1. Troubleshooting-Manual (Kaffeemaschine)

1. Datenart

→ **semi-strukturiertes Dokument**
(Fließtext, Abschnitte, Überschriften, wechselnde Länge)

2. Zugriffsmuster

→ **read-heavy**
Das Manual wird oft gelesen, aber fast nie geändert.

3. Zugriffsszenario

→ **Chunk-Retrieval / Dokumentlese-Zugriffe**
Ein RAG-System arbeitet nicht mit dem ganzen Dokument, sondern mit Chunks.

4. Geeignete Datenbanken

MongoDB
– Flexible Speicherung von Text, Abschnitten und variabler Struktur
– Sehr gute Eignung für Dokumente im JSON-Format
– Schnelles Laden mehrerer Chunks per $in

Alternativ: PostgreSQL JSONB
– Gut, wenn zusätzlich SQL-Filter, Versionierung oder Volltext wichtig sind

Nicht geeignet: Redis, Timescale, klassische Tabellen
→ Keine strukturierten Dokumentabfragen, keine Chunks, keine Textsuche.

---

2. Embedding von chunk_42

1. Datenart

→ **Embedding / Vektordaten**

2. Zugriffsmuster

→ **ANN-Suche (Ähnlichkeitssuche)**
„Finde die nächsten 8 Chunks."

3. Zugriffsszenario

→ **Vektor-Retrieval**
Ein eingehender Kundentext wird in einen Vektor umgewandelt und gegen alle Embeddings gesucht.

4. Geeignete Datenbanken

pgvector (PostgreSQL-Erweiterung)
– Hohe Qualität bei Filter + ANN (z. B. Kategorie, Sprache, Version)
– Gute Performance und verlässlicher Indexaufbau

Alternativ: Qdrant, Milvus
– Hochoptimierte ANN-Engines

Nicht geeignet: MongoDB (lokal), Redis
→ ANN-Suche ist dort entweder langsam oder gar nicht möglich.

---

3. Rate-Limit-Counter für user_123

1. Datenart

→ **ephemerer Zustand (State)**
Kurzlebig, muss nicht gespeichert werden.

2. Zugriffsmuster

→ **ultra-low-latency, viele kleine INCR-Writes**

3. Zugriffsszenario

→ **Rate-Limit Zähler (INCR + TTL)**
Jeder Request erhöht den Zähler, TTL löscht ihn automatisch.

4. Geeignete Datenbanken

Redis
– Atomare INCR-Operation
– EXPIRE für automatische Löschung
– Sub-Millisekunden-Latenz
– Perfekt für Rate-Limits, Session-State und flüchtige Daten

Nicht geeignet: Postgres, Mongo
→ Kein TTL, hoher Overhead, zu langsam für Live-Rate-Limits.

---

4. Kundenprofil (Adresse, Verträge)

1. Datenart

→ **strukturierte Daten**

2. Zugriffsmuster

→ **read/write mixed (OLTP)**
Korrekte Updates und verlässliche Konsistenz sind Pflicht.

3. Zugriffsszenario

→ **Kundenprofil-Lookup + Update**

4. Geeignete Datenbanken

PostgreSQL
– ACID-Transaktionen
– Foreign Keys
– Constraints
– Sicheres Schreiben, sichere Aktualisierung

Nicht geeignet: MongoDB, Redis
→ Keine referenzielle Integrität, keine ACID-Transaktionen.

---

5. Chat-Verlauf einer aktiven Support-Session

1. Datenart

→ **semi-strukturiertes Dokument**, das kontinuierlich wächst

2. Zugriffsmuster

→ **append-only**

3. Zugriffsszenario

→ **Chat-Logging (Nachrichten anhängen)**

4. Geeignete Datenbanken

MongoDB
– $push für neue Nachrichten
– Flexible JSON-Struktur
– Effizientes Laden des gesamten Verlaufs

Nicht geeignet: Postgres (teure JSON-Updates), Redis (nicht persistent)

🟩 **Zusammenfassungstabelle**

Objekt	Datenart	Zugriffsmuster	Zugriffsszenario	Geeignete DB
Troubleshooting-Manual	Dokument	read-heavy	Chunk Retrieval	Mongo / JSONB

Objekt	Datenart	Zugriffsmuster	Zugriffsszenario	Geeignete DB
Embedding	Vektor	ANN-Suche	Vektor-Retrieval	pgvector / Qdrant
Rate-Limit Counter	ephemeral State	low-latency INCR	Rate-Limit	Redis
Kundenprofil	strukturiert	OLTP	Lookup + Update	Postgres
Chat-Verlauf	semi-strukturiert	append-only	Chat-Log	Mongo

🟥 Falsche, aber plausible Lösungen (mit Erklärung)

Diese fünf Beispiele zeigen typische Denkfehler — jede wirkt im ersten Moment logisch, ist aber technisch falsch.

❌ 1. „Manuals sind häufig genutzt → ab in Redis, das ist schneller."

Warum es falsch ist:

- Redis ist ein Key-Value-Store, kein Dokumentenspeicher
- Keine Suchen, keine Filter, keine Chunks, keine Textverarbeitung
- TTL/Eviction würde Inhalte löschen

Richtig: Mongo / JSONB

❌ 2. „Embeddings in Mongo speichern und dort ANN-Search machen."

Warum es falsch ist:

- Mongo kann lokal keine effiziente ANN-Suche
- Filtering + ANN gleichzeitig funktioniert in pgvector besser
- ANN-Indexes fehlen oder performen schwach

Richtig: pgvector / Qdrant

❌ 3. „Rate-Limits einfach als Tabelle in Postgres speichern."

Warum es falsch ist:

- Jeder Update = WAL + Disk-Sync
- Zu hohe Latenz
- Kein TTL
- Rate-Limits erfordern atomare, schnelle INCR-Operationen

Richtig: Redis

❌ 4. „Chatverläufe als JSON-Feld in Postgres speichern und einfach updaten."

Warum es falsch ist:

- Jeder $push ersetzt das gesamte JSON
- Hoher Update-Overhead
- Race-Conditions
- JSON-Felder sind nicht für wachsende Listen gedacht

Richtig: MongoDB

❌ 5. „Kundenprofile in Mongo speichern — JSON ist flexibel und praktisch."

Warum es falsch ist:

- keine ACID-Garantien
- keine Foreign Keys
- keine verlässliche Konsistenz
- falsches Modell für Kundendaten mit Vertragsbezug

Richtig: PostgreSQL

🟩 Perfekte Chunk-Modellierung (zur Klarstellung)

Die Mini-Aufgabe verwendet vereinfachte Chunk-Beispiele.
Für ein funktionierendes RAG-System reicht das **nicht**.

Ein vollständiger Chunk benötigt üblicherweise:

- chunk_id

- text
- doc_id
- chunk_num (Reihenfolge im Dokument)
- section_title
- product_family
- document_type (Manual, Troubleshooting etc.)
- version
- language
- visibility (öffentlich/intern)
- embedding
- created_at

Warum diese Felder wichtig sind:

- Filter auf relevante Produktfamilie
- Laden mehrerer aufeinanderfolgender Chunks
- Versionierung (v1.0 vs. v3.2)
- Unterscheidung der Dokumenttypen
- Mehrsprachige Inhalte
- Präzise Metadaten für das Retrieval
- Grundlage für hochwertige Antworten
