**Projektdokumentation – Onboarding-Assistent (RAG, watsonx.ai, PostgreSQL/pgvector)**

**1. Überblick**

Der Onboarding-Assistent beantwortet Fragen neuer Mitarbeitender mit Retrieval-Augmented Generation (RAG).  
Kernprinzip: *Antwort nur aus dokumentierter Faktengrundlage*. Dafür werden die Onboarding-Dokumente ingestiert, in Chunks zerlegt, semantisch als Vektoren (Embeddings) im PostgreSQL (mit pgvector) gespeichert und zur Laufzeit passend zu einer Nutzerfrage abgerufen und mit einem LLM (watsonx.ai) verfasst.

**Technologie-Stack**

* **Backend**: Python 3.11, FastAPI, httpx (async), Uvicorn
* **Vektor-DB**: IBM Databases for PostgreSQL (Managed), Extension **pgvector**
* **Vektormodelle**: watsonx.ai Runtime – ibm/granite-embedding-278m-multilingual (768D, ~512 Token Input)
* **LLM**: watsonx.ai Runtime – z. B. ibm/granite-3-8b-instruct
* **Frontend**: React + Vite (im selben Container gebuildet & ausgeliefert)
* **Container/Hosting**: IBM Cloud Code Engine (ohne Serverbetrieb)
* **Auth**: IAM API Key → Bearer Token (watsonx.ai), TLS-DB (sslmode=verify-full)
* **Infra as Env**: ENV-Variablen + Secrets in Code Engine

**2. Architektur**

**Datenfluss (high-level)**

1. **Ingest**: Dateien (PDF/DOCX/MD) → Text → **Chunking** (≈350 Token, Overlap) → **Embeddings** (watsonx.ai) → **Upsert** in public.documents (Postgres/pgvector).
2. **Frage**: User → **Embedding der Frage** → **Vektor-Suche** (cosine) + optional Keyword-Score → Top-K Chunks → **Prompting** → **LLM-Antwort** (mit Quellenangaben).
3. **Personalisierung**: Audience/Location‐Filter via metadata->>'audience' (z. B. boeblingen, muenchen, all).

**Backend-Module**

* app/rag.py – Retrieval, Promptbau, LLM-Orchestrierung
* app/embeddings.py – Embedding-Client für watsonx.ai
* app/llm.py – Text-Generation via watsonx.ai /ml/v1/text/generation
* app/db.py – psycopg-Connection + pgvector-Adapterregistrierung
* ingest/\* – Loader (PDF/DOCX/MD), Chunker, Ingest-Pipeline (idempotent)
* server.py/app.py – FastAPI-App, CORS, Health, Chat- und Admin-Endpunkte

**Wichtige Endpunkte**

**1. Health-Check**

* **GET /healthz**  
  Prüft, ob der Service läuft.  
  **Response:**  
  { "status": "healthy", "service": "Boardy Onboarding Assistant" }
* **GET /api/health**  
  Prüft, ob die API erreichbar ist.  
  **Response:**  
  { "status": "healthy", "api": "Boardy API" }

**2. Locations**

* **GET /api/locations**  
  Gibt eine Liste vordefinierter Standorte zurück.  
  **Response:**

[

  { "id": "boeblingen", "name": "IBM Böblingen", "description": "Entwicklungs- und Innovationszentrum." },

  ...

**3. Chat/RAG**

* **POST /v1/ask**  
  Fragt den Onboarding-Assistenten (RAG) mit einer Nutzerfrage und optionalem Chatverlauf.  
  **Request:**

{ "query": "string", "history": [ { "role": "user|assistant", "content": "string" } ] }

**Response:**

{ "answer": "string", "sources": [ { "title": "...", "doc\_id": "...", "chunk\_id": ... } ] }

**4. Speech to Text**

* **OPTIONS /api/speech-to-text**  
  CORS/Preflight für Speech-to-Text.
* **POST /api/speech-to-text**  
  Wandelt Audio (base64) in Text um.  
  **Request:**

{ "audio\_data": "base64string", "content\_type": "audio/wav" }

**Response:**  
{ "transcript": "...", "confidence": 1.0, "success": true }

**5. Text to Speech**

* **POST /api/text-to-speech**  
  Wandelt Text in Audio um.  
  **Request:**

{ "text": "string", "voice": "string" }

**Response:**  
Audio-File (audio/wav)

**6. Datei-Upload & Ingest**

* **POST /api/upload-file**  
  Lädt eine Datei hoch und speichert sie temporär.  
  **Request:**  
  Multipart-Form-Data (Datei) **Response:**  
  { "filename": "...", "success": true }
* **POST /api/ask-with-file**  
  Nutzt den Inhalt einer hochgeladenen Datei als temporären Kontext für eine Frage (ohne Ingest ins RAG).  
  **Request:**  
  Multipart-Form-Data:
  + [query](vscode-file://vscode-app/Applications/Visual%20Studio%20Code.app/Contents/Resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Text
  + [file](vscode-file://vscode-app/Applications/Visual%20Studio%20Code.app/Contents/Resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Datei  
    **Response:**  
    Antwort wie bei /v1/ask
* **POST /api/ingest-uploaded-file**  
  Führt den Ingest-Prozess für eine zuvor hochgeladene Datei aus (speichert sie ins RAG).  
  **Request:**  
  Form-Data:
  + [filename](vscode-file://vscode-app/Applications/Visual%20Studio%20Code.app/Contents/Resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Name der Datei  
    **Response:**  
    { "ingested": true, "filename": "..." }

**7. Frontend/Static Files**

* **GET /**  
  Liefert die Frontend-Startseite aus.
* **GET /{path:path}**  
  Liefert statische Dateien des Frontends aus.

**3. Datenbankdesign**

**Tabelle** public.documents

CREATE EXTENSION IF NOT EXISTS vector;

CREATE EXTENSION IF NOT EXISTS pg\_trgm; -- optional für Hybrid-Suche

CREATE TABLE IF NOT EXISTS public.documents (

id text PRIMARY KEY,

doc\_id text NOT NULL,

chunk\_id int NOT NULL,

content text NOT NULL,

metadata jsonb NOT NULL,

embedding vector(768) NOT NULL, -- Granite Embedding 278m: 768D

checksum text, -- sha1(content) für Idempotenz

token\_count int,

embedding\_dim int,

created\_at timestamptz DEFAULT now(),

updated\_at timestamptz

);

-- (doc\_id, chunk\_id) erzwingt Idempotenz

ALTER TABLE public.documents

ADD CONSTRAINT documents\_doc\_chunk\_unique UNIQUE (doc\_id, chunk\_id);

-- ANN-Index (cosine). 'lists' je nach Datenmenge anpassen (100–200)

CREATE INDEX IF NOT EXISTS documents\_embedding\_ivfflat

ON public.documents USING ivfflat (embedding vector\_cosine\_ops) WITH (lists = 100);

-- Metadaten-Index für Filter (Audience etc.)

CREATE INDEX IF NOT EXISTS documents\_gin\_metadata ON public.documents USING gin (metadata);

-- optional: Keyword-Komponente

CREATE INDEX IF NOT EXISTS documents\_trgm\_idx ON public.documents USING gin (content gin\_trgm\_ops);

**Hinweise**

* **Dimension** des Vektorfelds muss **Embedding-Modell** entsprechen (Granite 278m: **768**).
* Nach großen Ingests: ANALYZE public.documents;
* Für EU-Regionen ggf. Runtimes/Project in **eu-de** betreiben; hier: **us-south (Dallas)**.

**4. Ingest-Pipeline (ingest/\*)**

**4.1 Loader (ingest/loaders.py)**

* Unterstützt **.md**, **.pdf**, **.docx**
* Liest Text robust (PDF via pypdf, DOCX via python-docx)
* Setzt metadata.audience aus dem **Ordnernamen** (z. B. onboarding\_docs/boeblingen/...)

**4.2 Chunker (ingest/chunker.py)**

* Zielgröße: **350 Tokens**, Overlap **≈60** Tokens
* Hard-Cap: **≤500 Tokens** (Embedding-Limit 512)
* Pipeline: **Absätze → Sätze → Hard-Split**
* Token-Schätzung: konservativ **1 Token ~ 4 Zeichen**
* Ergebnis: stabile, LLM-fähige Chunks, die Embedding-Limits sicher einhalten.

**4.3 Idempotenz & Upsert (ingest/ingest.py)**

* **Deterministische ID**: id = md5(f"{doc\_id}#{chunk\_id}")
* **Checksum**: sha1(content); nur geänderte Chunks werden **neu** embeded
* **Upsert**: ON CONFLICT (doc\_id,chunk\_id) – verhindert Duplikate

**Batching & Limits**

* Embedding-Batchgröße: 32–64 Texte/Request bewährt
* Pre-Trim bei >500 Tokens als Sicherheitsnetz

**5. Retrieval & Prompting (RAG)**

**5.1 Retrieval-SQL (Vektor-Suche + Audience-Filter)**

-- nur Vektor-Score

SELECT id, doc\_id, chunk\_id, content, metadata

FROM public.documents

WHERE (metadata->>'audience' IS NULL)

OR (metadata->>'audience' = $1) -- audience (z. B. "muenchen")

OR (metadata->>'audience' = 'all')

ORDER BY embedding <=> $2 -- $2 = pgvector.Vector(query\_embedding)

LIMIT $3;

**Hybrid-Variante (optional):** Vektor-Score + Keyword via Trigram

SELECT id, doc\_id, chunk\_id, content, metadata,

(0.7 \* (1 - (embedding <=> $2)) + 0.3 \* similarity(content, $4)) AS score

FROM public.documents

WHERE (metadata->>'audience' IS NULL)

OR (metadata->>'audience' = $1)

OR (metadata->>'audience' = 'all')

ORDER BY score DESC

LIMIT $3;

**5.2 Promptgestaltung**

**System-Prompt (Beispiel, deutsch & konservativ)**

„Du bist ein Onboarding-Assistent des Unternehmens. Antworte **kurz**, **korrekt** und **nur** auf Basis des bereitgestellten Kontexts (Quellen). Wenn etwas fehlt/unklar ist, weise transparent darauf hin und nenne den Eskalationsweg. Füge am Ende **„Quellen: <Titel#Chunk, …>“** an.“

**Optimierter - System-Prompt**

"Du bist ein empathischer Onboarding-Assistent der Firma. Antworte kurz, korrekt, auf Deutsch. Nutze bevorzugt die bereitgestellten Kontexte, aber du darfst auch auf eigenes Wissen zurückgreifen, wenn du dir sicher bist. Wenn du unsicher bist, sage dies klar und schlage einen Eskalationsweg vor. Nenne die Quellen (Titel#Chunk) nur, wenn du dir sehr sicher bist, dass die Information aus dem Kontext stammt. Du darfst empathisch reagieren und Smalltalk führen, wenn es zur Situation passt."

**User-Prompt**

FRAGE:

{question}

KONTEXT (verwende NUR Folgendes):

[filename#chunk\_id] {content}

...

ANTWORTFORMAT:

- knappe Antwort auf Deutsch

- bei Prozessen: nummerierte Schritte

- Abschlusszeile: 'Quellen: <Titel#Chunk, ...>'

**6. watsonx.ai – API-Details**

* **Base URL**: WATSONX\_BASE\_URL=https://us-south.ml.cloud.ibm.com (Region beachten)
* **Version**: WATSONX\_API\_VERSION=2024-05-01
* **Embeddings**: POST {base}/ml/v1/text/embeddings?version=...
  + Body: { "model\_id": "...","project\_id":"...","inputs":[ "...", ... ] }
  + Manche Deployments erwarten texts statt inputs → Fallback im Client implementiert.
  + Response üblicherweise { "data":[{"embedding":[...]}] }
* **Text-Generation**: POST {base}/ml/v1/text/generation?version=...
  + Body: { "model\_id":"...","project\_id":"...","input":"PROMPT","parameters":{...} }
  + Response: { "results":[{"generated\_text":"..."}] }
* **Auth**: IAM API Key → Bearer Token (Token-Manager/Cache im Code vorsehen)

**Typische Fehler & Fixes**

* 404 /ml/v1/text/generate → **richtiger Pfad** ist /generation.
* 400 invalid\_input\_argument ... texts → Payload-Feld von inputs auf texts wechseln.
* 403 project\_id ... not associated → Runtime-Service muss im watsonx-**Projekt** verknüpft sein.
* 400 token sequence length ... 512 → Chunking härter (≤500 Tokens) oder kürzen.

**7. Konfiguration (ENV)**

**Minimal**

WATSONX\_BASE\_URL=https://us-south.ml.cloud.ibm.com

WATSONX\_API\_VERSION=2024-05-01

WATSONX\_PROJECT\_ID=<GUID>

WATSONX\_API\_KEY=<secret>

EMBEDDINGS\_MODEL\_ID=ibm/granite-embedding-278m-multilingual

LLM\_MODEL\_ID=ibm/granite-3-8b-instruct

DATABASE\_URL=postgres://USER:PASS@HOST:PORT/DB?sslmode=verify-full&sslrootcert=system

CORS\_ORIGINS= <https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud>, http://localhost:5173

PORT=8000

In **Code Engine**: API-Key als Secret, DB-URL als Env, Zertifikat (CA-Cert Secret) als Volume.

**8. Deployment (IBM Cloud Code Engine)**

**Image builden & pushen**

ibmcloud cr region-set us-south

ibmcloud cr namespace-add boardy

export IMAGE=us.icr.io/boardy/boardy-onboarding:$(date +%Y%m%d%H%M)

docker build -t $IMAGE .

ibmcloud cr login

docker push $IMAGE

**App erstellen**

ibmcloud ce project create --name boardy-prj --region us-south

ibmcloud ce project select --name boardy-prj

ibmcloud ce app create \

--name boardy-app \

--image $IMAGE \

--port 8000 \

--cpu 1 --memory 1G \

--min-scale 0 --max-scale 3

**Secrets & ENV**

ibmcloud ce secret create --name wxaikey --from-literal WATSONX\_API\_KEY=<key>

ibmcloud ce app bind --name boardy-app --secret wxaikey

ibmcloud ce app env set --name boardy-app --env WATSONX\_BASE\_URL=https://us-south.ml.cloud.ibm.com

ibmcloud ce app env set --name boardy-app --env WATSONX\_API\_VERSION=2024-05-01

ibmcloud ce app env set --name boardy-app --env WATSONX\_PROJECT\_ID=<project-id>

ibmcloud ce app env set --name boardy-app --env EMBEDDINGS\_MODEL\_ID=ibm/granite-embedding-278m-multilingual

ibmcloud ce app env set --name boardy-app --env LLM\_MODEL\_ID=ibm/granite-3-8b-instruct

ibmcloud ce app env set --name boardy-app --env 'DATABASE\_URL=postgres://USER:PASS@HOST:PORT/DB?sslmode=verify-full&sslrootcert=system'

ibmcloud ce app env set --name boardy-app --env 'CORS\_ORIGINS= <https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud/>, http://localhost:5173'

**Smoke-Tests**

curl https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud/healthz

curl -X POST https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud/v1/ask \

-H "Content-Type: application/json" \

-d '{"query":"Wie beantrage ich Urlaub?"}'

**9. Betrieb, Sicherheit, Qualität**

**Sicherheit**

* TLS für DB (sslmode=verify-full, System-Truststore oder root.crt)
* Secrets niemals ins Image backen → Code Engine Secrets nutzen
* Least-Privilege-DB-Role (ibm-cloud-base-user Grants nur auf Schema/Tabelle)
* Logging: keine PII in Prompts/Logs persistieren

**Observability**

* Code Engine Logs (ibmcloud ce app logs --follow)
* Admin-Endpunkte (/admin/rag/stats, /admin/rag/sample)
* Metriken: Ingest-Dauer, Embedding-Latenz, Tokenverbrauch, Trefferquote (Top-K Overlap)

**Qualität**

* Golden-Set von ~20 Onboarding-Fragen (Erwartete Quelle & Antwort)
* Regression nach jedem Ingest/Modellwechsel
* Prompt-A/B-Tests (z. B. knapper/strenger Output vs. ausführlicher)

**Performance-Tuning**

* ivfflat lists erhöhen bei vielen Vektoren (100→200/400)
* k=6–8 meist gut, Re-Rank auf 12–16 Kandidaten
* Pre-Filter (Audience, Doc-Typ) **vor** ANN-Sortierung
* Caching von Query-Embeddings (z. B. 60s LRU)

**Fehlerbilder & Troubleshooting**

* UndefinedFunction: operator vector <=> double precision[]  
  → register\_vector(conn) vergessen oder Vector(emb) nicht benutzt.
* permission denied for table documents  
  → Grants auf ibm-cloud-base-user setzen.
* no\_associated\_service\_instance\_error  
  → watsonx-Projekt ↔ Runtime korrekt verknüpfen.
* path\_not\_found\_error /text/generate  
  → Endpunkt ist /text/generation.

**10. Erweiterungen & Roadmap**

* **Code-Engine-Job** für geplanten Ingest (CRON)
* **Re-Ranking** mit Cross-Encoder oder LLM-Score
* **Access Control** (Rollen/Audiences: HR, IT, Standort)
* **Feedback-Loop**: „War hilfreich?“ + Korrekturen → Redaktionsworkflow
* **Evaluation**: RAGAS oder eigene KPIs (Faithfulness, Context-Recall)
* **Ingest per Chat ansatt nur mit Dokumenten**
* Gamification?
* **RAG Datenbank strikt nach der Lokation trennen**. Abfrage im Homescreen soll Umgebungsvariable für RAG Endpunkt ändern.

**11. Betriebshandbuch (Kurz)**

* **Neues Dokument hinzufügen**  
  Datei nach onboarding\_docs/<audience>/ legen →  
  python -m ingest.ingest "<ABSOLUTER\_PFAD>"  
  (idempotent; nur neue/änderte Chunks werden neu eingebettet)
* **DB prüfen**
* SELECT COUNT(\*) FROM public.documents;
* SELECT doc\_id, chunk\_id, left(content,80), metadata->>'audience'
* FROM public.documents ORDER BY updated\_at DESC NULLS LAST LIMIT 10;
* **RAG testen**
  + curl -X POST https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud/v1/ask \

-H "Content-Type: application/json" \

-d '{"query":"Wie beantrage ich Urlaub?","user":{"location":"boeblingen"}}'

* **Deployment updaten**
  + export IMAGE=us.icr.io/boardy/boardy-onboarding:$(date +%Y%m%d%H%M)
  + docker build -t $IMAGE .
  + docker push $IMAGE
  + ibmcloud ce app update --name boardy-app --image $IMAGE

**12. Beispiel-Flow**

**Anfrage:** „  
curl -sS -X POST -H 'Content-Type: application/json' \  
-d '{"query":"Ich fühle mich gestresst."}' \  
https://boardy-backend-app.1zt0zkzab8pz.eu-de.codeengine.appdomain.cloud/v1/ask“

Die Anfrage wird entgegengenommen, relevante Kontexte werden gesucht, ein Prompt gebaut, das LLM antwortet empathisch, und die Quellen werden transparent gemacht. Das Ergebnis wird als JSON zurückgegeben.

**1. Empfang der Anfrage**

* Die FastAPI-Anwendung erhält einen POST-Request an /v1/ask mit dem JSON-Body.
* Die Anfrage wird als AskRequest geparst.

**2. Aufruf der RAG-Logik**

* Die Funktion ask\_rag in server.py ruft die Funktion answer aus rag.py auf:

result = await rag\_answer(req.query, chat\_history=req.history)

**3. Kontext-Retrieval**

* In rag.py wird die Funktion retrieve aufgerufen:
  + Die Query "Ich fühle mich gestresst." wird per Embedding in einen Vektor umgewandelt.
  + Mit diesem Vektor wird eine Ähnlichkeitssuche in der Datenbank (documents-Tabelle) durchgeführt.
  + Die relevantesten Chunks (max. 6) werden geladen.

**4. Prompt-Erstellung**

* Die gefundenen Kontexte werden mit der Nutzerfrage zu einem Prompt für das LLM kombiniert (format\_prompt).
* Falls kein Kontext gefunden wird, wird ein Fallback-Prompt erstellt, der das LLM anweist, aus eigenem Wissen zu antworten.

**5. Chatverlauf**

* Falls ein Chatverlauf (history) mitgegeben wurde, wird dieser in den Prompt eingebaut (hier: leer).

**6. Antwortgenerierung**

* Das LLM (WatsonxAILLM) erhält den System-Prompt und den zusammengesetzten Prompt.
* Das LLM generiert eine empathische Antwort auf Deutsch.

**7. Quellenangabe**

* Die verwendeten Kontext-Chunks werden als Quellen aufbereitet (Titel, doc\_id, chunk\_id).

**8. Response**

* Die Antwort und die Quellen werden als JSON zurückgegeben:

{

  "answer": "...",

  "sources": [

    { "title": "...", "doc\_id": "...", "chunk\_id": ... },

    ...

  ]

}

**13. Zusammenfassung**

* **RAG** sorgt für faktenbasierte Antworten mit **Quellenbezug**.
* **watsonx.ai** liefert **Embeddings** & **LLM-Antworten**, die Daten (Vektoren & Chunks) liegen **ausschließlich** in der **PostgreSQL/pgvector**-Instanz.
* **Idempotenter Ingest** + **Audience-Filter** → personalisierte, wiederholbar korrekte Ergebnisse.
* **Code Engine** vereinfacht Betrieb, Skalierung und Secret-Handling.
* Das System ist modular, erweiterbar und in Produktion gut zu betreiben (Logging, Metriken, Tests).