give me download able document or md file for above document

Here is a complete, ready-to-save Markdown document that consolidates the updated code, file roles, deployment steps, and how everything works. Save it as KnowledgeBot-README.md in the project root.

KnowledgeBot – Hybrid RAG Assistant

KnowledgeBot combines fast, deterministic responses for simple queries with high-quality, context-grounded answers for document questions using an improved RAG pipeline. It supports per-page PDF ingestion, BM25 retrieval with query expansion and version boosts, robust Ollama handling, and a streamlined Gradio UI.

Highlights

- Per-page PDF parsing for precise, page-cited answers.
- BM25 retrieval with synonyms and version boosts (e.g., 1.9.0/v1.9.0).
- Robust Ollama client (handles streaming and non-streaming JSON).
- Stepwise answers for install/procedure queries; concise answers otherwise.
- Simple UI that routes all queries through the improved RAG path.

Updated Files

Use these as the canonical, working files for this project.

1) src/knowledge_bot/tools/custom_tool.py

- Reads multiple file formats with graceful fallbacks.
- PDF is parsed per-page (pdfplumber preferred; PyPDF2 fallback).
- Restores GitHubRepoCloner and GoogleDriveReader placeholders to avoid import errors.

```
import os
from typing import List, Dict, Any

# Optional imports with graceful fallbacks
try:
    import pdfplumber
    HAS_PDFPLUMBER = True
except Exception:
    HAS_PDFPLUMBER = False
```

```
try:
    import PyPDF2
    HAS_PYPDF2 = True
except Exception:
    HAS_PYPDF2 = False

try:
    import docx
    HAS_DOCX = True
except Exception:
    HAS_DOCX = False

try:
    import pandas as pd
    HAS_PANDAS = True
except Exception:
    HAS_PANDAS = False

SUPPORTED_LOCAL = {".pdf", ".docx", ".txt", ".csv", ".xlsx", ".md"}

def _read_pdf_per_page(path: str) -> List[Dict[str, Any]]:
    docs: List[Dict[str, Any]] = []
    name = os.path.basename(path)

    if HAS_PDFPLUMBER:
        try:
            with pdfplumber.open(path) as pdf:
                for i, page in enumerate(pdf.pages, start=1):
                    try:
                        text = page.extract_text() or ""
                    except Exception:
                        text = ""
                    if text.strip():
                        st = os.stat(path)
                        docs.append({
                            "content": text,
                            "metadata": {
                                "source": path,
                                "path": path,
                                "name": name,
                                "type": ".pdf",
                                "size": st.st_size,
                                "last_modified": st.st_mtime,
                                "page": i,
                                "page_count": len(pdf.pages),
                            }
                        })
        except Exception:
            pass

    if not docs and HAS_PYPDF2:
        try:
            with open(path, "rb") as f:
                reader = PyPDF2.PdfReader(f)
                for i, page in enumerate(reader.pages, start=1):
```

```
                    try:
                        text = page.extract_text() or ""
                    except Exception:
                        text = ""
                    if text.strip():
                        st = os.stat(path)
                        docs.append({
                            "content": text,
                            "metadata": {
                                "source": path,
                                "path": path,
                                "name": name,
                                "type": ".pdf",
                                "size": st.st_size,
                                "last_modified": st.st_mtime,
                                "page": i,
                                "page_count": len(reader.pages),
                            }
                        })
        except Exception:
            pass

    return docs

def _read_docx(path: str) -> str:
    if not HAS_DOCX:
        return ""
    try:
        d = docx.Document(path)
        return "\n".join(p.text for p in d.paragraphs)
    except Exception:
        return ""

def _read_txt(path: str) -> str:
    try:
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            return f.read()
    except Exception:
        return ""

def _read_csv(path: str) -> str:
    if not HAS_PANDAS:
        return _read_txt(path)
    try:
        df = pd.read_csv(path)
        return df.head(1000).to_string(index=False)
    except Exception:
        return ""

def _read_xlsx(path: str) -> str:
    if not HAS_PANDAS:
        return ""
    try:
        df = pd.read_excel(path)
        return df.head(1000).to_string(index=False)
    except Exception:
```

```
        return ""

def LocalFileReader(folder: str) -> List[Dict[str, Any]]:
    docs: List[Dict[str, Any]] = []
    if not folder or not os.path.isdir(folder):
        return docs

    for root, _, files in os.walk(folder):
        for fn in files:
            ext = os.path.splitext(fn)[1].lower()
            if ext not in SUPPORTED_LOCAL:
                continue
            full_path = os.path.join(root, fn)

            try:
                if ext == ".pdf":
                    docs.extend(_read_pdf_per_page(full_path))
                else:
                    if ext == ".docx":
                        content = _read_docx(full_path)
                    elif ext in {".txt", ".md"}:
                        content = _read_txt(full_path)
                    elif ext == ".csv":
                        content = _read_csv(full_path)
                    elif ext == ".xlsx":
                        content = _read_xlsx(full_path)
                    else:
                        content = ""

                    if content and content.strip():
                        st = os.stat(full_path)
                        docs.append({
                            "content": content,
                            "metadata": {
                                "source": full_path,
                                "path": full_path,
                                "name": fn,
                                "type": ext,
                                "size": st.st_size,
                                "last_modified": st.st_mtime,
                            }
                        })
            except Exception:
                continue

    return docs

def GitHubRepoCloner(repo_url: str) -> tuple[str, List[Dict[str, Any]]]:
    if not repo_url:
        return "", []
    return "", []

def GoogleDriveReader(folder_id: str) -> List[Dict[str, Any]]:
    if not folder_id:
        return []
    return []
```

2) src/knowledge_bot/crew_simple.py

- Intent routing (personal_info, list_files, list_projects, rag_query).
- Chunking plus BM25 retrieval with synonyms/version boosts (keyword fallback).
- Page-aware contexts and robust Ollama client (stream + non-stream).
- Procedure queries return numbered, actionable steps where possible.

```
from typing import Any, Dict, List, Tuple
import os, re, json, requests

# Optional BM25 scorer
try:
    from rank_bm25 import BM25Okapi
    HAS_BM25 = True
except Exception:
    HAS_BM25 = False

def _detect_intent(q: str) -> str:
    ql = (q or "").lower().strip()
    if any(x in ql for x in ["who am i", "what is my name", "who i am"]) and "interest" r
        return "personal_info"
    if "list" in ql and (".py" in ql or ".md" in ql):
        return "list_files"
    if ("project" in ql or "projects" in ql) and any(y in ql for y in ["list", "show", "t
        return "list_projects"
    return "rag_query"

def _extract_versions(q: str) -> List[str]:
    versions = set(re.findall(r"\bv?\d+(?:[._]\d+){1,3}\b", (q or "").lower()))
    normalized = {v.replace("_", ".") for v in versions}
    return list(versions | normalized)

def _expand_query_terms(q: str) -> List[str]:
    ql = (q or "").lower()
    tokens = set(re.findall(r"\w+", ql))
    mapping = {
        "install": {"installation", "setup", "configure", "deploy", "deployment", "instal
        "procedure": {"procedure", "steps", "runbook", "guide", "howto"},
        "pod": {"pod", "pods"},
        "max": {"max", "m.a.x", "m_a_x"},
        "release": {"release", "version"},
        "ecp": {"ecp"},
    }
    for t in list(tokens):
        if t in mapping:
            tokens |= mapping[t]
    for v in _extract_versions(q):
        tokens.add(v)
        tokens.add(v.lstrip("v"))
        tokens.add(v.replace("_", "."))
        tokens.add(v.replace(".", "_"))
    return list(tokens)

def _chunk_text(text: str, chunk_size: int = 700, overlap: int = 120) -> List[str]:
```

```
    if not text:
        return []
    out, n, start = [], len(text), 0
    while start < n:
        end = min(n, start + chunk_size)
        out.append(text[start:end])
        if end >= n:
            break
        start = max(0, end - overlap)
    return out

def _load_documents_from_folder(folder_path: str) -> List[Dict[str, Any]]:
    try:
        import sys
        sys.path.append("./src")
        from knowledge_bot.tools.custom_tool import LocalFileReader

        raw_docs = LocalFileReader(folder_path)
        print(f"[RAG] Loaded {len(raw_docs)} documents from {folder_path}")

        chunked: List[Dict[str, Any]] = []
        for d in raw_docs:
            content = (d.get("content") or "").strip()
            meta = d.get("metadata", {})
            if not content or len(content) < 50:
                continue
            chunks = _chunk_text(content, 700, 120)
            for i, ch in enumerate(chunks):
                md = meta.copy()
                md["chunk_id"] = i
                md["total_chunks"] = len(chunks)
                md["name"] = md.get("name") or os.path.basename(md.get("path", "") or "")
                chunked.append({"content": ch, "metadata": md})

        print(f"[RAG] Created {len(chunked)} chunks")
        for c in chunked[:2]:
            md = c["metadata"]
            print(f"[RAG] Example chunk from {md.get('name', 'unknown')} page={md.get('pa
        return chunked
    except Exception as e:
        print(f"[RAG] Error loading documents: {e}")
        import traceback; traceback.print_exc()
        return []

def _bm25_search(query: str, docs: List[Dict[str, Any]], top_k: int = 7) -> List[Tuple[f]
    if not HAS_BM25 or not docs:
        return []
    tokenized_corpus = [re.findall(r"\w+", (d["content"] or "").lower()) for d in docs]
    bm25 = BM25Okapi(tokenized_corpus)
    expanded = _expand_query_terms(query)
    scores = [bm25.get_scores(re.findall(r"\w+", t)) for t in expanded]
    avg_score = [sum(vals) / max(1, len(vals)) for vals in zip(*scores)]

    versions = _extract_versions(query)
    phrase = (query or "").lower()
    scored: List[Tuple[float, Dict[str, Any]]] = []
```

```
    for idx, d in enumerate(docs):
        s = avg_score[idx]
        text = (d["content"] or "").lower()

        for v in versions:
            if v in text or v.lstrip("v") in text or v.replace(".", "_") in text:
                s += 2.0

        if any(k in phrase for k in ["install", "installation", "steps", "procedure", "ru
            if any(k in text for k in ["install", "installation", "steps", "procedure", '
                s += 1.5

        if any(k in text for k in ["prerequisites", "requirements", "overview", "installa
            s += 0.5

        scored.append((s, d))

    scored.sort(key=lambda x: x[0], reverse=True)
    return scored[:top_k]

def _simple_search(query: str, docs: List[Dict[str, Any]], top_k: int = 7) -> List[Tuple[
    q_terms = set(re.findall(r"\w+", (query or "").lower()))
    out: List[Tuple[float, Dict[str, Any]]] = []
    for d in docs:
        text = (d["content"] or "").lower()
        if not text:
            continue
        c_terms = set(re.findall(r"\w+", text))
        overlap = len(q_terms & c_terms)
        if overlap <= 0:
            continue
        phrase_bonus = 0.2 * sum(text.count(t) for t in q_terms if t in text)
        out.append((overlap + phrase_bonus, d))
    out.sort(key=lambda x: x[0], reverse=True)
    return out[:top_k]

def _compose_contexts(chosen: List[Dict[str, Any]]) -> List[str]:
    contexts: List[str] = []
    for d in chosen:
        md = d.get("metadata", {})
        name = md.get("name", "unknown")
        page = md.get("page")
        header = f"Source: {name}" + (f" (page {page})" if page else "")
        contexts.append(f"{header}\n\n{d['content']}")
    return contexts

def _ollama_chat_stream_or_json(base_url: str, payload: Dict[str, Any], timeout_connect:
    url = f"{base_url.rstrip('/')}/api/chat"
    payload = dict(payload)
    payload.setdefault("stream", True)
    try:
        with requests.post(url, json=payload, stream=True, timeout=(timeout_connect, time
            r.raise_for_status()
            ctype = (r.headers.get("Content-Type") or "").lower()

            if "application/json" in ctype and not r.headers.get("Transfer-Encoding") ==
```

```
                jd = r.json()
                return (jd.get("message", {}) or {}).get("content", "") or jd.get("respon

            parts: List[str] = []
            for line in r.iter_lines(decode_unicode=True):
                if not line:
                    continue
                try:
                    obj = json.loads(line)
                except json.JSONDecodeError:
                    continue
                msg = obj.get("message") or {}
                content = msg.get("content") or ""
                if content:
                    parts.append(content)
                if obj.get("done"):
                    break
            return "".join(parts).strip()
    except Exception as e:
        print(f"[OLLAMA] Error: {e}")
        return ""

def _generate_simple_answer(query: str, contexts: List[str]) -> str:
    if not contexts:
        return "No relevant context was found to answer this question."
    ql = (query or "").lower()
    if any(k in ql for k in ["install", "installation", "steps", "procedure", "runbook",
        lines = []
        for ctx in contexts:
            for line in ctx.splitlines():
                if re.search(r"^\s*\d+[\.\)]\s+", line) or any(k in line.lower() for k in
                    lines.append(line.strip())
        if lines:
            return "Installation steps derived from context:\n- " + "\n- ".join(lines[:20
    return f"Based on the context:\n\n{contexts[0][:900]}"

def _generate_answer_with_ollama(query: str, contexts: List[str], base_url: str = None, m
    base_url = base_url or os.getenv("OLLAMA_BASE_URL", "http://ollama-service.ollama.svc
    if not contexts:
        return "No relevant context was found to answer this question."

    context_text = "\n\n---\n\n".join(contexts[:3])[:3500]
    wants_steps = any(k in (query or "").lower() for k in ["install", "installation", "st
    style_instr = (
        "Return numbered, actionable steps with prerequisites and post-checks. Cite the p
        if wants_steps else
        "Return a concise, factual answer grounded only in the context. "
    )
    prompt = (
        f"{style_instr}"
        "If the context does not contain the answer, say exactly 'Not found in the provid
        f"Context:\n{context_text}\n\nQuestion: {query}\nAnswer:"
    )
    payload = {
        "model": model,
        "messages": [
```

```
            {"role": "system", "content": "You answer only from the provided context and
            {"role": "user", "content": prompt},
        ],
        "stream": True,
        "options": {"temperature": 0.2, "num_predict": 700, "num_ctx": 4096},
    }
    print(f"[OLLAMA] Querying {base_url} with model {model}")
    answer = _ollama_chat_stream_or_json(base_url, payload)
    if answer:
        return answer.strip()
    print("[OLLAMA] Falling back to simple answer")
    return _generate_simple_answer(query, [c.split("\n\n", 1)[-1] for c in contexts])

def process_query_direct(query: str, folder_path: str = "./knowledge") -> Dict[str, Any]:
    folder_path = os.path.abspath(folder_path)
    intent = _detect_intent(query)

    print(f"[DIRECT] Processing: '{query}'")
    print(f"[DIRECT] Intent: {intent}")
    print(f"[DIRECT] Folder: {folder_path}")

    if intent == "personal_info":
        try:
            pref = os.path.join(folder_path, "user_preference.txt")
            if os.path.exists(pref):
                with open(pref, "r", encoding="utf-8") as f:
                    for line in f:
                        if line.strip().lower().startswith("name:"):
                            return {"answer": line.split(":", 1)[1].strip(),
                                    "sources": [{"intent": "personal_info", "file": "user
        except Exception as e:
            print(f"[DIRECT] Profile read error: {e}")
        return {"answer": "Kapil", "sources": [{"intent": "personal_info", "source": "fal

    if intent == "list_files":
        base = folder_path
        m = re.search(r"\{([^}]+)\}", query or "")
        if m:
            base = m.group(1).strip()
        files: List[str] = []
        try:
            for root, _, fns in os.walk(base):
                for name in fns:
                    if name.lower().endswith((".py", ".md")):
                        files.append(os.path.join(root, name))
        except Exception:
            pass
        return {"answer": "\n".join(files) if files else f"No .py or .md files found in {
                "sources": [{"intent": "list_files", "base_path": base, "count": len(file

    if intent == "list_projects":
        base = folder_path
        m = re.search(r"\{([^}]+)\}", query or "")
        if m:
            base = m.group(1).strip()
        projects: List[str] = []
```

```
        try:
            for root, dirs, files in os.walk(base):
                if any(x in files for x in ["README.md", ".git", "requirements.txt", "pac
                    projects.append(root)
                    dirs[:] = []
                if root.count(os.sep) - base.count(os.sep) >= 3:
                    dirs[:] = []
        except Exception:
            pass
        return {"answer": "\n".join(projects) if projects else f"No projects found in {ba
                "sources": [{"intent": "list_projects", "base_path": base, "count": len(p

    docs = _load_documents_from_folder(folder_path)
    if not docs:
        return {"answer": "No documents found to search. Please add files to your knowled
                "sources": [{"intent": "rag_no_docs"}]}

    chosen_scored: List[Tuple[float, Dict[str, Any]]] = []
    if HAS_BM25:
        chosen_scored = _bm25_search(query, docs, top_k=7)
    if not chosen_scored:
        chosen_scored = _simple_search(query, docs, top_k=7)

    if not chosen_scored:
        return {"answer": f"I couldn't find information about '{query}' in your documents
                "sources": [{"intent": "rag_no_results"}]}

    chosen_docs = [d for _, d in chosen_scored[:5]]
    contexts = _compose_contexts(chosen_docs)
    base_url = os.getenv("OLLAMA_BASE_URL", "").strip() or None
    answer = _generate_answer_with_ollama(query, contexts, base_url=base_url, model=os.ge

    sources = []
    for d in chosen_docs:
        md = d.get("metadata", {})
        sources.append({
            "name": md.get("name") or os.path.basename(md.get("path", "") or ""),
            "page": md.get("page"),
            "path": md.get("path"),
        })
    return {"answer": answer, "sources": sources + [{"intent": "rag_query", "chunks_used"
```

3) src/knowledge_bot/main.py

- Simple UI that always uses the improved direct processor.
- Optional Ollama URL field to quickly point at NodePort or service.
- Annotates sources with page numbers in the UI.

```
import os
import socket
import warnings
import gradio as gr
from dotenv import load_dotenv
```

```
from knowledge_bot.crew_simple import process_query_direct

load_dotenv()
warnings.filterwarnings("ignore")
DEFAULT_KNOWLEDGE_DIR = "./knowledge"

def find_free_port(start_port=7861):
    for port in range(start_port, start_port + 10):
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.bind(("0.0.0.0", port))
                return port
        except OSError:
            continue
    return 7870

def kickoff_pipeline(
    query,
    folder_path,
    github_url,
    model,
    save_markdown,
    embed_model,
    use_drive,
    drive_folder_choice,
    top_k,
    knowledge_dir,
    profile_name,
    answer_sink_path,
    answer_max_tokens,
    continue_segments,
    num_ctx,
    ollama_url
):
    if not query or not query.strip():
        return "Please enter a query."

    knowledge_path = (knowledge_dir or DEFAULT_KNOWLEDGE_DIR).strip()
    if knowledge_path:
        os.environ["KNOWLEDGE_DIR"] = os.path.abspath(knowledge_path)

    if ollama_url and ollama_url.strip():
        os.environ["OLLAMA_BASE_URL"] = ollama_url.strip()
    if model and str(model).strip():
        os.environ["KB_MODEL"] = str(model).strip()

    folder_to_use = folder_path or os.path.abspath(knowledge_path)
    print(f"[MAIN] Processing query: '{query}'")
    print(f"[MAIN] Using folder: {folder_to_use}")

    try:
        result = process_query_direct(query, folder_to_use)
        answer = result.get("answer", "No answer generated")
        sources = result.get("sources", [])
        pages = []
        for s in sources:
```

```
            nm = s.get("name")
            pg = s.get("page")
            if nm:
                pages.append(nm if pg is None else f"{nm} (p.{pg})")
        if pages:
            answer += f"\n\n Sources: {', '.join(pages[:5])}"

        if save_markdown and save_markdown.strip():
            try:
                with open(save_markdown.strip(), "w", encoding="utf-8") as f:
                    f.write(f"# Query\n{query}\n\n# Answer\n{answer}\n")
                answer += f"\n\n Saved to: {save_markdown.strip()}"
            except Exception as e:
                answer += f"\n\n✖ Save error: {e}"
        return answer
    except Exception as e:
        return f"Error processing query: {e}"

with gr.Blocks(title="KnowledgeBot") as demo:
    gr.Markdown("#  KnowledgeBot")
    gr.Markdown("Ask questions about your documents, list files/projects, or get personal

    query = gr.Textbox(label="Your Question", lines=2, placeholder="Try: who am i? | inst
    with gr.Row():
        folder = gr.Textbox(label="Folder path", value="", placeholder="Leave empty to us
        github = gr.Textbox(label="GitHub repo URL (optional)", value="")
    with gr.Row():
        model = gr.Dropdown(choices=["mistral", "llama3.2", "llama3.2:1b", "llama2", "gem
        embed_model = gr.Dropdown(choices=["mistral", "nomic-embed-text", "mxbai-embed-la
        top_k = gr.Slider(1, 20, value=5, step=1, label="Retrieval count (top-k)")
    with gr.Row():
        knowledge_dir = gr.Textbox(label="Knowledge directory", value=DEFAULT_KNOWLEDGE_D
        profile_name = gr.Textbox(label="Profile name (optional)", value="")
    with gr.Row():
        use_drive = gr.Checkbox(label="Use Google Drive", value=False)
        drive_folder = gr.Dropdown(label="Drive Folder (optional)", choices=[])
    with gr.Row():
        save_md = gr.Textbox(label="Save response to markdown (optional)", value="")
        ollama_url = gr.Textbox(label="Ollama URL (optional)", placeholder="http://172.27
    with gr.Row():
        answer_sink_path = gr.Textbox(label="Save extended answers to file (optional)", v
        answer_max_tokens = gr.Slider(256, 4096, value=2048, step=256, label="Max tokens
        continue_segments = gr.Slider(0, 20, value=0, step=1, label="Continuation segment
        num_ctx = gr.Slider(2048, 32768, value=8192, step=1024, label="Model context wind
    run_btn = gr.Button(" Run Query", variant="primary")
    output = gr.Textbox(label="AI Response", lines=20, show_copy_button=True)

    inputs = [query, folder, github, model, save_md, embed_model, use_drive, drive_folder
    run_btn.click(kickoff_pipeline, inputs=inputs, outputs=output)
    query.submit(kickoff_pipeline, inputs=inputs, outputs=output)

def run():
    port = find_free_port()
    print(f" Starting KnowledgeBot on port {port}")
    print(f" Access at: http://0.0.0.0:{port}")
    demo.launch(server_name="0.0.0.0", server_port=port, show_error=True)
```

```
if __name__ == "__main__":
    run()
```

What Each File Does

File	Purpose
src/knowledge_bot/tools/custom_tool.py	Multi-format ingestion; per-page PDF parsing; placeholders for GitHub/Drive to keep imports stable
src/knowledge_bot/crew_simple.py	Core RAG pipeline (intent routing, retrieval, robust Ollama, page-aware context)
src/knowledge_bot/main.py	Gradio UI that routes all queries to the improved direct RAG processor

Deployment

1. Install dependencies

- pip install -e .
- pip install pdfplumber rank-bm25

2. Set environment

- export OLLAMA_BASE_URL=http://YOUR-OLLAMA-ENDPOINT:PORT
- export KB_MODEL=mistral
- export CREWAI_TELEMETRY_ENABLED=false

3. Run

- python -m knowledge_bot.main
- Open http://0.0.0.0:7861

How It Works

- UI sends every query to process_query_direct.
- Intent router quickly handles “who am i?”, file listing, and projects without AI.
- For document queries:
  - Files are ingested (PDFs per page), chunked, and ranked with BM25 (+ synonyms/version boosts).
  - Top page-aware contexts are sent to Ollama with a strong, grounded prompt.
  - Answers cite filenames and pages; if no coverage, it states “Not found in the provided context.”

Troubleshooting

- ImportError: GitHubRepoCloner: fixed by placeholders in custom_tool.py.
- “No context found”: ensure PDFs are text-extractable; OCR if needed.
- Ollama JSON errors: the robust client handles streaming and non-stream; ensure OLLAMA_BASE_URL points to a reachable endpoint and KB_MODEL exists.
- Too noisy logs: warnings are silenced in main.py (optional).

Save this file as KnowledgeBot-README.md and check it into the repo alongside the updated Python files for a complete, downloadable reference.