzahl	Charakteristik
A – rückgewinnbar	12	Rationale enthält benannte Version, klassen-gültige Regel („setting X to Y“, „X vs Y“, „change the name of“) oder „provides the new installer“.
B – uninformativ	8	abandoned-Bucket mit Rationale wie „agent asks for more details“ oder „thread ends unresolved“. Aus dem Rationale allein nicht rehabilitierbar.
C – ohne Klassen-Signal	5	consultation-Bucket mit echtem Instanz-Bezug. Nicht rückgewinnbar.

- **Prompt-Nudges** (System-Prompt): zwei zusätzliche Ausnahmeklauseln, die den escalated-Bucket zu solved umleiten, wenn eine konkrete Fix-Version genannt wird, und den consultation-Bucket zu solved, wenn eine generalisierbare Regel im Rationale erscheint.
- **Override E (umgeschrieben)**: escalated → solved bei benannter Version oder „fix scheduled for [Monat]“ oder „next release fix“, ohne enges Versionsnummern-Schema.
- **Override F (umgeschrieben)**: consultation+confirmed → solved bei „setting X to Y“, „X vs Y“, „change the name of“, „use X instead“; ohne den überaggressiven _INSTANCE_TIED-Anti-Pattern.
- **Override H (neu)**: solved + confirmed=False → confirmed=True, wenn der Agent eine konkrete Version oder einen Installer ausliefert („provides the new installer“, „delivers NET 13.1“).
- **Override I (neu)**: admin_only + confirmed=True → solved, wenn das Rationale eine klassen-gültige Technik beschreibt (z. B. Ticket #1066 „customer changing the name of the area“).

Live-Ergebnisse v3

Die v3-Konfiguration wurde am 2026-05-07 ab 14:30 Uhr auf Server-Hardware ausgeführt (Commit 923bd7a). Tabelle 5.13 zeigt die Konfusionsmatrizen, Tabelle 5.14 den direkten Vergleich gegen die v1-Baseline.

Auf Golden 45 erfüllt v3 die Erwartungen aus der Offline-Simulation exakt: prognostiziert wurden 86,7 %, gemessen wurden 86,7 %. Recall stieg um +21,8 Prozentpunkte (von 13/23 auf 18/23 wiedergefundene USEFUL-Tickets), Precision fiel um nur 5,3 Prozentpunkte. F1

Tabelle 5.13.: v3 Konfusionsmatrizen (Qwen 2.5 32B, Single-Pass)

Datensatz	TP	FP	TN	FN	Acc.	Prec.	Rec.
Golden 45	18	1	21	5	86,7 %	94,7 %	78,3 %
Set 200	41	9	129	21	85,0 %	82,0 %	66,1 %

Tabelle 5.14.: Vergleich v1 Baseline vs. v3 (Qwen 2.5 32B)

	Golden 45			Set 200		
	v1	v3	Δ	v1	v3	Δ
Accuracy	77,8 %	86,7 %	+8,9 PP	84,5 %	85,0 %	+0,5 PP
Precision	100,0 %	94,7 %	−5,3 PP	87,8 %	82,0 %	−5,8 PP
Recall	56,5 %	78,3 %	+21,8 PP	58,1 %	66,1 %	+8,0 PP
F1	72,2 %	85,7 %	+13,5 PP	69,9 %	73,2 %	+3,3 PP
TP / FP / TN / FN	13/0/22/10	18/1/21/5		36/5/133/26	41/9/129/21	

verbesserte sich um +13,5 Prozentpunkte. Auf Set 200 fiel das Resultat hingegen geringer aus als erwartet – die Offline-Simulation hatte +3,5 Prozentpunkte vorhergesagt, gemessen wurden +0,5 Prozentpunkte. Eine ticketweise Aufschlüsselung (Tabelle 5.15) zeigt warum.

Tabelle 5.15.: Per-Ticket-Flips Set 200, v1 → v3

Kategorie	Anzahl	Tickets
<i>Recoveries</i> (v3 korrigiert v1-Fehler)	9	#1196, #2795, #4413, #4890, #11372, #13073, #13613, #13797, #19023
<i>Regressionen</i> (v3 verschlechtert v1)	8	#1722, #1946, #6619, #11384, #11722, #13294, #13687, #14942
Netto	+1	–

Die acht Regressionen sind durchgängig Tickets, bei denen die Prompt-Nudges Qwen zu einer großzügigeren `solved`-Klassifikation veranlassen, obwohl die Wahrheit `USELESS` lautet. Auf Golden 45 spielt dieser Effekt keine Rolle, weil die nudges-anfälligen Konversationsmuster dort nicht enthalten sind; auf Set 200 verteilen sich diese Muster über die zusätzlichen 155 Tickets und reduzieren den durch Recoveries erzielten Gewinn fast vollständig.

Bucket-Verteilungen v1 vs. v3

Die Verschiebungen in den Modellausgaben (Tabelle 5.16) zeigen, dass die Prompt-Nudges das Klassifikationsverhalten gezielt umlenken: solved steigt, escalated und abandoned sinken.

Tabelle 5.16.: Bucket-Verteilung v1 → v3

Bucket	Golden 45			Set 200		
	v1	v3	Δ	v1	v3	Δ
solved	16	22	+6	46	56	+10
admin_only	10	10	0	52	53	+1
consultation	7	6	-1	22	22	0
escalated	6	3	-3	17	11	-6
abandoned	5	3	-2	33	28	-5
no_problem	1	1	0	30	30	0

Anwendung auf den Gesamtbestand

Die anschließende Anwendung der v3-Konfiguration auf den Full Corpus (16 987 Tickets) wurde nach 1 253 verarbeiteten Tickets abgebrochen. Die durchschnittliche Latenz von 50,7s/Ticket ergab eine Hochrechnung von ca. 9 Tagen Server-Laufzeit – nicht vertretbar für eine produktive Wissensbasis-Pipeline. Die 1 253 vorhandenen Vorhersagen liefern jedoch eine Verteilungsschätzung: 234 USEFUL (18,7%), 1 019 USELESS (81,3%), davon 157 Pre-Filter-Treffer (12,5%). Der Anteil verwertbarer Tickets liegt damit unter der ground-truth-basierten Set-200-Schätzung von 31 %, was teilweise auf Sampling-Bias der ersten 1 253 verarbeiteten Tickets zurückzuführen sein kann (sequenziell nach Erstellungsdatum).

5.2.9. Methodische Erkenntnis: leere applied_overrides

Eine Auswertung der v3-Live-Läufe ergab einen unerwarteten und methodisch wesentlichen Befund: das Feld `applied_overrides` ist über alle drei v3-Datensätze hinweg *für jedes einzelne Ticket leer* (Tabelle 5.17). Über 1 497 Vorhersagen hinweg hat keiner der vier sorgfältig auf realen Rationales kalibrierten Overrides E, F, H, I auch nur einmal gegriffen.

Daraus folgt: der gemessene Accuracy-Gewinn von +8,9 Prozentpunkten auf Golden 45 und +0,5 Prozentpunkten auf Set 200 ist *vollständig auf die zwei Prompt-Nudges im System-Prompt zurückzuführen*. Die nachgelagerten Regex-Overrides liefern keinerlei Beitrag; sie verbleiben funktional als Sicherheitsnetz, sind aber auf den vorliegenden Daten überflüssig.

Tabelle 5.17.: Override-Aktivierungen während v3 (alle Läufe)

Datensatz	Tickets	Override-Aktivierungen
Golden 45	45	0
Set 200	200	0
Full Corpus (partial)	1 252	0
Summe	1 497	0

Dieser Befund hat zwei Konsequenzen für die Methodologie:

1. **Prompt-Engineering schlägt Post-Processing auf einem starken Modell.** Die Hauptursache für die in Iteration v3 erzielten Verbesserungen liegt nicht in der Logik nach dem LLM-Aufruf, sondern in der Reformulierung des System-Prompts. Bei einem 32B-Parameter-Modell mit konsistentem Reasoning sind die Modellausgaben hinreichend stabil, dass eine direkte Korrektur am Eingabeende effektiver ist als eine korrigierende Schicht am Ausgabeende.
2. **Offline-Simulation muss zwischen Prompt- und Override-Effekten unterscheiden.** Die Offline-Simulation der v3-Konfiguration auf den Roh-JSONL-Daten der v1-Baseline hatte +3,5 Prozentpunkte für Set 200 vorhergesagt – gemessen wurden nur +0,5 Prozentpunkte. Die Diskrepanz erklärt sich daraus, dass die Simulation nur die Override-Stufe nachstellen konnte und nicht die Prompt-Wirkung; in der Live-Anwendung führten die Prompt-Nudges zu acht Regressionen auf Set 200, die offline nicht sichtbar waren.

Für die zukünftige Iteration bedeutet das, dass eine getrennte Bewertung von *Prompt-Wirkung* und *Override-Wirkung* notwendig ist. Eine vollständige A/B-Studie würde drei Konfigurationen erfordern: (i) Baseline-Prompt + Baseline-Overrides, (ii) Baseline-Prompt + neue Overrides, (iii) neuer Prompt + Baseline-Overrides. Im Rahmen der vorliegenden Arbeit wurde aufgrund des Latenz-Budgets von 50 s/Ticket auf eine solche vollständige Faktorisierung verzichtet.

Bestätigung durch den Llama-Server-Lauf

Der in Abschnitt 5.2.10 dokumentierte Wechsel zu Llama 3.1 8B Q5_K_M auf Server-Hardware liefert eine unabhängige Bestätigung der oben formulierten Methodik-Lehre. Die Llama-Konfiguration verwendet ausschließlich die ursprüngliche Llama-getunte `pipeline.py` mit den Confidence-Overrides P4/A/B/C; die für Qwen entwickelten Overrides E/F/H/I und die Prompt-Nudges des System-Prompts sind *nicht* aktiviert. Trotz dieses Verzichts auf die in Abschnitt 5.2.8 ausgearbeiteten Qwen-spezifischen Korrekturen

erreichte der Server-Lauf 91,1 % Accuracy auf Golden 45 und damit den höchsten Wert aller im Rahmen dieser Arbeit getesteten Konfigurationen. Daraus ergibt sich der Befund, dass die Overrides E/F/H/I keine universellen Verbesserungen darstellen, sondern eine *Qwen-spezifische Therapie* – gerichtet gegen jene Klassifikationsmuster, die Qwen 2.5 32B in seinem Rationale-Stil systematisch erzeugt. Die kleinere Llama-Architektur produziert diese Muster nicht, weshalb auf die nachgelagerte Korrektur verzichtet werden kann. Modellwahl und Prompt-Engineering ersetzen damit auch in der Produktionsumgebung den größten Teil der Post-Processing-Logik.

5.2.10. Wechsel zurück zu Llama 3.1 8B Q5_K_M

Aus den Ergebnissen der drei Qwen-Iterationen ergab sich der Schluss, dass das verfügbare Hardware-Budget eine produktive Anwendung von Qwen 2.5 32B auf den Gesamtbestand ausschließt: bei 50 s/Ticket dauert ein vollständiger Lauf über 16 987 Tickets etwa 9 Tage, und die erreichte Accuracy auf Golden 45 (86,7 %) liegt unter dem in der lokalen Forschungsphase mit Llama 3.1 8B Q5_K_M erreichten Wert von 88,9 %. Daher wurde die Entscheidung getroffen, für die produktive Pipeline zur *ursprünglich verwendeten Llama-Konfiguration* zurückzukehren – jedoch nun auf Server-Hardware ausgeführt, um den Gesamtbestand effizient verarbeiten zu können.

Die produktive Konfiguration ist:

- **Modell:** Meta-Llama-3.1-8B-Instruct-Q5_K_M.gguf (ca. 5,5 GB) – exakt das Modell, das in der lokalen Forschungsphase 88,9 % Accuracy auf Golden 45 erreichte.
- **Pipeline:** unverändert die ursprüngliche `pipeline.py` (Llama-tuned, Overrides P4/A/B/C, ohne die Qwen-spezifischen Erweiterungen E/F/H/I oder Prompt-Nudges).
- **Hardware:** Server WEBSRV38, 16-Core CPU, 31 GB RAM, kein GPU. Zur Inferenz wird `llama-cpp-python` im CPU-Modus mit allen 16 Threads betrieben.
- **Self-Consistency:** $N = 3$ mit Seeds $\{7, 11, 19, 23, 31\}$ und Temperatur 0,3, identisch zur lokalen Konfiguration.

Die Benchmark-Skripte (`benchmark_llama8b_q5_set200.py` und `benchmark_llama8b_q5_full_corpus.py`) wurden am 2026-05-08 ab 11:09 Uhr gestartet. Tabelle 5.18 fasst die finalen Werte des Llama-Vergleichs zusammen.

Diese Rückkehr zur Llama-Konfiguration ist methodisch keine Regression: Das Modell ist dasselbe wie in der lokalen Forschungsphase, einschließlich der dort iterativ entwickelten Pipeline und ihrer Confidence-Overrides. Der Wechsel betrifft ausschließlich die Hardware-Plattform (vom Mac mit Metal-Beschleunigung auf einen reinen CPU-Server),

Tabelle 5.18.: Llama-Vergleich lokal ↔ Server (finale Werte)

	Lokal (Mac M3, Metal)	Server (16-Core CPU)
Acc. Golden 45	88,9 % (40/45)	91,1 % (41/45)
TP / FP / TN / FN G45	20 / 2 / 20 / 3	20 / 1 / 21 / 3
Precision / Recall / F1 G45	90,9 % / 87,0 % / 88,9 %	95,2 % / 87,0 % / 90,9 %
Acc. Set 200	nicht durchgeführt	87,0 % (174/200)
TP / FP / TN / FN S200	–	47 / 11 / 127 / 15
Precision / Recall / F1 S200	–	81,0 % / 75,8 % / 78,3 %
Acc. Full Corpus 16 987	nicht durchgeführt	abgeschlossen: 5 411 USEFUL (31,9 %) / 11 576 USELESS (68,1 %)
Latenz/Ticket	11,65 s	ca. 20 s (CPU)
Wall-Time Full Corpus	–	abgeschlossen am 2026-05-12

die jedoch durch ihre größere RAM-Kapazität und permanente Verfügbarkeit für die Verarbeitung des Gesamtbestands besser geeignet ist. Auf der Server-Plattform übertrifft die Llama-Konfiguration sowohl den lokalen Spitzenwert (91,1 % vs. 88,9 % auf Golden 45) als auch den besten Qwen-Lauf (91,1 % vs. 86,7 % auf Golden 45; 87,0 % vs. 85,0 % auf Set 200). Die ursprüngliche Llama-Konfiguration bleibt damit das Forschungs- und Produktivmodell.

Bucket-Verteilung auf dem Gesamtbestand

Der mit der Llama-3.1-8B-Q5-Server-Konfiguration durchgeführte Full-Corpus-Lauf vom 12. Mai 2026 verarbeitete alle 16 987 Tickets und produzierte für jeden Ticket eine Klassifikation in eine der sechs Outcome-Bucket-Klassen (Abschnitt 3.3.1.4). Tabelle 5.19 zeigt die resultierende Verteilung.

Drei Beobachtungen sind methodisch interessant.

Erstens, der USEFUL-Anteil von 31,9 % stimmt fast exakt mit der Set-200-Schätzung (31 %, vgl. Tabelle 5.1) überein. Die in Abschnitt 5.2.1 bewusst gewählte Klassenverteilung des Validierungssatzes erweist sich damit *a posteriori* als sehr repräsentativ für den Gesamtbestand. Eine größere Abweichung zwischen Set 200 und Gesamtbestand hätte methodische Zweifel an der Generalisierbarkeit der Set-200-Ergebnisse aufgeworfen.

Zweitens, der dominierende USELESS-Bucket ist admin_only (27,9 %). Dieser umfasst Lizenz-, Account- und Berechtigungsänderungen, also überwiegend interne Provisionierungsaufgaben ohne technischen Konversationswert. Daneben dominiert no_problem (14,9 %) – typischerweise interne Notizen, Anruf-Nachfassungen oder

Tabelle 5.19.: Bucket-Verteilung der Llama-3.1-8B-Q5-Server-Konfiguration auf dem vollständigen Zendesk-Korpus von 16 987 Tickets. Die Spalte *USEFUL?* kennzeichnet, ob die jeweilige Klasse die nachgelagerte RAG-Extraktion erreicht. Die Decision-Rule ($USEFUL \Leftrightarrow outcome=solved \wedge customer_confirmed=true$) teilt den *solved*-Bucket in zwei Teilmengen: bestätigte Auflösungen (*USEFUL*) und Auflösungen ohne kundenseitige Bestätigung (*USELESS*).

Outcome-Bucket	Anzahl	Anteil	USEFUL?
solved + customer_confirmed=true	5 411	31,9 %	✓
solved, ohne customer_confirmed	909	5,4 %	
admin_only	4 747	27,9 %	
no_problem	2 538	14,9 %	
escalated	1 565	9,2 %	
abandoned	993	5,8 %	
consultation	824	4,9 %	
Gesamt	16 987	100 %	
USELESS-Summe	11 576	68,1 %	

rein organisatorische Tickets. Beide Klassen werden gezielt vom Pre-Filter und vom LLM-Klassifikator als nicht-verwertbar markiert, ohne dass sie als technische Probleme fehlinterpretiert werden.

Drittens, *escalated* (9,2 %) und *abandoned* (5,8 %) **zusammen umfassen 15 % der Tickets** – Fälle, in denen entweder eine Eskalation an die Produktentwicklung erfolgte (typischerweise ohne Workaround im Ticket) oder die Konversation ohne klare Lösung endete. Diese Tickets enthalten zwar oft ein technisches Problem, aber keine im Sinne der Wissensbasis verwertbare Lösung, weshalb sie konsequent als *USELESS* gefiltert werden. Der überwiegende Teil der „verlorenen“ potenziell-nützlichen Tickets entsteht in dieser Gruppe.

Die 5 411 als *solved* klassifizierten Tickets bilden die Eingabe für die in Abschnitt 5.3 dokumentierte RAG-Datenunit-Extraktion.

5.2.11. Vergleich aller eingesetzten Konfigurationen

Aus den in den vorhergehenden Abschnitten dokumentierten Iterationen lassen sich zwölf Konfigurationen rekonstruieren, für die jeweils die vollständigen Vorhersagen auf Golden 45 zur Verfügung stehen – entweder als JSONL-Datei (Qwen-7B-Pilot, alle Server-Iterationen) oder über die Rekonstruktion aus den Notebook-Auswertungszellen (Llama-Phase 2). Tabelle 5.20 fasst sie unter einer einheitlichen Metrik gegen die finale Ground-Truth-Annotation zusammen.

Die Tabelle macht drei Befunde unmittelbar sichtbar. Erstens steigt die Genauigkeit der

Tabelle 5.20.: Alle rekonstruierbaren Konfigurationen – einheitliche Auswertung gegen finale GT (Golden 45)

Konfiguration	Datum	Quant.	Acc.	Prec.	Rec.	F1	TP	FP	TN	FN
Qwen 2.5 7B v8 (lokal)	2026-05-04	Q4	62,2 %	100,0 %	26,1 %	41,4 %	6	0	22	17
Llama 8B 5d81d59	2026-05-04	Q4	77,8 %	81,0 %	73,9 %	77,3 %	17	4	18	6
Llama 8B dda7a86	2026-05-05	Q4	80,0 %	93,8 %	65,2 %	76,9 %	15	1	21	8
Llama 8B 4a0fef1	2026-05-05	Q4	82,2 %	89,5 %	73,9 %	81,0 %	17	2	20	6
Llama 8B 44750c8	2026-05-05	Q4	75,6 %	80,0 %	69,6 %	74,4 %	16	4	18	7
Llama 8B 7d45b0d SC $N = 3$	2026-05-05	Q4	84,4 %	90,0 %	78,3 %	83,7 %	18	2	20	5
Llama 8B 623917d	2026-05-05	Q5	86,7 %	90,5 %	82,6 %	86,4 %	19	2	20	4
Llama 8B 4560fa1 (lokaler Peak)	2026-05-05	Q5	88,9 %	90,9 %	87,0 %	88,9 %	20	2	20	3
Qwen 32B v1 Baseline (Server)	2026-05-07	Q4	77,8 %	100,0 %	56,5 %	72,2 %	13	0	22	10
Qwen 32B v2 (Server)	2026-05-07	Q4	77,8 %	93,3 %	60,9 %	73,7 %	14	1	21	9
Qwen 32B v3 (Server)	2026-05-07	Q4	86,7 %	94,7 %	78,3 %	85,7 %	18	1	21	5
Llama 8B Q5 (Server)	2026-05-09	Q5	91,1 %	95,2 %	87,0 %	90,9 %	20	1	21	3

Llama-Konfiguration über die sieben dokumentierten Iterationen monoton von 75,6 % auf 88,9 % (lokal) und schließlich auf 91,1 % (Server). Zweitens ist Qwen 32B in der Baseline-Konfiguration zwar maximal präzise (100 % Precision), aber durch geringen Recall begrenzt; v3 schließt den Rückstand bei Accuracy weitgehend, ohne den Llama-Spitzenwert zu erreichen. Drittens dominiert die finale Llama-Server-Konfiguration auf allen vier Hauptmetriken – Accuracy, Recall, F1 und (gemeinsam mit v1) Precision – mit Ausnahme der absoluten Precision von Qwen 32B v1, deren Vorteil durch die katastrophale Recall-Lücke jedoch sofort relativiert wird (10 falsch-negative Tickets vs. 3 bei Llama Q5 Server).

Universell schwierige Tickets

Eine ticketweise Übersicht der Vorhersagen aller zwölf Konfigurationen ist in Anhang A.4 (Tabelle A.5) wiedergegeben. Aus dieser Vergleichsmatrix lassen sich vier strukturell schwierige Tickets identifizieren, die von der überwiegenden Mehrheit oder allen Konfigurationen falsch klassifiziert werden (Tabelle 5.21).

Die vier Tickets aus Tabelle 5.21 sind methodisch relevant, weil sie eine harte Obergrenze für die durch Prompt-Engineering und nachgelagerte Heuristiken erreichbare Klassifikationsgüte markieren: Ticket #18717 wird von keiner Konfiguration korrekt klassifiziert, weil das Modell-Eingabematerial keine eindeutige Resolution-Signatur enthält. Auch Tickets, die nur in einer oder wenigen Konfigurationen korrekt klassifiziert werden, sind Hinweise auf annotatorische Grenzfälle, in denen die binäre USEFUL/USELESS-Zuordnung selbst nicht eindeutig festgelegt werden kann. Eine systematische Auswertung dieser Grenzfälle ist Gegenstand der weiterführenden Arbeiten (Abschnitt 5.2.15).

5.2.12. Externe Validierung durch Frontier-LLMs via VS Code Copilot

Zur unabhängigen Validierung der manuellen Ground-Truth-Annotation des Golden-45-Datensatzes und gleichzeitig zur Einordnung der lokal trainierten Pipeline gegen den Stand der kommerziellen Sprachmodelle wurde das Golden-45-Klassifikationsverfahren parallel von sieben kommerziellen Frontier-Modellen durchgeführt. Der Zugriff erfolgte über GitHub Copilot Pro [20] in Visual Studio Code [33], dessen Tarif Multi-Model-Zugriff auf die folgenden Anbieter bietet: Anthropic (Claude Opus 4.7 [3], Sonnet 4.6 [4], Haiku 4.5 [2]), OpenAI (GPT-5.5 [37], GPT-5.4 [36]), Google DeepMind (Gemini 3.1 [21]) und xAI (Grok 4.1 Fast [52]).

Methodik

Die Validierung erfolgte in zwei Phasen. *Phase 1*: Der Autor führte eine vollständige manuelle Annotation der 45 Tickets durch (siehe Abschnitt 5.2.1). *Phase 2*: Jedes der sieben

Tabelle 5.21.: Tickets, die in 11 oder 12 der 12 Konfigurationen falsch klassifiziert wurden

Ticket	Wahrheit	korr. Konfigs	Fehler	Erklärung
#18717 GF+ Data Transfer	USEFUL	0/12	12 FN	Echtes Problem, aber der Thread endet mit weiteren Rückfragen des Agents ohne dokumentierte Lösung.
#4413 Search address	USEFUL	1/12	11 FN	Funktionalitätsproblem wird an Entwicklung weitergeleitet; nur Qwen 32B v3 erkennt den fix-versionierten Charakter.
#8240 Adding crossings	USEFUL	5/12	7 FN	Beratungsfall mit kunden-spezifischen Daten; nur Qwen 32B-Varianten erkennen die generalisierbare Vorgehensweise im Rationale.
#2759 Bezugslinien Migr.	USELESS	6/12	6 FP	Migrationsbug mit Workaround; Llama interpretiert Workaround als Lösung, Qwen 32B bleibt strenger.

Modelle wurde mit demselben System-Prompt instruiert, sich in die Rolle eines Support-Agenten zu versetzen und für jeden Ticketverlauf die Frage zu beantworten, ob darin wiederverwendbare technische Information enthalten ist, deren Aufnahme in eine zukünftige Wissensbasis sinnvoll wäre (USEFUL/USELESS). Dieselben PII-maskierten Ticketkonversationen, die dem lokalen Klassifikator vorgelegt wurden, dienten als Eingabe. Eine ticketweise Mehrheitsentscheidung der sieben Modelle („AI-Vote“) wurde als zusätzliche Vergleichsgröße ausgewertet.

Ergebnisse

Tabelle 5.22 zeigt die Klassifikationsgüte jedes Modells gegen die menschliche Annotation. Die vollständige ticketweise Vergleichsmatrix mit allen sieben Modellausgaben ist in Anhang A.5 dokumentiert.

Tabelle 5.22.: Klassifikationsgüte der Frontier-Modelle auf Golden 45 (gegen menschliche Annotation)

Modell / Anbieter	Acc.	Prec.	Rec.	F1	TP	FP	TN	FN
Gemini 3.1 (Google DeepMind)	100,0 %	100,0 %	100,0 %	100,0 %	23	0	22	0
Grok 4.1 Fast (xAI)	97,8 %	95,8 %	100,0 %	97,9 %	23	1	21	0
AI-Vote (Mehrheit, 7 Modelle)	93,3 %	–	–	–	22	2	20	1
Sonnet 4.6 (Anthropic)	82,2 %	94,1 %	69,6 %	80,0 %	16	1	21	7
Opus 4.7 (Anthropic)	77,8 %	78,3 %	78,3 %	78,3 %	18	5	17	5
GPT-5.5 (OpenAI)	77,8 %	100,0 %	56,5 %	72,2 %	13	0	22	10
GPT-5.4 (OpenAI)	75,6 %	100,0 %	52,2 %	68,6 %	12	0	22	11
Haiku 4.5 (Anthropic)	64,4 %	62,1 %	78,3 %	69,2 %	18	11	11	5
<i>Referenz: Llama 3.1 8B Q5 Server (lokal entwickelt)</i>	<i>91,1 %</i>	<i>95,2 %</i>	<i>87,0 %</i>	<i>90,9 %</i>	<i>20</i>	<i>1</i>	<i>21</i>	<i>3</i>

Aus der Auswertung ergeben sich vier Beobachtungen:

- **Gemini 3.1 erreicht perfekte Übereinstimmung mit der menschlichen Annotation.** Bei keiner der 45 Klassifikationsentscheidungen weicht das Modell vom annotierten Label ab. Dies dient zugleich als externe Bestätigung der Konsistenz der Ground-Truth-Annotation: Es ist möglich, eine vollständig konvergente Klassifikation zu erreichen, was die Eindeutigkeit der Mehrzahl der Ticketverläufe untermauert.
- **Die lokal entwickelte Llama-Pipeline (91,1 %) übertrifft fünf von sieben kommerziellen Frontier-Modellen.** Lediglich Gemini 3.1 und Grok 4.1 Fast liegen vor der lokalen Pipeline; Opus 4.7, Sonnet 4.6, beide GPT-Varianten und insbesondere Haiku 4.5 erreichen geringere Genauigkeit. Dies ist ein bemerkenswerter Befund für eine domänenspezifisch kalibrierte Pipeline mit einem 8B-Parameter-Modell, das auf einem CPU-Server läuft, gegenüber Frontier-Modellen mit vermutlich um Größenordnungen mehr Parametern und API-basiertem GPU-Backend.
- **Die Anthropic- und OpenAI-Modelle zeigen unterschiedliche Fehler-Profile.** Sonnet 4.6 und beide GPT-Varianten sind ausgeprägt konservativ (Precision ≥ 94 %, niedriger Recall) – sie übergehen viele tatsächlich nützliche Tickets als USELESS. Haiku 4.5 ist umgekehrt deutlich liberal (Recall 78,3 %, aber Precision nur 62,1 %) – es klassifiziert systematisch mehr Tickets als USEFUL und produziert dadurch 11 falsch-positive.

Opus 4.7 ist das einzige Modell mit annähernd ausgeglichener Verteilung von FP und FN.

- **AI-Mehrheitsentscheidung erreicht 93,3 %.** Bei strikter Mehrheitsabstimmung über alle sieben Modelle entstehen drei Fehler (FP #16836, FP #13102 – bei diesen lag eine 3-zu-4-Aufteilung gegen den Trend vor – und FN #1196). Die Mehrheitsentscheidung liegt damit unter den Einzelresultaten von Gemini und Grok, aber deutlich über dem Durchschnitt der sieben Modelle (82,2 %). 15 der 45 Tickets erhielten ein einstimmiges Votum (7 zu 0); alle einstimmigen Voten stimmen mit der menschlichen Annotation überein – ein zusätzlicher Indikator für die Reliabilität der Annotation.

Methodische Einordnung

Die parallele Klassifikation durch Frontier-Modelle erfüllt in dieser Arbeit zwei Funktionen. Sie dient *erstens* als externe Validierung der Ground-Truth-Annotation: Da der Annotationsprozess durch einen einzelnen Autor durchgeführt wurde und damit keine Inter-Annotator-Agreement-Kennzahl ableitbar ist (vgl. Abschnitt 5.2.1), bietet die unabhängige Klassifikation durch sieben kommerzielle Modelle einen alternativen Reliabilitätsindikator – besonders die perfekte Übereinstimmung von Gemini 3.1 und die 15 einstimmigen 7-zu-0-Voten unterstreichen die Eindeutigkeit der Mehrzahl der Annotationen. Sie dient *zweitens* als Vergleichsmaßstab für die lokale Pipeline: dass die lokal entwickelte Llama-Konfiguration nicht nur kosten- und latenzseitig, sondern auch in der Klassifikationsgüte mit den meisten Frontier-Modellen mithält, motiviert den weiteren Einsatz lokaler Modelle in Produktivumgebungen, in denen Datenschutz, Latenz oder Lizenzkosten gegen API-basierte Frontier-Modelle sprechen.

5.2.13. Statistische Robustheitsanalyse

Die in den vorangegangenen Abschnitten berichteten Punktschätzungen für Accuracy, Precision und Recall beantworten nicht die Frage, wie zuverlässig die beobachteten Unterschiede zwischen Konfigurationen sind. Insbesondere auf Golden 45 entspricht ein einzelner Klassifikationsfehler 2,22 Prozentpunkten, sodass kleine berichtete Differenzen mit der Größenordnung zufälliger Einzelentscheidungen zusammenfallen. Dieser Abschnitt ergänzt die Auswertung um zwei standardisierte Robustheitsverfahren: Wilson-Konfidenzintervalle für jede Einzelmetrik und exakte McNemar-Tests für die drei zentralen Konfigurationsvergleiche der Arbeit. Alle Berechnungen sind in `compute_statistics.py` dokumentiert und werden im Anhang per Konfiguration ausgewiesen.

Wilson-Konfidenzintervalle

Tabelle 5.23 fasst die 95 %-Wilson-Konfidenzintervalle für die wichtigsten Konfigurationen auf Golden 45 zusammen. Die Intervallbreiten reichen für Accuracy bis zu rund 24 Prozentpunkten und verdeutlichen die geringe statistische Auflösung des Datensatzes. Beispielhaft ist der häufig zitierte Punktwert „100 % Precision“ der Qwen-Baseline v1 mit einem Wilson-Konfidenzintervall von [77,2 %; 100 %] verbunden – die Beobachtung ist statistisch nicht von einer Precision von beispielsweise 85 % unterscheidbar.

Tabelle 5.23.: Wilson-95 %-Konfidenzintervalle für die wichtigsten Konfigurationen auf Golden 45 ($n = 45$). Punktwerte und Intervallgrenzen in Prozent.

Konfiguration	Acc.	95 %-CI	Prec.	95 %-CI	Recall	95 %-CI
Llama Q4 SC ($N=3$, 7d45b0d)	84,4	[71,2; 92,3]	90,0	[69,9; 97,2]	78,3	[58,1; 90,3]
Llama Q5 Single-Shot (623917d)	86,7	[73,8; 93,7]	90,5	[71,1; 97,3]	82,6	[62,9; 93,0]
Llama Q5 Single-Shot + Overrides (4560fa1)	88,9	[76,5; 95,2]	90,9	[72,2; 97,5]	87,0	[67,9; 95,5]
Qwen 32B v1 (Server, Baseline)	77,8	[63,7; 87,5]	100,0	[77,2; 100,0]	56,5	[36,8; 74,4]
Qwen 32B v3 (Server, real-kalibriert)	86,7	[73,8; 93,7]	94,7	[75,4; 99,1]	78,3	[58,1; 90,3]
Llama 8B Q5 Server (produktiv)	91,1	[79,3; 96,5]	95,2	[77,3; 99,2]	87,0	[67,9; 95,5]

Auf Set 200 sind die Intervalle erwartungsgemäß deutlich enger (Tabelle 5.24). Die Genauigkeitsintervalle der drei Server-Konfigurationen überlappen vollständig, was eine wichtige inhaltliche Konsequenz hat: Aussagen wie „Llama Server ist genauer als Qwen v3“ sind auf Set 200 *datentechnisch nicht eindeutig belegt*; die Differenz wird erst durch die im folgenden Unterabschnitt durchgeführten McNemar-Tests einer formalen Prüfung unterzogen.

Tabelle 5.24.: Wilson-95 %-Konfidenzintervalle der drei Server-Konfigurationen auf Set 200 ($n = 200$). Punktwerte und Intervallgrenzen in Prozent.

Konfiguration	Acc.	95 %-CI	Prec.	95 %-CI	Recall	95 %-CI
Qwen 32B v1 (Server, Baseline)	84,5	[78,8; 88,9]	87,8	[74,5; 94,7]	58,1	[45,7; 69,5]
Qwen 32B v3 (Server, real-kalibriert)	85,0	[79,4; 89,3]	82,0	[69,2; 90,2]	66,1	[53,7; 76,7]
Llama 8B Q5 Server (produktiv)	87,0	[81,6; 91,0]	81,0	[69,1; 89,1]	75,8	[63,8; 84,8]

Bootstrap-Konfidenzintervalle für F1

Wilson-Intervalle setzen voraus, dass die zugrunde liegende Metrik eine reine Binomialproportion ist. Für den F1-Wert ist diese Annahme nicht erfüllt, da F1 eine zusammengesetzte Größe aus Precision und Recall darstellt. Aus diesem Grund wird F1 zusätzlich über ein nicht-parametrisches Bootstrap-Verfahren (1.000 Resamples, Perzentil-Methode) bewertet. Die ermittelten Bootstrap-Intervallbreiten stimmen mit dem Wilson-Befund qualitativ

überein: auf Golden 45 sind die Intervalle für alle Konfigurationen breit (typisch 15–25 Prozentpunkte), auf Set 200 deutlich enger (typisch 7–10 Prozentpunkte). Die vollständigen Werte sind in `output/statistics_summary.json` dokumentiert.

McNemar-Tests für drei zentrale Konfigurationsvergleiche

Wilson-Intervalle bewerten Einzelmetriken isoliert. Für den *paarweisen* Vergleich zweier Klassifikatoren auf identischen Tickets ist der exakte McNemar-Test methodisch korrekt, da er die ticketweise Übereinstimmung beider Klassifikatoren auswertet und nicht die aggregierten Genauigkeiten. Tabelle 5.25 dokumentiert drei für die Arbeit zentrale Vergleiche.

Die Spalte b gibt die Anzahl der Tickets an, bei denen Klassifikator A korrekt und Klassifikator B falsch lag; c ist umgekehrt. Unter der Nullhypothese gleicher Klassifikationsfähigkeit ist die Anzahl der Diskrepanzen ($b + c$) binomialverteilt mit Erfolgswahrscheinlichkeit 0,5.

Tabelle 5.25.: Exakte McNemar-Tests (Binomialvariante) für drei zentrale Konfigurationsvergleiche. b = Klassifikator A korrekt, B falsch; c = umgekehrt; p = exakter zweiseitiger Binomialwert. Keiner der Tests erreicht das übliche Signifikanzniveau $\alpha = 0,05$.

Vergleich	Datensatz	n	b	c	p	signifikant?
Q4 SC (7d45b0d) vs. Q5 Single-Shot (623917d)	Golden 45	45	1	2	1,000	nein
Qwen 32B v1 vs. Llama 8B Q5 Server	Golden 45	45	2	8	0,109	nein
Qwen 32B v1 vs. Llama 8B Q5 Server	Set 200	200	13	18	0,473	nein
Qwen 32B v3 vs. Llama 8B Q5 Server	Golden 45	45	3	5	0,727	nein
Qwen 32B v3 vs. Llama 8B Q5 Server	Set 200	200	12	16	0,572	nein

Drei Befunde sind methodisch relevant.

Erstens: Die +2,3-Prozentpunkte-Differenz Q4 → Q5 ist statistisch nicht von Rauschen unterscheidbar. Der McNemar-Test ergibt $p = 1,00$ ($b = 1$, $c = 2$). Anders ausgedrückt: zwischen Self-Consistency-Q4 und Single-Shot-Q5 differieren genau drei der 45 Tickets, davon zwei zugunsten der Q5-Konfiguration. Eine Stichprobe von 45 Tickets reicht nicht aus, um aus dieser Verteilung eine systematische Quantisierungs- oder Decoding-Wirkung abzuleiten. Die in Abschnitt 5.2.4 dargestellte Wahl zugunsten von Q5 wird damit nicht durch eine signifikante Genauigkeitsverbesserung, sondern durch die dort dokumentierten Latenz-, Speicher- und Wartbarkeitsvorteile getragen. Zwei weitere Aspekte stützen diese Wahl methodisch. Zum einen ist Single-Shot Q5 mit Temperatur $T = 0,0$ deterministisch reproduzierbar: identische Eingaben erzeugen identische Klassifikationen, während Self-Consistency Q4 mit $T = 0,3$ und $N = 3$ Inferenzläufen eine inhärente stochastische Streuung aufweist und eine bitidentische Reproduktion eines Einzellaufs ohne Fixierung sämtlicher Seeds und Sampling-Parameter nicht garantieren kann.

Diese Reproduzierbarkeit ist für die nachgelagerte Pipeline (Indexierung, Audit-Logs, Annotationsabgleich) eine harte Anforderung. Zum anderen liefert Q5 in der manuellen Sichtkontrolle der erzeugten Bucket-Rationales (vgl. Abschnitt 5.2.6) durchgängig kohärentere und besser strukturierte Begründungstexte; dieser Qualitätsunterschied wurde nicht quantitativ ausgewertet, ist aber für die Auditierbarkeit der Filter-Entscheidungen im Produktivbetrieb relevant.

Zweitens: Auch Qwen 32B vs. Llama 8B Q5 Server ist auf keinem Datensatz signifikant. Die p -Werte liegen zwischen 0,11 (Qwen v1 auf Golden 45) und 0,73 (Qwen v3 auf Golden 45); die übrigen Werte liegen zwischen 0,47 und 0,57. Die produktive Wahl zugunsten von Llama 8B Q5 wird damit nicht durch einen *statistisch belegten* Genauigkeitsvorsprung gegenüber Qwen v3 getragen, sondern durch die in Abschnitt 5.2.6 und 5.2.10 dokumentierten praktischen Kriterien: rund fünf- bis sechsmal niedrigere Latenz pro Ticket, ein etwa viermal geringerer Speicherbedarf und eine Hochrechnung von 3–4 statt 9 Tagen für die Verarbeitung des Gesamtbestands.

Drittens: Statistische Signifikanz und praktische Relevanz fallen hier auseinander. Die geringe Stichprobengröße schließt einen signifikanten Nachweis kleiner bis mittlerer Genauigkeitsunterschiede strukturell aus – $n = 45$ benötigt für einen zweiseitigen McNemar-Test bei Effektstärken von ca. 5 Prozentpunkten typischerweise $b + c \geq 12$, was die hier beobachteten Werte von $b + c \in \{3, 8, 10, 28, 31\}$ überwiegend unterschreiten. Eine größere annotierte Stichprobe (z. B. $n = 500$) wäre eine direkte Folgearbeit, um die berichteten Tendenzen statistisch abzusichern. Für die vorliegende Arbeit folgt aus dieser Beobachtung kein anderes Modelldesign, jedoch eine entsprechend vorsichtige Formulierung der berichteten Ergebnisse: die Punktwerte in den Tabellen 5.4, 5.5, 5.7 und 5.9 sind als Tendenzen, nicht als belastbare Genauigkeitsunterschiede zu lesen, und werden durch die ergänzenden Wilson-Konfidenzintervalle dieses Abschnitts kontextualisiert.

Out-of-Sample-Validierung auf den 155 nicht überlappenden Tickets

Da Golden 45 vollständig in Set 200 enthalten ist und gleichzeitig als Iterations- und Kalibrierungsdatsatz für Prompts und Confidence-Overrides diente, ist die berichtete Set-200-Accuracy potenziell zugunsten dieser 45 überlappenden Tickets verzerrt. Eine echte Out-of-Sample-Validierung umfasst daher nur die verbleibenden 155 Tickets ($200 - 45 = 155$), die ausschließlich in Set 200, nicht jedoch in Golden 45 enthalten sind. Auf dieser Teilmenge sind 39 Tickets als USEFUL und 116 als USELESS annotiert, was einer Klassenverteilung von 25,2 % USEFUL entspricht und damit etwas stärker zu USELESS verschoben ist als die 31 % des vollen Set 200. Tabelle 5.26 stellt die Ergebnisse aller sieben Klassifikatoren auf dem vollen Set 200 (Spalte 200) der Auswertung auf den 155 Out-of-Sample-Tickets (Spalte OOS 155) gegenüber.

Tabelle 5.26.: Vergleich der Klassifikationsleistung auf dem vollen Set 200 ($n = 200$, inklusive 45 Kalibrierungstickets) und dem Out-of-Sample-Anteil ($n = 155$, ohne Golden-45-Tickets). Die Spalte ΔAcc gibt die Differenz $\text{Acc}(\text{OOS}) - \text{Acc}(200)$ in Prozentpunkten an. Wilson-95 %-Konfidenzintervalle beziehen sich auf die OOS-Accuracy. Alle Werte in Prozent.

Klassifikator	Set 200		OOS 155				ΔAcc
	Acc.	F1	Acc.	95 %-CI	F1		
B1 Majority	69,0	0,0	74,8	[67,5; 81,0]	0,0		+5,8
B2 Status-only	33,0	47,7	27,1	[20,7; 34,6]	40,2		-5,9
B3 Status + ≥ 3 öff. Kommentare	61,0	59,8	60,6	[52,8; 68,0]	54,8		-0,4
B4 5-stufiger Pre-Filter (rule-based)	68,5	48,8	73,5	[66,1; 79,9]	50,6		+5,0
L1 Qwen 2.5 32B v1 (Server)	84,5	69,9	86,5	[80,2; 91,0]	68,7		+2,0
L2 Qwen 2.5 32B v3 (Server)	85,0	73,2	84,5	[78,0; 89,4]	65,7		-0,5
L3 Llama 3.1 8B Q5 (Server, produktiv)	87,0	78,3	85,8	[79,4; 90,4]	71,1		-1,2

Drei Beobachtungen sind aus dem Out-of-Sample-Vergleich abzuleiten.

(a) Der durch die Überlappung verursachte Bias der produktiven Llama-Konfiguration ist gering. L3 fällt auf der Out-of-Sample-Menge von 87,0 % Accuracy auf 85,8 % und damit um lediglich 1,2 Prozentpunkte (entspricht ca. zwei Tickets). Diese Differenz liegt weit innerhalb des Wilson-95 %-Konfidenzintervalls [79,4 %; 90,4 %] und ist damit nicht von statistischem Rauschen unterscheidbar. Auch L2 (Qwen v3) verändert sich nur marginal (-0,5 Prozentpunkte). Die durch die Überlappung verursachte günstige Verzerrung der Set-200-Resultate ist daher zwar real, aber klein; die im Hauptergebnis berichtete Accuracy von 87,0 % verschiebt sich bei sauberer Out-of-Sample-Bewertung auf 85,8 %.

(b) Regelbasierte Baselines und L1 zeigen größere Verschiebungen, jedoch primär durch die geänderte Klassenverteilung. B1 Majority steigt um +5,8 Prozentpunkte (69,0 $\rightarrow$ 74,8 %), weil der Out-of-Sample-Anteil mit 25,2 % USEFUL einen höheren USELESS-Anteil aufweist und der reine Mehrheits-Prädiktor entsprechend häufiger richtig liegt. B2 Status-only verliert um -5,9 Prozentpunkte aus demselben Grund: die Out-of-Sample-Tickets sind häufiger im resolved-Endzustand archiviert, ohne wiederverwertbares Wissen zu enthalten. L1 (Qwen v1) zeigt mit +2,0 Prozentpunkten (84,5 $\rightarrow$ 86,5 %) sogar eine leichte Verbesserung auf der Out-of-Sample-Menge, was wiederum durch die für eine recall-vorsichtige Konfiguration günstigere Klassenverteilung erklärbar ist (weniger USEFUL-Tickets, die der Klassifikator hätte erkennen müssen).

(c) Die Out-of-Sample-Auswertung bestätigt die Reihenfolge der Konfigurationen. L3 bleibt mit 85,8 % Accuracy und F1 von 71,1 % die stärkste Konfiguration im Vergleich; L1 (86,5 % Acc, F1 68,7 %) und L2 (84,5 % Acc, F1 65,7 %) verbleiben statistisch nicht von L3 unterscheidbar (vgl. McNemar-Ergebnisse in Tabelle 5.25), liegen aber im F1-Wert deut-

lich hinter L3. Die regelbasierten Baselines bleiben unverändert klar abgeschlagen: B3 (die vom Gutachter vorgeschlagene Heuristik) erreicht auf der Out-of-Sample-Menge nur 60,6 % Accuracy bzw. F1 54,8 %, und der ambitionierte 5-stufige Pre-Filter B4 erreicht trotz seiner deutlich höheren Accuracy (73,5 %) einen F1-Wert von lediglich 50,6 % – ein Symptom dafür, dass er die für die nachgelagerte Wissensbasis allein relevante Klasse *USEFUL* systematisch zu konservativ identifiziert. Die in dieser Arbeit getroffene Entscheidung zugunsten der LLM-basierten Pipeline bleibt auch unter strenger Out-of-Sample-Bewertung tragfähig.

5.2.14. Fehleranalyse

Eine Aufschlüsselung der falsch-negativen Tickets nach ursprünglich vom Modell zugewiesenem Bucket (Tabelle 5.27) verdeutlicht den systematischen Charakter der Recall-Lücke.

Tabelle 5.27.: FN-Verteilung nach Bucket (Golden 45 + Set 200, 36 Tickets)

Bucket × confirmed	Anzahl	Modell-Annahme	Tatsächlicher Inhalt
abandoned, False	11	Agent fragt Details, Thread tot	Lösung erscheint später im langen Thread
escalated, False	10	An Entwicklung weitergeleitet, kein Workaround	Konkrete Fix-Version genannt
consultation, True	7	Hinweise spezifisch zu Kundendaten	Klassen-gültige Regel-/Setting/Filter
consultation, False	3	wie oben	wie oben
admin_only, True	3	Reine Provisionierung	Wiederverwendbare technische Aktion
solved, False	2	Keine Kundenbestätigung	Generische UI-Prozedur (silent confirm)

Drei Erkenntnisse lassen sich ableiten: Erstens verteilen sich die FN-Tickets nicht zufällig, sondern konzentrieren sich in vier Bucket-Klassen, was die Override-basierte Behebung architektonisch sinnvoll macht. Zweitens treten die zehn falsch-negativen Tickets aus Golden 45 *alle* auch in Set 200 auf, was nahelegt, dass das Verhalten des Modells auf diesen Mustern systematisch und nicht zufällig ist. Drittens ergibt sich eine harte Obergrenze für rein regelbasierte Korrekturen: die acht Tickets der Gruppe B (abandoned mit uninformativem Rationale) sind aus der Modellausgabe allein nicht rehabilitierbar; sie würden einen tieferen Eingriff (z. B. eine zweite LLM-Stufe oder eine Reformulierung des Prompts) erfordern.

5.2.15. Diskussion

Die Qualitätsfilterung erreicht in der aktuellen Baseline-Konfiguration einen ausgeprägten Vorteil bei der Präzision (100 % auf Golden 45, 87,8 % auf Set 200) bei einer moderaten Recall (56-58 %). Für die Anforderung an die Wissensbasis ist dies das gewünschte Verhalten: ein einzelner falsch-positiver Ticket im Index beeinträchtigt potenziell sämtliche nachfolgenden Retrieval-Treffer, während falsch-negative Tickets durch eine spätere manuelle Sichtung jederzeit rehabilitiert werden können.

Der Übergang von Llama 3.1 8B (88,9 % auf Golden 45) zu Qwen 2.5 32B (77,8 % auf Golden 45, aber 84,5 % auf Set 200) ist auf den ersten Blick scheinbar paradox. Eine genauere Betrachtung zeigt jedoch, dass Llama auf dem kleinen Datensatz nicht nur lernte zu klassifizieren, sondern auch *die Verteilung des Datensatzes auswendig zu lernen*: Iterationen, in denen Llama 88,9 % auf Golden 45 erreichte, fielen auf größeren Datensätzen häufig auf 70-80 %. Qwen zeigt das umgekehrte Muster – höhere Set-200-Accuracy als Golden-45-Accuracy –, was auf eine bessere Generalisierung hinweist. Das größere Modell ist konservativer (höhere Precision) und systematischer (FN-Tickets konzentrieren sich in wenigen, klar charakterisierten Bucket-Klassen).

Der gescheiterte v2-Versuch verdeutlicht eine methodische Lehre: *regelbasierte Modellkorrekturen müssen gegen die tatsächlichen Modellausgaben kalibriert werden, nicht gegen synthetisch erzeugte Beispiele*. Die im Vorfeld verfassten Beispieltexte haben lediglich die Konsistenz der Regex-Muster mit dem antizipierten Wortlaut überprüft – nicht aber, ob das Modell diese Phrasings in der Realität auch tatsächlich produziert. Die v3-Strategie korrigiert diesen Fehler durch ausschließliche Kalibrierung an den realen Rationales aus der Baseline-JSONL-Datei.

5.3. RAG-Datenunit-Extraktion

Die zweite Stufe der Datenverarbeitungspipeline – die strukturierte Extraktion von RAG-Datenunits aus den als USEFUL klassifizierten Tickets (Abschnitt 3.3.1.11) – wurde nach demselben Schema wie die Qualitätsfilterung evaluiert: zunächst auf einem kleinen, handkuratierten Golden-Set zur schnellen Iteration und Quality-Gate-Validierung, anschließend in einem Full-Run über alle ca. 5 400 USEFUL-Tickets.

5.3.1. Quality-Gate auf Golden-23

Vor jedem Full-Run wird die Pipeline auf einem fixierten Datensatz aus 23 hand-kuratierten Tickets (`zendesk/stage2_extraction/data/test_golden_useful.json`) ausgeführt, die in 25 Referenz-Datenunits manuell zerlegt wurden (`data/golden_data_units.json`).

Das Skript `zendesk/stage2_extraction/src/snapshot.py` berechnet sieben deterministische Metriken; `zendesk/stage2_extraction/src/gate_check.py` prüft sie gegen die in Abschnitt 3.3.1.11 dokumentierten Hard-Schwellen. Tabelle 5.28 zeigt die bislang abgeschlossenen Gate-Läufe.

Tabelle 5.28.: Gate-Check-Läufe auf Golden-23 (alle Hard-Schwellen aus Tabelle 3.6)

Datum	Run-ID	Modell	units	split P/R	section	ent. rec.	lang
2026-05-13	baseline	Llama 3.1 8B Q5_K_M	25/25	1,00 / 1,00	0,565	0,40	1,00
2026-05-17	qwen32b_verifier	Qwen 2.5 32B Q4_K_M	23/25	1,00 / 0,00	0,783	0,39	1,00

Der *Baseline-Lauf* mit Llama 3.1 8B Q5_K_M (Commit-Hash `baseline`) bestand alle Schwellen: 25 von 25 Referenz-Units wurden erzeugt, beide Multi-Topic-Tickets (#11372 und #13102) korrekt zerlegt (Split-Recall = 1,00, Split-Precision = 1,00), Section-Match knapp über der Schwelle bei 56,5%. Die Entity-Recall liegt bei 40% – über der reinen Advisory-Schwelle von 30%.

Der zweite Lauf (*qwen32b_verifier*) wurde als Vergleichsexperiment mit dem größeren Qwen-Modell durchgeführt. Auffällig: Qwen produzierte nur 23 statt 25 Units, weil beide Multi-Topic-Tickets fälschlich als `single` klassifiziert wurden (Split-Recall = 0). Dafür stieg die Section-Match-Quote von 56,5% auf 78,3% – eine systematisch genauere Kategorisierung. Diese Beobachtung wiederholt das Muster aus der Filter-Evaluation: das größere Modell ist konservativer (vermeidet Multi-Topic-Splits), liefert aber bei den verbleibenden Aufgaben höhere Genauigkeit. Für den Produktivlauf wurde dennoch **Llama 8B Q5_K_M** gewählt, weil das Multi-Topic-Splitting für die Qualität der Wissensbasis kritischer ist als marginal höhere Section-Genauigkeit und die Inferenzzeit ca. 80 s pro Chunk gegenüber ca. 480 s bei Qwen 32B beträgt – bei 5 411 USEFUL-Tickets entspricht das einer Full-Corpus-Hochrechnung von ca. 5 Tagen (Llama) gegenüber ca. 20 Tagen (Qwen), was den produktiven Service `support-backend.service` fast einen Monat blockieren würde und nicht vertretbar ist.

5.3.2. Produktivlauf über 5 411 USEFUL-Tickets

Der Full-Run über alle 5 411 als USEFUL klassifizierten Tickets wurde mit der Llama-8B-Q5-Konfiguration auf WEBSRV38 abgeschlossen. Die Pipeline lieferte **6 060 strukturierte RAG-Datenunits** aus **5 407 Quell-Tickets** (4 Tickets verfehlten den 16 384-Token-Kontextrahmen und wurden als `errors` protokolliert; *erfolgreich verarbeitet*: 5 407/5 411 =

99,93 %). Die Differenz *Tickets in* vs. *Units out* entsteht durch die Multi-Topic-Zerlegung: 585 Tickets (10,8 %) zerfielen in mehrere thematisch eigenständige Einheiten ($\varnothing \approx 2,1$ Sub-Topics, Maximum: 11). Tabelle 5.29 fasst die Eckdaten zusammen.

Tabelle 5.29.: Produktivlauf der RAG-Datenunit-Extraktion

Kennzahl	Wert
Modell	Llama 3.1 8B Instruct Q5_K_M (5,4 GB GGUF)
Hardware	WEBSRV38, 16-Core CPU, 31 GB RAM, kein GPU
LLM-Runtime	llama-cpp-python 0.3.2, n_ctx=16384, n_threads=16, chat_format="llama-3"
LLM-Aufrufe	3 pro Chunk (Chunker, Extractor, Verifier), alle grammar-constrainer
Input	5 411 USEFUL-Tickets
Output	6 060 RAG-Datenunits aus 5 407 Quell-Tickets
Multi-Topic-Tickets	585 (10,8 %), $\varnothing$ 2,1 Sub-Topics (max. 11)
Errors (Kontext-Overflow)	4 (0,07 %)
Wall-Time	$\approx$ 9,7 Tage
Output-Dateien	rag_units_llama8b_5k.jsonl (7,5 MB), verifier_drops_5k.jsonl (2,5 MB)
Resume-Verhalten	inkrementelles JSONL-Append + flush, kill-9-resistant (load_done_ids())

Quality-Flag-Verteilung

Jede der 6 060 Datenunits trägt ein `quality`-Flag, das den Zustand nach Verifier-Stufe und beiden Safety-Checks (Fix 1, vgl. Abschnitt 3.3.1.11) widerspiegelt. Tabelle 5.30 zeigt die Produktiv-Verteilung.

Tabelle 5.30.: Quality-Flag-Verteilung der 6 060 produzierten Datenunits. Die Spalte *Gewicht* entspricht dem im Retrieval-Score angewandten Multiplikator (Abschnitt 5.3.3).

quality-Flag	Anzahl	Anteil	Gewicht
fixed	4 368	72,1 %	0,9
unverified_too_aggressive	938	15,5 %	0,7
unverified_structure_lost	356	5,9 %	0,7
verified	345	5,7 %	1,0
unverified	28	0,5 %	0,7
fallback	25	0,4 %	0,4
Summe	6 060	100 %	—

Die dominierende Klasse ist *fixed* (72,1 %): der Verifier modifizierte den Extractor-Output, ohne die strukturellen Marker zu zerstören. 21,9 % der Units wurden vom Safety-Check (Fix 1) als *unverified_** markiert, weil der Verifier zu aggressiv eingriff – in diesen Fällen wird der Extractor-Output als Quelle der Wahrheit behalten und das Flag signalisiert dem Retrieval, die Unit niedriger zu gewichten. Echte *verified*-Units (keine Verifier-Korrekturen nötig) machen nur 5,7 % aus, weil der Verifier praktisch immer mindestens einen Satz angreift (vgl. Verifier-Aktivität weiter unten). Die Fallback-Klasse umfasst 25 Tickets unter 30 Zeichen, für die ein deterministischer regex-basierter Mini-Pfad greift.

Strukturerhaltung – die zentrale Qualitätsmetrik

Die für die nachgelagerte Retrieval-Pipeline wichtigste Eigenschaft jeder Unit ist die Erhaltung der drei strukturellen Marker ****Problem:****, ****Investigation:**** und ****Resolution:****. Eine fehlende Resolution-Sektion etwa würde bedeuten, dass eine vermeintliche Wissens-Unit das tatsächlich gefragte Element nicht enthält. Tabelle 5.31 fasst die Erhaltung auf dem Produktiv-Korpus zusammen.

Tabelle 5.31.: Strukturerhaltung im Produktivlauf (6 060 Datenunits)

Strukturelement	Anteil
Problem: vorhanden	99,5 %
Investigation: vorhanden	93,4 %
Resolution: vorhanden	67,9 %
Mindestens Problem + (Investigation <i>oder</i> Resolution)	99,3 %
Vollständig Problem + Investigation + Resolution	62,0 %

Die Headline-Zahl 99,3 % – der Anteil der Units, die mindestens das Problem zusammen mit einem der beiden weiteren Marker tragen – ist das Designziel der Extraktionspipeline. Der *Resolution*-Anteil von 67,9 % ist niedriger, weil ein Teil der *USEFUL*-Tickets zwar ein klar formuliertes Problem und eine investigative Diskussion enthält, aber den Lösungsschritt im Konversationsfluss nicht explizit benennt; der Extractor markiert diese ehrlicherweise nicht als *Resolution*, statt eine zu erfinden.

Fix 1: Quantitative Wirkung des Safety-Checks

Die in Abschnitt 3.3.1.11 beschriebene Safety-Check-Logik („Fix 1“: Verifier-Output wird verworfen, wenn er die strukturellen Marker entfernt oder den Inhalt um mehr als 50 % schrumpft) wurde auf einem frühen, partiellen Lauf vor Einführung des Checks und auf dem finalen Full-Run vermessen. Tabelle 5.32 stellt die Wirkung gegenüber.

Tabelle 5.32.: Quantitative Wirkung von Fix 1 (Safety-Check) auf die Strukturhaltung. Vergleich: 531-Unit Partial-Run vor Einführung des Checks vs. 6 060-Unit Full-Run nach Einführung.

Konfiguration	Strukturhaltung	Korpusgröße
Ohne Fix 1 (Partial-Run, frühe Iteration)	41,5 %	531 Units
Mit Fix 1 (Produktivlauf)	99,3 %	6 060 Units

Der Unterschied ist drastisch: ohne den Safety-Check ließ sich bei rund 58 % der Units die strukturelle Form nicht halten, weil der Verifier ungeprüft alle als hallucinatorisch eingestuft Marker entfernte. Mit dem Safety-Check, der diese Eingriffe rückgängig macht und stattdessen ein `unverified_*`-Flag setzt, steigt die Strukturhaltung auf 99,3 %. Damit ist Fix 1 nicht nur eine kleine Verfeinerung, sondern die entscheidende Eingriff, der die produktive Nutzbarkeit der Pipeline erst herstellt: 21,9 % der finalen Units verdanken ihre Integrität diesem Check.

Verifier-Aktivität

Der Verifier protokolliert jeden Eingriff in einer separaten Datei `verifier_drops_5k.jsonl`. Tabelle 5.33 fasst die kumulierte Aktivität über den Full-Run zusammen.

Tabelle 5.33.: Aktivität des Verifiers über den Full-Run

Kennzahl	Wert
Protokollierte Drop-Records	5 660
Einzelne KEEP/FIX/DROP-Verdicts	13 883
Anteil DROP	93,9 %
Anteil FIX	6,1 %
Units mit mindestens einem Drop-Event	93 %

Diese Verteilung erklärt, warum echte `verified`-Units selten sind (5,7 %): der Verifier greift bei praktisch jeder Unit ein. Ohne den Safety-Check (Fix 1) wäre die hohe DROP-Quote unmittelbar in einen Verlust der Strukturhaltung umgeschlagen; mit dem Check wird sie zu einer feinkörnigen Quality-Annotation, die das Retrieval später nutzbar macht.

5.3.3. Quality-Weighted Retrieval – Übergang in die Suche

Die in Tabelle 5.30 dokumentierten Quality-Flags sind nicht nur ein passives Audit-Artefakt, sondern fließen als Multiplikator direkt in den finalen Retrieval-Score der nachgelagerten Suchpipeline (Abschnitt 5.4) ein. Die Abbildung der Flags auf Gewichte folgt der Designregel „nichts wegwerfen, nur deprioritisieren“: Units mit niedrigerer Verifier-Konfidenz erschei-

nen weiterhin im Index und werden weiterhin von Bi-Encoder und Cross-Encoder bewertet, ihr Endsore wird jedoch entsprechend reduziert.

Konkret gilt:

$$\text{score}_{\text{final}} = \text{score}_{\text{rerank}} \times w_q,$$

wobei w_q der in Tabelle 5.30 ausgewiesene Quality-Gewicht ist. Eine *verified*-Unit mit Rerank-Score 0,8 erhält $\text{score}_{\text{final}} = 0,8 \times 1,0 = 0,8$, eine *fallback*-Unit mit demselben Rerank-Score nur $0,8 \times 0,4 = 0,32$. Damit eine *fallback*-Unit eine *fixed*-Unit überholt, muss ihre Rerank-Roh-Punktzahl mindestens 2,25-fach höher ausfallen.

Zusätzlich zur Quality-Gewichtung unterstützt die Such-API mehrere optionale Metadaten-Filter, die vor dem Reranking angewandt werden und den Cross-Encoder-Aufwand auf plausible Kandidaten konzentrieren:

- `section` – Beschränkung auf eine der 12 Bucket-Sektionen (z. B. UI, Lizenz, Performance).
- `language` – Filterung auf German oder English.
- `product` – Filterung auf eines der zehn TKI-Produkte (NET Design, NET Engineering, ...).
- `min_quality_weight` – Mindest-Quality-Gewicht (z. B. $\geq 0,9$ für eine konservativere Suche).
- `quality_in` – explizite Quality-Flag-Mengenfilterung.
- `ticket_id` – Einschränkung auf einen einzelnen Zendesk-Ticket (Debug-Zweck).

Zur Erzeugung der Vektor-Repräsentation wurde der Input für den BGE-M3-Embedder ebenfalls strukturiert vorbereitet: ein Header-Block mit Title, Section, Product, Keywords und Entities (jeweils auf einer eigenen Zeile), ein Trenner `--`, und anschließend der `**Problem:** / **Investigation:** / **Resolution:**`-Block des `content`-Felds. Diese Struktur entspricht dem Feld `embedding_text` in der `units.parquet`-Datei und ist für den Embedder so vorgesehen, dass die Header-Tokens zusätzliches semantisches Gewicht für Domänenbegriffe (Produktname, Sektion, Entitäten) liefern, während der `content`-Block den eigentlichen Wissensgehalt trägt.

5.4. Qualität der semantischen Suche

Die finale Wissensbasis aus 6 060 RAG-Datenunits wurde in einen FAISS-Index überführt und über eine zweistufige Retrieval-Pipeline (Bi-Encoder-Suche → Cross-Encoder-

Reranking) angefragt. Die Auswertung dieses Abschnitts deckt zwei Aspekte ab: die *funktionale Korrektheit* des Indexes (Smoke-Test mit kanonischen Anfragen) und die *Antwortlatenz* der einzelnen Stufen unter Produktivkonfiguration.

5.4.1. Smoke-Test mit kanonischen Anfragen

Aus den drei thematisch relevantesten Domänen des Korpus – Layer- und Konvertierungsprobleme, Lizenz- und Account-Themen, Export- und Reporting-Endpoints – wurden sieben kanonische Anfragen formuliert (vollständig dokumentiert in `stage3_rag/examples/sample_queries.md`). Für jede Anfrage ist die erwartete Top-1-Unit aus dem Korpus a priori bekannt; die Pipeline wurde so konfiguriert wie im produktiven Backend (FAISS-IndexFlatIP, BGE-M3-Embedder, BGE-Reranker-v2-M3, Quality-Weighted-Scoring).

Auf allen sieben Anfragen liefert das System den erwarteten Treffer auf Rang 1 (7/7 = 100 % Top-1-Treffer). Die Smoke-Tests sind kein Ersatz für eine systematische $\text{Recall}@k$ -Evaluation – sie demonstrieren ausschließlich, dass der Index funktional korrekt aufgebaut ist und das Reranking die thematisch passende Unit selektiert. Eine umfassende Recall-Bewertung mit größerem Query-Set würde eine zusätzliche annotierte Query-Datensatz erfordern, die im Rahmen dieser Arbeit nicht erstellt werden konnte (vgl. Ausblick in Kapitel 6).

5.4.2. Index-Eigenschaften und Latenz

Tabelle 5.34 fasst die statischen Eigenschaften des Produktiv-Indexes zusammen.

Tabelle 5.34.: Eigenschaften des Zendesk-Stage-3-FAISS-Indexes (vor dem Merge mit den übrigen drei Quellen, vgl. Abschnitt 5.5.4).

Eigenschaft	Wert
Anzahl Vektoren	6 060
Dimension	1 024 (BGE-M3)
Datentyp	float32, ℓ_2 -normalisiert
Index-Typ	IndexFlatIP (exakter Inner-Product, $\equiv$ Kosinus)
Indexdatei (<code>units_faiss.index</code>)	ca. 24 MB
Vektor-Datei (<code>units_vectors.npy</code>)	ca. 24 MB
ID-Datei (<code>units_ids.npy</code>)	ca. 500 KB
Erzeugungszeit (Mac M3 MPS)	ca. 8 min für den gesamten Korpus

Tabelle 5.35 dokumentiert die per-Stufen-Latenz einer Suchanfrage auf der lokalen Entwicklungskonfiguration (Apple M3, Metal-beschleunigter Embedder und Cross-Encoder; IndexFlatIP läuft CPU-only).

Tabelle 5.35.: Latenz der einzelnen Retrieval-Stufen pro Anfrage (Median über 7 Smoke-Test-Anfragen, lokale Mac-M3-Konfiguration).

Stufe	Latenz
BGE-M3 Query-Encoding	≈ 50 ms
FAISS Top-100-Retrieval	< 1 ms
Cross-Encoder-Reranking (25 Kandidaten, frühe Iteration; produktiv 50)	≈ 200 ms
Summe pro Anfrage	≈ 250 ms

Die zweistufige Architektur ist damit auf der Retrieval-Seite deutlich unter einer Sekunde – die dominierende Latenz im Gesamtsystem entsteht in der nachfolgenden LLM-Generierung (Abschnitt 5.6). Der Cross-Encoder erhält nur die 25 vom Bi-Encoder vorgewählten Kandidaten – die übliche Two-Stage-Retrieval-Strategie aus Abschnitt 1.5 –, wodurch die quadratische Kostenkomponente des Cross-Encoders auf einen konstanten Budget-Anteil reduziert wird; ohne diese Pre-Filterung wären die 6 060 Units mit dem Cross-Encoder einzeln zu bewerten gewesen.

5.4.3. Qualitative Beobachtungen aus den Smoke-Tests

Drei qualitative Beobachtungen aus den sieben Smoke-Test-Anfragen:

Erstens, die mehrsprachige Embedding-Strategie funktioniert. Anfragen in Deutsch trafen problemlos auf Units, deren Konversationsverlauf primär englisch verlief (und umgekehrt), weil BGE-M3 als multilinguales Modell die semantische Brücke schlägt. Eine im späteren Verlauf der Arbeit auf dem Merged-Index (vgl. Abschnitt 5.5.4) wiederholte Anfrage „Wie exportiere ich Buchungspositionen?“ fand korrekt die Unit zum Vitruvi-Endpoint `POST /api/v1/reporting/accounts_payable_items_export`.

Zweitens, das Reranking ändert die Reihenfolge der Top-Treffer regelmäßig substantiell. Der Bi-Encoder produziert für viele Anfragen eine plausible Kandidatenmenge, in der jedoch nicht immer die thematisch passende Unit auf Rang 1 steht. Auf den sieben Smoke-Tests verschob der Cross-Encoder die korrekte Unit in 4 von 7 Fällen erst aus den unteren Rängen auf Rang 1 – ein Hinweis, dass die zweistufige Architektur den erwarteten Mehrwert liefert.

Drittens, das Quality-Weighted-Scoring (Abschnitt 5.3.3) wirkt erwartungsgemäß: fallback-Units (Gewicht 0,4) tauchten in den Top-5 bei keiner der sieben Anfragen auf, obwohl ihre Roh-Embeddings vereinzelt unter den Top-20 des Bi-Encoders waren. Die Multiplikation mit dem niedrigen Quality-Gewicht stuft sie zuverlässig unter die strukturell vollständigen Units.

5.5. Qualität der generierten Antworten

Die in Abschnitt 5.1 vorgestellten RAGAS-Metriken werden in diesem Abschnitt auf die produktive Endkonfiguration des RAG-Systems angewendet (L3 = Llama 3.1 8B Instruct Q5_K_M, BGE-M3 als Bi-Encoder, BGE-Reranker-v2-M3 mit MMR-Diversifizierung und Rerank-Score-Schwellenwert 0,40, Merged-FAISS-Index mit 14 118 Vektoren über die vier integrierten Quellcluster Zendesk-Tickets, Help Center, Vitruvi und TKI-Produkte-KB). Die Evaluation erfolgt vollständig lokal über die in `react_app/eval/` hinterlegte Harness; Produktionscode aus `react_app/backend/src/` wird ausschließlich importiert und nicht modifiziert (vgl. Listing A.7 im Anhang).

5.5.1. Test-Set-Konstruktion

Da der in Abschnitt 5.2.1 beschriebene `zendesk_test_set/` ausschließlich der Bewertung der Qualitätsfilterung dient und keine Frage-Antwort-Paare mit Ground Truth für die RAG-Endstrecke enthält, wurde für die End-zu-End-Evaluation des RAG-Systems ein separates Test-Set konstruiert (`react_app/eval/test_set.json`). Er besteht aus 20 hand-erstellten Frage-Antwort-Paaren, die so gewählt sind, dass jede der vier integrierten Quellen mit fünf Fragen vertreten ist, und die in Sprache (DE/EN) und Fragetyp (definitorisch, prozedural, vergleichend, scope-bezogen) ausgewogen ist. Tabelle 5.36 fasst die Verteilung zusammen.

Tabelle 5.36.: Balance des RAG-Test-Sets ($n = 20$).

Dimension	Ausprägung	Anzahl
Quelle	Zendesk-Tickets / Help Center / Vitruvi / TKI-Pr-KB	5 / 5 / 5 / 5
Sprache	Deutsch / Englisch	13 / 7
Fragetyp	prozedural	10
	definitorisch	5
	vergleichend	3
	scope-bezogen	2

Die Konstruktion folgte einem dreistufigen Vorgehen: (i) Inventur und Charakterisierung der Verteilung über alle vier Quellen direkt auf den Rohdaten (`units.parquet`, `chunks_articles.parquet`, `chunks_pdfs.parquet`, `chunks_openapi.parquet`, `chunks_tos.parquet` sowie die acht NET-Modul-Markdown-Dateien); (ii) stratifiziertes Sampling von je fünf Chunks pro Quelle, balanciert über die jeweiligen Sub-Typen (z. B. im Help Center vier Articles und ein PDF); (iii) manuelles Verfassen von Fragen im Support-Nutzer-Stil mit jeweils einer als Ground Truth dienenden Referenzantwort von 2–4 Sätzen, vollständig aus dem zugehörigen Roh-Chunk ableitbar. Die Methodik und die Limitationen des Sets

sind in `react_app/eval/test_set_manifest.md` dokumentiert; fünf repräsentative Beispiele aus dem Test-Set sind in Anhang A.8 aufgeführt.

Die gewählte Größe $n = 20$ ist ein bewusster Kompromiss: RAGAS empfiehlt $n \geq 50$ für robust verteilte Aggregate, jedoch erfordert jede Frage im hier verwendeten Setup eine vollständige End-zu-End-Inferenz inklusive Modell-Lade-Overhead der lokalen Evaluations-Harness (~ 47 s im Mittel; die produktive Server-Latenz im Warm-Betrieb liegt erheblich darunter, vgl. Tabelle 5.39) sowie vier zusätzliche Bewertungsaufrufe des lokalen Judges (~ 40 s je Aufruf). Eine Verdopplung auf 40 oder Verdreifachung auf 60 Fragen hätte die Gesamtlaufzeit über die für die Iterationsphase verfügbare Zeit hinausgeschoben, ohne den qualitativen Befund grundlegend zu verändern. Die resultierende Stichprobengröße wird in Abschnitt 5.5.7 als formale Limitation aufgenommen.

5.5.2. Judge-Konfiguration

Da das Erfordernis eines vollständig on-premise betriebenen Systems (vgl. Kapitel 3) auch für die Evaluationsschicht gilt, wurde auf API-basierte Judges (z. B. GPT-4o-mini) verzichtet. Als Judge-LLM dient *Llama 3.1 8B Instruct Q4_K_M* – dieselbe Modellfamilie wie die produktive L3, aber eine andere Quantisierung. Diese Trennung der Gewichte (Q4 gegen Q5) vermeidet den methodisch problematischen Fall identischer Judge- und Generator-Parameter, ohne die On-Premise-Anforderung zu verletzen. Die Embeddings für *Context Precision* und *Answer Relevance* liefert BGE-M3 in derselben Konfiguration wie die produktive Retrieval-Stufe – damit liegen die kontextbezogenen Metriken auf derselben semantischen Mannigfaltigkeit wie der Index.

Eine erste Iteration verwendete Qwen2-1.5B-Instruct als Judge, da das Modell deutlich kleiner und schneller ist. Es zeigte sich jedoch, dass die kleine Variante das von RAGAS verlangte strikte JSON-Ausgabeschema von Pydantic-Output-Parsern nicht zuverlässig einhält: Auf den `nli_statements`- und `context_recall_classification`-Prompts lieferte Qwen2 wiederholt freitextuelle Antworten ohne valide JSON-Hülle, was zum Abbruch der jeweiligen Metrik führte. Der Wechsel zu Llama 3.1 8B löste dieses Problem, weil das größere Modell der Instruktion „respond strictly in the requested format“ signifikant besser folgt; vereinzelte Parse-Fehler treten dennoch auf und werden in Abschnitt 5.5.7 dokumentiert. Die vollständige Adapter-Implementierung mit dem Llama-3.1-Chat-Template und der expliziten JSON-Disziplin ist in Listing A.8 im Anhang abgedruckt.

Aus technischer Sicht ist zu erwähnen, dass das Judge-Backend (`llama-cpp-python`) ein Singleton-Modell pro Prozess hält. Eine parallele Auswertung von Fragen über mehrere RAGAS-Worker führt zu Wettlauf-Bedingungen am gemeinsamen KV-Cache, was sich in den ersten Versuchen in stummen Ausfällen einzelner Bewertungsaufrufe äußerte. Die Auswertung erfolgt daher seriell (`RunConfig(max_workers=1)`). Das Kontextfenster des

Judges wurde auf 16 384 Tokens vergrößert, da *context_recall* und *context_precision* interne Prompts mit der vollständigen Liste der retrieveden Chunks zusammenstellen und mit dem RAGAS-Standardwert von 4 096 Tokens in mehreren Fällen die Kontextgrenze sprengen.

5.5.3. Interpretationsschwellen

Die RAGAS-Metriken liefern Werte im Intervall $[0; 1]$, deren Aussagekraft erst durch eine Einordnung in praxistaugliche Schwellenwerte konkret wird. Tabelle 5.37 schlägt eine Klassifikation in vier Bereiche vor, die sich an der RAGAS-Dokumentation und an in der RAG-Literatur etablierten Konventionen orientiert.

Tabelle 5.37.: Vorgeschlagene Interpretationsschwellen für RAGAS-Metriken.

Wertebereich	Klassifikation	Praktische Bedeutung
$\geq 0,85$	sehr gut	Produktiv einsetzbar ohne Einschränkung.
$\geq 0,70$	gut	Produktiv einsetzbar, gelegentliche Stichprobenprüfung sinnvoll.
$\geq 0,50$	akzeptabel	Brauchbar als Assistenz, jedoch nicht ohne menschliche Validierung.
$< 0,50$	ungenügend	Nicht freigabefähig; Datenquelle oder Pipeline überarbeiten.

Die Schwellen sind absichtlich konservativ gewählt: Im Support-Kontext, in dem fehlerhafte Antworten als verbindliche Auskunft an Endkunden weitergegeben werden könnten, kann eine niedrige Faithfulness oder Context Recall reale Folgekosten verursachen. Die Einstufung dient sowohl der späteren Diskussion einzelner Quellen (Abschnitt 5.5.5) als auch der Bewertung der Zielerreichung (FA-2, FA-5) in Abschnitt 5.5.6.

5.5.4. RAGAS-Ergebnistabelle für die finale Konfiguration

Tabelle 5.38 fasst die vier RAGAS-Kernmetriken für die produktive Endkonfiguration zusammen, sowohl als Gesamt-Aggregat als auch aufgeschlüsselt nach den vier Quellen des Merged-Index. Die Werte werden als Mittelwert $\pm$ Stichprobenstandardabweichung berichtet; die effektive Stichprobe n pro Metrik ist im Zusammenhang mit Parser-Fehlern in Abschnitt 5.5.7 erläutert.

Im Aggregat erreicht das System auf der *Answer-Relevance*- und *Context-Precision*-Metrik den Bereich „sehr gut“, auf *Faithfulness* und *Context Recall* den Bereich „gut“ (vgl. Tabelle 5.37). Die hohe Context Precision bei gleichzeitig hoher Answer Relevance zeigt,

Tabelle 5.38.: RAGAS-Scores der produktiven Systemkonfiguration (L3 = Llama 3.1 8B Q5_K_M, Reranker BGE-Reranker-v2-M3, Schwellenwert 0,40). Test-Set $n = 20$, Judge: Llama 3.1 8B Q4_K_M (lokal).

Quelle (n)	Faithfulness	Answer Rel.	Ctx. Precision	Ctx. Recall
Gesamt (20)	$0,816 \pm 0,198$	$0,891 \pm 0,064$	$0,957 \pm 0,078$	$0,835 \pm 0,216$
Zendesk-Tickets (5)	$0,750 \pm 0,250$	$0,859 \pm 0,066$	$0,945 \pm 0,098$	$0,675 \pm 0,192$
Help Center (5)	$0,929 \pm 0,124$	$0,912 \pm 0,043$	$0,954 \pm 0,092$	$0,917 \pm 0,144$
Vitruvi (API+ToS) (5)	$0,659 \pm 0,165$	$0,905 \pm 0,084$	$0,937 \pm 0,066$	$0,875 \pm 0,216$
TKI-Produkte-KB (5)	$0,871 \pm 0,124$	$0,887 \pm 0,035$	$0,993 \pm 0,013$	$0,875 \pm 0,216$

dass das zweistufige Retrieval mit Schwellenwert-Filter dem Drafter konsequent fragebezogene Kontexte liefert und dieser daraus relevante Antworten formuliert. Die etwas niedrigere Faithfulness deutet darauf hin, dass das Modell zwar eng am Kontext bleibt, vereinzelt aber Verbindungs- und Erklärungssätze hinzufügt, die vom Judge nicht in einem Chunk verankert werden können. Beide Befunde werden in den nachfolgenden Abschnitten quellbezogen weiter differenziert.

5.5.5. Quellbezogene Diskussion

Die Aufschlüsselung in Tabelle 5.38 offenbart deutliche Qualitätsunterschiede zwischen den vier integrierten Datenquellen.

Help Center – stärkste Quelle. Mit Mittelwerten von $\geq 0,91$ über alle vier Metriken liegt das Help Center klar über dem Gesamtdurchschnitt. Erklärbar ist dies durch die kuratierten Artikel-Chunks mit semantisch konsistenten Breadcrumb-Headings („NET Cloud > Webportal > Projektmanagement“), die der Drafter zuverlässig als Zitations-Anker übernimmt. Auch eingebundene PDF-Inhalte (z. B. Stonex-S580-Handbuch) liefern hier strukturierte Antworten.

TKI-Produkte-KB – höchste Context Precision. Mit 0,993 erreicht diese Quelle praktisch die Obergrenze: Die acht NET-Modul-Markdown-Dokumente sind strikt hierarchisch (Kurzprofil, Funktionsumfang, Abgrenzung), wodurch der Reranker fragenah passende Abschnitte mit minimalem Rauschen identifiziert. Die geringfügig niedrigere Faithfulness (0,871) deutet darauf hin, dass das LLM bei vergleichenden Fragen (T3, T5) zusätzliche Markenkontext-Sätze einfügt, die zwar zutreffend, aber nicht 1:1 belegbar sind.

Vitruvi – Faithfulness als Schwachpunkt. Mit 0,659 liegt Vitruvi am untersten Rand des Bereichs „akzeptabel“. Strukturelle Ursache: OpenAPI-Endpoint-Chunks bestehen aus

knappen Methoden-Header-Blöcken, Parameter-Tabellen und Response-Codes; eine flüssige Antwort in natürlicher Sprache erfordert verbindende Erklärungen, die der Judge als nicht im Kontext verankert wertet. Die hohe Context Precision (0,937) und der Context Recall (0,875) zeigen jedoch, dass das Retrieval die richtigen Endpoints findet – das Problem ist generationsseitig.

Zendesk-Tickets – niedrigster Context Recall. Mit 0,675 unterschreitet diese Quelle als einzige den „gut“-Schwellenwert. Ticket-Resolutions sind oft auf mehrere Chunks verteilt (Problem-, Investigation- und Resolution-Abschnitte separater Tickets); der Reranker holt einen passenden Chunk, lässt aber gelegentlich komplementäre Hintergrunddetails im Index zurück. Dies ist ein klarer Hinweis, dass eine ticketzentrierte Chunk-Aggregation oder eine Erhöhung von `rerank_top_k` für Ticket-typische Queries die Recall-Lücke schließen könnte; eine solche Anpassung wird im Limitationen-Abschnitt 5.9.5 als operative Einschränkung aufgegriffen.

5.5.6. Bezug zu FA-2 und FA-5

Die RAGAS-Ergebnisse erlauben erstmals eine quantitative Bewertung der zwei funktionalen Anforderungen, die am stärksten von der Antwortgenerierung abhängen.

FA-2 (Antwortgenerierung). Die Anforderung wird durch das Zusammenspiel von Answer Relevance und Faithfulness adressiert. Ein Gesamtwert von 0,891 auf Answer Relevance belegt, dass die generierten Antworten konsequent auf die gestellte Frage eingehen und nicht in tangential Beschreibungen abdriften. Ein Faithfulness-Wert von 0,816 – mit ausgeprägter Quellenheterogenität (vgl. Abschnitt 5.5.5) – zeigt, dass die Anweisung im System-Prompt, ausschließlich auf Basis der retrieveden Dokumente zu antworten, mehrheitlich befolgt wird, gleichzeitig aber bei knapp strukturierten Quellen wie OpenAPI-Endpoints Spielraum für Verfeinerung besteht. FA-2 ist damit auf der Aggregatebene als erfüllt zu betrachten, mit dem expliziten Vorbehalt der quellspezifischen Faithfulness-Reduktion bei Vitruvi.

FA-5 (Konfidenzbewertung). Die Standardabweichung der Faithfulness von 0,198 über das gesamte Test-Set rechtfertigt empirisch die in `config.py` (DraftConfidenceConfig) hinterlegte Differenzierung zwischen `low_conf_threshold = 0,40` und `hard_reject_threshold = 0,25`: ein einziger Konfidenzwert ohne abgestufte Reaktion würde die ausgeprägten Quellunterschiede nicht adäquat abbilden. Der vom Drafter selbst gelieferte <CONFIDENCE>-Wert ist Teil der vier Signale des Produktiv-Scores (Gewicht 0,35); die korrelative Untermauerung durch eine externe Faithfulness-Messung –

wie sie diese Evaluation liefert – stützt seine Verwendung als Komponente der 4-Signal-Konfidenz. FA-5 ist damit ebenfalls als erfüllt zu betrachten.

5.5.7. Limitationen der RAGAS-Evaluation

Die Aussagekraft der berichteten Werte ist durch mehrere methodische Faktoren begrenzt, die hier explizit gemacht werden und im strukturierten Limitationen-Abschnitt 5.9 in die übergreifende Validitäts-Diskussion eingeordnet werden.

Parse-Ausfälle des lokalen Judges. Trotz der bewussten Wahl des größeren Llama-3.1-8B-Judges blieben vier von 20 Bewertungen auf der Metrik Faithfulness sowie vier von 20 auf Context Recall ohne valides JSON-Output und gehen als NaN weder in Mittelwert noch Standardabweichung ein. Die effektive Stichprobe für diese beiden Metriken beträgt $n = 16$. Eine post-hoc-Analyse der ausgefallenen Frage-IDs zeigt, dass die Parser-Fehler auf alle vier Quellen, beide Sprachen und drei der vier Fragetypen verteilt sind und keinem inhaltlichen Muster folgen; die berichteten Mittelwerte bleiben damit für das Test-Set repräsentativ. Antwortrelevanz und Context Precision basieren auf der vollen Stichprobe ($n = 20$).

Weitere Limitationen. Die übrigen für die Aussagekraft dieser Evaluation relevanten Einschränkungen – die geringe Stichprobengröße $n = 20$, der Einzelannotator-Vorbehalt sowie die auf vier Quellcluster begrenzte Abdeckung des Merged-Index – gelten gleichermaßen für andere Teile der Arbeit und werden zur Vermeidung von Wiederholungen im strukturierten Limitationen-Abschnitt 5.9 gebündelt aufgegriffen (insbesondere statistische Auflösung in Abschnitt 5.9.4, IAA-Mangel in 5.9.1 und Quellabdeckung in 5.9.2).

5.6. Systemperformance

Die Latenzzeiten der einzelnen Pipeline-Stufen wurden über mehrere Durchläufe gemessen:

Die Gesamtlatenz wird maßgeblich von der LLM-Inferenz bestimmt, die je nach Antwortlänge und Komplexität variiert. Durch das SSE-Streaming wird die wahrgenommene Antwortzeit deutlich reduziert, da der Benutzer bereits nach Abschluss der Suche (ca. 300–700 ms) erste Ergebnisse sieht und die Token des Antwortentwurfs in Echtzeit erscheinen.

Mittelwert vs. p95 – Begründung der gewählten Berichtsgröße. Für eine NFA-2-Konformitätsaussage liefert in der Regel das 95. Perzentil der End-zu-End-Latenz eine

Tabelle 5.39.: Durchschnittliche Latenzzeiten der Pipeline-Stufen

Stufe	Durchschnitt	Bereich
Semantische Suche (FAISS)	100–200 ms	50–500 ms
Reranking (Cross-Encoder)	200–500 ms	100–1000 ms
KB-Laden	<50 ms	<100 ms
Draft-Generierung (LLM)	2–10 s	1–15 s
Gesamt-Pipeline	2,5–10,5 s	1,5–16 s

schärfere Aussage als der Mittelwert, da Ausreißer mit großem Kontextfenster oder langer Antwortgenerierung in einem Mittelwert untergehen können. Im vorliegenden Anwendungsfall berichten wir bewusst den Mittelwert ergänzt durch den Bereich, weil drei Eigenschaften des Systems die p95-Aussagekraft relativieren. *Erstens* ist die UX-relevante Größe nicht die Gesamtdauer bis zum letzten generierten Token, sondern die *Time-to-First-Token*: Über SSE-Streaming sehen Support-Mitarbeiter den Antwortbeginn typischerweise nach 300–700 ms (Sichtbarkeit der Quellen nach der Reranking-Phase) und das erste Antwort-Token binnen weniger Sekunden danach. Eine längere Draft-Phase verschlechtert nicht das wahrgenommene Antwortverhalten, weil der Entwurf bereits inkrementell sichtbar ist. *Zweitens* dient der Hard-Reject-Mechanismus (`hard_reject_threshold`, vgl. Abschnitt 5.5.6) als implizite Latenz-Begrenzung an der oberen Schranke: Bei sehr großen Retrieval-Treffer-Listen wird die Pipeline abgebrochen, bevor die Draft-Phase beginnt, so dass extreme Ausreißer auf der Total-Latency-Achse strukturell selten sind. *Drittens* zeigt der bereits in Tabelle 5.39 ausgewiesene Bereich von 1,5–16 s den wesentlichen Streubefund – die obere Grenze des beobachteten Bereichs entspricht in dieser Stichprobe einer Größenordnung, die einem klassischen p95 bei interaktiver Nutzung näherkommt als ein theoretisches Tail-Perzentil aus seltenen Worst-Case-Kontexten. Die für NFA-2 relevante Aussage „für die Mehrzahl der Benutzeranfragen wird die 15-Sekunden-Vorgabe eingehalten“ wird damit durch Mittelwert plus Bereich angemessen abgebildet; eine formale Tail-Perzentil-Studie auf einer eigenen Last-Test-Stichprobe wäre erst für eine produktive Deployment-Validierung mit höherem Durchsatz und definierten SLOs erforderlich und ist in Abschnitt 5.9.5 als operative Erweiterungsrichtung aufgenommen.

Der Speicherverbrauch im Betrieb liegt bei ca. 10–12 GB, verteilt auf:

- FAISS-Index und Embedding-Modell (BGE-M3): ca. 2–3 GB
- Cross-Encoder (BGE-Reranker-v2-M3): ca. 1 GB
- LLM (Llama 3.1 8B Q5_K_M): ca. 5,7 GB
- Betriebssystem und Framework: ca. 2 GB

5.7. Vergleich verschiedener Konfigurationen

Das System unterstützt den Wechsel zwischen verschiedenen GGUF-Modellen zur Laufzeit. Im Rahmen der Evaluation wurden verschiedene Konfigurationen verglichen:

Einfluss der Temperatur: Bei niedriger Temperatur ($T = 0.1$) erzeugt das Modell konservativere, aber konsistentere Antworten. Bei höherer Temperatur ($T = 0.7$) sind die Antworten kreativer, aber weniger zuverlässig. Der Standardwert von $T = 0.3$ bietet einen guten Kompromiss für den Support-Kontext.

Einfluss der Rerank-Top-K: Eine Erhöhung der Rerank-Top-K-Werte von 5 auf 10 verbessert die Context Coverage, da dem LLM mehr potentiell relevante Dokumente zur Verfügung stehen. Ab Top-K = 15 nimmt die Verbesserung ab, während die Prompt-Länge zunimmt.

Presets: Die drei vordefinierten Presets zeigen unterschiedliche Kompromisse:

- **Fast:** Niedrigere Top-K-Werte, kürzere Antworten – geeignet für schnelle, einfache Anfragen.
- **Balanced:** Standardkonfiguration – guter Kompromiss zwischen Qualität und Geschwindigkeit.
- **Deep:** Höhere Top-K-Werte, längere Antworten – geeignet für komplexe technische Probleme.

5.8. Diskussion

5.8.1. Stärken des Systems

- Die **Multi-Source-Integration** ermöglicht eine umfassende Wissensabdeckung. Insbesondere die Kombination von Zendesk-Tickets (historische Lösungen), HC-Artikeln (offizielle Dokumentation) und der TKI-Pr-KB (Modul-Funktionsbeschreibungen) liefert komplementäre Informationen.
- Das **zweistufige Retrieval** verbessert die Präzision erheblich gegenüber einem einstufigen Ansatz, insbesondere bei mehrdeutigen Anfragen.
- Die **Konfidenzbewertung** hilft den Support-Mitarbeitern, die Verlässlichkeit der generierten Antworten einzuschätzen, und verhindert durch den Hard-Reject-Mechanismus die Anzeige unzuverlässiger Entwürfe.
- Der **lokale LLM-Betrieb** gewährleistet Datenschutz ohne Abhängigkeit von externen APIs.

- Das **Streaming** verbessert die User Experience durch progressive Darstellung der Ergebnisse.

5.9. Limitationen und Threats to Validity

Über die in den vorausgehenden Abschnitten am jeweiligen Ort dokumentierten Einzeleinschränkungen hinaus fasst dieser Abschnitt die methodischen Grenzen der Arbeit gebündelt zusammen. Die Gliederung orientiert sich an den vier in der empirischen Forschung etablierten Kategorien von *Threats to Validity* – interner, externer, Konstrukt- und statistischer Konklusionsvalidität – und schließt mit einem fünften Block zu *operativen* Einschränkungen, die das System in der praktischen Anwendung betreffen, ohne unmittelbar die Geltung der berichteten Befunde zu gefährden. Die einzelnen Grenzen sind, wo möglich, mit der bereits im Text genannten Mitigation versehen; wo eine Mitigation strukturell nicht möglich war, wird dies offen benannt.

5.9.1. Interne Validität

Einzelannotator und fehlendes Inter-Annotator-Agreement. Sowohl der Qualitätsfilter-Datensatz (Golden 45 und Set 200, Abschnitt 5.2.1) als auch das RAGAS-Test-Set (Abschnitt 5.5.1) wurden vollständig vom Verfasser allein annotiert bzw. konstruiert. Eine Inter-Annotator-Agreement-Bewertung wie Cohen's κ (vgl. Abschnitt 3.3.1.10 der Grundlagen) ist damit strukturell nicht möglich. Diese Einschränkung lässt sich nachträglich nicht beheben; sie wird durch eine externe Validierung mit sieben kommerziellen Frontier-LLMs auf Golden 45 (Abschnitt 5.2.12) abgemildert, die mit 93,3 % Mehrheits-AI-Übereinstimmung gegenüber der Annotation des Verfassers die Stabilität der Labels indiziert, ohne die strukturelle Lücke vollständig zu schließen.

Kalibrierungs- und Testdatensatz teilen sich Tickets (Golden 45 $\subset$ Set 200). Da Golden 45 als echte Teilmenge in Set 200 enthalten ist und Prompts sowie Confidence-Overrides auf Golden 45 kalibriert wurden, sind die auf den vollen 200 Tickets berichteten Werte zugunsten der dort enthaltenen 45 Kalibrierungstickets verzerrt (*Data Leakage*). Mitigation: Die separate Auswertung der 155 Out-of-Sample-Tickets in Abschnitt 5.2.13 quantifiziert die Verzerrung und ermöglicht eine unverzerrte Generalisierungsaussage.

Konfundierung zweier Konfigurationsänderungen Q4 SC $\rightarrow$ Q5 Single-Shot. Beim Wechsel von Llama 3.1 8B Q4_K_M mit Self-Consistency ($N = 3$) zu Q5_K_M im Single-Shot-Modus wurden gleichzeitig zwei Parameter verändert: die Quantisierungsstufe und das Inferenzverfahren. Eine sauber isolierte Wirkungszuschreibung ist daher allein aus der

Accuracy-Differenz nicht möglich. Mitigation: Der McNemar-Test in Abschnitt 5.2.13 prüft die Quantisierungsentscheidung gegen die Nullhypothese statistischer Gleichheit und ergibt $p = 1,00$; die Wahl zugunsten von Q5 wird in Abschnitt 5.2.4 explizit nicht über eine signifikante Accuracy-Verbesserung, sondern über Latenz- und Wartbarkeitsvorteile begründet.

Judge und Generator aus derselben Modellfamilie (RAGAS). Der lokale RAGAS-Judge ist Llama 3.1 8B Q4_K_M, das produktive Generator-Modell Llama 3.1 8B Q5_K_M. Die beiden teilen Architektur und Trainingsdaten, was eine systematische Überschätzung der Faithfulness durch „Familien-Bias“ nicht vollständig ausschließt. Mitigation: Der Wechsel der Quantisierungsstufe (Abschnitt 5.5.2) entkoppelt zumindest die konkreten Modellgewichte; ein API-basierter Frontier-Judge wurde aus den in Abschnitt 5.5.2 genannten On-Premise-Gründen verworfen.

5.9.2. Externe Validität

TKI-spezifische Bucket-Definitionen und Override-Logik. Die im Pipeline-Schema verwendeten Bucket-Definitionen für `subject_kind` und `outcome` sowie die Confidence-Overrides (P1–P5) sind auf die im TKI-Support beobachteten Ticket-Phänomene zugeschnitten. Eine Übertragung auf andere Domänen (z. B. allgemeines SaaS-Support, Healthcare, E-Commerce) würde mindestens eine Neuannotation eines domänenspezifischen Golden-Sets und eine Neukalibrierung der Override-Regeln erfordern. Die in dieser Arbeit berichteten Genauigkeitswerte gelten ausdrücklich für die TKI-Domäne.

Sprach-Mix. Die annotierten Datensätze umfassen einen Mix aus deutschen und englischen Tickets (Set 200: überwiegend Deutsch; RAGAS-Test-Set: 13 DE / 7 EN). Eine reine englischsprachige Support-Pipeline wurde nicht evaluiert; bei stark abweichender Sprachverteilung können Embedding-Recall und Drafter-Faithfulness von den hier berichteten Werten abweichen. BGE-M3 ist multilingual trainiert und mildert dieses Risiko teilweise.

Quellabdeckung des produktiven Merged-Index. Der produktive FAISS-Index integriert vier logische Quellcluster (Zendesk-Tickets, Help Center, Vitruvi und TKI-Produkte-KB) mit insgesamt 14 118 Vektoren. Die im Konzeptkapitel diskutierten weiteren Quellen Confluence, Jira und Slack sowie separat aufgeführte PDF-Dokumentationen sind im finalen Index nicht enthalten und werden daher in der RAGAS-Evaluation nicht abgedeckt (vgl. Abschnitt 5.5.7). Die in Abschnitt 5.5.4 berichteten Aggregate gelten somit für die produktive Endkonfiguration, nicht für die konzeptionelle Vollzahl der Datenquellen.

Hardware-Bindung der Latenz- und Speichermessungen. Sämtliche Latenz- und Speicherangaben (Abschnitt 5.6) wurden auf einem Apple M3 mit Metal-Beschleunigung und einem CPU-basierten Server (WEBSRV38, vgl. Tabelle 5.2) erhoben. Die in Abschnitt 5.6 berichtete NFA-2-Konformitätsaussage gilt damit ausdrücklich für diese Hardware-Klassen; eine Übertragung auf abweichende Plattformen – insbesondere GPU-beschleunigte Inferenz mit ν LLM oder MLX – bleibt empirisch neu zu verifizieren.

5.9.3. Konstrukt-Validität

Binäres Label „USEFUL/NOT_USEFUL“. Das im Konzept (vgl. Abschnitt 5.2.1) gewählte Binärlabel reduziert das eigentlich kontinuierliche Konstrukt „Wiederverwendbarkeit für künftigen Support“ auf eine harte Ja/Nein-Entscheidung. Inkrementell oder kontextabhängig nützliche Tickets (z. B. Tickets mit nützlichem Problem-Beschreibung, aber unklarer Lösung) werden zwangsweise einer der beiden Klassen zugeordnet. Eine mehrstufige Skala wäre konstrukt näher gewesen, hätte aber die Verfügbarkeit von Vergleichs-LLMs und die Anwendbarkeit der McNemar- und Wilson-Statistik reduziert.

LLM-as-Judge bei RAGAS. Sämtliche RAGAS-Verdicts (Faithfulness, Answer Relevance, Context Precision, Context Recall) sind Ausgaben eines weiteren LLM und nicht objektive Ground-Truth-Bewertungen. Sie messen, wie gut das Generator-Ergebnis aus Sicht des Judge-LLMs zum Ground Truth passt; systematische Verzerrungen des Judges (etwa Bevorzugung langer Antworten) sind nicht auszuschließen. Mitigation: Eine streng schemabasierte Pydantic-Validierung der Judge-Outputs (Abschnitt 5.5.2) filtert zumindest unstrukturierte Antworten heraus.

Heuristisch gewichteter 4-Signal-Konfidenz-Score. Die produktive Konfidenz aggregiert vier Signale – Rerank-Score, Trefferseparation, LLM-Selbstkonfidenz und Coverage – mit fest gewählten Gewichten (0,30/0,20/0,35/0,15, vgl. `DraftConfidenceConfig`). Diese Gewichte sind expertenbasiert gesetzt und nicht aus annotierten Daten gelernt. Mitigation: Die empirische Faithfulness-Standardabweichung in Abschnitt 5.5.6 stützt die abgestufte Trennung von `low_conf_threshold` und `hard_reject_threshold` und liefert damit eine indirekte Validierung des Score-Designs, ersetzt aber kein datengetriebenes Tuning der Gewichte.

5.9.4. Statistische Konklusionsvalidität

Effektive Auflösung der Stichproben. Auf Golden 45 entspricht ein einzelnes Ticket bereits 2,22 Prozentpunkten Accuracy; Differenzen unterhalb dieser Schwelle sind statistisch

nicht von zufälligen Einzelentscheidungen unterscheidbar (vgl. Abschnitt 5.2.1). Set 200 verbessert diese Auflösung auf 0,5 Prozentpunkte. Das RAGAS-Test-Set mit $n = 20$ liegt bewusst am unteren Rand der RAGAS-Empfehlung; Konfidenzintervalle pro Quelle sind entsprechend breit.

Wilson-Konfidenzintervalle als Härtest. Die zentralen Genauigkeitswerte werden konsequent mit Wilson-95 %-Konfidenzintervallen berichtet (Tabellen 5.23 und 5.24). Diese sind insbesondere auf Golden 45 ausgeprägt breit (z. B. Precision 100 % mit Intervall $[77,2; 100]$ % aufgrund $n = 13$ positiver Vorhersagen). Punktwerte ohne explizite Intervallangabe sollten daher nicht als belastbarer Vergleichsmaßstab gelesen werden.

Nicht signifikante Quantisierungsdifferenz. Der McNemar-Test für den Übergang $Q4 \rightarrow Q5$ ergibt $p = 1,00$; der Accuracy-Zugewinn von 84,4 % $\rightarrow$ 86,7 % entspricht einem einzelnen Ticket und ist statistisch nicht von Zufall unterscheidbar (Abschnitt 5.2.13). Die Produktionsentscheidung zugunsten von Q5 wird daher ausdrücklich nicht über die Punkt-Accuracy, sondern über Latenz und Wartbarkeit begründet.

Parser-Ausfälle in der RAGAS-Auswertung. Vier von 20 Bewertungen auf der Metrik Faithfulness sowie vier von 20 auf Context Recall blieben mangels valider Judge-JSON-Ausgabe ohne Score und gehen als NaN weder in Mittelwert noch Standardabweichung ein (Abschnitt 5.5.7). Die effektive Stichprobe für diese beiden Metriken beträgt $n = 16$. Eine Verteilungsanalyse der ausgefallenen IDs zeigt, dass die Fehler über alle vier Quellen, beide Sprachen und drei der vier Fragetypen verteilt sind und keinem inhaltlichen Muster folgen.

Keine Multiple-Comparisons-Korrektur. Die RAGAS-Evaluation berichtet vier Metriken über vier Quellen, was zu 16 Einzelvergleichen je Gesamtaussage führt. Eine Bonferroni- oder Benjamini-Hochberg-Korrektur wurde nicht angewandt, da die Aussagen primär deskriptiv erfolgen und nicht als formale Hypothesentests interpretiert werden. Bei einer formalen Inferenz an gleichen Daten ist diese Korrektur nachzuholen.

5.9.5. Operative Einschränkungen

Diese Einschränkungen betreffen den produktiven Betrieb des Systems und nicht die Geltung der berichteten Evaluationsergebnisse. Sie werden hier gesammelt, um die im Fazitkapitel diskutierten Folgeentwicklungen ankerbar zu machen.

- **Lokale LLM-Qualität gegenüber Frontier-Modellen.** Llama 3.1 8B liefert bei offenen, mehrstufigen Reasoning-Aufgaben außerhalb der hier evaluierten domänen-spezifischen Klassifikationsaufgabe eine geringere Antwortqualität als kommerzielle Frontier-Modelle wie GPT-4 oder Claude Opus; auf der binären Klassifikationsaufgabe übertrifft die hier kalibrierte Pipeline jedoch fünf der sieben in Abschnitt 5.2.12 verglichenen Frontier-Modelle. Der Kompromiss zwischen allgemeiner Reasoning-Stärke und On-Premise-Datenschutz wurde bewusst zugunsten des Datenschutzes entschieden.
- **Latenz auf CPU- und MPS-Hardware.** Die Antwortgenerierung ist auf den verfügbaren Hardware-Klassen (Apple M3 mit Metal, Server-CPU) langsamer als auf GPU-beschleunigter Hardware mit vLLM/MLX-Optimierung. Folgewirkungen auf die NFA-2-Konformitätsaussage sind in Abschnitt 5.6 diskutiert.
- **Keine inkrementelle Index-Aktualisierung.** Der FAISS-Index muss bei Änderungen an der Wissensbasis manuell neu aufgebaut werden; ein inkrementeller Aktualisierungs-Workflow ist nicht implementiert.
- **Konfidenz an Grenzfällen unzuverlässig.** Bei Anfragen, die semantisch sehr nah an vorhandenen Dokumenten liegen, ohne von diesen direkt beantwortet zu werden, kann der 4-Signal-Konfidenz-Score überoptimistisch ausfallen. Die Schwellen `low_conf_threshold` und `hard_reject_threshold` mitigieren, aber eliminieren diesen Effekt nicht vollständig.
- **Kein Feedback-Loop.** Es fehlt ein Mechanismus, der es den Support-Mitarbeitern erlaubt, die Qualität der generierten Antworten zu bewerten und diese Rückmeldungen in eine kontinuierliche Verbesserung des Systems (Override-Anpassung, Re-Ranking-Tuning) einfließen zu lassen.
- **Manuelle Pflege der Override-Regeln.** Die Confidence-Overrides P1–P5 wurden händisch aus Fehleranalysen abgeleitet. Eine automatische Ableitung neuer Overrides aus zukünftigen Annotationen ist nicht implementiert.
- **Keine formale Tail-Perzentil-Latenzstudie.** Die in Tabelle 5.39 berichteten Latenzwerte basieren auf Stichproben aus dem Iterationsbetrieb und werden mit Mittelwert plus Bereich angegeben; eine systematische p95-/p99-Studie unter definierten Lastprofilen würde eine separate Last-Test-Stichprobe erfordern und ist Voraussetzung für eine quantifizierte SLA-Aussage im produktiven Deployment (vgl. Begründung in Abschnitt 5.6).

Kapitel 6.

Zusammenfassung und Ausblick

6.1. Zusammenfassung

In der vorliegenden Bachelorarbeit wurde ein KI-gestützter Support-Agent als Webanwendung konzipiert, implementiert und evaluiert. Das System unterstützt Support-Mitarbeiter bei der Beantwortung technischer Kundenanfragen, indem es relevante Informationen aus verteilten Wissensquellen automatisch abrufen und daraus Antwortvorschläge generiert.

Das System integriert Daten aus vier produktiv eingebundenen Quellen – Zendesk-Tickets, HC, Vitruvi und der TKI-Pr-KB – in einer einheitlichen, semantisch durchsuchbaren Struktur. Drei weitere unternehmensinterne Quellen (Confluence, Jira, Slack) wurden analysiert, aus zeitlichen Scope-Gründen jedoch nicht in die produktive Pipeline aufgenommen und verbleiben als Erweiterungsrichtung für zukünftige Arbeiten. Die Datenverarbeitungspipeline umfasst die Extraktion, Normalisierung, PII-Maskierung, Chunking und Vektoreinbettung der Quelldaten.

Für die Retrieval-Phase wurde eine zweistufige Pipeline implementiert, die einen schnellen Bi-Encoder (BGE-M3 mit FAISS-Indexierung) für die Vorauswahl mit einem präzisen Cross-Encoder (BGE-Reranker-v2-M3) für das Reranking kombiniert. Die MMR-basierte Diversitätsselektion und ein Score-Schwellenwertfilter verbessern die Qualität der an das LLM weitergeleiteten Dokumente.

Die Antwortgenerierung erfolgt durch ein lokal betriebenes Llama 3.1 8B-Modell in der Q5_K_M-Quantisierung, das über `llama-cpp-python` ohne GPU-Abhängigkeit ausgeführt wird. Die Token-für-Token-Ausgabe wird über Server-Sent Events an das React-Frontend gestreamt, was eine progressive Darstellung der Ergebnisse ermöglicht.

Für das vorgelagerte Subsystem der *Qualitätsfilterung* der Zendesk-Tickets, das die Eignung jedes Tickets für die Wissensbasis automatisiert beurteilt, wurde ein eigener Klassifikator entwickelt. Dieser durchlief mehrere Modellkonfigurationen – von Llama 3.1 8B Q4_K_M mit Self-Consistency in der lokalen Iterationsphase über einen verworfenen Pilottest mit Qwen 2.5 7B und eine Qwen 2.5 32B Q4_K_M-Vergleichsbaseline bis zur produktiven Endkonfiguration mit Llama 3.1 8B Q5_K_M im Single-Shot-Modus auf Server-Hardware. Diese erreicht 91,1 % Accuracy auf dem Entwicklungsdatensatz Golden 45 und

87,0 % Accuracy auf dem Validierungsdatensatz Set 200.

Eine mehrstufige Konfidenzbewertung, die vier Signale (Rerank-Score, Score-Separation, LLM-Selbstbewertung und Abfrageterm-Abdeckung) kombiniert, unterstützt die Support-Mitarbeiter bei der Einschätzung der Antwortqualität. Ein Hard-Reject-Mechanismus verhindert die Anzeige von Antworten mit zu niedriger Konfidenz.

Das Frontend macht den Antwortpfad in vier Live-Phasen sichtbar (Suche, Reranking, Entwurfsgenerierung, Abschluss) und unterstützt die Quellenüberprüfung durch semantische Zitations-Anker, die im Antworttext als klickbare Chips erscheinen und auf die zugehörige Quellkarte in der Quellenleiste verweisen. Der Modellwechsel zur Laufzeit, drei modellfamilien-spezifische Presets, eine optionale Debug-Sicht sowie die Bearbeitbarkeit des Entwurfs vor dem Übernehmen ins Ticketsystem komplettieren die Werkzeugkette. Die Architektur ist modular aufgebaut und ermöglicht die Erweiterung um zusätzliche Datenquellen und Modelle.

Der vollständige lokale Betrieb des LLM stellt sicher, dass keine Kundendaten an externe Dienste übertragen werden, was den Datenschutzerfordernungen des Unternehmens entspricht.

Grenzen der Arbeit. Die methodischen Grenzen der Arbeit sind im strukturierten Abschnitt 5.9 (*Limitationen und Threats to Validity*) gebündelt und nach den vier Kategorien interne, externe, Konstrukt- und statistische Konklusionsvalidität geordnet. Die wesentlichen Befunde sind: (i) die Annotation aller Datensätze erfolgte durch einen einzelnen Annotator, sodass ein klassisches Inter-Annotator-Agreement strukturell nicht erhebbar ist und nur durch die unabhängige Frontier-LLM-Cross-Validierung indirekt gestützt wird; (ii) Bucket-Definitionen und Confidence-Overrides der Qualitätsfilterung sind auf die TKI-Domäne kalibriert und nicht ohne Neukalibrierung übertragbar; (iii) die Auflösung der berichteten Genauigkeitswerte ist durch die Stichprobengröße begrenzt (Golden 45: $\pm 2,22$ Prozentpunkte je Ticket; RAGAS-Test-Set $n = 20$ am unteren Rand der Empfehlung); und (iv) der McNemar-Test belegt, dass der Quantisierungsübergang Q4 $\rightarrow$ Q5 statistisch nicht signifikant ist, sodass die Produktionsentscheidung explizit nicht über die Accuracy, sondern über Latenz- und Wartbarkeitsargumente begründet wird. Die berichteten Ergebnisse sind vor diesem Hintergrund zu lesen.

6.2. Ausblick

Das entwickelte System bietet mehrere Ansatzpunkte für Weiterentwicklungen:

Inkrementelle Indexaktualisierung: Derzeit muss der FAISS-Index bei Änderungen an der Wissensbasis vollständig neu aufgebaut werden. Eine inkrementelle Aktualisierung, die

nur geänderte oder neue Dokumente verarbeitet, würde die Aktualität der Wissensbasis verbessern und den Aufwand für die Pflege reduzieren.

Feedback-Mechanismus: Die Integration eines Bewertungssystems (z. B. „Hilfreich“/„Nicht hilfreich“-Buttons) würde es ermöglichen, die Nutzung und Qualität der Antworten systematisch zu erfassen. Diese Daten könnten für die Feinabstimmung der Konfidenzgewichte oder das Fine-Tuning des Modells verwendet werden.

Größere oder spezialisierte Modelle: Mit zunehmender Verfügbarkeit leistungsfähigerer Hardware könnten größere Modelle (z. B. 70B-Parameter-Varianten) oder auf den Support-Kontext feinabgestimmte Modelle eingesetzt werden, um die Antwortqualität weiter zu verbessern.

Multi-Agenten-Architektur: Eine Erweiterung auf mehrere spezialisierte Agenten – etwa ein Agent für die Klassifikation der Anfrage, ein Agent für die Recherche und ein Agent für die Formulierung – könnte die Gesamtqualität verbessern, indem jede Teilaufgabe von einem optimierten Modell bearbeitet wird.

Integration in bestehende Systeme: Die Anbindung des Systems als Plugin oder Widget direkt in die Zendesk-Agent-Oberfläche würde die Nutzung im täglichen Workflow erleichtern und den Kontextwechsel für die Support-Mitarbeiter reduzieren.

Erweiterte Evaluation: Eine umfassendere Evaluation mit einer größeren Stichprobe, A/B-Tests im produktiven Einsatz und Nutzerbefragungen würde aussagekräftigere Ergebnisse über den tatsächlichen Mehrwert des Systems liefern.

Telemetrie und Monitoring: Die Implementierung einer Telemetrie-Schicht zur Erfassung von Anfrage-Antwort-Metriken im Produktivbetrieb würde die kontinuierliche Überwachung und Optimierung des Systems ermöglichen.

Literaturverzeichnis

- [1] ABETLEN ; CONTRIBUTORS: *llama-cpp-python – Python bindings for llama.cpp*. <https://github.com/abetlen/llama-cpp-python>, 2026. – Python-Wrapper für llama.cpp mit Unterstützung für GGUF-Quantisierung und GBNF-Grammatiken; eingesetzt für die lokale Inferenz aller LLM-Aufrufe der Pipeline (Filter- und Extraktions-Pipeline), abgerufen am 2026-05-18
- [2] ANTHROPIC: *Claude Haiku 4.5 – Modellbeschreibung*. <https://www.anthropic.com/claude>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [3] ANTHROPIC: *Claude Opus 4.7 – Modellbeschreibung*. <https://www.anthropic.com/claude>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [4] ANTHROPIC: *Claude Sonnet 4.6 – Modellbeschreibung*. <https://www.anthropic.com/claude>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [5] ASAI, Akari ; WU, Zeqiu ; WANG, Yizhong ; SIL, Avirup ; HAJISHIRZI, Hannaneh: Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. In: *International Conference on Learning Representations (ICLR)*, 2024
- [6] ATLISSIAN: *Evaluate the performance of your Rovo agent*. <https://support.atlassian.com/rovo/docs/evaluate-the-performance-of-your-rovo-agent/>, 2026. – Hersteller-Dokumentation zu Evaluationsmetriken, abgerufen am 2026-05-28
- [7] ATLISSIAN: *Knowledge sources for Rovo subagents*. <https://support.atlassian.com/rovo/docs/knowledge-sources-for-agents/>, 2026. – Atlassian Support-Dokumentation zu Wissensquellen und Permission-Modell, abgerufen am 2026-05-28
- [8] ATLISSIAN: *Rovo – Unlock organizational knowledge with GenAI*. <https://www.atlassian.com/software/rovo>, 2026. – Produktdokumentation, abgerufen am 2026-05-28
- [9] ATLISSIAN: *Rovo Plans and Trial – Lizenzmodell und Credit-System*. <https://www.atlassian.com/licensing/rovo>, 2026. – abgerufen am 2026-05-28

- [10] BROWN, Tom ; MANN, Benjamin ; RYDER, Nick ; SUBBIAH, Melanie ; KAPLAN, Jared D. ; DHARIWAL, Prafulla ; NEELAKANTAN, Arvind ; SHYAM, Pranav ; SASTRY, Girish ; ASKELL, Amanda u. a.: Language models are few-shot learners. In: *Advances in Neural Information Processing Systems* 33 (2020), S. 1877–1901
- [11] CARBONELL, Jaime ; GOLDSTEIN, Jade: The use of MMR, diversity-based reranking for reordering documents and producing summaries. In: *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1998, S. 335–336
- [12] CHEN, Jianlv ; XIAO, Shitao ; ZHANG, Peitian ; LUO, Kun ; LIAN, Defu ; LIU, Zheng: BGE M3-Embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In: *arXiv preprint arXiv:2402.03216* (2024)
- [13] DEEPSET: *Haystack – Open-Source LLM Framework*. <https://haystack.deepset.ai/>, 2026. – Produktions-orientiertes RAG-Framework von deepset, abgerufen am 2026-05-28
- [14] DETTMERS, Tim ; PAGNONI, Artidoro ; HOLTZMAN, Ari ; ZETTLEMOYER, Luke: QLoRA: Efficient finetuning of quantized LLMs. In: *Advances in Neural Information Processing Systems* 36 (2023)
- [15] DEVLIN, Jacob ; CHANG, Ming-Wei ; LEE, Kenton ; TOUTANOVA, Kristina: BERT: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of NAACL-HLT*, 2019, S. 4171–4186
- [16] ES, Shahul ; JAMES, Jithin ; ESPINOSA-ANKE, Luis ; SCHOCKAERT, Steven: RAGAS: Automated Evaluation of Retrieval Augmented Generation. In: *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024, S. 150–158
- [17] FRESHWORKS INC.: *Freshworks Launches New AI Capabilities to Minimize Customer Service Fragmentation*. <https://freshworks.gcs-web.com/news-releases/news-release-details/freshworks-launches-new-ai-capabilities-minimize-customer>, 2025. – Hersteller-Pressemitteilung mit Deflection-Daten zu Freddy AI (Ø 65 %, bis 80 %), abgerufen am 2026-05-28
- [18] GAO, Luyu ; MA, Xueguang ; LIN, Jimmy ; CALLAN, Jamie: Precise Zero-Shot Dense Retrieval without Relevance Labels. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023, S. 1762–1777
- [19] GERGANOV, Georgi ; CONTRIBUTORS: *llama.cpp – LLM inference in C/C++ with GGUF quantization and GBNF grammar-constrained generation*. <https://github.com/>

- ggerganov/llama.cpp, 2026. – Definiert das GGUF-Modellformat und die GBNF-Grammatik-Syntax für grammar-constrained Generation; siehe grammars/README.md im Repository, abgerufen am 2026-05-18
- [20] GITHUB: *GitHub Copilot Pro – Plan und Modellzugang*. <https://github.com/features/copilot/plans>, 2026. – Multi-Model-Zugriff (Anthropic, OpenAI, Google, xAI) in Visual Studio Code, abgerufen am 2026-05-12
- [21] GOOGLE DEEPMIND: *Gemini 3.1 – Modellbeschreibung*. <https://deepmind.google/technologies/gemini/>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [22] HOLTZMAN, Ari ; BUYS, Jan ; DU, Li ; FORBES, Maxwell ; CHOI, Yejin: The curious case of neural text degeneration. In: *International Conference on Learning Representations (ICLR)*, 2020
- [23] IBM RESEARCH: *Docling – Document conversion for AI workflows*. <https://github.com/DS4SD/docling>, 2026. – Open-Source-Bibliothek mit HybridChunker für heading-aware Dokument-Segmentierung, abgerufen am 2026-05-28
- [24] INTERCOM: *From resolutions to outcomes: Evolving how Fin delivers value*. <https://www.intercom.com/blog/from-resolutions-to-outcomes-evolving-how-fin-delivers-value/>, 2026. – Hersteller-Bericht zur Resolution Rate von Fin v2 (67 % Durchschnitt, Stand März 2026), abgerufen am 2026-05-28
- [25] JIANG, Zhengbao ; XU, Frank F. ; GAO, Luyu ; SUN, Zhiqing ; LIU, Qian ; DWIVEDI-YU, Jane ; YANG, Yiming ; CALLAN, Jamie ; NEUBIG, Graham: Active Retrieval Augmented Generation. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023, S. 7969–7992
- [26] JOHNSON, Jeff ; DOUZE, Matthijs ; JÉGOU, Hervé: Billion-scale similarity search with GPUs. In: *IEEE Transactions on Big Data* 7 (2019), Nr. 3, S. 535–547
- [27] KARPUKHIN, Vladimir ; OĞUZ, Barlas ; MIN, Sewon ; LEWIS, Patrick ; WU, Ledell ; EDUNOV, Sergey ; CHEN, Danqi ; YIH, Wen-tau: Dense passage retrieval for open-domain question answering. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020, S. 6769–6781
- [28] LANGCHAIN: *LangChain Documentation*. <https://python.langchain.com/docs/>, 2026. – Open-Source-Orchestrierungsframework für LLM-Anwendungen, abgerufen am 2026-05-28

- [29] LEWIS, Patrick ; PEREZ, Ethan ; PIKTUS, Aleksandra ; PETRONI, Fabio ; KARPUKHIN, Vladimir ; GOYAL, Naman ; KÜTTLER, Heinrich ; LEWIS, Mike ; YIH, Wen-tau ; ROCKTÄSCHEL, Tim u. a.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: *Advances in Neural Information Processing Systems* 33 (2020), S. 9459–9474
- [30] LLAMAINDEX: *LlamaIndex – Data Framework for LLM Applications*. <https://docs.llamaindex.ai/>, 2026. – Open-Source-Framework mit Fokus auf Retrieval und Indexing, abgerufen am 2026-05-28
- [31] MALKOV, Yury A. ; YASHUNIN, Dmitry A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42 (2020), Nr. 4, S. 824–836
- [32] MANNING, Christopher D. ; RAGHAVAN, Prabhakar ; SCHÜTZE, Hinrich: *Introduction to Information Retrieval*. Cambridge University Press, 2008
- [33] MICROSOFT: *Visual Studio Code*. <https://code.visualstudio.com/>, 2026. – Editor mit GitHub-Copilot-Integration für Multi-Model-LLM-Zugriff, abgerufen am 2026-05-12
- [34] NOGUEIRA, Rodrigo ; JIANG, Zhiying ; PRADEEP, Ronak ; LIN, Jimmy: Document ranking with a pretrained sequence-to-sequence model. In: *Findings of EMNLP*, 2020, S. 708–718
- [35] OORD, Aäron van den ; LI, Yazhe ; VINYALS, Oriol: Representation learning with contrastive predictive coding. In: *arXiv preprint arXiv:1807.03748* (2018)
- [36] OPENAI: *GPT-5.4 – Modellbeschreibung*. <https://platform.openai.com/docs/models>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [37] OPENAI: *GPT-5.5 – Modellbeschreibung*. <https://platform.openai.com/docs/models>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [38] OPENAI: *GPTs (ChatGPT Enterprise version)*. <https://help.openai.com/en/articles/8555535-gpts-chatgpt-enterprise-version>, 2026. – OpenAI Help Center – Funktionsbeschreibung Custom GPTs für Unternehmens-Workspaces, abgerufen am 2026-05-28
- [39] OPENAI: *Managing GPT access in Enterprise and Edu workspaces*. <https://help.openai.com/en/articles/9083988-how-to-share-gpts-within-workspaces>, 2026. – OpenAI Help Center – Sharing-Modell für Custom GPTs (Link-Sharing, Workspace-Zugriff), abgerufen am 2026-05-28

- [40] OPENAPI INITIATIVE: *OpenAPI Specification, Version 3.0.3*. <https://spec.openapis.org/oas/v3.0.3>, 2020. – Sprachspezifikation für die Beschreibung von REST-APIs, abgerufen am 2026-05-28
- [41] OUYANG, Long ; WU, Jeffrey ; JIANG, Xu ; ALMEIDA, Diogo ; WAINWRIGHT, Carroll ; MISHKIN, Pamela ; ZHANG, Chong ; AGARWAL, Sandhini ; SLAMA, Katarina ; RAY, Alex u. a.: Training language models to follow instructions with human feedback. In: *Advances in Neural Information Processing Systems* 35 (2022), S. 27730–27744
- [42] README INC.: *ReadMe.io – Developer Documentation Platform*. <https://readme.com/>, 2026. – Hosting-Plattform für die Vitruvi-Entwicklerdokumentation, abgerufen am 2026-05-28
- [43] REIMERS, Nils ; GUREVYCH, Iryna: Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In: *arXiv preprint arXiv:1908.10084* (2019)
- [44] ROBERTSON, Stephen ; ZARAGOZA, Hugo: The probabilistic relevance framework: BM25 and beyond. In: *Foundations and Trends in Information Retrieval* 3 (2009), Nr. 4, S. 333–389
- [45] SALESFORCE: *Salesforce Announces Einstein GPT for Service and New Service Automation Capabilities Powered by Data Cloud*. <https://www.salesforce.com/news/stories/einstein-gpt-service-cloud-news/>, 2024. – Hersteller-Ankündigung zu Einstein GPT for Service, abgerufen am 2026-05-28
- [46] SENNRICH, Rico ; HADDOW, Barry ; BIRCH, Alexandra: Neural machine translation of rare words with subword units. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, S. 1715–1725
- [47] STREAMLIT INC.: *Streamlit – A faster way to build and share data apps*. <https://streamlit.io/>, 2026. – Python-Framework für interaktive Web-Anwendungen, verwendet für die Ticket-Annotationsoberfläche `annotate_tickets.py`, abgerufen am 2026-05-15
- [48] VASWANI, Ashish ; SHAZEER, Noam ; PARMAR, Niki ; USZKOREIT, Jakob ; JONES, Llion ; GOMEZ, Aidan N. ; KAISER, Łukasz ; POLOSUKHIN, Illia: Attention is all you need. In: *Advances in neural information processing systems* 30 (2017)
- [49] VITRUVI SOFTWARE: *Vitruvi Developer Portal – Documentation and API Reference*. <https://developers.vitruvi.cc/docs/getting-started>, 2026. – Öffentliche Entwicklerdokumentation mit Konzept-Guides (`/docs`) und OpenAPI-Referenz (`/reference`), abgerufen am 2026-05-28

- [50] VITRUVI SOFTWARE: *Vitruvi Help Center – Knowledge Base*. <https://vitruvi.zendesk.com/hc/en-us>, 2026. – Geschlossene Wissensdatenbank für Vitruvi-Kunden auf Zendesk-Basis, Zugriff nur für authentifizierte Nutzer, abgerufen am 2026-05-28
- [51] WANG, Xuezhi ; WEI, Jason ; SCHUURMANS, Dale ; LE, Quoc ; CHI, Ed ; NARANG, Sharan ; CHOWDHERY, Aakanksha ; ZHOU, Denny: Self-consistency improves chain of thought reasoning in language models. In: *International Conference on Learning Representations (ICLR)*, 2023
- [52] xAI: *Grok 4.1 Fast – Modellbeschreibung*. <https://x.ai/>, 2026. – Zugriff über GitHub Copilot Pro in Visual Studio Code, abgerufen am 2026-05-12
- [53] YAN, Shi-Qi ; GU, Jia-Chen ; ZHU, Yun ; LING, Zhen-Hua: Corrective Retrieval Augmented Generation. In: *arXiv preprint arXiv:2401.15884* (2024)
- [54] ZENDESK: *Zendesk CX Trends Report 2024*. <https://www.zendesk.com/blog/cx-trends/>, 2024. – Branchenbericht zur Automatisierung im Kundensupport (30–50 % Tier-1-Auflösung durch AI-Chatbots), abgerufen am 2026-05-28

Kapitel 7.

Übersicht verwendeter Hilfsmittel

Im Sinne der Selbstständigkeitserklärung listet dieser Abschnitt sämtliche KI-basierten Werkzeuge auf, die im Rahmen der vorliegenden Arbeit verwendet wurden. Tabelle 7.1 dokumentiert jeweils den Produktnamen, den genutzten Funktionsumfang sowie die Textstellen, an denen die Werkzeuge zum Einsatz kamen. Die geistige Leistung, die fachliche Bewertung sowie die Verantwortung für sämtliche übernommenen Passagen liegen vollständig beim Autor.

Sämtliche oben aufgeführten Werkzeuge dienten ausschließlich unterstützender Funktion. Die wissenschaftliche Fragestellung, das Forschungsdesign, die Auswahl der Methoden, die Auswertung und Interpretation der Ergebnisse sowie die finale Fassung aller Texte und Quellcode-Bestandteile wurden vom Autor eigenständig verantwortet.

Produkt	Funktionsumfang im Rahmen der Arbeit	Textstellen
Claude Versionen 4.6, Haiku 4.5), Sonnet 4.7, genutzt über die Claude Code-CLI sowie über GitHub Copilot Pro in Visual Studio Code	(Anthropic; Opus 4.7, Haiku 4.5), Refaktorisierung von Quellcode-Skizzen, Unterstützung bei $\text{\LaTeX}$ -Tabellen- und Formelsatz sowie Zusammenfassung und Ver gleich wissenschaftlicher Quellen. Die Endfassung sämtlicher Texte, Code- Listings und Ergebnisinterpretationen wurde vom Autor manuell überprüft.	Kapitel 3, 4, 5; Anhang A.7
ChatGPT (OpenAI)	Prototyping der Ticket-zu-Dateneinheit-Extraktion: In der Konzeptphase wurde ChatGPT eine exemplarische Zendesk-Konversation übergeben, um zu prüfen, in welcher Form ein kommerzielles Frontier-Modell eine struktu- rierte Dateneinheit (<i>Data-Unit</i>) aus einer Support-Korrespondenz extrahie- ren kann. Das resultierende Ausgabeschema diente anschließend als Re- ferenzmuster für die Reproduktion dieses Extraktionsschritts mit lokal be- triebenen Modellen im Implementierungsteil. Die endgültige Festlegung des Schemas und sämtlicher Auswertungen wurde vom Autor manuell überprüft und verantwortet.	Kapitel 3, 4
GitHub Copilot Pro (in Vi- sual Studio Code)	Code-Autovervollständigung während der Implementierung sowie Zugriff auf kommerzielle Frontier-Modelle (Claude Opus 4.7, Sonnet 4.6, Haiku 4.5; GPT-5.5, GPT-5.4; Gemini 3.1; Grok 4.1 Fast) im Rahmen der externen Va- lidierung des Useful-/Useless-Klassifikators.	Kapitel 4; Anhang A.5

Tabelle 7.1.: Im Rahmen dieser Bachelorarbeit verwendete KI-basierte Hilfsmittel.

Anhang A.

Anhang

A.1. Technologieübersicht

Tabelle A.1.: Übersicht der eingesetzten Technologien

Komponente	Technologie	Version / Details
Frontend	React	19.2.0
Frontend Build	Vite	7.3.1
Frontend Sprache	TypeScript	5.9.3
Frontend Tests	Vitest	4.1.1
Backend Framework	FastAPI	≥ 0.104
Backend Server	Uvicorn	≥ 0.24
Embeddings	Sentence-Transformers	BGE-M3, 1024 dim
Vektorsuche	FAISS	IndexFlatIP
Reranking	CrossEncoder	BGE-Reranker-v2-M3
LLM Runtime	llama-cpp-python	≥ 0.2
LLM Modelle	siehe Tabelle A.2	verschiedene Rollen / Quantisierungen
Datenformat	Apache Parquet	via Pandas
PDF-Verarbeitung	Docling	mit Granite Vision
Evaluation	RAGAS	$\geq 0.2.0$
Rate Limiting	slowapi	$\geq 0.1.9$

Tabelle A.2.: Eingesetzte LLM-Modelle nach Rolle

Rolle	Modell	Quant.	Einsatz
Antwort-Gen. (RAG)	Llama 3.1 8B	Q5_K_M	Backend, lokal über llama-cpp-python; produktive Endkonfiguration
Qualitätsfilter (Iter.)	Llama 3.1 8B	Q5_K_M	Iterationsphase Mo–Di 2026-05-04/05; Spitzenwert 88,9 % (Golden 45)
Qualitätsfilter (Pilot)	Qwen 2.5 7B	Q4_K_M	Pilottest am 2026-05-04, ca. 64 % Accuracy, verworfen
Qualitätsfilter (Baseline)	Qwen 2.5 32B	Q4_K_M	Vergleichsmodell auf Server WEBSRV38, 16-Core CPU, Single-Pass-Klassifikator (86,7 % Golden 45)
Qualitätsfilter (Prod.)	Llama 3.1 8B	Q5_K_M	Produktive Endkonfiguration auf Server WEBSRV38, Single-Shot $T = 0$, 91,1 % Golden 45
RAGAS Judge	Llama 3.1 8B	Q4_K_M	Evaluations-Harness (react_app/eval/judge.py); on-premise JSON-strict Bewertung der vier RAGAS-Metriken

A.2. API-Endpunkte

Tabelle A.3.: Vollständige API-Endpunktübersicht

Endpunkt	Methode	Auth	Beschreibung
/	GET	Nein	Systemstatus
/health	GET	Nein	Modell- und Queue-Status
/api/models	GET	Nein	Verfügbare LLM-Modelle
/api/models/switch	POST	Nein	Modellwechsel
/api/search	POST	Nein	Semantische FAISS-Suche
/api/rerank	POST	Nein	Cross-Encoder Reranking
/api/draft	POST	Nein	LLM-Antwortgenerierung
/api/pipeline	POST	Nein	Synchrone Pipeline
/api/pipeline-stream	POST	Nein	SSE-Streaming-Pipeline

A.3. Umgebungsvariablen

Tabelle A.4.: Konfigurierbare Umgebungsvariablen

Variable	Standard	Beschreibung
LLM_MODEL	Llama-3.1-8B	GGUF-Dateiname des LLM
LLM_N_CTX	4096	Kontextfenstergröße
LLM_N_THREADS	8	CPU-Threads für LLM
LLM_N_GPU_LAYERS	0	GPU-Schichten (0 = CPU)
RATE_LIMIT	10/min	API Rate Limiting
CORS_ALLOW_ORIGINS	localhost	Erlaubte CORS-Origins
MAX_CONCURRENT	2	Max. parallele Pipelines
DRAFT_TIMEOUT	300s	Timeout pro Draft

A.4. Per-Ticket-Vergleichsmatrix Golden 45

Tabelle A.5 dokumentiert die Vorhersagen aller zwölf in Abschnitt 5.2.11 verglichenen Konfigurationen für jeden der 45 Tickets des Golden-Datensatzes. Die Symbole bezeichnen die Vorhersagelabel: **U** = USEFUL, **X** = USELESS. Die Hochstellung gibt an, ob die Vorhersage der Wahrheit entspricht: ✓ = korrekt, × = Fehler.

Spaltenkürzel:

- **q7**: Qwen 2.5 7B Q4 v8 (lokal, 2026-05-04)
- **c1–c7**: Llama 3.1 8B in Phase 2 (lokal Mac M3) – 5d81d59, dda7a86, 4a0fef1, 44750c8, 7d45b0d (SC $N=3$), 623917d (Q5), 4560fa1 (Q5)
- **v1–v3**: Qwen 2.5 32B Q4 auf Server – Baseline, v2, v3
- **LS**: Llama 3.1 8B Q5_K_M auf Server (*produktiv*)

A.5. Externe Frontier-LLM-Validierung – Vergleichsmatrix Golden 45

Tabelle A.6 dokumentiert die ticketweisen Vorhersagen von sieben kommerziellen Frontier-Modellen, die in Abschnitt 5.2.12 als externe Validierungsbasis verwendet wurden. Der Zugriff auf alle Modelle erfolgte über GitHub Copilot Pro [20] in Visual Studio Code [33];

Tabelle A.5.: Per-Ticket-Vergleichsmatrix Golden 45 (alle 12 Konfigurationen)

#	Ticket	Titel	GT	q7	c1	c2	c3	c4	c5	c6	c7	v1	v2	v3	LS
USEFUL (Wahrheit U)															
1	1196	DXF Einmessung	U	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	U ^r	X ^x	U ^r	U ^r	U ^r
2	1594	NET Optionen-Einstellungen	U	X ^x	U ^r	U ^r	U ^r	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
3	4413	Search address functionality	U	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	U ^r	X ^x
4	4890	Cluster ID Problem	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	U ^r	U ^r
5	6654	Missing Verstärker in View	U	X ^x	U ^r	X ^x	X ^x	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
6	6909	Net Installer Update	U	X ^x	X ^x	U ^r	U ^r	X ^x	U ^r	U ^r	U ^r	X ^x	X ^x	X ^x	U ^r
7	7467	Expired licenses	U	U ^r	U ^r	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
8	8067	Copying Doopack plot sequence	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
9	8240	Adding crossings to table	U	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	U ^r	U ^r	U ^r	X ^x
10	8626	Export format issue	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	X ^x	U ^r
11	9227	VMware Resolution Issue	U	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
12	9259	Duck Connections Editor	U	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	X ^x	X ^x	U ^r
13	11372	AutoCAD Installer Request	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	X ^x	U ^r	U ^r
14	12177	Trassen/Rohrzuordnungen	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
15	12909	Performance Issue	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
16	13073	Fiber Optic Connection	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	X ^x	U ^r	U ^r
17	13102	HLD to LLD Conversion	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	X ^x	X ^x	X ^x	U ^r
18	13930	Layer Import Issue	U	U ^r	U ^r	X ^x	X ^x	U ^r	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
19	16401	License Server Connection	U	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
20	17425	AutoCAD Stützpunkte	U	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
21	18653	Kabel Geometrie hinzufügen	U	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
22	18717	GF+ Data Transfer	U	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x	X ^x
23	19476	NVT Name Fix	U	X ^x	X ^x	X ^x	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r	U ^r
USELESS (Wahrheit X)															
24	1085	Documentation Service Training	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
25	1472	NET 12.0.4 Changes	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
26	1946	Ticket filtering/sorting	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	U ^x	X ^r	X ^r
27	2135	PDF report changes	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
28	2759	Bezugslinien Migration	X	X ^r	U ^x	X ^r	U ^x	U ^x	X ^r	U ^x	X ^r	X ^r	X ^r	X ^r	U ^x
29	2772	Label Migration Issue	X	X ^r	X ^r	X ^r	X ^r	U ^x	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
30	4061	OneDrive Installation Issue	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
31	4689	Push demand type	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
32	4812	Net Build Login	X	X ^r	U ^x	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
33	5946	Database & Plans Issues	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
34	7424	Access to designs	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
35	8599	Location Status Change	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
36	8619	User Access	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
37	9553	Kabel teilen Fehler	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
38	12609	NET Cloud Unavailability	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
39	15012	GfPlus Validierung	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
40	15310	Remote Desktop Access	X	X ^r	U ^x	U ^x	U ^x	U ^x	U ^x	U ^x	X ^r	X ^r	X ^r	X ^r	X ^r
41	16171	NET Scan Einrichtung	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
42	16836	TKI cable design	X	X ^r	X ^r	X ^r	X ^r	U ^x	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
43	16915	Aufhme Server Problem	X	X ^r	U ^x	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
44	18130	Change host owners	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r
45	20191	Polygon Redundancy Issue	X	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r	X ^r

jedem Modell wurde derselbe Support-Agenten-Prompt vorgelegt wie dem lokalen Klassifikator. Die Symbole **U** = USEFUL, **X** = USELESS; die Spalte *AI-Vote* gibt die Stimmenzahl (USEFUL/USELESS) der sieben Modelle wieder, die letzte Spalte (**✓/X**) entspricht dem menschlich annotierten Wahrheitswert.

Die letzte Spalte (**✓/X**) visualisiert die menschliche Wahrheit (USEFUL/USELESS) und dient als schnelle Orientierungshilfe beim Vergleich mit den Modellausgaben. Die ticketweisen Übereinstimmungen jedes Modells lassen sich aus den Spalten direkt ablesen; eine zusammenfassende Klassifikationsgüte für jedes Modell wurde in Tabelle 5.22 berechnet.

A.6. Kanonische Produktliste

Das System unterstützt die folgenden TKI-Softwareprodukte:

1. NET Design
2. NET Engineering
3. NET Build
4. NET Cloud
5. NET Scan
6. NET Operations
7. NET Inhouse
8. NET REST
9. NET Field Survey
10. Dynamics Sales

Die Produktnormalisierung erfolgt durch Substring-Matching, sodass verschiedene Schreibweisen (z. B. „netdesign“, „NET-Design“, „Net Design“) auf den kanonischen Namen abgebildet werden.

A.7. Vollständige Quellcode-Listings zum Implementierungskapitel

Dieser Abschnitt enthält die vollständigen Versionen der vier längeren Quellcode-Listings aus Kapitel 4, die im Haupttext aus Gründen der Lesbarkeit auf einen Schlüsselausschnitt

Tabelle A.6.: Vergleichsmatrix: Menschliche Annotation vs. sieben Frontier-Modelle (Golden 45)

Ticket	Human	Opus 4.7	Sonnet 4.6	Haiku 4.5	GPT 5.5	GPT 5.4	Gemini 3.1	Grok 4.1 F.	AI-Vote	H.
#7467	U	U	U	U	U	U	U	U	7/0	✓
#8599	X	X	X	X	X	X	X	X	0/7	X
#18130	X	X	X	X	X	X	X	X	0/7	X
#12609	X	X	X	X	X	X	X	U	1/6	X
#1594	U	U	U	U	U	U	U	U	7/0	✓
#6909	U	X	U	U	X	X	U	U	4/3	✓
#9553	X	U	X	X	X	X	X	X	1/6	X
#8619	X	X	X	X	X	X	X	X	0/7	X
#17425	U	U	U	U	U	U	U	U	7/0	✓
#4061	X	X	X	U	X	X	X	X	1/6	X
#15310	X	X	X	U	X	X	X	X	1/6	X
#1946	X	X	X	U	X	X	X	X	1/6	X
#13073	U	U	U	U	U	U	U	U	7/0	✓
#16915	X	X	X	X	X	X	X	X	0/7	X
#8626	U	X	U	U	X	X	U	U	4/3	✓
#16836	X	U	X	U	X	X	X	X	2/5	X
#8240	U	U	X	U	X	X	U	U	4/3	✓
#16171	X	X	X	X	X	X	X	X	0/7	X
#13102	U	X	X	U	X	X	U	U	3/4	✓
#2135	X	X	X	U	X	X	X	X	1/6	X
#9227	U	X	X	U	X	X	U	U	3/4	✓
#1196	U	U	X	X	X	X	U	U	3/4	✓
#18653	U	U	U	X	U	U	U	U	6/1	✓
#13930	U	U	X	U	U	U	U	U	6/1	✓
#9259	U	U	U	X	U	U	U	U	6/1	✓
#8067	U	U	U	U	U	U	U	U	7/0	✓
#16401	U	U	U	U	U	U	U	U	7/0	✓
#18717	U	U	X	U	X	X	U	U	4/3	✓
#20191	X	X	X	X	X	X	X	X	0/7	X
#1472	X	X	X	X	X	X	X	X	0/7	X
#7424	X	X	X	U	X	X	X	X	1/6	X
#19476	U	X	U	U	X	X	U	U	4/3	✓
#1085	X	U	X	X	X	U	X	X	1/6	X
#12177	U	U	U	U	U	U	U	U	7/0	✓
#4812	X	U	U	X	X	X	X	X	2/5	X
#4689	X	X	X	U	X	X	X	X	1/6	X
#2772	X	X	X	U	X	X	X	X	1/6	X
#2759	X	X	X	U	X	X	X	X	1/6	X
#4890	U	U	U	X	U	U	U	U	6/1	✓
#15012	X	U	X	U	X	X	X	X	2/5	X
#5946	X	X	X	U	X	X	X	X	1/6	X
#6654	U	U	U	U	U	U	U	U	7/0	✓
#11372	U	U	U	X	X	X	U	U	4/3	✓
#12909	U	U	U	U	U	X	U	U	6/1	✓
#4413	U	U	X	U	X	X	U	U	4/3	✓

reduziert sind. Die Auswahl umfasst die Ticket-Extraktionsfunktion, die deterministische Vorfilterregel-Menge, die Implementierung des Overrides P4 sowie die Aggregationsschleife der Self-Consistency-Variante. Querverweise aus dem Haupttext zeigen jeweils auf die hier dokumentierten Listings.

A.7.1. Zendesk-Ticket-Extraktion mit Cursor-Paginierung und Checkpointing

Die Funktionen in `zendesk/stage0_acquisition/fetch_zendesk.py` rufen die Zendesk *Search Export API* cursorbasiert ab (Phase 1) und reichern anschließend jedes Ticket über einen separaten Kommentar-Endpunkt an (Phase 2). Rate-Limit-Antworten (HTTP 429) werden mit *Retry-After-Backoff* behandelt; nach jeweils 100 angereicherten Tickets schreibt die Phase-2-Schleife einen Checkpoint, der bei einem Abbruch eine wiederaufnahmesichere Fortsetzung ermöglicht (vgl. Abschnitt 4.2.2.1 im Haupttext).

```
def fetch_all_tickets(email, api_token, subdomain):
    """Phase 1: cursor-Schleife ueber alle Tickets ohne Kommentare."""
    url = f"https://{subdomain}.zendesk.com/api/v2/search/export.json"
    auth = (f"{email}/token", api_token)
    tickets = []
    params = {"query": "type:ticket", "filter[type]": "ticket"}
    while url:
        r = requests.get(url, auth=auth, params=params)
        if r.status_code == 429:
            # Rate-Limit
            time.sleep(int(r.headers.get("Retry-After", 60)))
            continue
        data = r.json()
        tickets.extend(data.get("results", []))
        if data.get("meta", {}).get("has_more"):
            url = data.get("links", {}).get("next")
            params = {}
            # Cursor liegt im Next
            -Link
        else:
            url = None
    return tickets

def enrich_with_comments(tickets, email, api_token, subdomain,
                        checkpoint_file):
    """Phase 2: Kommentare nachladen + Checkpoint je 100 Tickets."""
    processed = 0
    for ticket in tickets:
        if ticket.get("comments"):
            # bereits angereichert
            continue
```

```

        ticket["comments"] = fetch_comments_for_ticket(
            ticket["id"], email, api_token, subdomain)
    processed += 1
    if processed % 100 == 0:
        save_checkpoint(checkpoint_file, tickets)
    time.sleep(0.1) # API-Rate-Schonung
    return tickets

```

Listing A.1: Zendesk-Ticket-Extraktion mit Cursor-Paginierung und Checkpointing (vollständig)

A.7.2. Deterministische Pre-Filter-Regeln

Die deterministische Vorfilterstufe verwirft zwei Klassen von Tickets, bevor sie an den LLM-Klassifikator weitergereicht werden: Tickets ohne öffentliche Kommentare (interne Provisionierungstickets) sowie kurze Telefonnotiz-Tickets („Anruf mit ...“). Beide Regelpfade geben unmittelbar eine Prediction mit source="prefilter" zurück; ansonsten gibt die Funktion None zurück und die Klassifikation übernimmt das LLM (vgl. Abschnitt 4.2.2.5 im Haupttext).

```

def prefilter(ticket: dict) -> Prediction | None:
    public = [c for c in ticket.get("comments", [])
               if c.get("public", True)]
    if not public:
        return Prediction(label="USELESS",
                          outcome="no_problem",
                          rationale="no public comments -- "
                                   "internal-only ticket",
                          source="prefilter")
    subj = (ticket.get("subject") or "").strip().lower()
    if (subj.startswith("anruf mit") and len(public) <= 1
        and len(public[0].get("body", "")) < 80):
        return Prediction(label="USELESS",
                          outcome="no_problem",
                          rationale="call-note ticket -- "
                                   "internal call summary",
                          source="prefilter")
    return None # an LLM weiterreichen

```

Listing A.2: Pre-Filter-Regeln (vollständig)


```

majority_confirmed = confirmed_votes >= (n // 2 + 1)
rep = next(s for s in samples if s.outcome == winning)
return Prediction(
    label=_decide(winning, majority_confirmed),
    outcome=winning,
    customer_confirmed=majority_confirmed,
    rationale=rep.rationale,
    subject_kind=rep.subject_kind,
    votes=dict(votes),
    source=f"sc_n{n}",
)

```

Listing A.4: Self-Consistency-Aggregation (vollständig)

A.7.5. LLM-Initialisierung für Qwen 2.5 32B

Der LLM-Aufruf erfolgt über llama-cpp-python mit dem Split-GGUF des Qwen-Modells. Neben den im Haupttext (Abschnitt 4.2.2.6) benannten Parametern (Kontextfenster, Threads, Chat-Format, Seed) zeigt die vollständige Initialisierung die Server-CPU-Konfiguration mit `n_gpu_layers=0` sowie das Unterdrücken der llama-cpp-internen Verbose-Ausgabe.

```

from llama_cpp import Llama

llm = Llama(
    model_path=str(MODEL_PATH), # split GGUF, Teil 00001-of-00005
    n_ctx=8192,
    n_gpu_layers=0, # reiner CPU-Betrieb (Server)
    n_threads=16,
    chat_format="chatml", # Qwen-2.x-spezifisch
    seed=7,
    verbose=False,
)

```

Listing A.5: LLM-Initialisierung für Qwen 2.5 32B (vollständig)

A.7.6. SSE-Streaming-Pipeline

Der Streaming-Endpunkt `/api/pipeline-stream` kapselt den vollständigen Antwortpfad aus semantischer Suche, Cross-Encoder-Reranking und Token-Streaming der Antwortgenerierung in einem einzigen asynchronen Generator. Die im Haupttext (Abschnitt 4.2.9) gezeigte Kernschleife ist Teil dieser Funktion; die vollständige Implementierung umfasst alle drei Pipeline-Stufen sowie die Auslagerung der CPU-intensiven Operationen über `asyncio.to_thread()`.

```

@router.post("/api/pipeline-stream")
async def pipeline_stream(req: PipelineRequest):
    async def event_generator():
        # Stage 1: Semantic Search
        hits = await asyncio.to_thread(
            semantic_search, req.query, req.search_k,
            products=req.products)
        yield {"event": "search_done",
              "data": json.dumps({"count": len(hits)})}

        # Stage 2: Reranking
        reranked = await asyncio.to_thread(
            rerank, req.query, hits, req.rerank_top_k)
        yield {"event": "rerank_done",
              "data": json.dumps({"count": len(reranked)})}

        # Stage 3: Draft Generation (Token-Streaming)
        for token in draft_answer_stream(
            req.query, reranked):
            if token["type"] == "token":
                yield {"event": "draft_token",
                      "data": token["token"]}
            elif token["type"] == "final":
                yield {"event": "complete",
                      "data": json.dumps(result)}

    return EventSourceResponse(event_generator())

```

Listing A.6: SSE-Streaming-Pipeline (vollständig)

A.7.7. Thin Wrapper über die produktive RAG-Pipeline (RAGAS-Evaluation)

Die in Abschnitt 5.5.2 angesprochene Eigenschaft „Produktivcode wird nicht modifiziert“ ist im Wrapper `react_app/eval/pipeline_wrapper.py` architektonisch umgesetzt. Die Funktion `production_answer()` reproduziert exakt die Stufenfolge des produktiven SSE-Endpunkts (semantische Suche, Reranking mit Schwellenwert, Antwortentwurf), jedoch synchron und ohne FastAPI-Overhead. Die produktive Quantisierung `Q5_K_M` wird über die Umgebungsvariable `LLM_MODEL` erzwungen, bevor das Backend-Konfigurationsmodul importiert wird, sodass die in Abschnitt 5.5.4 berichteten Werte tatsächlich für die produktive Endkonfiguration gelten.

```

import os, sys, time
from pathlib import Path

```

```

# Q5_K_M produktive Quantisierung erzwingen, BEVOR
# backend.config importiert wird.
os.environ.setdefault(
    "LLM_MODEL",
    "Meta-Llama-3.1-8B-Instruct-Q5_K_M.gguf",
)
sys.path.insert(0, str(
    Path(__file__).resolve().parent.parent / "backend" / "src"
))

from config import RERANKER_CFG, MODELS
from embedding.search import semantic_search
from embedding.reranker import rerank
from rag.answer_drafter import draft_answer

def production_answer(question: str) -> dict:
    """Eine Frage durch die produktive Pipeline schicken.

    Reproduziert pipeline_stream.py 1:1, aber synchron:
        semantic_search(top_k=50) -> rerank(top_k=10)
        -> Threshold-Filter 0.40 -> draft_answer().
    """
    t0 = time.perf_counter()
    hits = semantic_search(question, top_k=50,
                           use_reranker=False)
    t_search = time.perf_counter() - t0

    t0 = time.perf_counter()
    reranked = rerank(question, hits, top_k=10)
    filtered = [
        h for h in reranked
        if h.get("rerank_score", h.get("score", 0))
           >= RERANKER_CFG.score_threshold
    ]
    t_rerank = time.perf_counter() - t0

    t0 = time.perf_counter()
    draft = draft_answer(question, filtered)
    t_draft = time.perf_counter() - t0

    return {
        "question": question,
        "answer": draft.get("draft", ""),
        "contexts": [h.get("text") or h.get("embedding_text")]
    }

```

```

        for h in filtered],
    "source_breakdown": [h.get("source") for h in filtered],
    "latency_ms": {
        "search": round(t_search * 1000, 1),
        "rerank": round(t_rerank * 1000, 1),
        "draft": round(t_draft * 1000, 1),
    },
}

```

Listing A.7: Thin Wrapper über die produktive RAG-Pipeline für die RAGAS-Evaluation
(Auszug aus `react_app/eval/pipeline_wrapper.py`)

A.7.8. Lokaler RAGAS-Judge mit Llama-3.1-Chat-Template

Der in Abschnitt 5.5.2 beschriebene lokale Judge ist als LangChain-LLM-Adapter um eine resident gehaltene llama-cpp-python-Instanz herum implementiert. Entscheidend ist die Einbettung der RAGAS-Prompts in das Llama-3.1-Instruct-Chat-Format mit einem System-Turn, der das Modell explizit auf reines JSON-Output ohne Markdown-Hülle festlegt – ohne diese Disziplinierung würde RAGAS' Pydantic-Output-Parser auf unstrukturierte Antworten mit Parse-Fehlern reagieren.

```

class LlamaJudgeLLM(LLM):
    """Minimaler LangChain-LLM um Llama 3.1 8B Q4 GGUF."""

    model_path: str = str(JUDGE_MODEL_PATH)
    # 16k Tokens: RAGAS' context_recall- und
    # context_precision-Prompts mit bis zu 10 Chunks
    # sprengen den Standardwert 4096.
    n_ctx: int = 16384
    n_threads: int = 8
    n_gpu_layers: int = -1 # alle Layer auf Metal (M-Series)

    def _call(self, prompt: str, stop=None,
              run_manager=None, **kwargs):
        llama = _get_judge_llama()
        # Llama 3.1 Instruct Chat-Template mit JSON-Disziplin.
        formatted = (
            "<|start_header_id|>system<|end_header_id|>\n\n"
            "You are a careful evaluator. Follow the user's "
            "instructions and respond strictly in the requested "
            "format. When the user asks for JSON, output ONLY a "
            "valid JSON object -- no prose, no markdown fences, "
            "no commentary.<|eot_id|>"
            "<|start_header_id|>user<|end_header_id|>\n\n"
            f"{prompt}<|eot_id|>"

```

```

        "<|start_header_id|>assistant<|end_header_id|>\n\n"
    )
    stops = list(stop or []) + ["<|eot_id|>",
                                "<|end_of_text|>"]

    out = llama(
        formatted,
        max_tokens=kwargs.get("max_tokens", 512),
        temperature=kwargs.get("temperature", 0.01),
        stop=stops,
    )
    return out.get("choices", [{}])[0].get("text", "").strip()

```

Listing A.8: Lokaler RAGAS-Judge – LangChain-Adapter um Llama 3.1 8B Q4 (Auszug aus `react_app/eval/judge.py`)

A.8. RAGAS-Test-Set – repräsentative Beispiele

Die folgende Auswahl zeigt fünf Frage-Antwort-Paare aus dem in Abschnitt 5.5.1 beschriebenen Test-Set – je ein typisches Beispiel pro Quelle sowie ein scope-bezogener Edge-Case. Die vollständige Liste aller 20 Paare liegt in `react_app/eval/test_set.json`; die Konstruktionsmethodik ist in `react_app/eval/test_set_manifest.md` dokumentiert. Ground Truths sind hier zur besseren Lesbarkeit gekürzt; die unverkürzten Texte stehen im JSON-Dokument.

Die Auswahl deckt drei Fragetypen (prozedural, definitorisch, scope-bezogen) ab und enthält bewusst sowohl deutsche als auch englische Anfragen, um die in Tabelle 5.36 berichtete Balance zu illustrieren. Das scope-bezogene Beispiel T2 ist insofern instruktiv, als es eine *negativ formulierte* Frage darstellt – der korrekte Ground Truth besteht hier in einer Aufzählung dessen, was die Dokumentation *nicht* enthält, und prüft damit die Fähigkeit des Systems, scope-definierende Aussagen zu erkennen statt zu paraphrasieren.

ID	Quelle	Spr.	Typ	Frage / Ground Truth (gekürzt)
Z3	Zendesk-Tickets	DE	prozedural	<i>Frage:</i> „Beim Start von AutoCAD Map 3D bekommen unsere Nutzer Lizenzfehler. Wie wurde dieses Problem in der Vergangenheit gelöst?“ <i>Ground Truth:</i> Der Lizenzfehler beim AutoCAD-Start wurde durch einen Serverneustart behoben. Nach dem Neustart funktionierten die Lizenzen wieder bei allen betroffenen Nutzern.
H3	Help Center	DE	definitivisch	<i>Frage:</i> „Was bietet das Projektmanagement-Modul in NET Cloud und wofür ist es geeignet?“ <i>Ground Truth:</i> Das Modul ermöglicht das Behalten der Teilaufgaben eines Planungsprojektes im Überblick. Sein Funktionsumfang ist geringer als bei regulären Projektmanagement-Tools und eignet sich vor allem für die Grobplanung; zentrales Element ist der per Drag-and-Drop bedienbare Projektplan.
V3	Vitruvi	EN	definitivisch	<i>Frage:</i> „How many authentication methods does the Vitruvi API support?“ <i>Ground Truth:</i> The Vitruvi API currently supports three different authentication methods.
T1	TKI-Produkte-KB	DE	definitivisch	<i>Frage:</i> „Welche Aufgaben deckt NET Engineering ab und auf welcher Plattform basiert das Modul?“ <i>Ground Truth:</i> NET Engineering ist das Fachschalenmodul für Ausführungsplanung und Dokumentation von Telekommunikationsnetzen auf Basis von Autodesk AutoCAD Map 3D. Es überführt Strukturplanungen aus NET Design in die Feinplanung, dokumentiert Trassen, Glasfaser- und Rohrnetze und erzeugt alle erforderlichen Reports.
T2	TKI-Produkte-KB	DE	scope-bezogen	<i>Frage:</i> „Was ist in der TKI-NET-Cloud-Produktdokumentation explizit nicht enthalten?“ <i>Ground Truth:</i> Die Dokumentation beschreibt ausschließlich Standardfunktionen für Standard-Workspaces, NET Scan und NET Maps. Kunden- und Partnerspezifische Erweiterungen werden nicht berücksichtigt (Ausnahme: die Deutsche Telekom-Integration); auf interne Implementierungsdetails wird nicht detailliert eingegangen.

Tabelle A.7.: Fünf repräsentative Frage-Antwort-Paare aus dem RAGAS-Test-Set.