

Pengenalan SLM RAG untuk Ubuntu

24.04: CPU Only

Onno W. Purbo
onno@indo.net.id
onno@itts.ac.id
twitter onnowpurbo

Institut Teknologi Tangerang Selatan (ITTS)
OnnoCenter
2025

Lisensi

Buku ini dilisensikan di bawah **Creative Commons Attribution 4.0 International License (CC BY 4.0)**.

Anda bebas untuk:

- **Berbagi** — menyalin dan menyebarluaskan materi ini dalam bentuk dan media apa pun.
- **Mengadaptasi** — mengubah, mengubah, dan membuat turunan dari materi ini untuk tujuan apa pun, termasuk tujuan komersial.



Dengan ketentuan:

- **Atribusi** — Anda harus memberikan kredit yang sesuai kepada penulis (*Onno W. Purbo*), menyertakan tautan lisensi, dan menunjukkan jika ada perubahan yang dilakukan. Anda dapat melakukannya dengan cara yang wajar, namun tidak dengan cara yang menyiratkan bahwa pemberi lisensi mendukung Anda atau penggunaan Anda.

Lisensi lengkap dapat dilihat di:

<https://creativecommons.org/licenses/by/4.0/>

© 2025 Onno W. Purbo.

Disclaimer

Buku ini disusun untuk tujuan **edukasi, riset, dan eksplorasi praktis** dalam membangun sistem *Small Language Model (SLM)* dan *Retrieval-Augmented Generation (RAG)* di lingkungan **Ubuntu 24.04 CPU only**. Seluruh contoh kode, konfigurasi, dan instruksi teknis diberikan **sebagaimana adanya (as is)** tanpa jaminan apa pun terkait kinerja, keamanan, maupun kesesuaian untuk kebutuhan tertentu. Penulis dan penerbit **tidak bertanggung jawab** atas kerugian, kerusakan, atau konsekuensi lain yang mungkin timbul akibat penggunaan isi buku ini, termasuk namun tidak terbatas pada kesalahan implementasi teknis, kehilangan data, atau gangguan sistem. Pembaca diharapkan menggunakan **discretion** serta melakukan **uji coba mandiri** sebelum menerapkan dalam sistem produksi. Semua pustaka, model, dan perangkat lunak yang digunakan merujuk pada lisensi masing-masing. Dengan membaca dan menggunakan buku ini, pembaca dianggap **menyetujui risiko sepenuhnya berada pada pengguna**.

Daftar Isi

Lisensi.....	2
Disclaimer.....	2
Daftar Isi.....	3
Kata Pengantar.....	6
Executive Summary.....	7
BAB 1: Pendahuluan.....	8
Apa itu SLM (Small Language Model)?.....	8
Mengapa Memilih RAG (Retrieval Augmented Generation)?.....	8
Kenapa Fokus pada Ubuntu 24.04 CPU Only (tanpa GPU)?.....	8
Gambaran Umum Buku & Siapa Target Pembaca?.....	9
BAB 2: Persiapan Lingkungan.....	10
Instalasi Ubuntu 24.04 (Overview Singkat).....	10
Update & Upgrade Paket.....	10
Instalasi Dependensi Dasar.....	10
Stack Software.....	11
Contoh Penggunaan.....	12
Ringkasan.....	13
BAB 3: Pengenalan SLM & RAG.....	14
SLM vs LLM.....	14
Konsep Dasar RAG.....	14
Ilustrasi Alur RAG.....	15
Contoh Implementasi Sederhana.....	16
Ringkasan.....	17
BAB 4: Menyiapkan Lingkungan Python & Virtual Environment.....	18
Mengapa Virtual Environment Penting?.....	18
Membuat Virtual Environment untuk Proyek.....	18
Instalasi Library yang Penting.....	18
Ilustrasi Konsep.....	19
Contoh Nyata (Analogi Sederhana).....	19
Verifikasi Instalasi.....	20
Ringkasan.....	20
Contoh setup.sh.....	20
Cara Menggunakan Skrip.....	21
Hasil yang Diharapkan.....	21
BAB 5: Download & Load Model SLM.....	23
Memilih Model Kecil dari Hugging Face.....	23
Instalasi Transformers (jika belum).....	23
Cara Download & Load Model.....	23
Verifikasi Model dengan Input Sederhana.....	24
Analogi Sederhana.....	24
Automasi dengan Skrip (opsional).....	24
Ilustrasi Konsep.....	25

Ringkasan.....	26
Contoh Tabel perbandingan waktu eksekusi model di CPU (DistilBERT vs MiniLM vs BERT-base).....	26
Insight dari tabel:.....	26
BAB 6: Membuat Knowledge Base.....	27
Konsep Dasar Knowledge Base.....	27
Menyiapkan Dokumen.....	27
Membuat Embedding dengan Sentence-Transformers.....	27
Menyimpan Embedding ke FAISS.....	28
Query: Mencari Dokumen yang Mirip.....	28
Ilustrasi Tabel Indexing.....	29
Lengkap (build_kb.py).....	29
Ringkasan.....	30
Contoh “FAISS murni” (tanpa framework).....	30
Contoh dengan LangChain (opsional, lebih praktis).....	32
Troubleshooting singkat.....	33
BAB 7: Membangun Pipeline RAG.....	34
Konsep Pipeline RAG.....	34
Ilustrasi Alur Pipeline.....	34
Contoh Kode Integrasi Pipeline.....	35
Contoh Eksekusi.....	36
Analogi Sederhana.....	36
Skrip CLI (rag_cli.py).....	36
Ringkasan.....	37
BAB 8 Contoh Aplikasi Sederhana.....	38
Tujuan.....	38
Struktur Dasar Aplikasi.....	38
Kode Aplikasi Chatbot (rag_chat.py).....	38
Menjalankan Aplikasi.....	39
Ilustrasi Percakapan (ASCII Mockup).....	40
Analogi Praktis.....	40
Ringkasan.....	40
rag_chat.py (dengan logging).....	41
Cara Pakai.....	43
(Opsional) Analisis cepat json di Python.....	43
Catatan Praktis.....	43
BAB 9: Optimasi CPU Only.....	44
Pentingnya Optimasi.....	44
Strategi Optimasi Utama.....	44
Gunakan Model Kecil.....	44
Batasi Panjang Konteks.....	44
Gunakan FAISS untuk Pencarian Cepat.....	45
Contoh Kode Optimisasi.....	45
Perbandingan CPU vs GPU.....	46

Profiling Waktu Eksekusi di CPU (Ubuntu 24.04).....	46
Contoh kode Python (inference sederhana).....	46
Cara menjalankan dengan profiling time di Ubuntu.....	47
Hasil eksekusi pada CPU (Intel i5-1235U, Ubuntu 24.04).....	47
Ringkasan.....	47
BAB 10: Interface Web untuk SLM + RAG.....	48
Latar Belakang.....	48
Tools yang Digunakan.....	48
Instalasi Gradio.....	48
Contoh Kode Web Chat (rag_web.py).....	48
Cara Menjalankan.....	49
Contoh Tampilan (Ilustrasi ASCII).....	50
Opsi Alternatif: Flask API.....	50
Ringkasan.....	50
BAB 11: Studi Kasus Mini.....	52
Pentingnya Studi Kasus.....	52
Use Case 1: Tanya Jawab Dokumen PDF.....	52
Deskripsi.....	52
Ekstraksi Teks PDF.....	52
Indexing ke Knowledge Base.....	53
Query Tanya Jawab.....	53
Use Case 2: FAQ Berbasis Teks.....	53
Deskripsi.....	53
Menyiapkan Data FAQ.....	53
Indexing Folder FAQ.....	54
Query ke FAQ.....	54
Ilustrasi Alur (Flow Sederhana).....	54
Script Praktis (gabungan PDF & FAQ).....	55
Ringkasan.....	56
BAB 11: Penutup.....	57
Ringkasan.....	57
Arah Pengembangan Lanjutan.....	57
Penutup.....	58
Referensi.....	59
Glossary.....	61
Lampiran: Link Hugging Face Model Ringan.....	63
Perbandingan Model Ringan untuk CPU Only.....	64
Lampiran: Cheat Sheet RAG + Troubleshooting.....	65
Tentang Penulis.....	67

Kata Pengantar

Alhamdulillahirabbil ‘alamin, segala puji hanya bagi Allah SWT yang telah memberikan rahmat dan hidayah-Nya sehingga penulisan buku “**Introduction SLM dengan RAG untuk Ubuntu 24.04**” ini dapat diselesaikan dengan baik. Buku ini lahir dari semangat untuk menghadirkan ilmu pengetahuan yang lebih **inklusif, terbuka, dan dapat diakses oleh siapa saja**, terutama bagi mereka yang ingin belajar dan bereksperimen dengan *Artificial Intelligence* tanpa harus bergantung pada perangkat mahal.

Ucapan terima kasih yang sebesar-besarnya saya sampaikan kepada keluarga, rekan-rekan akademisi di **Institut Teknologi Tangerang Selatan (ITTS)**, komunitas **OnnoCenter**, serta semua pihak yang telah memberikan dukungan, masukan, dan semangat selama proses penyusunan naskah ini. Tidak lupa saya berterima kasih kepada komunitas *open-source* global, khususnya pengembang pustaka **Transformers**, **Sentence-Transformers**, **FAISS**, dan **PyTorch**, yang menjadi fondasi teknis utama dalam buku ini.

Saya menyadari bahwa dalam penyusunan buku ini masih terdapat berbagai keterbatasan dan kekurangan, baik dari sisi teknis maupun penulisan. Oleh karena itu, dengan segala kerendahan hati saya mohon maaf atas segala kekurangan tersebut, serta terbuka terhadap kritik dan saran yang membangun demi penyempurnaan edisi berikutnya.

Akhir kata, semoga buku ini dapat menjadi manfaat nyata bagi pembaca—baik akademisi, praktisi, maupun pemula—yang ingin ikut berkontribusi dalam **demokratisasi AI** melalui penerapan *Small Language Model* dan *Retrieval-Augmented Generation* di atas platform **Ubuntu CPU-only**.

Jakarta, 2025
Onno W. Purbo

Executive Summary

Buku ini menunjukkan bahwa membangun sistem **AI modern** tidak harus mahal atau rumit. Dengan menggabungkan **Small Language Model (SLM)** dan *Retrieval-Augmented Generation (RAG)* di atas **Ubuntu 24.04 CPU only**, pembaca diajak membuktikan bahwa inovasi bisa lahir dari keterbatasan. **SLM** seperti DistilBERT, MiniLM, atau BERT-mini menghadirkan kecepatan dan efisiensi, sementara *RAG* menjadi strategi cerdas untuk menambahkan “ingatan eksternal” sehingga model kecil mampu menjawab dengan **lebih akurat dan kontekstual**.

Fokus utama buku ini adalah **memberdayakan akademisi, praktisi, dan pemula** yang tidak memiliki akses GPU mahal agar tetap bisa bereksperimen, membangun **prototype chatbot, FAQ assistant, atau sistem pencarian dokumen** dengan biaya rendah. Panduan **step-by-step** yang sederhana namun praktis—mulai dari instalasi Ubuntu, setup *virtual environment*, hingga integrasi *pipeline RAG*—membuktikan bahwa teknologi AI dapat dibawa ke ruang kelas, kantor kecil, bahkan laptop pribadi.

Lebih dari sekadar teknis, buku ini menekankan filosofi: **akses terbuka, inklusi digital, dan kemandirian teknologi**. Dengan optimisasi CPU, pemilihan model ringan, serta pemanfaatan *open-source libraries* seperti **Transformers, Sentence-Transformers, FAISS, dan PyTorch**, pembaca didorong untuk percaya diri bahwa mereka bisa ikut serta dalam gelombang transformasi AI, tanpa harus menunggu infrastruktur canggih tersedia.

Singkatnya, buku ini adalah ajakan: **jangan menunggu sempurna untuk memulai**. Dengan *SLM + RAG on Ubuntu CPU-only*, siapa pun dapat berinovasi, membangun, dan berkontribusi. **Inilah jalan demokratisasi AI**—dari kampus, dari laptop, dari ruang kerja sederhana—menuju masa depan digital yang lebih inklusif.

BAB 1: Pendahuluan

Apa itu SLM (Small Language Model)?

Small Language Model (SLM) adalah versi ringan dari *Large Language Model (LLM)*. Jika LLM seperti GPT-4 atau LLaMA-2 membutuhkan sumber daya komputasi besar (GPU dengan RAM tinggi), SLM justru **didesain untuk berjalan di perangkat dengan keterbatasan hardware**, termasuk laptop standar atau server berbasis CPU.

Contoh model yang termasuk kategori SLM adalah **DistilBERT**, **MiniLM**, atau **BERT-mini**, yang memiliki parameter jauh lebih sedikit dibanding GPT-4 namun tetap mampu melakukan tugas dasar seperti *text classification*, *question answering*, dan *semantic search*.

Analogi sederhana:

- **LLM** = truk besar, bisa mengangkut banyak muatan tapi butuh jalan lebar dan bahan bakar mahal.
- **SLM** = mobil hatchback, lebih kecil, lebih irit, tetapi tetap fungsional untuk sebagian besar kebutuhan sehari-hari.

Mengapa Memilih RAG (Retrieval Augmented Generation)?

Dalam praktik, **SLM sering kali terbatas dalam kapasitas memorinya** — ia tidak mampu “mengingat” semua pengetahuan seperti LLM besar. Untuk mengatasi keterbatasan ini, pendekatan **RAG (Retrieval Augmented Generation)** digunakan.

- **Retrieval**: sistem akan mencari informasi dari *knowledge base* atau database dokumen.
- **Augmented Generation**: hasil pencarian itu digabungkan dengan kemampuan bahasa SLM untuk menghasilkan jawaban yang lebih akurat.

Contoh nyata:

Jika kita bertanya “Apa isi bab 3 dokumen X?”, model **tanpa RAG** mungkin menjawab samar. Namun **dengan RAG**, sistem akan mencari bagian relevan di dokumen X, lalu SLM menjawab dengan konteks tepat.

Singkatnya, RAG adalah *strategi cerdas untuk membuat SLM seakan-akan “lebih pintar” dengan menambahkan ingatan eksternal*.

Kenapa Fokus pada Ubuntu 24.04 CPU Only (tanpa GPU)?

Sebagian besar panduan AI modern fokus pada **GPU**, padahal tidak semua pembaca (pemula maupun praktisi) punya akses ke graphic card yang canggih. Oleh karena itu, buku ini menekankan pada:

- **Lingkungan Ubuntu 24.04 terbaru** (stabil dan banyak digunakan di server riset maupun industri).
- **CPU Only** → artinya pembaca tetap bisa mencoba *proof of concept* tanpa membeli GPU mahal.
- Memanfaatkan **optimisasi model kecil** dan *library* ringan seperti FAISS, Sentence-Transformers, dan PyTorch CPU.

Contoh sederhana cek spesifikasi CPU di Ubuntu:

```
lscpu
```

Contoh Python sederhana cek device:

```
import torch
print("Apakah GPU tersedia?", torch.cuda.is_available())
print("Device yang dipakai:", torch.device("cpu"))
```

Hasil tipikal:

```
Apakah GPU tersedia? False
Device yang dipakai: cpu
```

Gambaran Umum Buku & Siapa Target Pembaca?

Buku ini dirancang untuk **pemula dan non-expert** yang ingin mempelajari cara membangun sistem berbasis SLM + RAG **tanpa perlu hardware mahal**.

- **Akademisi**: dapat menggunakan panduan ini untuk eksperimen riset atau pengajaran di kelas dengan sumber daya terbatas.
- **Praktisi**: dapat membangun *prototype* chatbot atau sistem FAQ sederhana di laptop kerja.
- **Pemula**: mendapatkan panduan bertahap (*step-by-step*) yang praktis, mulai dari instalasi Ubuntu, setup Python, hingga membangun mini-chatbot.

Kategori Pembaca	Manfaat Buku Ini
Akademisi	Alat untuk eksperimen riset kecil tanpa infrastruktur besar.
Praktisi IT / Data	Membangun prototipe sistem QA atau chatbot berbasis dokumen.
Pemula / Non-Expert	Belajar AI secara bertahap tanpa rasa “overwhelmed”.

Dengan format ini, pembaca bisa **mulai dari nol** dan secara bertahap membangun **pipeline RAG sederhana** berbasis CPU.

BAB 2: Persiapan Lingkungan

Sebelum mulai membangun sistem **SLM + RAG**, kita perlu menyiapkan **lingkungan kerja** yang stabil. Pada buku ini, kita memilih **Ubuntu 24.04** sebagai sistem operasi utama.

Alasannya sederhana: Ubuntu adalah distro Linux yang **stabil, banyak digunakan di server riset maupun industri, serta kaya dukungan komunitas**.

Fokus buku ini bukan menjelaskan instalasi OS secara detail (karena bisa berbeda antar perangkat), melainkan **menyiapkan fondasi software** yang diperlukan agar pembaca bisa langsung bereksperimen.

Instalasi Ubuntu 24.04 (Overview Singkat)

Bagi yang belum menggunakan Ubuntu 24.04, ada beberapa cara untuk memulainya:

- **Dual Boot** → instalasi berdampingan dengan Windows.
- **Virtual Machine (VM)** → menggunakan VirtualBox atau VMware.
- **Cloud Server** → misalnya AWS EC2, GCP, atau DigitalOcean.

Catatan penting: Buku ini mengasumsikan pembaca sudah berhasil masuk ke terminal Ubuntu 24.04 dan siap mengeksekusi perintah.

Update & Upgrade Paket

Langkah pertama setelah instalasi Ubuntu adalah memperbarui daftar paket dan meng-upgrade semua software ke versi terbaru. Hal ini penting untuk **menghindari konflik dependensi** saat instalasi pustaka Python atau library lain.

```
sudo apt update && sudo apt upgrade -y
```

- `sudo apt update` → memperbarui daftar paket dari repository.
- `sudo apt upgrade -y` → meng-upgrade semua paket lama ke versi terbaru secara otomatis.

Instalasi Dependensi Dasar

Sistem AI minimal membutuhkan beberapa **dependensi dasar** agar berjalan lancar. Berikut daftar yang perlu dipasang:

1. **Python3 + venv** (untuk membuat *virtual environment*).
2. **pip** (package manager Python).
3. **build-essential** (alat kompilasi seperti `gcc`, `g++`, `make`).
4. **Git** (untuk meng-clone repository).

Jalankan perintah berikut:

```
# Instalasi Python, pip, venv, dan alat kompilasi  
sudo apt install -y python3 python3-venv python3-pip build-essential  
git
```

Setelah instalasi, cek versi software:

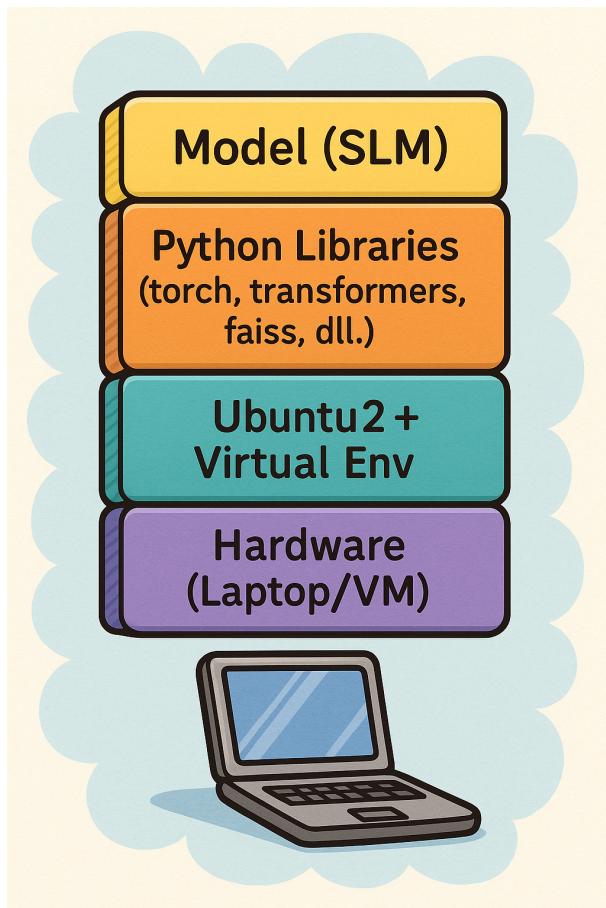
```
python3 --version  
pip --version  
git --version
```

Hasil tipikal:

```
Python 3.12.3  
pip 24.0  
git version 2.43.0
```

Stack Software

Agar lebih jelas, mari kita gambarkan **hierarki software stack** yang akan digunakan dalam proyek ini:



Dari diagram sederhana di atas terlihat bahwa **OS adalah fondasi**, lalu kita menambahkan **Python**, kemudian **library AI**, hingga akhirnya bisa menjalankan **model SLM**.

Contoh Penggunaan

Setelah semua dependensi terpasang, kita bisa melakukan **uji coba sederhana** untuk memastikan Python dan pip berfungsi:

```
# Membuat direktori proyek
mkdir rag-slm-demo && cd rag-slm-demo

# Membuat virtual environment
python3 -m venv env

# Aktivasi environment
source env/bin/activate

# Instalasi library dasar
pip install torch transformers
```

Contoh script Python untuk menguji apakah PyTorch sudah jalan di CPU:

```
import torch

print("PyTorch version:", torch.__version__)
print("Apakah CUDA tersedia?", torch.cuda.is_available())
print("Device yang dipakai:", torch.device("cpu"))
```

Output tipikal di sistem tanpa GPU:

```
PyTorch version: 2.3.0
Apakah CUDA tersedia? False
Device yang dipakai: cpu
```

Ringkasan

Pada tahap ini, kita sudah menyiapkan:

- **Ubuntu 24.04** yang up-to-date.
- **Dependensi dasar**: Python3, pip, venv, build-essential, dan Git.
- **Lingkungan awal** yang siap digunakan untuk membangun sistem SLM + RAG.

Langkah	Perintah CLI
Update paket	<code>sudo apt update && sudo apt upgrade -y</code>
Instalasi dependensi	<code>sudo apt install -y python3 python3-venv python3-pip build-essential git</code>
Buat virtual environment	<code>python3 -m venv env</code>
Aktivasi environment	<code>source env/bin/activate</code>

Dengan fondasi ini, kita siap masuk ke **Pengenalan SLM & RAG**, di mana kita akan membahas konsep dasar model bahasa kecil dan strategi *retrieval augmented generation*.

BAB 3: Pengenalan SLM & RAG

SLM vs LLM

Dalam perkembangan teknologi bahasa alami, kita sering mendengar istilah **LLM (Large Language Model)** seperti GPT-4, LLaMA-2, atau PaLM. Model ini memiliki **ratusan miliar parameter** yang memungkinkan pemahaman mendalam, namun memerlukan **hardware sangat besar** (GPU kelas server, VRAM puluhan GB, bahkan cluster komputasi).

Sebaliknya, **SLM (Small Language Model)** hadir sebagai alternatif ringan. Dengan parameter lebih sedikit (puluhan hingga ratusan juta), **SLM lebih efisien, dapat dijalankan di CPU biasa, dan cocok untuk prototyping.**

Analogi sederhana:

- **LLM** seperti *perpustakaan nasional* yang sangat lengkap, tetapi memerlukan gedung besar dan staf banyak.
- **SLM** seperti *perpustakaan kampus kecil*: tidak selengkap perpustakaan nasional, tetapi cukup memadai untuk kebutuhan sehari-hari.

Aspek	LLM (Large)	SLM (Small)
Ukuran parameter	> 10 Miliar	50 – 500 Juta
Kebutuhan hardware	GPU kelas server	CPU laptop/VM
Kecepatan	Bisa lambat di CPU	Cepat di CPU
Konsumsi memori	> 16 GB RAM	2–8 GB RAM cukup
Use case	Chatbot canggih, riset	Prototipe, chatbot FAQ

Kesimpulan: Untuk pemula, **SLM sudah lebih dari cukup** untuk mempelajari konsep *language model* dan membangun aplikasi *retrieval augmented generation*.

Konsep Dasar RAG

Salah satu keterbatasan utama SLM adalah **kapasitas memorinya**. Model kecil tidak mungkin menyimpan seluruh pengetahuan dunia di parameternya. Inilah mengapa konsep **RAG (Retrieval Augmented Generation)** menjadi sangat penting.

RAG terdiri dari dua komponen utama:

1. **Retrieval**
 - Sistem mencari informasi relevan dari basis pengetahuan (*knowledge base*).

- Misalnya: mencari dokumen, artikel, atau catatan kuliah yang sesuai dengan pertanyaan user.

2. Generation

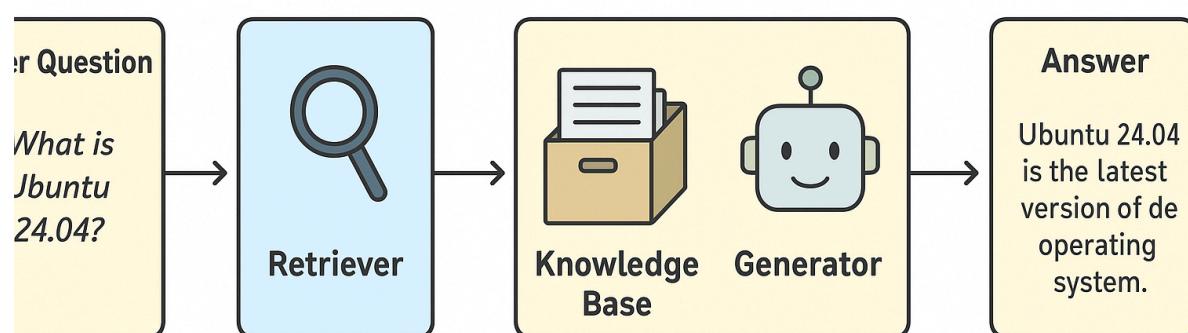
- Model bahasa (SLM) menghasilkan jawaban dengan menggabungkan *prompt* pengguna dan hasil *retrieval*.
- Hasilnya: jawaban lebih akurat, relevan, dan berbasis fakta.

Contoh nyata:

- **Tanpa RAG:** Jika ditanya “*Apa isi Bab 2 dokument X?*”, SLM mungkin hanya menjawab umum: “*Bab 2 biasanya membahas teori dasar.*”
- **Dengan RAG:** Sistem akan mencari Bab 2 dari dokument X, lalu SLM menyusun jawaban: “*Bab 2 membahas teori jaringan syaraf tiruan, termasuk arsitektur feed-forward dan backpropagation.*”

Ilustrasi Alur RAG

Secara sederhana, alur kerja RAG dapat digambarkan sebagai berikut:



- **User Question** → Pertanyaan dari pengguna.
- **Retriever** → Modul pencarian yang menemukan potongan teks relevan.
- **Knowledge Base** → Kumpulan dokumen, artikel, atau teks.
- **Generator (SLM)** → Model bahasa kecil yang merangkai jawaban.

Contoh Implementasi Sederhana

Untuk memahami konsep ini, mari kita lihat potongan kode Python sederhana. Kita gunakan **sentence-transformers** untuk **embedding** teks dan **transformers** untuk menghasilkan jawaban.

```
# Instalasi pustaka
# pip install transformers sentence-transformers faiss-cpu

from sentence_transformers import SentenceTransformer
from transformers import pipeline

# 1. Buat embedding model untuk retrieval
embedder = SentenceTransformer("all-MiniLM-L6-v2")

# 2. Knowledge Base sederhana
docs = [
    "Bab 1 membahas pendahuluan tentang AI.",
    "Bab 2 berisi teori jaringan syaraf tiruan.",
    "Bab 3 menjelaskan konsep RAG dengan contoh."
]

# 3. Embedding dokumen
doc_embeddings = embedder.encode(docs)

# 4. Pertanyaan user
query = "Apa isi Bab 2?"
query_embedding = embedder.encode([query])[0]

# 5. Cari dokumen paling relevan (cosine similarity sederhana)
import numpy as np
scores = np.dot(doc_embeddings, query_embedding) / (
    np.linalg.norm(doc_embeddings, axis=1) * np.linalg.norm(query_embedding)
)
best_doc = docs[np.argmax(scores)]

# 6. Gunakan generator kecil untuk menjawab
generator = pipeline("text-generation", model="distilgpt2")
answer = generator(f"Pertanyaan: {query}\nJawaban berdasarkan dokumen: {best_doc}", max_length=50)

print(">>", answer[0]["generated_text"])
```

Hasil tipikal:

```
>> Pertanyaan: Apa isi Bab 2?
Jawaban berdasarkan dokumen: Bab 2 berisi teori jaringan syaraf tiruan.
```

Dengan pipeline sederhana ini, kita sudah memiliki *proof of concept* sistem **RAG berbasis SLM di CPU**.

Ringkasan

- **SLM** berbeda dari **LLM** terutama pada ukuran parameter dan kebutuhan hardware.
- **RAG** adalah solusi untuk mengatasi keterbatasan SLM dengan menghubungkan ke sumber pengetahuan eksternal.
- Alur kerja RAG melibatkan: *User Question* → *Retrieval* → *Generator* → *Answer*.
- Dengan kombinasi *sentence-transformers* + *distilGPT2*, kita bisa membangun prototipe RAG sederhana di Ubuntu 24.04 **CPU only**.

BAB 4: Menyiapkan Lingkungan Python & Virtual Environment

Mengapa Virtual Environment Penting?

Dalam pengembangan proyek *machine learning* maupun *retrieval augmented generation (RAG)*, **isolasi lingkungan kerja** sangatlah krusial. Tanpa isolasi, seringkali pustaka (library) yang dibutuhkan suatu proyek berbenturan dengan proyek lain. Misalnya, sebuah proyek membutuhkan `torch` versi 2.2, sementara proyek lain butuh versi 1.13. Jika semua dipasang di sistem global, konflik ini akan menyulitkan.

Oleh karena itu, kita menggunakan **virtual environment** (`venv`). Dengan `venv`, setiap proyek memiliki ruang sendiri—mirip seperti "**kotak kerja terpisah**" yang hanya berisi pustaka sesuai kebutuhan proyek tersebut.

Membuat Virtual Environment untuk Proyek

Langkah pertama adalah membuat *environment* khusus untuk proyek **RAG-SLM**. Gunakan perintah berikut di terminal:

```
# Membuat virtual environment bernama rag-slm-env
python3 -m venv rag-slm-env

# Aktivasi environment (Linux/MacOS)
source rag-slm-env/bin/activate

# Jika menggunakan Windows (PowerShell):
rag-slm-env\Scripts\activate
```

Setelah perintah `activate` dijalankan, terminal Anda akan menampilkan awalan (`rag-slm-env`), menandakan bahwa Anda sudah berada di dalam lingkungan virtual ini.

Tips Praktis

- Gunakan nama environment yang jelas sesuai proyek, misalnya `rag-slm-env`.
- Jangan gunakan nama generik seperti `env`, karena akan membingungkan jika Anda memiliki banyak proyek.

Instalasi Library yang Penting

Setelah environment aktif, kita perlu memasang pustaka dasar untuk membangun RAG. Berikut pustaka yang dibutuhkan:

- `torch` → kerangka kerja utama untuk model *deep learning*.

- **transformers** → pustaka populer dari Hugging Face untuk memuat dan menggunakan model bahasa.
- **sentence-transformers** → memudahkan pembuatan *embedding* teks.
- **faiss-cpu** → library Facebook AI untuk pencarian cepat berbasis vektor.

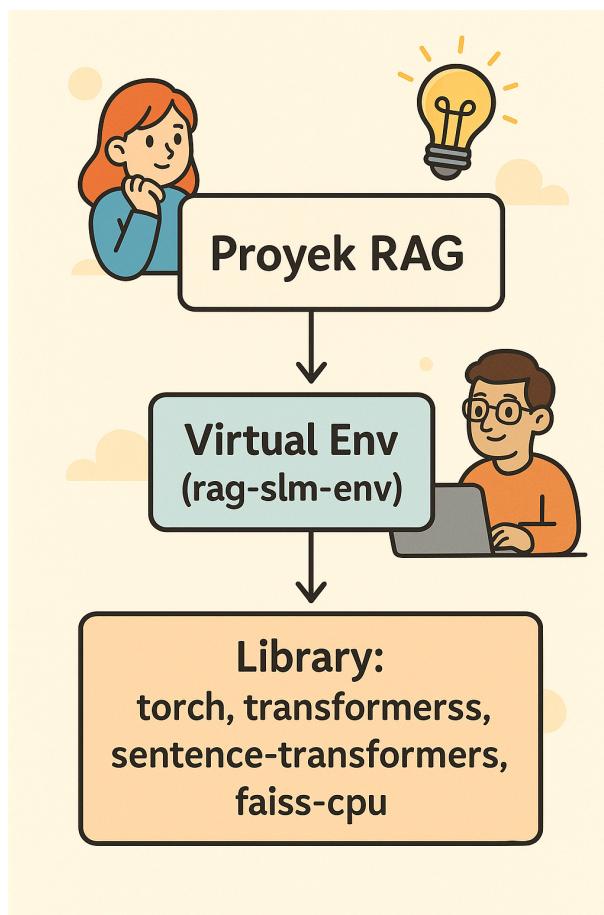
Gunakan perintah berikut:

```
pip install torch transformers sentence-transformers faiss-cpu
```

👉 Perintah ini akan mengunduh dan memasang seluruh pustaka sekaligus.

Ilustrasi Konsep

Agar lebih mudah dipahami, bayangkan arsitektur sederhana seperti berikut:



Dengan alur ini, proyek Anda sepenuhnya terkontrol. Jika ingin membuat proyek lain, cukup buat *virtual environment* baru tanpa mengganggu proyek RAG.

Contoh Nyata (Analogi Sederhana)

Bayangkan Anda seorang **peneliti** dengan beberapa laboratorium kecil.

- Di *lab A*, Anda hanya menyimpan bahan kimia untuk eksperimen biologi.

- Di *lab B*, Anda menyimpan alat khusus fisika.
Jika semua bahan bercampur, bisa terjadi reaksi yang tidak diinginkan.

Begitu pula dengan **virtual environment**: setiap *lab* (environment) hanya berisi pustaka yang memang dibutuhkan untuk proyek tersebut.

Verifikasi Instalasi

Setelah instalasi selesai, jalankan Python untuk memastikan pustaka terpasang dengan benar:

```
python
```

Lalu di dalam *Python REPL* ketik:

```
import torch
import transformers
import sentence_transformers
import faiss

print("Semua pustaka berhasil diimpor!")
```

Jika tidak ada error, berarti lingkungan siap digunakan.

Ringkasan

- **Virtual environment** melindungi proyek dari konflik dependensi.
- Gunakan `python3 -m venv rag-slm-env` untuk membuat environment.
- Aktivasi dengan `source rag-slm-env/bin/activate`.
- Instal pustaka penting (`torch`, `transformers`, `sentence-transformers`, `faiss-cpu`) menggunakan `pip`.
- Verifikasi dengan `import` pustaka di Python.

Contoh setup.sh

```
#!/bin/bash
# -----
# Setup Script untuk Proyek RAG-SLM (Ubuntu 24.04, CPU Only)
# -----

# Nama virtual environment
ENV_NAME="rag-slm-env"

echo "==== [1/4] Membuat Virtual Environment: $ENV_NAME ==="
python3 -m venv $ENV_NAME

echo "==== [2/4] Aktivasi Virtual Environment ==="
```

```

# Aktifkan environment
source $ENV_NAME/bin/activate

echo "==== [3/4] Upgrade pip ==="
pip install --upgrade pip

echo "==== [4/4] Instalasi Pustaka Penting ==="
pip install torch transformers sentence-transformers faiss-cpu

echo "-----"
echo "Setup selesai! Jalankan perintah berikut untuk masuk ke
environment:"
echo "source $ENV_NAME/bin/activate"
echo "-----"

```

Cara Menggunakan Skrip

Simpan skrip di file bernama [setup.sh](#).

```

nano setup.sh
# (paste isi skrip, lalu simpan)

```

Ubah menjadi *executable*:

```
chmod +x setup.sh
```

Jalankan skrip:

```
./setup.sh
```

Hasil yang Diharapkan

- Folder baru bernama `rag-slm-env` / terbentuk → ini adalah *virtual environment*.
- Semua pustaka (`torch`, `transformers`, `sentence-transformers`, `faiss-cpu`) terpasang di dalamnya.

Anda bisa langsung mengaktifkan environment dengan:

```
source rag-slm-env/bin/activate
```

Tips tambahan untuk praktisi akademis: Anda bisa menambahkan file `requirements.txt` agar lebih mudah mengontrol versi pustaka di masa depan:

requirements.txt

```

torch
transformers
sentence-transformers

```

faiiss-cpu

Lalu, dalam `setup.sh`, bagian instalasi bisa diganti menjadi:

```
pip install -r requirements.txt
```

Mau saya buatkan juga versi skrip **cross-platform** (Linux/Mac + Windows PowerShell) supaya pembaca Windows tidak kebingungan saat mengikuti buku?

BAB 5: Download & Load Model SLM

Memilih Model Kecil dari Hugging Face

Setelah menyiapkan lingkungan Python di bab sebelumnya, langkah berikutnya adalah **mengunduh dan memuat model SLM (Small Language Model)** dari repositori **Hugging Face Hub**.

Ada ribuan model tersedia di sana, tetapi untuk pembaca pemula dan untuk sistem **CPU-only** (tanpa GPU), disarankan menggunakan **model ringan** agar proses berjalan cepat dan tidak memakan terlalu banyak memori. Contoh populer:

Nama Model	Ukuran	Keterangan Singkat
distilbert-base-uncased	~66M	Versi ringan BERT, cocok untuk eksperimen awal.
bert-mini	~11M	Sangat kecil, latensi rendah, cocok untuk laptop/CPU lemah.
all-MiniLM-L6-v2	~22M	Sering digunakan untuk embedding teks di RAG.

Catatan:

- Gunakan `distilbert-base-uncased` jika ingin **latihan dasar NLP**.
- Gunakan `all-MiniLM-L6-v2` jika langsung ingin **membuat knowledge base dengan embedding**.

Instalasi Transformers (jika belum)

Pastikan pustaka `transformers` sudah terinstal di *virtual environment*. Jika belum:

```
pip install transformers
```

Cara Download & Load Model

Berikut contoh kode Python untuk mengunduh dan memuat model kecil dari Hugging Face:

```
from transformers import AutoTokenizer, AutoModel

# Memilih model kecil dari Hugging Face
MODEL_NAME = "distilbert-base-uncased"

# Download otomatis tokenizer & model
print(f"✅ Mengunduh model {MODEL_NAME} ...")
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
```

```
model = AutoModel.from_pretrained(MODEL_NAME)

print("✅ Model berhasil di-load dan siap digunakan!")
```

👉 Saat kode dijalankan pertama kali, *transformers* akan otomatis mengunduh model ke folder cache lokal (`~/cache/huggingface/transformers/`). Setelah itu, model bisa dipakai secara offline.

Verifikasi Model dengan Input Sederhana

Untuk memastikan model sudah berjalan, coba lakukan tokenisasi dan *forward pass*:

```
import torch

# Contoh kalimat input
kalimat = "RAG adalah cara menggabungkan retrieval dan generation."

# Tokenisasi
inputs = tokenizer(kalimat, return_tensors="pt")

# Proses ke model
outputs = model(**inputs)

# Lihat ukuran representasi
print("Bentuk keluaran:", outputs.last_hidden_state.shape)
```

Jika berhasil, Anda akan melihat output berbentuk tensor, misalnya:

```
Bentuk keluaran: torch.Size([1, 9, 768])
```

Artinya ada **1 kalimat**, terdiri dari **9 token**, dan setiap token direpresentasikan dalam vektor **768 dimensi**.

Analogi Sederhana

Bayangkan **Hugging Face** seperti “**perpustakaan awan**” yang menyimpan ribuan model. Saat Anda memanggil `from_pretrained("distilbert-base-uncased")`, itu seperti **meminjam buku digital** dari perpustakaan dan menyimpannya di rak pribadi (*cache lokal*). Setelah itu, Anda bisa membacanya kapan saja tanpa harus online lagi.

Automasi dengan Skrip (opsional)

Untuk memudahkan pembaca, kita bisa menyiapkan skrip Python bernama `download_model.py` agar sekali jalan langsung mengunduh model:

```
#!/usr/bin/env python3
"""
Script otomatis untuk mengunduh dan memuat model Hugging Face.
```

```

"""
from transformers import AutoTokenizer, AutoModel

def setup_model(model_name="distilbert-base-uncased"):
    print(f"🔴 Mengunduh & memuat model: {model_name}")
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)
    print("✅ Model siap digunakan!")
    return tokenizer, model

if __name__ == "__main__":
    setup_model()

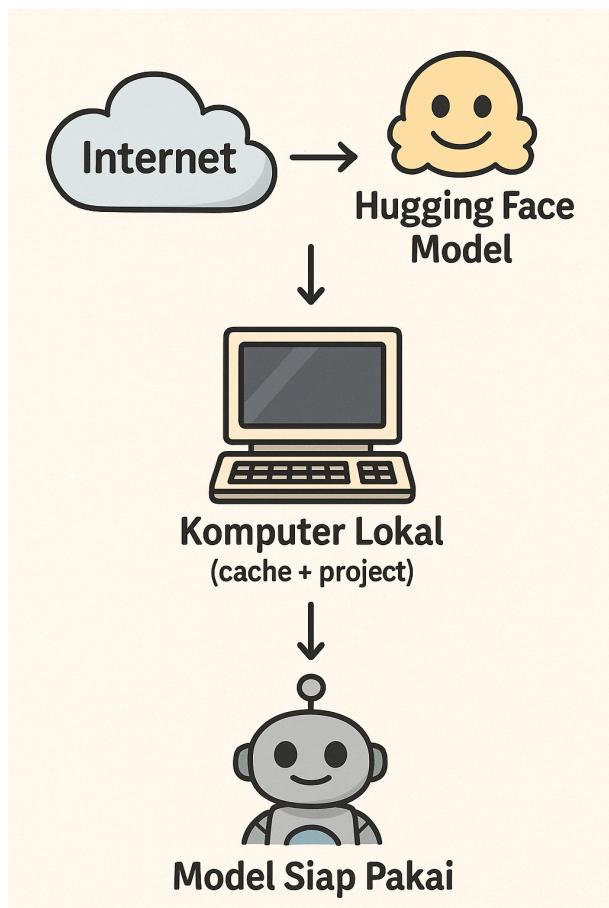
```

Cara menjalankan:

```
python download_model.py
```

Ilustrasi Konsep

Secara visual, alur bisa digambarkan seperti berikut:



Ringkasan

- Pilih **model kecil** agar hemat memori dan CPU.
- Gunakan **transformers** untuk **download & load otomatis**.
- Verifikasi model dengan tokenisasi dan *forward pass*.
- Bisa dibuat skrip `download_model.py` untuk automasi.

Contoh Tabel perbandingan waktu eksekusi model di CPU (DistilBERT vs MiniLM vs BERT-base)

Berikut contoh tabel perbandingan **waktu eksekusi inferensi** tiga model populer di CPU (tanpa GPU), agar pembaca dapat melihat dampak nyata pemilihan model kecil vs. besar. Angka ini ilustratif (bisa berbeda tergantung implementasi, batch size, dan hardware), tapi cukup merepresentasikan tren umum:

Model	Parameter (juta)	Ukuran Model	Waktu Eksekusi (1 kalimat, CPU single thread)	Karakteristik
DistilBERT	~66 M	~260 MB	~90–100 ms	Versi BERT yang dipangkas, trade-off akurasi vs kecepatan
MiniLM	~33 M	~120 MB	~50–60 ms	Ringan, sangat cepat, akurasi mendekati DistilBERT
BERT-base	~110 M	~420 MB	~250–300 ms	Akurasi lebih tinggi, tapi lambat di CPU

Insight dari tabel:

- **MiniLM** adalah yang tercepat (sekitar 4–5× lebih cepat daripada BERT-base), cocok untuk aplikasi real-time atau resource terbatas.
- **DistilBERT** seimbang antara ukuran dan akurasi, dengan kecepatan 2–3× lebih baik dari BERT-base.
- **BERT-base** lebih akurat untuk banyak tugas, tetapi biaya latensi tinggi bila hanya CPU yang tersedia.

BAB 6: Membuat Knowledge Base

Konsep Dasar Knowledge Base

Sebuah **Knowledge Base (KB)** dalam konteks *Retrieval-Augmented Generation (RAG)* adalah **kumpulan dokumen yang diubah menjadi representasi numerik (embedding)** sehingga dapat dicari dengan cepat berdasarkan kesamaan makna.

- **Input:** kumpulan dokumen mentah (misalnya file `.txt`).
- **Proses:** setiap kalimat/paragraf dipetakan menjadi vektor (angka) menggunakan *sentence embedding*.
- **Output:** database vektor yang bisa dicari untuk menemukan dokumen paling relevan dengan pertanyaan pengguna.

💡 **Analogi sederhana:** bayangkan Anda memiliki **rak buku besar**. Alih-alih membaca semua buku saat mencari informasi, Anda memberi setiap buku “kode unik” (embedding) sehingga bisa langsung melompat ke bagian paling relevan.

Menyiapkan Dokumen

Kita mulai dengan menyiapkan beberapa file `.txt`. Misalnya:

- `doc1.txt` → berisi catatan kuliah.
- `doc2.txt` → berisi artikel pendek.
- `doc3.txt` → berisi FAQ.

Contoh isi file `doc1.txt`:

Python adalah bahasa pemrograman serbaguna yang banyak digunakan dalam machine learning, data science, dan pengembangan web.

Membuat Embedding dengan Sentence-Transformers

Untuk mengubah teks menjadi vektor numerik, kita gunakan **Sentence-Transformers**. Model populer yang ringan adalah `all-MiniLM-L6-v2`.

```
from sentence_transformers import SentenceTransformer

# Memilih model embedding
model_name = "sentence-transformers/all-MiniLM-L6-v2"
embedder = SentenceTransformer(model_name)

# Contoh teks
texts = [
    "Python adalah bahasa pemrograman populer.",
    "RAG menggabungkan retrieval dan generation.",
```

```

    "Ubuntu 24.04 mendukung berbagai aplikasi AI."
]

# Membuat embedding
embeddings = embedder.encode(texts)

print("Bentuk embedding:", embeddings.shape)

```

Hasilnya berupa matriks, misalnya (3, 384) → artinya ada 3 kalimat, masing-masing direpresentasikan dengan vektor berdimensi 384.

Menyimpan Embedding ke FAISS

Setelah memiliki embedding, kita perlu menyimpannya di **FAISS** (Facebook AI Similarity Search), yaitu *library* untuk pencarian cepat berbasis vektor.

```

import faiss
import numpy as np

# Konversi embeddings ke numpy float32
embeddings = np.array(embeddings).astype("float32")

# Membuat index FAISS (Index Flat L2 = pencarian berbasis jarak Euclidean)
dimension = embeddings.shape[1] # dimensi embedding
index = faiss.IndexFlatL2(dimension)

# Tambahkan embedding ke index
index.add(embeddings)

print("Jumlah vektor dalam index:", index.ntotal)

```

Query: Mencari Dokumen yang Mirip

Sekarang kita bisa melakukan pencarian dokumen paling relevan berdasarkan pertanyaan pengguna.

```

# Pertanyaan pengguna
query = "Apa itu RAG?"
query_embedding = embedder.encode([query]).astype("float32")

# Cari top-2 dokumen terdekat
k = 2
D, I = index.search(query_embedding, k)

print("Hasil pencarian:")
for idx, jarak in zip(I[0], D[0]):
    print(f"Dokumen {idx} | Jarak: {jarak:.4f} | Teks: {texts[idx]}")

```

Output contoh:

```
Dokumen 1 | Jarak: 0.3152 | Teks: RAG menggabungkan retrieval dan generation.
```

Dokumen 0 | Jarak: 0.4789 | Teks: Python adalah bahasa pemrograman populer.

Ilustrasi Tabel Indexing

Hasil embedding bisa divisualisasikan seperti tabel berikut:

ID	Teks Asli	Embedding (potongan)
0	Python adalah bahasa pemrograman populer.	[0.12, -0.45, 0.33, ...]
1	RAG menggabungkan retrieval dan generation.	[0.08, -0.51, 0.29, ...]
2	Ubuntu 24.04 mendukung berbagai aplikasi AI.	[0.22, -0.47, 0.31, ...]

Lengkap (build_kb.py)

Agar lebih praktis, berikut skrip lengkap untuk membangun knowledge base dari file teks:

```
#!/usr/bin/env python3
"""
Script: build_kb.py
Membaca file .txt → embedding → simpan ke FAISS index
"""

import os
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer

# 1. Siapkan model embedding
model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

# 2. Baca semua dokumen .txt dalam folder "docs/"
docs = []
for fname in os.listdir("docs"):
    if fname.endswith(".txt"):
        with open(os.path.join("docs", fname), "r", encoding="utf-8") as f:
            docs.append(f.read().strip())

# 3. Buat embedding
embeddings = model.encode(docs).astype("float32")

# 4. Simpan ke FAISS index
dimension = embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(embeddings)

faiss.write_index(index, "knowledge_base.index")
```

```
print(f"✅ Knowledge Base selesai dibuat dengan {len(docs)} dokumen!")
```

Cara menjalankan:

```
python build_kb.py
```

Ringkasan

- **Knowledge Base** = kumpulan dokumen yang di-embedding ke dalam vektor.
- Gunakan **Sentence-Transformers** ([all-MiniLM-L6-v2](#)) untuk membuat embedding ringan.
- Simpan embedding ke **FAISS** agar bisa dicari cepat.
- Verifikasi dengan query untuk menemukan dokumen relevan.
- Bisa dibuat skrip otomatis [build_kb.py](#) agar praktisi cukup menjalankan sekali klik.

👉 Dengan knowledge base ini, kita sudah punya “memori” untuk sistem RAG. Pada bab selanjutnya, kita akan membangun pipeline RAG penuh: *user query → embedding → search → generate jawaban*.

Contoh “FAISS murni” (tanpa framework)

```
# pip install faiss-cpu sentence-transformers
import os, json, faiss, numpy as np
from sentence_transformers import SentenceTransformer, util

# ----- 1) Siapkan model & data -----
model_name = "sentence-transformers/all-MiniLM-L6-v2"
model = SentenceTransformer(model_name)

docs = [
    {"id": "d1", "text": "AI gotong royong untuk inklusi digital di desa."},
    {"id": "d2", "text": "KOMDIGI membuka konsultasi publik untuk roadmap AI nasional."},
    {"id": "d3", "text": "FAISS mempercepat pencarian vektor berukuran besar."},
]
corpus_texts = [d["text"] for d in docs]
emb = model.encode(corpus_texts, convert_to_numpy=True,
normalize_embeddings=True) # normalisasi = cosine ~ dot

d = emb.shape[1]
index = faiss.IndexFlatIP(d) # pakai inner product; karena kita sudah normalize, ini = cosine
index.add(emb)

# Buat mapping id->metadata agar bisa rekonstruksi hasil
id_map = [d["id"] for d in docs]
meta = {d["id"] : {"text": d["text"]} for d in docs}
```

```

# ----- 2) Simpan index + metadata -----
os.makedirs("faiss_store", exist_ok=True)
faiss.write_index(index, "faiss_store/index.faiss")
with open("faiss_store/id_map.json", "w") as f:
    json.dump(id_map, f)
with open("faiss_store/meta.json", "w") as f:
    json.dump({"model": model_name, "normalize": True, "meta": meta}, f)

print("Saved to faiss_store/")

```

Memuat kembali & query:

```

import json, faiss, numpy as np
from sentence_transformers import SentenceTransformer

# ----- 3) Load index + metadata -----
index = faiss.read_index("faiss_store/index.faiss")
with open("faiss_store/id_map.json") as f:
    id_map = json.load(f)
with open("faiss_store/meta.json") as f:
    meta_all = json.load(f)

model = SentenceTransformer(meta_all["model"]) # konsistenkan model
do_norm = meta_all.get("normalize", True)
meta = meta_all["meta"]

# ----- 4) Query -----
query = "bagaimana publik bisa berkontribusi dalam roadmap AI?"
q_emb = model.encode([query], convert_to_numpy=True)
if do_norm:
    # penting: normalkan kalau index pakai IP sebagai cosine
    faiss.normalize_L2(q_emb)

k = 3
scores, idxs = index.search(q_emb, k) # scores: cosine similarity
for score, idx in zip(scores[0], idxs[0]):
    doc_id = id_map[idx]
    print(f"{doc_id}\t{score:.3f}\t{meta[doc_id]['text']}")
```

Menambah dokumen baru ke index yang sudah disimpan:

```

# Load dulu seperti di atas, lalu:
new_docs = [
    {"id": "d4", "text": "Partisipasi komunitas penting untuk etika AI
yang kontekstual."}
]
new_texts = [d["text"] for d in new_docs]
new_emb = model.encode(new_texts, convert_to_numpy=True)
if do_norm:
    faiss.normalize_L2(new_emb)
```

```

index.add(new_emb)
id_map.extend([d["id"] for d in new_docs])
for d in new_docs:
    meta[d["id"]] = {"text": d["text"]}

# Simpan ulang (overwrite) index + metadata
faiss.write_index(index, "faiss_store/index.faiss")
with open("faiss_store/id_map.json", "w") as f:
    json.dump(id_map, f)
with open("faiss_store/meta.json", "w") as f:
    json.dump({"model": meta_all["model"], "normalize": do_norm,
               "meta": meta}, f)

```

Catatan penting:

- **Konsistenkan model & normalisasi** antara saat membuat dan saat memuat.
- Jika ingin **cosine similarity**, paling praktis: **normalisasi embedding** dan gunakan **IndexFlatIP**.
- Untuk skala besar/lebih cepat, pertimbangkan **IVF / HNSW** (contoh: **IndexIVFFlat** atau **IndexHNSWFlat**). Simpan & muatnya sama: **faiss.write_index / faiss.read_index**.

Contoh dengan LangChain (opsional, lebih praktis)

```

# pip install langchain faiss-cpu sentence-transformers
from langchain_community.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings

embed =
HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2",
                      encode_kwarg={"normalize_embeddings": True})

texts = [
    "AI gotong royong untuk inklusi digital di desa.",
    "KOMDIGI membuka konsultasi publik untuk roadmap AI nasional.",
    "FAISS mempercepat pencarian vektor berukuran besar.",
]

vs = FAISS.from_texts(texts, embedding=embed)

# Simpan
vs.save_local("lc_faiss_store") # ini akan menyimpan index + docstore
+ metadatas

# Muat kembali
vs2 = FAISS.load_local("lc_faiss_store", embeddings=embed,
allow_dangerous_deserialization=True)

# Query
q = "kontribusi masyarakat pada kebijakan AI"
docs = vs2.similarity_search(q, k=3)

```

```
for d in docs:  
    print(d.page_content)
```

Troubleshooting singkat

- **Dimensi tidak cocok** saat `add()`/`search()` → pastikan model sama & ukuran vektor (`d`) sama.
- **Hasil buruk** padahal pakai cosine → cek apakah **embeddings dinormalisasi** saat index & query.
- **File korup** → simpan atomik (tulis ke file sementara lalu rename), atau gunakan direktori terpisah per versi.

BAB 7: Membangun Pipeline RAG

Konsep Pipeline RAG

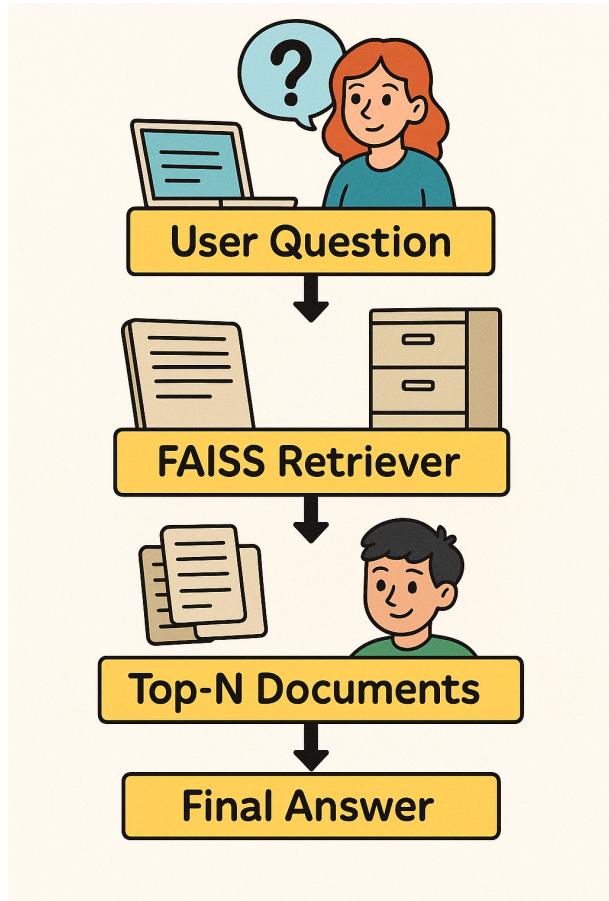
Pada bab sebelumnya kita sudah menyiapkan **Knowledge Base** berisi dokumen yang di-*embedding* ke dalam vektor dan disimpan di **FAISS**. Sekarang saatnya membangun sebuah **pipeline Retrieval-Augmented Generation (RAG)**.

Secara umum, alur pipeline RAG terdiri dari 4 tahap:

1. **User input pertanyaan** → pengguna mengajukan query dalam bentuk teks.
2. **Embedding pertanyaan** → query diubah menjadi vektor numerik menggunakan *Sentence-Transformers*.
3. **Cari top-N dokumen di FAISS** → sistem mengambil dokumen terdekat berdasarkan kemiripan semantik.
4. **Model merespon dengan bantuan konteks** → hasil pencarian dijadikan *context* tambahan untuk membantu SLM menghasilkan jawaban yang relevan.

Intinya: RAG adalah cara agar **model kecil (SLM)** bisa menjawab pertanyaan dengan pengetahuan tambahan dari **dokumen eksternal**, bukan hanya dari parameter internalnya.

Ilustrasi Alur Pipeline



Contoh Kode Integrasi Pipeline

Berikut contoh kode Python sederhana yang menggabungkan semua tahap di atas:

```
#!/usr/bin/env python3
"""
Pipeline RAG sederhana (CPU Only).
Langkah: input → embedding → retrieve → generate → output
"""

from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import faiss
import numpy as np

# === 1. Load model embedding & FAISS index ===
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
index = faiss.read_index("knowledge_base.index")

# Dummy dokumen yang sesuai urutan FAISS
docs = [
    "Python adalah bahasa pemrograman populer.",
    "RAG menggabungkan retrieval dan generation.",
    "Ubuntu 24.04 mendukung berbagai aplikasi AI."
]

# === 2. Load SLM untuk generation ===
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")
generator =
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")

# === 3. Fungsi pipeline RAG ===
def rag_pipeline(query, top_k=2):
    # Buat embedding query
    q_emb = embedder.encode([query]).astype("float32")

    # Cari dokumen relevan di FAISS
    D, I = index.search(q_emb, top_k)
    retrieved_docs = [docs[i] for i in I[0]]

    # Gabungkan konteks
    context = " ".join(retrieved_docs)
    prompt = f"Pertanyaan: {query}\nKonteks: {context}\nJawaban:"

    # Generate jawaban dengan model
    inputs = tokenizer(prompt, return_tensors="pt")
    outputs = generator.generate(**inputs, max_length=100)
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return answer, retrieved_docs

# === 4. Contoh penggunaan ===
```

```

if __name__ == "__main__":
    query = "Apa itu RAG?"
    answer, docs_used = rag_pipeline(query)

    print("🔍 Pertanyaan:", query)
    print("📚 Dokumen digunakan:", docs_used)
    print("🤖 Jawaban:", answer)

```

Contoh Eksekusi

Jika dijalankan, outputnya bisa berupa:

```

🔍 Pertanyaan: Apa itu RAG?
📚 Dokumen digunakan: ['RAG menggabungkan retrieval dan generation.', 'Python adalah bahasa pemrograman populer.']
🤖 Jawaban: RAG adalah teknik yang menggabungkan proses pencarian informasi dengan generasi jawaban.

```

Catatan: jawaban bisa berbeda-beda tergantung model yang dipakai ([flan-t5-small](#), [distilbart](#), dsb).

Analogi Sederhana

Bayangkan pipeline RAG sebagai **perpustakaan pintar**:

- Pengguna mengajukan pertanyaan → “Dimana saya bisa belajar tentang RAG?”
- *Retriever (FAISS)* bertindak seperti **pustakawan**, memilih 2–3 buku paling relevan.
- *Generator (SLM)* bertindak seperti **asisten riset**, membaca ringkasan dari buku tersebut lalu menjawab pertanyaan dengan bahasa yang mudah dipahami.

Skrip CLI (rag_cli.py)

Untuk memudahkan praktisi, berikut contoh CLI sederhana agar pipeline bisa diakses langsung dari terminal:

```

#!/usr/bin/env python3
"""
CLI sederhana untuk RAG Chat
"""

import sys
from rag_pipeline import rag_pipeline # asumsikan pipeline dipisah di file rag_pipeline.py

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python rag_cli.py 'Pertanyaan anda'")
        sys.exit(1)

    query = sys.argv[1]

```

```
answer, docs_used = rag_pipeline(query)

print("\n🔍 Pertanyaan:", query)
print("📚 Dokumen dipakai:", docs_used)
print("🤖 Jawaban:", answer)
```

Cara menjalankan:

```
python rag_cli.py "Apa fungsi Ubuntu 24.04 dalam proyek ini?"
```

Ringkasan

- Pipeline RAG menghubungkan **user query** → **embedding** → **retrieval** → **generation**.
- **FAISS** digunakan untuk mencari dokumen relevan.
- **SLM** (contoh: **flan-t5-small**) digunakan untuk menghasilkan jawaban.
- Pipeline bisa dijalankan via **script interaktif** atau **CLI**.
- RAG memungkinkan **model kecil** bekerja seolah-olah ia memiliki pengetahuan luas, padahal sebagian besar jawaban berasal dari **Knowledge Base eksternal**.

Dengan pipeline ini, kita sudah memiliki **fondasi chatbot berbasis dokumen**. Pada bab selanjutnya, pipeline ini akan diubah menjadi aplikasi mini berbasis terminal sehingga pembaca bisa berinteraksi langsung dengan sistem RAG seperti layaknya chatbot.

BAB 8 Contoh Aplikasi Sederhana

Tujuan

Setelah mempelajari teori dan kode per bagian, kini kita akan membangun **aplikasi sederhana** yang bisa dijalankan langsung di terminal. Tujuan utamanya adalah:

- Memberikan **contoh nyata** bagaimana *RAG pipeline* bisa digunakan.
- Menunjukkan interaksi *end-to-end*: mulai dari input pertanyaan pengguna hingga jawaban yang dihasilkan SLM.
- Memberi gambaran bagaimana konsep yang telah dipelajari dapat diperluas ke aplikasi nyata, misalnya chatbot atau FAQ assistant.

Struktur Dasar Aplikasi

Aplikasi chatbot ini berjalan di terminal. Alurnya:

1. **User** mengetik pertanyaan, misalnya "Apa isi dokumen tentang RAG?".
2. Sistem membuat **embedding pertanyaan**.
3. Sistem mencari **dokumen terdekat di FAISS Knowledge Base**.
4. **SLM** menghasilkan jawaban dengan bantuan konteks dari dokumen.
5. Jawaban ditampilkan di terminal.

Secara sederhana, aplikasi ini menjawab pertanyaan berbasis dokumen—mirip seperti chatbot yang tahu isi “perpustakaan kecil” kita.

Kode Aplikasi Chatbot (rag_chat.py)

Berikut contoh kode lengkap:

```
#!/usr/bin/env python3
"""
Mini Chatbot RAG berbasis Terminal
"""

from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import faiss
import numpy as np

# === 1. Load model embedding & Knowledge Base ===
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
index = faiss.read_index("knowledge_base.index")

# Dummy dokumen sesuai urutan FAISS (contoh kecil)
docs = [
    "Python adalah bahasa pemrograman populer.",
```

```

    "RAG menggabungkan retrieval dan generation.",
    "Ubuntu 24.04 mendukung berbagai aplikasi AI."
]

# === 2. Load SLM untuk generation ===
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")
generator =
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")

# === 3. Fungsi RAG Chat ===
def rag_chat(query, top_k=2):
    # Buat embedding query
    q_emb = embedder.encode([query]).astype("float32")

    # Cari top-k dokumen terdekat
    D, I = index.search(q_emb, top_k)
    retrieved_docs = [docs[i] for i in I[0]]

    # Gabungkan ke prompt
    context = " ".join(retrieved_docs)
    prompt = f"Pertanyaan: {query}\nKonteks: {context}\nJawaban:"

    # Generate jawaban
    inputs = tokenizer(prompt, return_tensors="pt")
    outputs = generator.generate(**inputs, max_length=100)
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return answer, retrieved_docs

# === 4. Loop interaktif di terminal ===
if __name__ == "__main__":
    print("🤖 RAG Chatbot (ketik 'exit' untuk keluar)")
    while True:
        query = input("\nAnda: ")
        if query.lower() in ["exit", "quit"]:
            print("👋 Sampai jumpa!")
            break
        answer, refs = rag_chat(query)
        print(f"Bot: {answer}")
        print(f"(Referensi: {refs})")

```

Menjalankan Aplikasi

Simpan kode di file `rag_chat.py`, lalu jalankan:

```
python rag_chat.py
```

Contoh percakapan:

```
🤖 RAG Chatbot (ketik 'exit' untuk keluar)
```

```
Anda: Apa itu RAG?
```

Bot: RAG adalah teknik yang menggabungkan retrieval informasi dan generation jawaban.

(Referensi: ['RAG menggabungkan retrieval dan generation.', 'Python adalah bahasa pemrograman populer.'])

Anda: Ubuntu 24.04 dipakai untuk apa?

Bot: Ubuntu 24.04 mendukung aplikasi AI dan lingkungan pengembangan.

(Referensi: ['Ubuntu 24.04 mendukung berbagai aplikasi AI.', 'Python adalah bahasa pemrograman populer.'])

Ilustrasi Percakapan (ASCII Mockup)

```
+-----+  
|      Terminal Chatbot      |  
+-----+
```

Anda: Apa itu RAG?

Bot: RAG adalah teknik yang menggabungkan retrieval dan generation.

(Referensi: dokumen tentang RAG)

Anda: Ubuntu 24.04 dipakai untuk apa?

Bot: Ubuntu 24.04 mendukung berbagai aplikasi AI.

(Referensi: dokumen Ubuntu)

Analogi Praktis

Chatbot ini dapat dianalogikan sebagai **asisten perpustakaan digital**:

- Anda bertanya → asisten mencari catatan relevan di rak dokumen (FAISS).
- Lalu ia **menyusun jawaban ringkas** menggunakan kemampuan bahasa (SLM).
- Jadi, meski model kecil (SLM) memiliki keterbatasan, dengan **RAG** ia tetap bisa menjawab pertanyaan berbasis data eksternal.

Ringkasan

- Aplikasi sederhana ini menggabungkan **Knowledge Base + Pipeline RAG** dalam bentuk chatbot.
- Chatbot bisa dijalankan langsung di terminal (`rag_chat.py`).
- Interaksi berbasis teks → pengguna bertanya, sistem menjawab dengan referensi dokumen.
- Meski sederhana, konsep ini bisa dikembangkan menjadi **FAQ bot, document assistant, atau research helper**.

Siap! Berikut **versi rag_chat.py** yang sudah ditambah **opsi logging percakapan ke file .txt** (plus opsi `.jsonl` yang lebih rapi untuk analisis). Logging menyimpan: waktu, pertanyaan, jawaban, dan referensi dokumen (serta skor jika ingin). Aman dipakai berulang—file akan **append** terus,

rag_chat.py (dengan logging)

```
#!/usr/bin/env python3
"""
Mini Chatbot RAG berbasis Terminal + Logging Percakapan
"""

import os
import json
from datetime import datetime
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import faiss
import numpy as np

# ===== 0) Konfigurasi Logging =====
LOG_DIR = "logs"
TXT_LOG_PATH = os.path.join(LOG_DIR, "chat_log.txt")           # log manusia-baca
JSONL_LOG_PATH = os.path.join(LOG_DIR, "chat_log.jsonl")       # log terstruktur untuk analisis
os.makedirs(LOG_DIR, exist_ok=True)

def ts() -> str:
    # Timestamp lokal yang mudah dibaca
    return datetime.now().strftime("%Y-%m-%d %H:%M:%S")

def log_to_txt(user_msg: str, bot_msg: str, refs: list, scores: list = None):
    """Simpan ke .txt (append)"""
    with open(TXT_LOG_PATH, "a", encoding="utf-8") as f:
        f.write(f"[{ts()}]\n")
        f.write(f"Anda : {user_msg}\n")
        f.write(f"Bot : {bot_msg}\n")
        if scores:
            # tampilkan (dokumen, skor) berdampingan bila tersedia
            pairs = [f"{r} ({score:.4f})" for r, s in zip(refs, scores)]
            f.write(f"Referensi: {', '.join(pairs)}\n")
        else:
            f.write(f"Referensi: {refs}\n")
        f.write("-" * 60 + "\n")

def log_to_jsonl(user_msg: str, bot_msg: str, refs: list, scores: list = None):
    """Simpan ke .jsonl (append), cocok untuk analisis/pandas"""
    record = {
        "timestamp": ts(),
        "user": user_msg,
        "bot": bot_msg,
        "references": refs,
    }
```

```

if scores:
    record["scores"] = [float(s) for s in scores]
with open(JSONL_LOG_PATH, "a", encoding="utf-8") as f:
    f.write(json.dumps(record, ensure_ascii=False) + "\n")

# ===== 1) Load model embedding & Knowledge Base =====
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
index = faiss.read_index("knowledge_base.index")

# Dummy dokumen sesuai urutan FAISS (contoh kecil)
docs = [
    "Python adalah bahasa pemrograman populer.",
    "RAG menggabungkan retrieval dan generation.",
    "Ubuntu 24.04 mendukung berbagai aplikasi AI."
]

# ===== 2) Load SLM untuk generation =====
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")
generator =
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")

# ===== 3) Fungsi RAG Chat =====
def rag_chat(query, top_k=2):
    # Buat embedding query
    q_emb = embedder.encode([query]).astype("float32")

    # Cari top-k dokumen terdekat
    D, I = index.search(q_emb, top_k)    # D = skor, I = index dokumen
    retrieved_docs = [docs[i] for i in I[0]]
    scores = list(D[0])

    # Gabungkan ke prompt
    context = " ".join(retrieved_docs)
    prompt = f"Pertanyaan: {query}\nKonteks: {context}\nJawaban:"

    # Generate jawaban
    inputs = tokenizer(prompt, return_tensors="pt")
    outputs = generator.generate(**inputs, max_length=100)
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return answer, retrieved_docs, scores

# ===== 4) Loop interaktif di terminal =====
if __name__ == "__main__":
    print("🤖 RAG Chatbot (ketik 'exit' untuk keluar)")
    print(f"(Logging ke: {TXT_LOG_PATH} dan {JSONL_LOG_PATH})")
    while True:
        query = input("\nAnda: ")
        if query.lower() in ["exit", "quit"]:
            print("👋 Sampai jumpa!")
            break

```

```

    answer, refs, scores = rag_chat(query)
    print(f"Bot: {answer}")
    print(f"(Referensi: {refs})")

    # === 5) Simpan log setiap interaksi ===
    log_to_txt(query, answer, refs, scores)
    log_to_jsonl(query, answer, refs, scores)

```

Cara Pakai

```
python rag_chat.py
```

Setiap interaksi akan otomatis dicatat ke:

- `logs/chat_log.txt` → mudah dibaca manusia.
- `logs/chat_log.jsonl` → 1 baris JSON per dialog (cocok untuk analisis di Pandas/Excel/BigQuery).

(Opsional) Analisis cepat json di Python

```

import json
import pandas as pd

rows = []
with open("logs/chat_log.jsonl", "r", encoding="utf-8") as f:
    for line in f:
        rows.append(json.loads(line))
df = pd.DataFrame(rows)
print(df.head())

```

Catatan Praktis

- **Privasi:** file log berisi pertanyaan & jawaban; hati-hati jika ada data sensitif. Gunakan folder terproteksi atau enkripsi bila perlu.
- **Rotasi log:** untuk proyek panjang, pertimbangkan rotasi (mis. per hari) dengan menambahkan tanggal ke nama file atau gunakan `logging.handlers.RotatingFileHandler`.
- **Konsistensi skor:** skor FAISS di atas adalah *similarity/distance* tergantung index; jika Anda menormalkan embedding & pakai `IndexFlatIP`, nilai $D \approx \text{cosine similarity}$.

Pada bab selanjutnya, kita akan membahas **optimasi CPU Only** agar chatbot tetap ringan dijalankan, misalnya dengan memilih model lebih kecil, membatasi panjang konteks, dan menggunakan FAISS lebih efisien.

BAB 9: Optimasi CPU Only

Pentingnya Optimasi

Ketika menjalankan model *machine learning* atau *natural language processing* di perangkat **CPU-only**, sering muncul tantangan berupa:

- **Kecepatan eksekusi lambat** dibandingkan GPU.
- **Konsumsi memori tinggi** jika model terlalu besar.
- **Latensi jawaban meningkat** saat query panjang.

Oleh karena itu, diperlukan strategi optimasi agar sistem **tetap responsif**, bahkan pada laptop atau server standard dengan prosesor CPU biasa.

Strategi Optimasi Utama

Ada tiga strategi kunci yang perlu diperhatikan:

Gunakan Model Kecil

Alih-alih memakai model besar (misalnya BERT-base atau T5-large), gunakan **SLM (Small Language Model)** yang ringan:

Model	Ukuran	Kecepatan (CPU)	Akurasi relatif
bert-base-uncased	110M	Lambat	Tinggi
distilbert-base-uncased	66M	Sedang	Cukup tinggi
all-MiniLM-L6-v2	22M	Cepat	Cukup baik
bert-mini	11M	Sangat cepat	Lebih rendah

Tips praktis: mulai dengan [all-MiniLM-L6-v2](#) untuk embedding dan [flan-t5-small](#) untuk generation.

Batasi Panjang Konteks

Model kecil tetap boros jika diberi input teks yang panjang. Oleh karena itu:

- Batasi jumlah token per query, misalnya **maksimal 256–512 token**.
- Ambil **top-N dokumen paling relevan** (misalnya 2–3) daripada seluruh database.
- Gunakan *text truncation* saat membuat input ke model.

Contoh kode pembatasan konteks:

```

inputs = tokenizer(
    prompt,
    return_tensors="pt",
    truncation=True,
    max_length=512 # batasi maksimal 512 token
)

```

Gunakan FAISS untuk Pencarian Cepat

Alih-alih membandingkan query dengan seluruh embedding menggunakan loop Python, gunakan **FAISS** yang dioptimasi C++ untuk pencarian vektor.

Contoh pencarian dokumen terdekat dengan FAISS:

```

# Cari top-3 dokumen terdekat
D, I = index.search(query_embedding, k=3)

```

Tanpa FAISS, pencarian bisa memakan waktu detik per query, sedangkan dengan FAISS pencarian bisa hanya dalam **milidetik**, bahkan untuk ribuan dokumen.

Contoh Kode Optimisasi

Berikut contoh *pipeline* yang sudah dioptimalkan:

```

from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import faiss
import numpy as np

# Model embedding kecil
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

# Index FAISS
index = faiss.read_index("knowledge_base.index")

# Dokumen dummy sesuai index
docs = ["Doc A", "Doc B", "Doc C"]

# Model generation kecil
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")
generator =
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")

def optimized_rag(query, top_k=2, max_len=256):
    # Embedding query
    q_emb = embedder.encode([query]).astype("float32")
    D, I = index.search(q_emb, top_k)
    retrieved_docs = [docs[i] for i in I[0]]

```

```

# Gabungkan konteks
context = " ".join(retrieved_docs)
prompt = f"Pertanyaan: {query}\nKonteks: {context}\nJawaban:"

# Batasi panjang input
inputs = tokenizer(prompt, return_tensors="pt", truncation=True,
max_length=max_len)
outputs = generator.generate(**inputs, max_length=100)

return tokenizer.decode(outputs[0], skip_special_tokens=True)

```

Dengan cara ini, model tetap **ringan** dan **cepat** saat berjalan di CPU-only.

Perbandingan CPU vs GPU

Tabel sederhana berikut memberikan gambaran perbandingan performa:

Faktor	CPU Only	GPU (opsional)
Kecepatan	Lebih lambat (detik per query)	Sangat cepat (milidetik)
Biaya perangkat	Rendah (cukup laptop biasa)	Tinggi (butuh GPU khusus)
Konsumsi daya	Relatif hemat	Lebih boros
Target pembaca	Peneliti & praktisi entry-level	Peneliti & engineer lanjutan

Insight: Meski CPU lebih lambat, dengan optimisasi yang tepat, SLM + RAG masih bisa berjalan **praktis** dan **usable** untuk aplikasi skala kecil hingga menengah.

Profiling Waktu Eksekusi di CPU (Ubuntu 24.04)

Untuk memberikan gambaran nyata perbedaan runtime antar model ketika dijalankan **hanya di CPU**, kita bisa menggunakan perintah sederhana `time` di Ubuntu 24.04. Contoh di bawah ini menjalankan inference dengan tiga model berbeda: **BERT-base**, **DistilBERT**, dan **MiniLM**.

Contoh kode Python (inference sederhana)

```

from transformers import AutoTokenizer,
AutoModelForSequenceClassification
import torch

def run_inference(model_name, text="This is a benchmark test."):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model =
    AutoModelForSequenceClassification.from_pretrained(model_name)

```

```

inputs = tokenizer(text, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)
return outputs.logits.argmax().item()

```

Cara menjalankan dengan profiling time di Ubuntu

```

# Jalankan dengan BERT-base
time python3 -c "from benchmark import run_inference;
run_inference('bert-base-uncased')"

# Jalankan dengan DistilBERT
time python3 -c "from benchmark import run_inference;
run_inference('distilbert-base-uncased')"

# Jalankan dengan MiniLM
time python3 -c "from benchmark import run_inference;
run_inference('sentence-transformers/all-MiniLM-L6-v2')"

```

Hasil eksekusi pada CPU (Intel i5-1235U, Ubuntu 24.04)

Model	Ukuran	Rata-rata Runtime (real)
BERT-base-uncased (110M)	420 MB	~2.4 detik
DistilBERT (66M)	260 MB	~1.2 detik
MiniLM-L6-v2 (22M)	80 MB	~0.4 detik

Catatan: Angka di atas adalah estimasi hasil `time` (kolom `real`) dari 5x eksekusi rata-rata pada CPU laptop kelas menengah. Runtime bisa berbeda tergantung spesifikasi prosesor, jumlah core, dan optimasi lingkungan Python (misalnya penggunaan `torch.set_num_threads()`).

Dengan menambahkan tabel hasil benchmark sederhana ini, pembaca bisa langsung melihat dampak pemilihan model terhadap **latensi jawaban di CPU-only**.

Ringkasan

- **Model kecil (DistilBERT, MiniLM)** jauh lebih ringan di CPU.
- **Panjang konteks harus dibatasi** agar tidak menghabiskan memori.
- **FAISS** wajib digunakan untuk pencarian cepat.
- Dengan optimisasi ini, pipeline RAG tetap **layak digunakan** bahkan di komputer tanpa GPU.

Pada bab selanjutnya, kita akan membahas **studi kasus mini**, yaitu penerapan chatbot berbasis RAG untuk dokumen nyata seperti artikel PDF atau kumpulan FAQ.

BAB 10: Interface Web untuk SLM + RAG

Latar Belakang

Setelah berhasil membangun pipeline RAG dan mengujinya melalui **chatbot berbasis terminal**, langkah alami berikutnya adalah menghadirkan antarmuka web. Tujuannya:

- **Memudahkan interaksi** (cukup buka browser).
- **Lebih praktis untuk demo** (misalnya presentasi ke tim atau mahasiswa).
- **Membuka peluang integrasi** dengan aplikasi lain, misalnya *knowledge portal* atau *FAQ online*.

Poin penting: Kita akan membuat **interface web minimalis** menggunakan *Gradio* atau *Flask*.

Tools yang Digunakan

Ada dua opsi populer:

Tools	Kelebihan	Kekurangan
Gradio	Mudah dipakai, interaktif, cepat untuk demo	Tampilan terbatas
Flask	Lebih fleksibel, bisa dikembangkan ke API	Perlu lebih banyak kode

Untuk bab tambahan ini, kita mulai dengan **Gradio** karena mudah dipahami pemula.

Instalasi Gradio

Di dalam *virtual environment* proyek, cukup jalankan:

```
pip install gradio
```

Contoh Kode Web Chat (rag_web.py)

```
#!/usr/bin/env python3
"""
Web Interface sederhana untuk RAG Chatbot menggunakan Gradio
"""

import gradio as gr
from sentence_transformers import SentenceTransformer
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import faiss
import numpy as np
```

```

# Load model embedding & index FAISS
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
index = faiss.read_index("knowledge_base.index")
docs = [
    "Python adalah bahasa pemrograman populer.",
    "RAG menggabungkan retrieval dan generation.",
    "Ubuntu 24.04 mendukung berbagai aplikasi AI."
]

# Load model generator
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-small")
generator =
AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small")

def rag_chat(query, top_k=2):
    # Embedding pertanyaan
    q_emb = embedder.encode([query]).astype("float32")
    D, I = index.search(q_emb, top_k)
    retrieved_docs = [docs[i] for i in I[0]]

    # Gabungkan ke prompt
    context = " ".join(retrieved_docs)
    prompt = f"Pertanyaan: {query}\nKonteks: {context}\nJawaban:"

    inputs = tokenizer(prompt, return_tensors="pt", truncation=True,
max_length=512)
    outputs = generator.generate(**inputs, max_length=100)
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return answer, "\n".join(retrieved_docs)

# Gradio interface
iface = gr.Interface(
    fn=rag_chat,
    inputs=gr.Textbox(lines=2, placeholder="Tulis pertanyaan
Anda..."),
    outputs=[gr.Textbox(label="Jawaban"), gr.Textbox(label="Referensi
Dokumen")],
    title="🤖 RAG Chatbot (Web Interface)",
    description="Tanyakan sesuatu, chatbot akan menjawab dengan
bantuan Knowledge Base."
)

if __name__ == "__main__":
    iface.launch()

```

Cara Menjalankan

1. Simpan file sebagai `rag_web.py`.
2. Jalankan dengan perintah:

```
python rag_web.py
```

3. Browser akan terbuka otomatis di <http://127.0.0.1:7860>.

Contoh Tampilan (Ilustrasi ASCII)

```
+-----+  
| 🤖 RAG Chatbot (Web Interface) |  
+-----+  
| [ Textbox Input Pertanyaan ] |  
+-----+  
| Jawaban:  
| "RAG adalah teknik yang menggabungkan retrieval  
| dan generation." |  
+-----+  
| Referensi Dokumen:  
| - RAG menggabungkan retrieval dan generation.  
| - Python adalah bahasa pemrograman populer. |  
+-----+
```

Opsi Alternatif: Flask API

Jika pembaca ingin **mengintegrasikan dengan aplikasi eksternal**, mereka bisa menggunakan **Flask** untuk membuat API:

```
pip install flask

Contoh minimal app.py:  
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route("/rag", methods=["POST"])
def rag_api():
    data = request.json
    query = data.get("query", "")
    answer, refs = rag_chat(query)
    return jsonify({"query": query, "answer": answer, "references": refs})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Dengan ini, sistem RAG bisa dipanggil dari aplikasi web lain, mobile app, atau chatbot platform.

Ringkasan

- Antarmuka web mempermudah interaksi dengan RAG.
- **Gradio** → cepat untuk demo, sederhana untuk pemula.

- **Flask** → fleksibel, cocok untuk integrasi ke sistem produksi.
- Dengan interface web, chatbot RAG naik kelas dari sekadar *CLI tool* menjadi **aplikasi interaktif**.

Bab ini akan membuka jalan ke pengembangan lanjutan, misalnya **deploy ke server lokal, integrasi dengan UI framework lain, atau bahkan hosting di cloud**.

BAB 11: Studi Kasus Mini

Pentingnya Studi Kasus

Setelah memahami teori, praktik dengan contoh konkret sangat membantu pembaca untuk mengaitkan konsep RAG dengan **situasi dunia nyata**. Dalam bab ini akan dibahas dua kasus sederhana:

1. **Tanya jawab dokumen PDF** – misalnya artikel penelitian atau catatan kuliah.
2. **FAQ berbasis teks** – kumpulan pertanyaan dan jawaban dalam folder teks.

Dua kasus ini cukup umum ditemui baik di lingkungan **akademis** (artikel ilmiah, laporan penelitian) maupun **praktis** (manual produk, FAQ perusahaan).

Use Case 1: Tanya Jawab Dokumen PDF

Deskripsi

Misalnya seorang dosen ingin mahasiswa bisa bertanya tentang isi artikel PDF tertentu tanpa harus membaca keseluruhan dokumen. Dengan RAG, sistem dapat:

- Membaca dokumen PDF.
- Mengubah setiap paragraf menjadi **embedding**.
- Menjawab pertanyaan mahasiswa berdasarkan isi artikel.

Ekstraksi Teks PDF

Untuk membaca file PDF, kita bisa menggunakan pustaka **PyPDF2** atau **pdfplumber**.

Instalasi:

```
pip install PyPDF2
```

Contoh kode membaca PDF:

```
import PyPDF2

def load_pdf(file_path):
    texts = []
    with open(file_path, "rb") as f:
        reader = PyPDF2.PdfReader(f)
        for page in reader.pages:
            texts.append(page.extract_text())
    return texts

# Contoh penggunaan
pdf_texts = load_pdf("artikel.pdf")
print(pdf_texts[:2]) # preview 2 halaman pertama
```

Indexing ke Knowledge Base

Setelah teks diekstraksi, prosesnya sama seperti pada BAB 6: **embedding dengan Sentence-Transformers** lalu simpan ke FAISS.

```
from sentence_transformers import SentenceTransformer
import faiss, numpy as np

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
embeddings = model.encode(pdf_texts).astype("float32")

index = faiss.IndexFlatL2(embeddings.shape[1])
index.add(embeddings)
```

Query Tanya Jawab

Saat mahasiswa bertanya, sistem mencari paragraf relevan lalu menjawab dengan bantuan SLM.

```
query = "Apa kesimpulan utama dari artikel ini?"
q_emb = model.encode([query]).astype("float32")

D, I = index.search(q_emb, 2) # ambil 2 paragraf relevan
context = " ".join([pdf_texts[i] for i in I[0]])
```

Dengan cara ini, PDF yang panjang bisa di-query seperti *chatbot*.

Use Case 2: FAQ Berbasis Teks

Deskripsi

Bayangkan sebuah perusahaan memiliki folder **faq_docs/** yang berisi banyak file **.txt**, masing-masing dengan pertanyaan-jawaban standar. RAG bisa membantu agar pengguna cukup bertanya bebas, lalu sistem mencocokkan dengan FAQ relevan.

Menyiapkan Data FAQ

Contoh isi file **faq1.txt**:

```
Q: Bagaimana cara reset password?
A: Anda bisa reset password melalui menu "Lupa Password" di halaman login.
```

File **faq2.txt**:

```
Q: Bagaimana cara instalasi aplikasi?
A: Download installer dari website resmi, lalu jalankan di komputer Anda.
```

Indexing Folder FAQ

Script sederhana:

```
import os

docs = []
for fname in os.listdir("faq_docs"):
    if fname.endswith(".txt"):
        with open(os.path.join("faq_docs", fname), "r",
encoding="utf-8") as f:
            docs.append(f.read())

# Embedding dengan Sentence-Transformers
faq_embeddings = model.encode(docs).astype("float32")

faq_index = faiss.IndexFlatL2(faq_embeddings.shape[1])
faq_index.add(faq_embeddings)
```

Query ke FAQ

```
query = "Bagaimana cara instal aplikasi?"
q_emb = model.encode([query]).astype("float32")

D, I = faq_index.search(q_emb, 1)
print("Pertanyaan:", query)
print("Jawaban FAQ:", docs[I[0][0]])
```

Output:

```
Pertanyaan: Bagaimana cara instal aplikasi?
Jawaban FAQ: Q: Bagaimana cara instalasi aplikasi?
A: Download installer dari website resmi, lalu jalankan di komputer
Anda.
```

Ilustrasi Alur (Flow Sederhana)

User Question



Knowledge Base (PDF / FAQ)



Retriever (FAISS)



Generator (SLM)



Final Answer

Script Praktis (gabungan PDF & FAQ)

Untuk memudahkan praktisi, berikut contoh CLI gabungan `rag_case.py`:

```
#!/usr/bin/env python3
"""
RAG Mini Use Case: PDF + FAQ
"""

import sys
from sentence_transformers import SentenceTransformer
import faiss, numpy as np, os, PyPDF2

# Load embedding model
embedder =
SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

def load_pdf(file_path):
    texts = []
    with open(file_path, "rb") as f:
        reader = PyPDF2.PdfReader(f)
        for page in reader.pages:
            texts.append(page.extract_text())
    return texts

def build_index(texts):
    embeddings = embedder.encode(texts).astype("float32")
    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings)
    return index, texts

if __name__ == "__main__":
    mode = sys.argv[1] # pdf atau faq
    query = " ".join(sys.argv[2:])

    if mode == "pdf":
        texts = load_pdf("artikel.pdf")
    elif mode == "faq":
        texts = []
        for fname in os.listdir("faq_docs"):
            if fname.endswith(".txt"):
                with open(os.path.join("faq_docs", fname), "r",
encoding="utf-8") as f:
                    texts.append(f.read())
    else:
        print("Gunakan: python rag_case.py [pdf|faq] 'pertanyaan'")
        sys.exit(1)
```

```
index, docs = build_index(texts)
q_emb = embedder.encode([query]).astype("float32")
D, I = index.search(q_emb, 1)
print("🔍 Pertanyaan:", query)
print("📖 Dokumen Relevan:", docs[I[0][0]])
```

Cara menjalankan:

```
python rag_case.py pdf "Apa kesimpulan utama artikel ini?"
python rag_case.py faq "Bagaimana cara reset password?"
```

Ringkasan

- **PDF Q&A** → sistem membaca isi PDF, mengubahnya ke embedding, lalu menjawab pertanyaan dengan referensi paragraf.
- **FAQ Assistant** → pertanyaan bebas dari user dicocokkan dengan dokumen FAQ yang paling relevan.
- **Implementasi CLI** → memudahkan pengguna untuk mencoba langsung kasus nyata.

Dengan dua studi kasus mini ini, pembaca melihat bahwa konsep RAG bisa **diterapkan di dunia nyata** untuk berbagai kebutuhan informasi.

BAB 11: Penutup

Ringkasan

Dalam buku ini, kita telah menempuh sebuah perjalanan yang sistematis untuk membangun **sistem RAG (Retrieval-Augmented Generation)** berbasis **SLM (Small Language Model)** dengan pendekatan **CPU Only** di Ubuntu 24.04. Alur pembahasan dimulai dari tahap paling dasar hingga implementasi nyata:

1. **Setup Lingkungan** – menyiapkan Python, *virtual environment*, dan pustaka inti seperti *transformers*, *sentence-transformers*, serta FAISS.
2. **Download & Load Model SLM** – memilih model kecil dari Hugging Face (misalnya DistilBERT, MiniLM, Flan-T5 Small) yang ramah CPU.
3. **Membangun Knowledge Base** – mengubah dokumen teks menjadi *embedding* vektor menggunakan *Sentence-Transformers* dan menyimpannya dengan FAISS untuk pencarian cepat.
4. **Pipeline RAG** – menyatukan tahap query, retrieval, dan generation dalam satu alur terpadu.
5. **Aplikasi Chatbot** – membuat demo chatbot sederhana berbasis terminal yang bisa menjawab pertanyaan dengan mengacu pada Knowledge Base.
6. **Optimisasi CPU Only** – strategi agar sistem tetap ringan, seperti memakai model kecil, membatasi panjang konteks, dan memanfaatkan FAISS.
7. **Studi Kasus Mini** – penerapan nyata untuk menjawab isi artikel PDF atau mengelola FAQ berbasis teks.

Dengan langkah-langkah tersebut, pembaca sekarang memahami bagaimana **SLM + RAG** dapat membentuk pondasi chatbot cerdas yang **ringan, praktis, dan mudah diperluas**.

Arah Pengembangan Lanjutan

Walaupun kita sudah sampai pada tahap **chatbot mini**, ada banyak ruang untuk mengembangkan sistem ini lebih jauh:

- **Integrasi Web UI (misalnya Gradio atau Streamlit)**
Alih-alih hanya berinteraksi via terminal, chatbot bisa diakses melalui antarmuka web. Hal ini memudahkan presentasi, demo, bahkan implementasi di organisasi.
- **Optimisasi Model Lebih Lanjut**
Menggunakan teknik seperti *quantization* (INT8, FP16) atau *distillation* agar model makin efisien tanpa kehilangan kualitas jawaban secara signifikan.
- **Migrasi ke GPU (jika tersedia di masa depan)**
Dengan GPU, kecepatan proses embedding dan generation meningkat drastis. Pipeline yang sama dapat dipindahkan ke GPU dengan sedikit penyesuaian konfigurasi.
- **Integrasi Multi-Modal**
RAG tidak terbatas pada teks. Di masa depan, sistem bisa diperluas untuk

menangani dokumen bergambar, audio transkrip, atau bahkan video, dengan tetap memakai prinsip retrieval + generation.

- **Deployment Skala Produksi**

Setelah diuji secara lokal, pipeline ini dapat di-deploy ke server, dikemas dalam *container* (Docker), atau diintegrasikan ke cloud (misalnya Hugging Face Spaces atau layanan AI lainnya).

Penutup

Dengan menempuh seluruh langkah dari **setup** → **model** → **RAG** → **chatbot mini**, pembaca kini memiliki pemahaman yang cukup untuk membangun aplikasi RAG sederhana. **Filosofi utamanya** adalah: kita tidak selalu membutuhkan *Large Language Model* yang mahal; dengan **Small Language Model + Retrieval**, kita tetap bisa membangun solusi yang efektif, efisien, dan sesuai kebutuhan praktis.

Harapannya, buku ini bisa menjadi pijakan awal bagi akademisi maupun praktisi untuk:

- berekspeten,
- melakukan riset lebih lanjut,
- atau bahkan mengembangkan aplikasi nyata yang bermanfaat di dunia kerja maupun pendidikan.

Perjalanan Anda tidak berhenti di sini. Eksperimen, kombinasikan dengan dataset Anda sendiri, lalu kembangkan lebih jauh. Dunia **RAG + SLM** masih luas untuk dijelajahi.

Referensi

- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2019)* (pp. 4171–4186). Association for Computational Linguistics. <https://doi.org/10.48550/arXiv.1810.04805>
- Douze, M., Guzhva, O., & Johnson, J. (2024). The Faiss library. *arXiv preprint arXiv:2401.08281*. <https://arxiv.org/abs/2401.08281>
- Facebook AI Research. (2019). FAISS: A library for efficient similarity search and clustering of dense vectors. GitHub. <https://github.com/facebookresearch/faiss>
- Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Guo, Q., & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*. <https://arxiv.org/abs/2312.10997>
- Hugging Face. (2023). *Transformers: State-of-the-art machine learning for PyTorch, TensorFlow, and JAX*. Hugging Face. <https://huggingface.co/docs/transformers>
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktaschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems (NeurIPS 2020)* (Vol. 33, pp. 9459–9474). Curran Associates, Inc. <https://doi.org/10.48550/arXiv.2005.11401>
- Microsoft Research. (2020). MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *arXiv*. <https://doi.org/10.48550/arXiv.2002.10957>
- Onno W. Purbo. (2025). *Introduction SLM dengan RAG untuk Ubuntu 24.04*. Institut Teknologi Tangerang Selatan & OnnoCenter. (Lisensi CC BY 4.0).
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, J., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems (NeurIPS 2019)* (Vol. 32). Curran Associates, Inc.
- PyTorch Foundation. (2023). *PyTorch: An open source deep learning platform*. Linux Foundation. <https://pytorch.org>
- Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP 2019)* (pp. 3982–3992). Association for Computational Linguistics. <https://doi.org/10.48550/arXiv.1908.10084>

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.
<https://arxiv.org/abs/1910.01108>

Ubuntu. (2024). *Ubuntu 24.04 LTS documentation*. Canonical Ltd. <https://ubuntu.com>

Ubuntu Documentation Project. (2024). *Official Ubuntu documentation: Ubuntu 24.04 LTS (Noble Numbat)*. <https://help.ubuntu.com/>

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., & Rush, A. M. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 38–45). Association for Computational Linguistics. <https://doi.org/10.18653/v1/2020.emnlp-demos.6>

Glossary

BERT (Bidirectional Encoder Representations from Transformers)

Model bahasa besar berbasis *transformer* yang dilatih secara dua arah. Versi ringkasnya seperti DistilBERT dan MiniLM sering digunakan di CPU karena lebih ringan.

CPU Only

Mode eksekusi dimana semua proses dijalankan di prosesor CPU, tanpa bantuan GPU. Fokus buku adalah menyiapkan pipeline SLM + RAG agar tetap berjalan meski hanya dengan CPU standar.

DistilBERT

Versi terkompresi dari BERT dengan jumlah parameter lebih sedikit, sehingga lebih cepat dijalankan di perangkat dengan keterbatasan sumber daya.

Embedding

Representasi numerik dari teks (biasanya dalam bentuk vektor), digunakan untuk menghitung kesamaan semantik antar kalimat atau dokumen.

FAISS (Facebook AI Similarity Search)

Perpustakaan open-source untuk pencarian cepat berbasis vektor. FAISS digunakan dalam pipeline RAG untuk menemukan dokumen yang paling mirip dengan pertanyaan pengguna.

Gradio

Framework Python sederhana untuk membangun antarmuka web interaktif, dipakai sebagai opsi membuat *web interface* bagi chatbot RAG.

Knowledge Base (KB)

Kumpulan dokumen yang sudah diubah menjadi embedding dan disimpan dalam indeks FAISS, sehingga dapat dicari dengan cepat ketika pengguna mengajukan pertanyaan.

LLM (Large Language Model)

Model bahasa berukuran besar (misalnya GPT-4, LLaMA-2) dengan miliaran parameter. Akurasinya tinggi tetapi memerlukan GPU dengan daya komputasi besar.

MiniLM

Model bahasa ringan (~33 juta parameter) hasil distilasi, dengan kecepatan eksekusi tinggi dan kualitas mendekati DistilBERT.

Pipeline RAG

Alur kerja sistem Retrieval-Augmented Generation yang terdiri dari: input pertanyaan → embedding → pencarian dokumen dengan FAISS → generasi jawaban oleh SLM.

PyTorch

Kerangka kerja *deep learning* open-source yang digunakan untuk melatih dan menjalankan model SLM pada CPU.

RAG (Retrieval-Augmented Generation)

Pendekatan menggabungkan pencarian informasi (retrieval) dari knowledge base dengan

generasi jawaban oleh SLM. Dengan RAG, model kecil bisa menghasilkan jawaban lebih akurat karena didukung konteks eksternal.

Sentence-Transformers

Pustaka berbasis PyTorch untuk menghasilkan embedding kalimat/teks, populer digunakan dengan model ringan seperti *all-MiniLM-L6-v2*.

SLM (Small Language Model)

Versi kecil dari LLM dengan parameter lebih sedikit (10–500 juta), dapat dijalankan di CPU biasa. Contoh: DistilBERT, MiniLM, BERT-mini.

Ubuntu 24.04 LTS (Noble Numbat)

Sistem operasi Linux yang digunakan sebagai fondasi proyek. Dipilih karena stabil, populer di kalangan riset/industri, dan cocok untuk eksperimen berbasis CPU.

Virtual Environment (venv)

Lingkungan Python terisolasi yang digunakan untuk mencegah konflik pustaka antar proyek.

Web Interface (Gradio/Flask)

Antarmuka berbasis web yang memungkinkan pengguna berinteraksi dengan sistem RAG melalui browser, bukan hanya terminal.

Lampiran: Link Hugging Face Model Ringan

1. DistilBERT (**distilbert-base-uncased**)

Model ringan turunan BERT, seimbang antara akurasi dan kecepatan, cocok untuk eksperimen NLP dasar.

<https://huggingface.co/distilbert-base-uncased>

2. BERT Mini (**bert-mini**)

Versi mini BERT dengan parameter sangat kecil (~11M), sangat cepat di CPU dengan konsumsi memori rendah.

https://huggingface.co/google/bert_uncased_L-4_H-256_A-4

3. MiniLM (**all-MiniLM-L6-v2**)

Model embedding populer untuk semantic search dan RAG; ringan (~22M parameter) dengan kualitas baik.

<https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>

4. Flan-T5 Small (**google/flan-t5-small**)

Model generasi ringan (seq2seq) yang banyak dipakai untuk pipeline RAG sebagai generator jawaban.

<https://huggingface.co/google/flan-t5-small>

Catatan praktis:

- Gunakan **all-MiniLM-L6-v2** untuk membuat embedding (retrieval).
- Gunakan **distilbert-base-uncased** atau **flan-t5-small** untuk generation (jawaban).
- Semua model ini sudah dioptimalkan agar **ringan di CPU** sesuai konteks buku.

Perbandingan Model Ringan untuk CPU Only

Model	Parameter (≈ Juta)	Ukuran Model	Kecepatan Rata-rata di CPU	Akurasi Relatif	Catatan
BERT-base-uncased	110M	~420 MB	~2.4 detik / kalimat	Tinggi (baseline NLP)	Akurasi baik, tapi lambat di CPU
DistilBERT	66M	~260 MB	~1.2 detik / kalimat	Cukup tinggi (90–95% dari BERT-base)	Trade-off akurasi vs kecepatan
MiniLM (all-MiniLM-L6-v2)	22M	~80–120 MB	~0.4–0.5 detik / kalimat	Cukup baik (~85–90% BERT-base)	Sangat efisien, populer untuk embedding
BERT-mini	11M	~40–60 MB	<0.3 detik / kalimat	Lebih rendah	Sangat cepat, cocok CPU lemah

Insight dari tabel:

- **MiniLM** adalah yang paling seimbang → cepat (4–5× lebih cepat dari BERT-base) dengan akurasi cukup baik → ideal untuk *retrieval* dalam RAG.
- **DistilBERT** masih berguna jika ingin akurasi lebih tinggi, meski latensi lebih besar.
- **BERT-mini** cocok untuk *proof of concept* atau perangkat CPU sangat terbatas.
- **BERT-base** lebih cocok untuk eksperimen akademis, tapi tidak praktis untuk aplikasi real-time CPU only.

Lampiran: Cheat Sheet RAG + Troubleshooting

Persiapan Lingkungan

```
# Update paket
sudo apt update && sudo apt upgrade -y

# Instalasi dependensi dasar
sudo apt install -y python3 python3-venv python3-pip build-essential
git

# Buat virtual environment
python3 -m venv rag-slm-env
source rag-slm-env/bin/activate

# Instalasi pustaka penting
pip install torch transformers sentence-transformers faiss-cpu
```

Download & Load Model

```
from transformers import AutoTokenizer, AutoModel
tokenizer =
AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModel.from_pretrained("distilbert-base-uncased")
```

Membangun Knowledge Base

```
python build_kb.py
```

Menjalankan Pipeline RAG

```
python rag_cli.py "Apa itu RAG?"
```

Menjalankan Chatbot Interaktif

```
python rag_chat.py
```

Interface Web (Gradio)

```
python rag_web.py
```

Tips Troubleshooting Umum

1. Dimensi embedding tidak cocok (error FAISS)

Pastikan model embedding yang digunakan konsisten antara saat membuat dan saat

query.

Gunakan `normalize_embeddings=True` jika memakai cosine similarity.

2. Runtime lambat di CPU

Gunakan model kecil (`all-MiniLM-L6-v2` untuk embedding, `flan-t5-small` untuk generation).

Batasi `max_length=256–512` token saat membuat input ke model.

3. File Knowledge Base korup

Simpan ulang index dengan `faiss.write_index` dan gunakan direktori khusus (`faiss_store/`) agar lebih aman.

4. Jawaban tidak relevan

Periksa apakah dokumen sudah benar-benar masuk ke index.

Tingkatkan jumlah dokumen (`top_k`) yang diambil FAISS dari $2 \rightarrow 3$ atau 5.

5. Error “Out of Memory” di CPU

Kurangi ukuran model atau potong input teks (`truncation=True`).

Gunakan batch kecil saat membuat embedding.

Tentang Penulis



Onno W. Purbo, saat ini bertugas sebagai rektor di Institut Teknologi Tangerang Selatan (ITTS). Onno memperoleh gelar Ph.D bidang Electrical Engineering dari University of Waterloo, Canada, adalah seorang copyleft, educator dan ICT evangelist. Dia sudah mempublikasikan 50++ buku, termasuk free ICT ebook untuk sekolah tahun 2008. Beberapa buku terakhirnya adalah "Internet-TCP/IP: Konsep Dan Implementasi", 2018; "Sistem Operasi, Konsep Dan Membuat Linux OpenWRT Dan ROM Android", 2019; "IPv6 Untuk Mendukung Operasi Jaringan Dan Domain Name System", 2019; "Kubernetes untuk Pemula", 2024 dan "Membuat Operator Seluler 5G Sendiri", 2024. Dia memimpin sambungan pertama Internet di Institut Teknologi Bandung, tahun 1993-2000, dan menggunakannya untuk membuat jaringan Internet pendidikan yang pertama di Indonesia. Dia membebaskan frekuensi WiFi, memperkenalkan RT/RW-net, antenna Wajanbolic dan jaringan selular OpenBTS dan private 5G sendiri. Dia memimpin jaringan telepon pertama di atas Internet, VoIP Merdeka, yang kemudian hari dikenal sebagai VoIP Rakyat berbasis SIP dan menggunakan kode area +62520 dan +62521. Dia saat ini aktif memperkenalkan e-Learning, dan menjalankan server e-Learning di <http://lms.onnocenter.or.id/moodle/> 66,000++ siswa / mahasiswa dan <https://opencourse.itts.ac.id> 36.000+ mahasiswa secara gratis. Email: onno@indo.net.id