Trial_Modified

January 18, 2026

1 Chubbyemu-Style Medical Storytelling Twin - Enhanced RAG System

This notebook implements an advanced Retrieval-Augmented Generation (RAG) system that: 1. **Retrieves STYLE** from ~12 Chubbyemu YouTube transcripts 2. **Retrieves FACTS** from medical PDF case reports 3. **Generates** educational medical stories in three formats: - **FULL_STORY** (600-1400 words): Complete narrative with dramatic pacing - **SHORT_AWARENESS** (150-350 words): Concise educational piece - **MECHANISM_TAKEAWAYS**: Mechanism explanation + 5 key bullet points 4. **Evaluates** baseline vs RAG performance 5. **Compares** RAG modes against each other (Full Story vs Short Awareness vs Mechanism Takeaways)

SAFETY: All outputs are educational only. No diagnosis, treatment, or medication dosing.

1.1 1. Setup and Installation

[1]:
```
!pip install -q google-generativeai sentence-transformers faiss-cpu PyMuPDF␣
↪pdfplumber tiktoken numpy pandas matplotlib seaborn scipy scikit-learn
```

```
                                        43.6/43.6 kB
1.4 MB/s eta 0:00:00
                                        67.8/67.8 kB
1.9 MB/s eta 0:00:00
                                       23.7/23.7 MB
39.5 MB/s eta 0:00:00
                                       24.1/24.1 MB
13.1 MB/s eta 0:00:00
                                       60.0/60.0 kB
1.1 MB/s eta 0:00:00
                                       5.6/5.6 MB
21.4 MB/s eta 0:00:00
                                       3.0/3.0 MB
19.3 MB/s eta 0:00:00
```

[2]:
```
import os
import json
import re
```

```
import pickle
import time
import numpy as np
import pandas as pd
from pathlib import Path
from typing import List, Dict, Tuple, Optional
from collections import defaultdict, Counter
from dataclasses import dataclass
from itertools import combinations

# PDF processing
import fitz  # PyMuPDF
import pdfplumber

# Embeddings and vector search
from sentence_transformers import SentenceTransformer
import faiss
from sklearn.metrics.pairwise import cosine_similarity
from scipy import stats

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Gemini API
import google.generativeai as genai

# Tokenization
import tiktoken

print("All imports successful")
```

```
WARNING:torchao.kernel.intmm:Warning: Detected no triton, on systems without
Triton certain kernels will not work

All imports successful
```

1.2 2. Configuration and API Setup

[3]:
```
PATHS = {
    "style_json": Path("data/style_json/"),
    "facts_pdfs": Path("data/facts_pdfs/"),
    "processed": Path("data/processed/"),
    "embedding_cache": Path("data/processed/embedding_cache/"),
    "indexes": Path("indexes/"),
    "eval": Path("eval/"),
    "eval_outputs": Path("eval/outputs/"),
    "eval_charts": Path("eval/charts/"),
```

```
}

for path in PATHS.values():
    path.mkdir(parents=True, exist_ok=True)

print("Directory structure created")
```

```
Directory structure created
```

[4]:
```
GOOGLE_API_KEY = "AIzaSyDOZ8t4JvAySgmD5KYqQqPJ-IXmPgYwAEI"    # Replace with␣
 ↪your actual API key

genai.configure(api_key=GOOGLE_API_KEY)

generation_config = {
    "temperature": 0.7,
    "top_p": 0.95,
    "top_k": 40,
    "max_output_tokens": 2048,
}

GEMINI_MODEL = genai.GenerativeModel(
    model_name="models/gemini-2.0-flash",
    generation_config=generation_config
)

eval_config = {
    "temperature": 0.1,
    "top_p": 0.95,
    "max_output_tokens": 1024,
}

EVAL_MODEL = genai.GenerativeModel(
    model_name="gemini-1.5-flash",
    generation_config=eval_config
)

print("Gemini API configured")
```

```
Gemini API configured
```

[5]:
```
EMBEDDING_MODEL_NAME = "all-MiniLM-L6-v2"
embedding_model = SentenceTransformer(EMBEDDING_MODEL_NAME)
EMBEDDING_DIM = embedding_model.get_sentence_embedding_dimension()

print(f"Loaded embedding model: {EMBEDDING_MODEL_NAME}")
print(f"Embedding dimension: {EMBEDDING_DIM}")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
modules.json:   0%|          | 0.00/349 [00:00<?, ?B/s]
config_sentence_transformers.json:   0%|          | 0.00/116 [00:00<?, ?B/s]
README.md: 0.00B [00:00, ?B/s]
sentence_bert_config.json:   0%|          | 0.00/53.0 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/612 [00:00<?, ?B/s]
model.safetensors:   0%|          | 0.00/90.9M [00:00<?, ?B/s]
tokenizer_config.json:   0%|          | 0.00/350 [00:00<?, ?B/s]
vocab.txt: 0.00B [00:00, ?B/s]
tokenizer.json: 0.00B [00:00, ?B/s]
special_tokens_map.json:   0%|          | 0.00/112 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/190 [00:00<?, ?B/s]
Loaded embedding model: all-MiniLM-L6-v2
Embedding dimension: 384
```

[6]:
```python
tokenizer = tiktoken.get_encoding("cl100k_base")

def count_tokens(text: str) -> int:
    return len(tokenizer.encode(text))

print("Tokenizer initialized")
```

```
Tokenizer initialized
```

1.3 3. Data Loading

[7]:
```python
def load_transcripts(json_dir: Path) -> List[Dict]:
    """Load all transcript JSON files from directory."""
    transcripts = []
    json_files = sorted(list(json_dir.glob("*.json")))

    if not json_files:
```

```
        print(f"WARNING: No JSON files found in {json_dir}")
        return transcripts

    print(f"Found {len(json_files)} JSON files:")

    for json_file in json_files:
        try:
            with open(json_file, "r", encoding="utf-8") as f:
                data = json.load(f)

            required_fields = ["video_id", "url", "transcript_segments"]
            missing = [fld for fld in required_fields if fld not in data]
            if missing:
                print(f"  X {json_file.name}: Missing fields {missing}")
                continue

            for seg in data["transcript_segments"]:
                if "text" in seg:
                    seg["text"] = re.sub(r"\s+", " ", seg["text"].replace("\n",␣
↪" ")).strip()

            if "full_transcript" not in data:
                data["full_transcript"] = " ".join([s.get("text", "") for s in␣
↪data["transcript_segments"]])

            transcripts.append(data)
            print(f"  OK {json_file.name}: {len(data['transcript_segments'])}␣
↪segments")

        except Exception as e:
            print(f"  X {json_file.name}: Error - {e}")

    print(f"\nSuccessfully loaded {len(transcripts)} transcripts")
    return transcripts


transcripts = load_transcripts(PATHS["style_json"])
```

```
Found 10 JSON files:
  OK transcript_video_12d4AiNS1JM.json: 151 segments
  OK transcript_video_LUGdQnxuixI.json: 217 segments
  OK transcript_video_Q8REcF4MRjQ.json: 151 segments
  OK transcript_video_UR3bM5qHBtY.json: 293 segments
  OK transcript_video_VHkrLMd5wmA.json: 164 segments
  OK transcript_video_Y3nsnruuowU.json: 199 segments
  OK transcript_video_cdjR99LAl8k.json: 232 segments
  OK transcript_video_ijgdfEKuXEU.json: 240 segments
```

```
  OK transcript_video_lDYvukPE73o.json: 147 segments
  OK transcript_video_yftBiNu0ZNU.json: 221 segments

Successfully loaded 10 transcripts
```

[8]:
```python
def extract_pdf_text(pdf_path: Path) -> str:
    """Extract text from PDF."""
    text = ""

    try:
        with fitz.open(pdf_path) as doc:
            for page in doc:
                text += page.get_text()
        if text.strip():
            return re.sub(r"\s+", " ", text).strip()
    except Exception:
        print("  PyMuPDF failed, trying pdfplumber...")

    try:
        with pdfplumber.open(pdf_path) as pdf:
            for page in pdf.pages:
                page_text = page.extract_text()
                if page_text:
                    text += page_text + " "
        if text.strip():
            return re.sub(r"\s+", " ", text).strip()
    except Exception as e:
        print(f"  pdfplumber also failed: {e}")

    return ""

print("\n" + "=" * 80)
print("LOADING MEDICAL PDF FACTS")
print("=" * 80)

pdf_files = sorted(list(PATHS["facts_pdfs"].glob("*.pdf")))
print(f"Found {len(pdf_files)} PDF files\n")

pdf_texts = {}
for pdf_file in pdf_files:
    print(f"Processing: {pdf_file.name}")
    text = extract_pdf_text(pdf_file)
    if text:
        pdf_texts[pdf_file.stem] = text
        print(f"  OK Extracted {len(text):,} characters")
    else:
        print("  X No text extracted")
```

```python
print(f"\nSuccessfully extracted text from {len(pdf_texts)} PDFs")
```

```
================================================================================
LOADING MEDICAL PDF FACTS
================================================================================
Found 21 PDF files

Processing: 001. Benefits Of Waking Early.pdf
  OK Extracted 2,749 characters
Processing: 2012-02-vitalsigns.pdf
  OK Extracted 8,423 characters
Processing: Benefits of vitamins and minerals_En202226733.pdf
  OK Extracted 13,455 characters
Processing: Caffeine.pdf
  OK Extracted 3,212 characters
Processing: DGA_FactSheet_AddedSugars_2021-06_508c.pdf
  OK Extracted 3,890 characters
Processing: Dietary_Guidelines_for_Americans_2020-2025.pdf
  OK Extracted 403,613 characters
Processing: Health-hydration-leaflet-spotting-signs-of-dehydration.pdf
  OK Extracted 3,522 characters
Processing: High-Fiber-Diet-Handout.pdf
  OK Extracted 7,567 characters
Processing: Probiotics-Consumer.pdf
  OK Extracted 8,920 characters
Processing: Scoop_August_2024.pdf
  OK Extracted 3,444 characters
Processing: Sleep-hygiene-3276-PIL.pdf
  OK Extracted 6,117 characters
Processing: Step_By_Step_Booklet_01_20.pdf
  OK Extracted 6,140 characters
Processing: VitaminK-Consumer.pdf
  OK Extracted 7,587 characters
Processing: WeightLoss-Consumer.pdf
  OK Extracted 20,494 characters
Processing: cdc_45851_DS1.pdf
  OK Extracted 3,347 characters
Processing: healthy_sleep.pdf
  OK Extracted 101,561 characters
Processing: hydration-poster-v-2.pdf
  OK Extracted 1,848 characters
Processing: obstructive-sleep-apnea-in-adults.pdf
  OK Extracted 7,401 characters
Processing: strength-training-tips.pdf
  OK Extracted 7,283 characters
```

```
Processing: vitamind-consumer.pdf
  OK Extracted 13,827 characters
Processing: water.pdf
  OK Extracted 4,128 characters

Successfully extracted text from 21 PDFs
```

1.4 4. Chunking and Embedding

[9]:
```python
def build_segment_list(transcripts: List[Dict]) -> List[Dict]:
    segments = []
    for transcript in transcripts:
        video_id = transcript["video_id"]
        for i, seg in enumerate(transcript["transcript_segments"]):
            segments.append({
                "seg_id": f"{video_id}_seg{i:04d}",
                "video_id": video_id,
                "start": seg.get("start", 0),
                "duration": seg.get("duration", 0),
                "text": seg.get("text", "")
            })
    return segments


def merge_segments_by_time(segments: List[Dict], window_seconds: Tuple[float,␣
 ↪float] = (60, 90)) -> List[Dict]:
    merged_blocks = []
    by_video = defaultdict(list)
    for seg in segments:
        by_video[seg["video_id"]].append(seg)

    for video_id, video_segs in by_video.items():
        video_segs.sort(key=lambda x: x["start"])
        current_block = []
        block_start = None
        block_id = 0

        for seg in video_segs:
            if not current_block:
                current_block = [seg]
                block_start = seg["start"]
            else:
                block_end = seg["start"] + seg["duration"]
                block_duration = block_end - block_start

                if block_duration <= window_seconds[1]:
                    current_block.append(seg)
```

```
                else:
                    if len(current_block) > 0:
                        merged_text = " ".join([s["text"] for s in␣
↪current_block])
                        merged_blocks.append({
                            "block_id": f"{video_id}_block{block_id:03d}",
                            "video_id": video_id,
                            "start": block_start,
                            "end": current_block[-1]["start"] +␣
↪current_block[-1]["duration"],
                            "text": merged_text
                        })
                        block_id += 1
                    current_block = [seg]
                    block_start = seg["start"]

        if current_block:
            merged_text = " ".join([s["text"] for s in current_block])
            merged_blocks.append({
                "block_id": f"{video_id}_block{block_id:03d}",
                "video_id": video_id,
                "start": block_start,
                "end": current_block[-1]["start"] +␣
↪current_block[-1]["duration"],
                "text": merged_text
            })

    return merged_blocks


def chunk_by_tokens(blocks: List[Dict], target_tokens: Tuple[int, int] = (450,␣
↪800), overlap_tokens: int = 100) -> List[Dict]:
    chunks = []
    chunk_global_id = 0

    for block in blocks:
        text = block["text"]
        words = text.split()

        target_words = (int(target_tokens[0] * 0.75), int(target_tokens[1] * 0.
↪75))
        overlap_words = int(overlap_tokens * 0.75)

        start_idx = 0
        while start_idx < len(words):
            end_idx = min(start_idx + target_words[1], len(words))
            chunk_text = " ".join(words[start_idx:end_idx])
```

```
            token_count = count_tokens(chunk_text)

            if token_count >= target_tokens[0] or end_idx == len(words):
                chunks.append({
                    "chunk_id": f"style_{chunk_global_id:05d}",
                    "source_type": "style",
                    "video_id": block["video_id"],
                    "text": chunk_text,
                    "token_count": token_count
                })
                chunk_global_id += 1

            if end_idx == len(words):
                break

            start_idx = end_idx - overlap_words

    return chunks


segments = build_segment_list(transcripts)
merged_blocks = merge_segments_by_time(segments)
style_chunks = chunk_by_tokens(merged_blocks)
print(f"Created {len(style_chunks)} style chunks")
```

```
Created 118 style chunks
```

[10]:
```
def chunk_pdf_text(doc_id: str, text: str, target_tokens: Tuple[int, int] =␣
↪(550, 900)) -> List[Dict]:
    chunks = []
    paragraphs = re.split(r"\n\n+|\. {2,}|\.[\n\r]+", text)
    paragraphs = [p.strip() for p in paragraphs if p.strip() and len(p.strip())␣
↪> 20]

    current_chunk = []
    current_tokens = 0
    chunk_id = 0

    for para in paragraphs:
        para_tokens = count_tokens(para)

        if current_tokens + para_tokens > target_tokens[1] and current_chunk:
            chunk_text = " ".join(current_chunk)
            chunks.append({
                "chunk_id": f"facts_{doc_id}_{chunk_id:03d}",
                "source_type": "facts",
                "doc_id": doc_id,
```

```
                "text": chunk_text,
                "token_count": current_tokens
            })
            chunk_id += 1
            current_chunk = [current_chunk[-1]] if len(current_chunk) > 1 else␣
↪[]
            current_tokens = count_tokens(current_chunk[0]) if current_chunk␣
↪else 0

        current_chunk.append(para)
        current_tokens += para_tokens

    if current_chunk and current_tokens >= target_tokens[0]:
        chunk_text = " ".join(current_chunk)
        chunks.append({
            "chunk_id": f"facts_{doc_id}_{chunk_id:03d}",
            "source_type": "facts",
            "doc_id": doc_id,
            "text": chunk_text,
            "token_count": current_tokens
        })

    return chunks


facts_chunks = []
for doc_id, text in pdf_texts.items():
    doc_chunks = chunk_pdf_text(doc_id, text)
    facts_chunks.extend(doc_chunks)
    print(f"  {doc_id}: {len(doc_chunks)} chunks")

print(f"\nCreated {len(facts_chunks)} facts chunks total")
```

```
  001. Benefits Of Waking Early: 1 chunks
  2012-02-vitalsigns: 1 chunks
  Benefits of vitamins and minerals_En202226733: 1 chunks
  Caffeine: 1 chunks
  DGA_FactSheet_AddedSugars_2021-06_508c: 1 chunks
  Dietary_Guidelines_for_Americans_2020-2025: 1 chunks
  Health-hydration-leaflet-spotting-signs-of-dehydration: 1 chunks
  High-Fiber-Diet-Handout: 1 chunks
  Probiotics-Consumer: 1 chunks
  Scoop_August_2024: 1 chunks
  Sleep-hygiene-3276-PIL: 1 chunks
  Step_By_Step_Booklet_01_20: 1 chunks
  VitaminK-Consumer: 1 chunks
  WeightLoss-Consumer: 1 chunks
```

```
  cdc_45851_DS1: 1 chunks
  healthy_sleep: 1 chunks
  hydration-poster-v-2: 0 chunks
  obstructive-sleep-apnea-in-adults: 1 chunks
  strength-training-tips: 1 chunks
  vitamind-consumer: 1 chunks
  water: 1 chunks

Created 20 facts chunks total
```

[11]:
```python
with open(PATHS["processed"] / "style_chunks.jsonl", "w", encoding="utf-8") as␣
↪f:
    for chunk in style_chunks:
        f.write(json.dumps(chunk, ensure_ascii=False) + "\n")

with open(PATHS["processed"] / "facts_chunks.jsonl", "w", encoding="utf-8") as␣
↪f:
    for chunk in facts_chunks:
        f.write(json.dumps(chunk, ensure_ascii=False) + "\n")

print("Chunks saved")
```

```
Chunks saved
```

[12]:
```python
class EmbeddingCache:
    def __init__(self, cache_dir: Path):
        self.cache_dir = cache_dir
        self.cache_dir.mkdir(parents=True, exist_ok=True)
        self.cache_file = self.cache_dir / "embeddings.pkl"
        self.cache = self._load_cache()

    def _load_cache(self) -> Dict:
        if self.cache_file.exists():
            try:
                with open(self.cache_file, "rb") as f:
                    return pickle.load(f)
            except Exception:
                return {}
        return {}

    def get(self, chunk_id: str) -> Optional[np.ndarray]:
        return self.cache.get(chunk_id)

    def set(self, chunk_id: str, embedding: np.ndarray):
        self.cache[chunk_id] = embedding

    def save(self):
```

```
        with open(self.cache_file, "wb") as f:
            pickle.dump(self.cache, f)


embedding_cache = EmbeddingCache(PATHS["embedding_cache"])


def embed_chunks(chunks: List[Dict], batch_size: int = 32) -> np.ndarray:
    embeddings = []
    to_embed = []
    to_embed_ids = []

    for chunk in chunks:
        cached = embedding_cache.get(chunk["chunk_id"])
        if cached is not None:
            embeddings.append((chunk["chunk_id"], cached))
        else:
            to_embed.append(chunk["text"])
            to_embed_ids.append(chunk["chunk_id"])

    if to_embed:
        for i in range(0, len(to_embed), batch_size):
            batch_texts = to_embed[i:i+batch_size]
            batch_ids = to_embed_ids[i:i+batch_size]
            batch_embeddings = embedding_model.encode(batch_texts,␣
 ↪convert_to_numpy=True)
            for chunk_id, emb in zip(batch_ids, batch_embeddings):
                embedding_cache.set(chunk_id, emb)
                embeddings.append((chunk_id, emb))
        embedding_cache.save()

    id_to_idx = {chunk["chunk_id"]: i for i, chunk in enumerate(chunks)}
    ordered_embeddings = [None] * len(chunks)
    for chunk_id, emb in embeddings:
        if chunk_id in id_to_idx:
            ordered_embeddings[id_to_idx[chunk_id]] = emb

    return np.array(ordered_embeddings)


print("Embedding style chunks...")
style_embeddings = embed_chunks(style_chunks)
print(f"Style embeddings shape: {style_embeddings.shape}")

print("\nEmbedding facts chunks...")
facts_embeddings = embed_chunks(facts_chunks) if facts_chunks else np.array([])
```

```
print(f"Facts embeddings shape: {facts_embeddings.shape if␣
 ↪len(facts_embeddings) > 0 else '(empty)'}")
```

```
Embedding style chunks…
Style embeddings shape: (118, 384)

Embedding facts chunks…
Facts embeddings shape: (20, 384)
```

1.5 5. FAISS Index Building

[13]:
```
def build_faiss_index(embeddings: np.ndarray) -> Optional[faiss.IndexFlatIP]:
    if len(embeddings) == 0:
        return None
    embeddings = embeddings.astype("float32")
    faiss.normalize_L2(embeddings)
    index = faiss.IndexFlatIP(embeddings.shape[1])
    index.add(embeddings)
    return index


style_index = build_faiss_index(style_embeddings)
print(f"Style index: {style_index.ntotal if style_index else 0} vectors")

facts_index = build_faiss_index(facts_embeddings) if len(facts_embeddings) > 0␣
 ↪else None
print(f"Facts index: {facts_index.ntotal if facts_index else 0} vectors")

if style_index:
    faiss.write_index(style_index, str(PATHS["indexes"] / "style.faiss"))
    with open(PATHS["indexes"] / "style_meta.json", "w", encoding="utf-8") as f:
        json.dump(style_chunks, f, ensure_ascii=False)

if facts_index:
    faiss.write_index(facts_index, str(PATHS["indexes"] / "facts.faiss"))
    with open(PATHS["indexes"] / "facts_meta.json", "w", encoding="utf-8") as f:
        json.dump(facts_chunks, f, ensure_ascii=False)

print("Indexes saved")
```

```
Style index: 118 vectors
Facts index: 20 vectors
Indexes saved
```

[14]:
```
def retrieve_top_k(query: str, index: faiss.Index, metadata: List[Dict], k: int␣
 ↪= 5) -> List[Dict]:
    if index is None or len(metadata) == 0:
```

```
        return []

    query_embedding = embedding_model.encode([query], convert_to_numpy=True).
↪astype("float32")
    faiss.normalize_L2(query_embedding)

    k = min(k, index.ntotal)
    distances, indices = index.search(query_embedding, k)

    results = []
    for idx, dist in zip(indices[0], distances[0]):
        if 0 <= idx < len(metadata):
            chunk = metadata[idx].copy()
            chunk["similarity"] = float(dist)
            results.append(chunk)

    return results


print("Retrieval functions defined")
```

```
Retrieval functions defined
```

1.6 6. Style and Evaluation Metrics

[15]:
```
CHUBBYEMU_STYLE_MARKERS = {
    'dramatic_hooks': [
        r'presenting to the emergency',
        r'this is what happened',
        r'is a \d+-year-old',
        r'presented to the emergency room',
        r'was found',
        r'was brought to',
        r'something was wrong'
    ],
    'medical_terms': [
        r'-emia\b',
        r'-itis\b',
        r'-osis\b',
        r'-pathy\b',
        r'hyper-?\w+',
        r'hypo-?\w+',
        r'serum',
        r'plasma',
        r'presenting to',
        r'diagnosis',
        r'symptoms'
```

```
    ],
    'explanation_patterns': [
        r'meaning\s+\w+',
        r'which means',
        r'referring to',
        r'this is when',
        r'this means',
        r'in the blood',
        r'in medical terms'
    ],
    'dramatic_elements': [
        r'but then',
        r'suddenly',
        r'immediately',
        r'at this point',
        r'it turns out',
        r'the problem',
        r"something wasn't right",
        r'began to',
        r'started to'
    ],
    'progression_markers': [
        r'over the next',
        r'days later',
        r'weeks later',
        r'after \d+',
        r'by the time',
        r'eventually',
        r'finally'
    ]
}

if style_chunks:
    reference_samples = [chunk['text'] for chunk in style_chunks[:20]]
    CHUBBYEMU_REFERENCE_EMBEDDINGS = embedding_model.encode(reference_samples,␣
 ↪convert_to_numpy=True)
    CHUBBYEMU_MEAN_EMBEDDING = np.mean(CHUBBYEMU_REFERENCE_EMBEDDINGS, axis=0)
    print(f"Created Chubbyemu reference embedding from {len(reference_samples)}␣
 ↪samples")
else:
    CHUBBYEMU_MEAN_EMBEDDING = None
    print("WARNING: No transcripts loaded - style metrics will be limited")
```

```
Created Chubbyemu reference embedding from 20 samples
```

[16]:
```
def calculate_style_marker_score(text: str) -> Dict[str, float]:
    text_lower = text.lower()
```

```
    scores = {}

    for category, patterns in CHUBBYEMU_STYLE_MARKERS.items():
        matches = 0
        for pattern in patterns:
            matches += len(re.findall(pattern, text_lower))
        scores[category] = (matches / max(len(text), 1)) * 1000

    weights = {
        'dramatic_hooks': 2.0,
        'medical_terms': 1.5,
        'explanation_patterns': 1.5,
        'dramatic_elements': 1.0,
        'progression_markers': 1.0
    }

    weighted_sum = sum(scores[k] * weights[k] for k in scores)
    total_weight = sum(weights.values())
    scores['overall_marker_score'] = weighted_sum / total_weight

    return scores

def calculate_embedding_similarity(text: str) -> float:
    if CHUBBYEMU_MEAN_EMBEDDING is None:
        return 0.0

    text_embedding = embedding_model.encode([text], convert_to_numpy=True)[0]
    similarity = cosine_similarity(
        text_embedding.reshape(1, -1),
        CHUBBYEMU_MEAN_EMBEDDING.reshape(1, -1)
    )[0][0]

    return float(similarity)

def calculate_sentence_structure_metrics(text: str) -> Dict[str, float]:
    sentences = re.split(r'[.!?]+', text)
    sentences = [s.strip() for s in sentences if s.strip()]

    if not sentences:
        return {'avg_sentence_length': 0, 'sentence_length_variance': 0,␣
 ↪'short_sentence_ratio': 0}

    lengths = [len(s.split()) for s in sentences]
    avg_length = np.mean(lengths)
    variance = np.var(lengths)
    short_sentences = sum(1 for l in lengths if l <= 8)
    short_ratio = short_sentences / len(sentences)
```

```
    return {
        'avg_sentence_length': avg_length,
        'sentence_length_variance': variance,
        'short_sentence_ratio': short_ratio
    }

def calculate_full_style_score(text: str) -> Dict[str, any]:
    marker_scores = calculate_style_marker_score(text)
    embedding_sim = calculate_embedding_similarity(text)
    structure_metrics = calculate_sentence_structure_metrics(text)

    composite_score = (
        embedding_sim * 50 +
        min(marker_scores['overall_marker_score'] * 10, 30) +
        min(structure_metrics['short_sentence_ratio'] * 20, 20)
    )

    return {
        'composite_style_score': composite_score,
        'embedding_similarity': embedding_sim,
        'marker_scores': marker_scores,
        'structure_metrics': structure_metrics
    }

print("Style fidelity metrics defined")
```

```
Style fidelity metrics defined
```

[17]:
```
def count_medical_specifics(text: str) -> Dict[str, int]:
    patterns = {
        'dosages_mg': r'\d+\s*mg\b',
        'dosages_ml': r'\d+\s*mL\b',
        'dosages_mcg': r'\d+\s*mcg\b',
        'percentages': r'\d+(\.\d+)?\s*%',
        'lab_values': r'\d+(\.\d+)?\s*(mmol|mg/dL|mEq|IU|ng)',
        'specific_numbers': r'\b\d{3,}\b',
        'medication_regimens': r'\d+\s*times?\s+(daily|per day|a day)',
        'treatment_durations': r'for\s+\d+\s+(day|week|month|hour)s?'
    }

    counts = {}
    total = 0
    for name, pattern in patterns.items():
        matches = re.findall(pattern, text, re.IGNORECASE)
        count = len(matches)
        counts[name] = count
```

```
        total += count

    counts['total_specifics'] = total
    return counts

def count_citations(text: str) -> Dict[str, any]:
    facts_citations = re.findall(r'\[(facts_[^\]]+)\]', text)
    style_citations = re.findall(r'\[(style_[^\]]+)\]', text)
    has_citation_section = bool(re.search(r'\bcitations?\b\s*[:\-]', text, re.
 ↪IGNORECASE))
    return {
        'style_citation_count': len(style_citations),
        'facts_citation_count': len(facts_citations),
        'total_citations': len(style_citations) + len(facts_citations),
        'has_citation_section': has_citation_section,
        'unique_style_citations': len(set(style_citations)),
        'unique_facts_citations': len(set(facts_citations))
    }

def calculate_claim_density(text: str) -> float:
    claim_patterns = [
        r'\bcauses?\b',
        r'\bresults? in\b',
        r'\bleads? to\b',
        r'\baccording to\b',
        r'\bstudies? (show|indicate|suggest)\b',
        r'\bhas been (shown|proven|demonstrated)\b',
        r'\d+%',
        r'\bdeficiency\b',
        r'\btoxicity\b'
    ]

    total_claims = 0
    for pattern in claim_patterns:
        total_claims += len(re.findall(pattern, text, re.IGNORECASE))

    word_count = len(text.split())
    return (total_claims / max(word_count, 1)) * 100

def calculate_factual_grounding_score(text: str, is_rag: bool = False) ->␣
 ↪Dict[str, any]:
    specifics = count_medical_specifics(text)
    citations = count_citations(text)
    claim_density = calculate_claim_density(text)

    if is_rag:
```

```
        uncited_specifics = max(0, specifics['total_specifics'] -␣
↪citations['total_citations'])
        hallucination_risk = min(100, uncited_specifics * 10 + claim_density *␣
↪2)
    else:
        hallucination_risk = min(100, specifics['total_specifics'] * 15 +␣
↪claim_density * 3)

    grounding_score = 100 - hallucination_risk

    return {
        'grounding_score': grounding_score,
        'hallucination_risk': hallucination_risk,
        'medical_specifics': specifics,
        'citations': citations,
        'claim_density': claim_density
    }

print("Factual accuracy metrics defined")
```

```
Factual accuracy metrics defined
```

[18]:
```
STYLE_EVALUATION_PROMPT = """
You are evaluating how well a generated text matches Chubbyemu's distinctive␣
↪medical storytelling style.

## Chubbyemu's Style Characteristics:
1. **Dramatic Opening**: Starts with "A [age]-year-old presenting to the␣
↪emergency room..."
2. **Medical Terminology with Explanations**: Uses terms followed by␣
↪explanations
3. **Progressive Tension**: Builds suspense with dramatic phrases
4. **Short Punchy Sentences**: Mixed with longer explanatory sentences
5. **Educational Asides**: Explains mechanisms clearly
6. **Case-Based Narrative**: Tells a story about a specific patient

## Text to Evaluate:
{text}

## Evaluation (score each 1-5):
1. Dramatic Hook (1-5)
2. Medical Terminology Usage (1-5)
3. Narrative Tension (1-5)
4. Sentence Variety (1-5)
5. Educational Clarity (1-5)
6. Overall Chubbyemu Voice (1-5)
```

```
Respond in JSON format:
{{"dramatic_hook": <score>, "medical_terminology": <score>, "narrative_tension":
 ↪ <score>, "sentence_variety": <score>, "educational_clarity": <score>,␣
 ↪"overall_voice": <score>, "total_style_score": <sum>, "brief_justification":␣
 ↪"<explanation>"}}
"""

FACTUAL_EVALUATION_PROMPT = """
You are evaluating the factual accuracy of a medical education text.

## Text to Evaluate:
{text}

## Medical Topic:
{topic}

## Evaluation (score each 1-5):
1. Medical Accuracy (1-5)
2. Appropriate Hedging (1-5)
3. No Fabricated Details (1-5)
4. Logical Consistency (1-5)
5. Safety Awareness (1-5)

Respond in JSON format:
{{"medical_accuracy": <score>, "appropriate_hedging": <score>,␣
 ↪"no_fabrications": <score>, "logical_consistency": <score>,␣
 ↪"safety_awareness": <score>, "total_factual_score": <sum>,␣
 ↪"potential_hallucinations": [], "brief_justification": "<explanation>"}}
"""

def llm_evaluate_style(text: str) -> Dict:
    try:
        prompt = STYLE_EVALUATION_PROMPT.format(text=text[:3000])
        response = EVAL_MODEL.generate_content(prompt)
        json_match = re.search(r'\{[^{}]*\}', response.text, re.DOTALL)
        if json_match:
            return json.loads(json_match.group())
        return {"error": "Could not parse response"}
    except Exception as e:
        return {"error": str(e)}

def llm_evaluate_factual(text: str, topic: str) -> Dict:
    try:
        prompt = FACTUAL_EVALUATION_PROMPT.format(text=text[:3000], topic=topic)
        response = EVAL_MODEL.generate_content(prompt)
        json_match = re.search(r'\{[^{}]*\}', response.text, re.DOTALL)
        if json_match:
```

```
            return json.loads(json_match.group())
        return {"error": "Could not parse response"}
    except Exception as e:
        return {"error": str(e)}

print("LLM evaluation functions defined")
```

```
LLM evaluation functions defined
```

1.7 7. Generation Functions

[19]:
```
DISCLAIMER = """EDUCATIONAL CONTENT ONLY
This is educational content for learning purposes. NOT medical advice.
Always consult a qualified healthcare professional for medical concerns.
"""

FORMAT_INSTRUCTIONS = {
    "FULL_STORY": """Write a complete narrative medical story (600-1400 words)␣
 ↪in Chubbyemu's style:
- Open with dramatic hook ("A [age]-year-old...presenting to the emergency␣
 ↪room")
- Build tension as symptoms progressively worsen
- Explain medical terms ("hypo- meaning low, -emia meaning in blood")
- Use short punchy sentences mixed with explanations
- Reveal diagnosis and explain mechanism
- End with outcome and educational takeaway""",

    "SHORT_AWARENESS": """Write a concise medical awareness piece (150-350␣
 ↪words):
- Quickly establish the key medical issue
- Explain why this matters
- Provide 2-3 key educational points
- End with actionable takeaway""",

    "MECHANISM_TAKEAWAYS": """Provide two parts:
1. MECHANISM (200-300 words): Detailed physiological explanation
2. KEY TAKEAWAYS: Exactly 5 bullet points (1-2 sentences each)"""
}

PROMPT_BASELINE = """You are a medical educator creating content inspired by␣
 ↪Chubbyemu's storytelling style.

Create a {mode} about:
{case_prompt}

Guidelines:
```

```
- Use Chubbyemu's signature style: dramatic pacing, medical terminology with␣
 ↪explanations
- EDUCATIONAL CONTENT ONLY - no specific medication dosages or treatment␣
 ↪protocols
- make up specific statistics or lab values to a little extent

Format:
{format_instructions}
"""

PROMPT_RAG = """You are a medical educator creating content inspired by␣
 ↪Chubbyemu's storytelling style.

You have reference material:

================================================================================
STYLE EXAMPLES (use for narrative structure and tone):
================================================================================
{style_context}

================================================================================
MEDICAL FACTS (use for factual information - CITE sources):
================================================================================
{facts_context}

================================================================================

Create a {mode} about:
{case_prompt}

CRITICAL RULES:
1. Use STYLE EXAMPLES for narrative voice only - NOT for medical facts
2. Use MEDICAL FACTS for all factual claims - CITE [chunk_id] after each fact
3. If information isn't in sources, say so or generalize appropriately
4. EDUCATIONAL CONTENT ONLY - no specific dosages or treatment protocols
5. End with: "Citations: [list all chunk_ids used]"

Format:
{format_instructions}
"""

print("Prompt templates defined")
```

```
Prompt templates defined
```

[20]:
```
def generate_baseline(case_prompt: str, mode: str = "FULL_STORY") -> str:
    prompt = PROMPT_BASELINE.format(
```

```
        mode=mode,
        case_prompt=case_prompt,
        format_instructions=FORMAT_INSTRUCTIONS[mode]
    )

    try:
        response = GEMINI_MODEL.generate_content(prompt)
        return DISCLAIMER + "\n\n" + response.text
    except Exception as e:
        return f"Error: {e}"

def generate_rag(
    case_prompt: str,
    mode: str = "FULL_STORY",
    top_k_style: int = 15,
    top_k_facts: int = 15
) -> Tuple[str, Dict]:
    style_results = retrieve_top_k(case_prompt, style_index, style_chunks,␣
 ↪k=top_k_style)
    facts_results = retrieve_top_k(case_prompt, facts_index, facts_chunks,␣
 ↪k=top_k_facts) if facts_index else []

    style_context = "\n\n".join([
        f"[{r['chunk_id']}]\n{r['text'][:500]}..." for r in style_results
    ]) if style_results else "No style examples available."

    facts_context = "\n\n".join([
        f"[{r['chunk_id']}]\nSource: {r['doc_id']}\n{r['text'][:500]}..." for r␣
 ↪in facts_results
    ]) if facts_results else "No medical facts available."

    prompt = PROMPT_RAG.format(
        style_context=style_context,
        facts_context=facts_context,
        mode=mode,
        case_prompt=case_prompt,
        format_instructions=FORMAT_INSTRUCTIONS[mode]
    )

    try:
        response = GEMINI_MODEL.generate_content(prompt)
        metadata = {
            "style_chunks": [r["chunk_id"] for r in style_results],
            "facts_chunks": [r["chunk_id"] for r in facts_results],
            "num_style_chunks": len(style_results),
            "num_facts_chunks": len(facts_results)
        }
```

```
        return DISCLAIMER + "\n\n" + response.text, metadata
    except Exception as e:
        return f"Error: {e}", {}

print("Generation functions defined")
```

```
Generation functions defined
```

1.8 8. Evaluation Questions

[24]:
```
EVAL_QUESTIONS = [
    {"id": "Q01", "prompt": "A person feels tired, gets headaches, and notices␣
 ↪darker urine after not drinking much water for days. What could be happening␣
 ↪inside the body?", "topic": "dehydration and fluid balance", "category":␣
 ↪"hydration"},
    {"id": "Q02", "prompt": "What are the common signs of dehydration, and why␣
 ↪do they occur?", "topic": "physiology of dehydration", "category":␣
 ↪"hydration"},
    {"id": "Q03", "prompt": "What does vitamin D do in the body, and what␣
 ↪happens if someone has too little or too much?", "topic": "vitamin D␣
 ↪function, deficiency, and toxicity", "category": "nutrition"}

]

ALL_MODES = ["FULL_STORY", "SHORT_AWARENESS", "MECHANISM_TAKEAWAYS"]

with open(PATHS["eval"] / "questions.json", "w", encoding="utf-8") as f:
    json.dump(EVAL_QUESTIONS, f, indent=2)

print(f"Created {len(EVAL_QUESTIONS)} evaluation questions")
print(f"Each question will be tested in {len(ALL_MODES)} modes: {ALL_MODES}")
print(f"Total evaluations: {len(EVAL_QUESTIONS) * len(ALL_MODES) * 2} (Baseline␣
 ↪+ RAG for each)")
```

```
Created 3 evaluation questions
Each question will be tested in 3 modes: ['FULL_STORY', 'SHORT_AWARENESS',
'MECHANISM_TAKEAWAYS']
Total evaluations: 18 (Baseline + RAG for each)
```

1.9 9. Run Full Evaluation (Baseline vs RAG + Cross-Mode)

[25]:
```
def run_full_evaluation(questions: List[Dict], modes: List[str], run_llm_eval:␣
 ↪bool = True) -> pd.DataFrame:
    results = []
    total_evals = len(questions) * len(modes)
    print("=" * 100)
    print("RUNNING COMPREHENSIVE EVALUATION (Baseline vs RAG + Cross-Mode)")
```

```
    print("=" * 100)
    print(f"\nEvaluating {len(questions)} questions x {len(modes)} modes =␣
↪{total_evals} total evaluations...\n")

    eval_count = 0
    for q in questions:
        for mode in modes:
            eval_count += 1
            print(f"\n[{eval_count}/{total_evals}] {q['id']} - {mode}")
            print(f"  Prompt: {q['prompt'][:50]}...")

            result = {
                "question_id": q["id"],
                "prompt": q["prompt"],
                "mode": mode,
                "topic": q["topic"],
                "category": q["category"],
            }

            print("  [BASELINE] Generating...")
            baseline_output = generate_baseline(q["prompt"], mode)
            result["baseline_output"] = baseline_output
            result["baseline_word_count"] = len(baseline_output.split())
            time.sleep(1)

            print("  [RAG] Generating...")
            rag_output, rag_metadata = generate_rag(q["prompt"], mode)
            result["rag_output"] = rag_output
            result["rag_word_count"] = len(rag_output.split())
            result["rag_style_chunks"] = rag_metadata.get("num_style_chunks", 0)
            result["rag_facts_chunks"] = rag_metadata.get("num_facts_chunks", 0)
            time.sleep(1)

            print("  [METRICS] Calculating style metrics...")
            baseline_style = calculate_full_style_score(baseline_output)
            rag_style = calculate_full_style_score(rag_output)

            result["baseline_style_composite"] =␣
↪baseline_style["composite_style_score"]
            result["baseline_style_embedding_sim"] =␣
↪baseline_style["embedding_similarity"]
            result["baseline_style_markers"] =␣
↪baseline_style["marker_scores"]["overall_marker_score"]

            result["rag_style_composite"] = rag_style["composite_style_score"]
            result["rag_style_embedding_sim"] =␣
↪rag_style["embedding_similarity"]
```

```
            result["rag_style_markers"] =␣
↪rag_style["marker_scores"]["overall_marker_score"]

            print("  [METRICS] Calculating factual metrics...")
            baseline_factual =␣
↪calculate_factual_grounding_score(baseline_output, is_rag=False)
            rag_factual = calculate_factual_grounding_score(rag_output,␣
↪is_rag=True)

            result["baseline_grounding_score"] =␣
↪baseline_factual["grounding_score"]
            result["baseline_hallucination_risk"] =␣
↪baseline_factual["hallucination_risk"]
            result["baseline_medical_specifics"] =␣
↪baseline_factual["medical_specifics"]["total_specifics"]
            result["baseline_claim_density"] = baseline_factual["claim_density"]

            result["rag_grounding_score"] = rag_factual["grounding_score"]
            result["rag_hallucination_risk"] = rag_factual["hallucination_risk"]
            result["rag_medical_specifics"] =␣
↪rag_factual["medical_specifics"]["total_specifics"]
            result["rag_claim_density"] = rag_factual["claim_density"]
            result["rag_total_citations"] =␣
↪rag_factual["citations"]["total_citations"]
            result["rag_has_citation_section"] =␣
↪rag_factual["citations"]["has_citation_section"]

            if run_llm_eval:
                print("  [LLM] Running LLM evaluation...")
                baseline_llm_style = llm_evaluate_style(baseline_output)
                time.sleep(1)
                rag_llm_style = llm_evaluate_style(rag_output)
                time.sleep(1)
                baseline_llm_factual = llm_evaluate_factual(baseline_output,␣
↪q["topic"])
                time.sleep(1)
                rag_llm_factual = llm_evaluate_factual(rag_output, q["topic"])
                time.sleep(1)

                result["baseline_llm_style_score"] = baseline_llm_style.
↪get("total_style_score", 0)
                result["baseline_llm_factual_score"] = baseline_llm_factual.
↪get("total_factual_score", 0)
                result["rag_llm_style_score"] = rag_llm_style.
↪get("total_style_score", 0)
```

```
                result["rag_llm_factual_score"] = rag_llm_factual.
↪get("total_factual_score", 0)

            results.append(result)
            print(f"    Baseline: Style={result['baseline_style_composite']:.
↪1f}, Grounding={result['baseline_grounding_score']:.1f}")
            print(f"    RAG:      Style={result['rag_style_composite']:.1f},␣
↪Grounding={result['rag_grounding_score']:.1f},␣
↪Citations={result['rag_total_citations']}")

    return pd.DataFrame(results)

print("Evaluation function defined")
```

```
Evaluation function defined
```

[26]:
```
# Run the evaluation (set run_llm_eval=False for faster testing)
results_df = run_full_evaluation(EVAL_QUESTIONS, ALL_MODES, run_llm_eval=True)
```

```
================================================================================
====================
RUNNING COMPREHENSIVE EVALUATION (Baseline vs RAG + Cross-Mode)
================================================================================
====================

Evaluating 3 questions x 3 modes = 9 total evaluations…


[1/9] Q01 - FULL_STORY
  Prompt: A person feels tired, gets headaches, and notices …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.91ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 130.36ms
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 152.92ms

    Baseline: Style=34.9, Grounding=0.0
```

```
    RAG:      Style=39.4, Grounding=99.0, Citations=9

[2/9] Q01 - SHORT_AWARENESS
  Prompt: A person feels tired, gets headaches, and notices …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…
  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.72ms
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 153.30ms
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 228.64ms

    Baseline: Style=21.6, Grounding=85.0
    RAG:      Style=21.1, Grounding=99.3, Citations=7

[3/9] Q01 - MECHANISM_TAKEAWAYS
  Prompt: A person feels tired, gets headaches, and notices …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…
  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 179.23ms
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.93ms

    Baseline: Style=21.6, Grounding=85.0
    RAG:      Style=24.2, Grounding=97.5, Citations=14

[4/9] Q02 - FULL_STORY
  Prompt: What are the common signs of dehydration, and why …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.65ms
```

```
  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 178.60ms

    Baseline: Style=34.7, Grounding=9.2
    RAG:      Style=39.2, Grounding=98.6, Citations=10


[5/9] Q02 - SHORT_AWARENESS
  Prompt: What are the common signs of dehydration, and why …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 178.32ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 178.80ms

    Baseline: Style=26.4, Grounding=38.5
    RAG:      Style=18.9, Grounding=98.3, Citations=9


[6/9] Q02 - MECHANISM_TAKEAWAYS
  Prompt: What are the common signs of dehydration, and why …
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.94ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 128.28ms
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.54ms

    Baseline: Style=27.1, Grounding=69.3
    RAG:      Style=18.4, Grounding=98.1, Citations=11
```

```
[7/9] Q03 - FULL_STORY
  Prompt: What does vitamin D do in the body, and what happe…
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.79ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 304.44ms

    Baseline: Style=34.7, Grounding=21.9
    RAG:      Style=27.8, Grounding=97.5, Citations=8


[8/9] Q03 - SHORT_AWARENESS
  Prompt: What does vitamin D do in the body, and what happe…
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 127.89ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 152.94ms

    Baseline: Style=24.9, Grounding=0.0
    RAG:      Style=18.6, Grounding=97.4, Citations=8


[9/9] Q03 - MECHANISM_TAKEAWAYS
  Prompt: What does vitamin D do in the body, and what happe…
  [BASELINE] Generating…
  [RAG] Generating…
  [METRICS] Calculating style metrics…

WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 128.98ms

  [METRICS] Calculating factual metrics…
  [LLM] Running LLM evaluation…
```

```
WARNING:tornado.access:404 POST
/v1beta/models/gemini-1.5-flash:generateContent?%24alt=json%3Benum-
encoding%3Dint (::1) 228.99ms
```

```
    Baseline: Style=25.0, Grounding=22.8
    RAG:      Style=24.7, Grounding=91.7, Citations=8
```

[27]:
```python
# Save detailed results
results_csv_path = PATHS["eval"] / "detailed_results.csv"
results_df.to_csv(results_csv_path, index=False)
print(f"Saved detailed results to {results_csv_path}")

outputs_path = PATHS["eval_outputs"] / "all_outputs.json"
outputs_data = results_df[["question_id", "prompt", "mode", "baseline_output",␣
 ↪"rag_output"]].to_dict("records")
with open(outputs_path, "w", encoding="utf-8") as f:
    json.dump(outputs_data, f, indent=2)
print(f"Saved all outputs for review to {outputs_path}")
```

```
Saved detailed results to eval/detailed_results.csv
Saved all outputs for review to eval/outputs/all_outputs.json
```

1.10 10. Results Summary: Baseline vs RAG

[28]:
```python
print("\n" + "=" * 100)
print("EVALUATION RESULTS SUMMARY: BASELINE vs RAG (All Modes Combined)")
print("=" * 100)

summary = {
    "Total Evaluations": len(results_df),
    "Baseline Avg Style Score": results_df["baseline_style_composite"].mean(),
    "RAG Avg Style Score": results_df["rag_style_composite"].mean(),
    "Style Score Improvement": results_df["rag_style_composite"].mean() -␣
 ↪results_df["baseline_style_composite"].mean(),
    "Baseline Avg Embedding Similarity":␣
 ↪results_df["baseline_style_embedding_sim"].mean(),
    "RAG Avg Embedding Similarity": results_df["rag_style_embedding_sim"].
 ↪mean(),
    "Baseline Avg Grounding Score": results_df["baseline_grounding_score"].
 ↪mean(),
    "RAG Avg Grounding Score": results_df["rag_grounding_score"].mean(),
    "Grounding Score Improvement": results_df["rag_grounding_score"].mean() -␣
 ↪results_df["baseline_grounding_score"].mean(),
    "Baseline Avg Hallucination Risk":␣
 ↪results_df["baseline_hallucination_risk"].mean(),
    "RAG Avg Hallucination Risk": results_df["rag_hallucination_risk"].mean(),
```

```
    "Hallucination Risk Reduction": results_df["baseline_hallucination_risk"].
 ↪mean() - results_df["rag_hallucination_risk"].mean(),
    "RAG Avg Citations": results_df["rag_total_citations"].mean(),
    "RAG Citation Rate": (results_df["rag_has_citation_section"].sum() /␣
 ↪len(results_df)) * 100
}

print("\n" + "-" * 80)
print("OVERALL STATISTICS")
print("-" * 80)
for key, value in summary.items():
    if isinstance(value, float):
        print(f"{key:.<50} {value:.2f}")
    else:
        print(f"{key:.<50} {value}")
```

```
================================================================================
====================
EVALUATION RESULTS SUMMARY: BASELINE vs RAG (All Modes Combined)
================================================================================
====================

--------------------------------------------------------------------------------
OVERALL STATISTICS
--------------------------------------------------------------------------------
Total Evaluations… 9
Baseline Avg Style Score… 27.88
RAG Avg Style Score… 25.81
Style Score Improvement… -2.07
Baseline Avg Embedding Similarity… 0.30
RAG Avg Embedding Similarity… 0.34
Baseline Avg Grounding Score… 36.86
RAG Avg Grounding Score… 97.50
Grounding Score Improvement… 60.64
Baseline Avg Hallucination Risk… 63.14
RAG Avg Hallucination Risk… 2.50
Hallucination Risk Reduction… 60.64
RAG Avg Citations… 9.33
RAG Citation Rate… 100.00
```

[29]:
```
print("\n" + "=" * 100)
print("BASELINE vs RAG COMPARISON TABLE")
print("=" * 100)

comparison_data = {
    "Metric": [
```

```
        "Style Composite Score (0-100)",
        "Embedding Similarity (0-1)",
        "Style Markers Score",
        "Grounding Score (0-100)",
        "Hallucination Risk (0-100)",
        "Medical Specifics Count",
        "Claim Density",
        "Total Citations",
        "Word Count"
    ],
    "Baseline Mean": [
        results_df["baseline_style_composite"].mean(),
        results_df["baseline_style_embedding_sim"].mean(),
        results_df["baseline_style_markers"].mean(),
        results_df["baseline_grounding_score"].mean(),
        results_df["baseline_hallucination_risk"].mean(),
        results_df["baseline_medical_specifics"].mean(),
        results_df["baseline_claim_density"].mean(),
        0,
        results_df["baseline_word_count"].mean()
    ],
    "RAG Mean": [
        results_df["rag_style_composite"].mean(),
        results_df["rag_style_embedding_sim"].mean(),
        results_df["rag_style_markers"].mean(),
        results_df["rag_grounding_score"].mean(),
        results_df["rag_hallucination_risk"].mean(),
        results_df["rag_medical_specifics"].mean(),
        results_df["rag_claim_density"].mean(),
        results_df["rag_total_citations"].mean(),
        results_df["rag_word_count"].mean()
    ]
}

comparison_data["Difference"] = [
    comparison_data["RAG Mean"][i] - comparison_data["Baseline Mean"][i]
    for i in range(len(comparison_data["Metric"]))
]

comparison_df = pd.DataFrame(comparison_data)
print("\n")
print(comparison_df.to_string(index=False, float_format=lambda x: f"{x:.2f}"))
```

```
================================================================================
====================
BASELINE vs RAG COMPARISON TABLE
```

```
================================================================================
====================
```

Metric	Baseline Mean	RAG Mean	Difference
Style Composite Score (0-100)	27.88	25.81	-2.07
Embedding Similarity (0-1)	0.30	0.34	0.04
Style Markers Score	0.29	0.16	-0.13
Grounding Score (0-100)	36.86	97.50	60.64
Hallucination Risk (0-100)	63.14	2.50	-60.64
Medical Specifics Count	4.67	1.22	-3.44
Claim Density	0.49	1.25	0.75
Total Citations	0.00	9.33	9.33
Word Count	570.89	472.89	-98.00

1.11 11. Cross-Mode Comparison (RAG: Full Story vs Short Awareness vs Mechanism Takeaways)

[30]:
```python
print("\n" + "=" * 100)
print("CROSS-MODE COMPARISON: RAG Performance by Mode")
print("=" * 100)

mode_stats = results_df.groupby("mode").agg({
    "rag_style_composite": ["mean", "std"],
    "rag_style_embedding_sim": ["mean", "std"],
    "rag_style_markers": ["mean", "std"],
    "rag_grounding_score": ["mean", "std"],
    "rag_hallucination_risk": ["mean", "std"],
    "rag_total_citations": ["mean", "std"],
    "rag_word_count": ["mean", "std"],
    "rag_claim_density": ["mean", "std"]
}).round(2)

mode_stats.columns = ['_'.join(col).strip() for col in mode_stats.columns.
 ↪values]
mode_stats = mode_stats.reset_index()

print("\n--- RAG Performance by Mode ---")
print(mode_stats.to_string(index=False))
```

```
================================================================================
====================
CROSS-MODE COMPARISON: RAG Performance by Mode
================================================================================
====================
```

```
--- RAG Performance by Mode ---
                mode  rag_style_composite_mean  rag_style_composite_std
rag_style_embedding_sim_mean  rag_style_embedding_sim_std
rag_style_markers_mean  rag_style_markers_std  rag_grounding_score_mean
rag_grounding_score_std  rag_hallucination_risk_mean  rag_hallucination_risk_std
rag_total_citations_mean  rag_total_citations_std  rag_word_count_mean
rag_word_count_std  rag_claim_density_mean  rag_claim_density_std
         FULL_STORY                      35.47                      6.64
0.47                         0.16                      0.28                      0.08
98.36                       0.79                        1.64
0.79                         9.0                       1.0                 815.00
52.31                       0.82                      0.39
MECHANISM_TAKEAWAYS                      22.43                      3.46
0.30                         0.04                      0.19                      0.33
95.78                       3.53                        4.22
3.53                        11.0                       3.0                 345.00
46.12                       2.11                      1.77
    SHORT_AWARENESS                      19.55                      1.37
0.25                         0.02                      0.02                      0.04
98.37                       0.95                        1.63
0.95                         8.0                       1.0                 258.67
40.13                       0.82                      0.48
```

[31]:
```python
print("\n" + "=" * 100)
print("DETAILED CROSS-MODE COMPARISON TABLE (RAG Only)")
print("=" * 100)

metrics_to_compare = [
    ("Style Composite Score", "rag_style_composite"),
    ("Embedding Similarity", "rag_style_embedding_sim"),
    ("Style Markers Score", "rag_style_markers"),
    ("Grounding Score", "rag_grounding_score"),
    ("Hallucination Risk", "rag_hallucination_risk"),
    ("Total Citations", "rag_total_citations"),
    ("Word Count", "rag_word_count"),
    ("Claim Density", "rag_claim_density")
]

cross_mode_data = {"Metric": []}
for mode in ALL_MODES:
    cross_mode_data[mode] = []

for metric_name, col_name in metrics_to_compare:
    cross_mode_data["Metric"].append(metric_name)
    for mode in ALL_MODES:
        mode_data = results_df[results_df["mode"] == mode][col_name]
```

```
        cross_mode_data[mode].append(f"{mode_data.mean():.2f} +/- {mode_data.
 ↪std():.2f}")

cross_mode_df = pd.DataFrame(cross_mode_data)
print("\n")
print(cross_mode_df.to_string(index=False))
```

```
================================================================================
====================
DETAILED CROSS-MODE COMPARISON TABLE (RAG Only)
================================================================================
====================
```

Metric	FULL_STORY	SHORT_AWARENESS	MECHANISM_TAKEAWAYS
Style Composite Score	35.47 +/- 6.64	19.55 +/- 1.37	22.43 +/- 3.46
Embedding Similarity	0.47 +/- 0.16	0.25 +/- 0.02	0.30 +/- 0.04
Style Markers Score	0.28 +/- 0.08	0.02 +/- 0.04	0.19 +/- 0.33
Grounding Score	98.36 +/- 0.79	98.37 +/- 0.95	95.78 +/- 3.53
Hallucination Risk	1.64 +/- 0.79	1.63 +/- 0.95	4.22 +/- 3.53
Total Citations	9.00 +/- 1.00	8.00 +/- 1.00	11.00 +/- 3.00
Word Count	815.00 +/- 52.31	258.67 +/- 40.13	345.00 +/- 46.12
Claim Density	0.82 +/- 0.39	0.82 +/- 0.48	2.11 +/- 1.77

[32]:
```
print("\n" + "=" * 100)
print("PAIRWISE MODE COMPARISONS (RAG)")
print("=" * 100)

def compare_modes(df, mode1, mode2, metric_col):
    data1 = df[df["mode"] == mode1][metric_col].values
    data2 = df[df["mode"] == mode2][metric_col].values
    mean_diff = data1.mean() - data2.mean()
    if len(data1) == len(data2):
        t_stat, p_value = stats.ttest_rel(data1, data2)
    else:
        t_stat, p_value = stats.ttest_ind(data1, data2)
    return {
        "mode1_mean": data1.mean(),
        "mode2_mean": data2.mean(),
        "mean_diff": mean_diff,
        "t_stat": t_stat,
        "p_value": p_value,
        "significant": p_value < 0.05
    }

mode_pairs = list(combinations(ALL_MODES, 2))
```

```
key_metrics = [
    ("Style Composite", "rag_style_composite"),
    ("Grounding Score", "rag_grounding_score"),
    ("Hallucination Risk", "rag_hallucination_risk"),
    ("Citations", "rag_total_citations"),
    ("Word Count", "rag_word_count")
]

for mode1, mode2 in mode_pairs:
    print(f"\n--- {mode1} vs {mode2} ---")
    for metric_name, col_name in key_metrics:
        comp = compare_modes(results_df, mode1, mode2, col_name)
        sig_marker = "*" if comp["significant"] else ""
        print(f"  {metric_name:20s}: {mode1}={comp['mode1_mean']:.2f},␣
 ↪{mode2}={comp['mode2_mean']:.2f}, diff={comp['mean_diff']:+.2f},␣
 ↪p={comp['p_value']:.3f}{sig_marker}")
```

```
================================================================================
====================
PAIRWISE MODE COMPARISONS (RAG)
================================================================================
====================

--- FULL_STORY vs SHORT_AWARENESS ---
  Style Composite     : FULL_STORY=35.47, SHORT_AWARENESS=19.55, diff=+15.92,
p=0.043*
  Grounding Score     : FULL_STORY=98.36, SHORT_AWARENESS=98.37, diff=-0.01,
p=0.973
  Hallucination Risk  : FULL_STORY=1.64, SHORT_AWARENESS=1.63, diff=+0.01,
p=0.973
  Citations           : FULL_STORY=9.00, SHORT_AWARENESS=8.00, diff=+1.00,
p=0.225
  Word Count          : FULL_STORY=815.00, SHORT_AWARENESS=258.67, diff=+556.33,
p=0.007*

--- FULL_STORY vs MECHANISM_TAKEAWAYS ---
  Style Composite     : FULL_STORY=35.47, MECHANISM_TAKEAWAYS=22.43,
diff=+13.04, p=0.130
  Grounding Score     : FULL_STORY=98.36, MECHANISM_TAKEAWAYS=95.78, diff=+2.57,
p=0.252
  Hallucination Risk  : FULL_STORY=1.64, MECHANISM_TAKEAWAYS=4.22, diff=-2.57,
p=0.252
  Citations           : FULL_STORY=9.00, MECHANISM_TAKEAWAYS=11.00, diff=-2.00,
p=0.321
  Word Count          : FULL_STORY=815.00, MECHANISM_TAKEAWAYS=345.00,
diff=+470.00, p=0.011*
```

```
--- SHORT_AWARENESS vs MECHANISM_TAKEAWAYS ---
  Style Composite     : SHORT_AWARENESS=19.55, MECHANISM_TAKEAWAYS=22.43,
diff=-2.88, p=0.268
  Grounding Score     : SHORT_AWARENESS=98.37, MECHANISM_TAKEAWAYS=95.78,
diff=+2.58, p=0.257
  Hallucination Risk  : SHORT_AWARENESS=1.63, MECHANISM_TAKEAWAYS=4.22,
diff=-2.58, p=0.257
  Citations           : SHORT_AWARENESS=8.00, MECHANISM_TAKEAWAYS=11.00,
diff=-3.00, p=0.286
  Word Count          : SHORT_AWARENESS=258.67, MECHANISM_TAKEAWAYS=345.00,
diff=-86.33, p=0.002*
```

[33]:
```
print("\n" + "=" * 100)
print("CROSS-MODE COMPARISON: BASELINE Performance by Mode")
print("=" * 100)

baseline_mode_stats = results_df.groupby("mode").agg({
    "baseline_style_composite": ["mean", "std"],
    "baseline_style_embedding_sim": ["mean", "std"],
    "baseline_grounding_score": ["mean", "std"],
    "baseline_hallucination_risk": ["mean", "std"],
    "baseline_word_count": ["mean", "std"]
}).round(2)

baseline_mode_stats.columns = ['_'.join(col).strip() for col in␣
 ↪baseline_mode_stats.columns.values]
baseline_mode_stats = baseline_mode_stats.reset_index()

print("\n--- Baseline Performance by Mode ---")
print(baseline_mode_stats.to_string(index=False))
```

```
================================================================================
====================
CROSS-MODE COMPARISON: BASELINE Performance by Mode
================================================================================
====================

--- Baseline Performance by Mode ---
               mode  baseline_style_composite_mean  baseline_style_composite_std
baseline_style_embedding_sim_mean  baseline_style_embedding_sim_std
baseline_grounding_score_mean  baseline_grounding_score_std
baseline_hallucination_risk_mean  baseline_hallucination_risk_std
baseline_word_count_mean  baseline_word_count_std
         FULL_STORY                           34.77                           0.08
0.43                             0.02                            10.37
10.97                            89.63                              10.97
```

```
1106.67                    104.70
MECHANISM_TAKEAWAYS                          24.58                          2.82
0.24                       0.18                          59.03
32.37                      40.97                            32.37
328.67                     281.65
    SHORT_AWARENESS                          24.30                          2.48
0.23                       0.18                          41.18
42.56                      58.82                            42.56
277.33                     233.25
```

[34]:
```python
print("\n" + "=" * 100)
print("RAG IMPROVEMENT OVER BASELINE BY MODE")
print("=" * 100)

improvement_data = {"Mode": [], "Style Score Change": [], "Grounding␣
 ↪Improvement": [], "Hallucination Risk Reduction": []}

for mode in ALL_MODES:
    mode_df = results_df[results_df["mode"] == mode]
    improvement_data["Mode"].append(mode)
    improvement_data["Style Score Change"].append(
        mode_df["rag_style_composite"].mean() -␣
 ↪mode_df["baseline_style_composite"].mean()
    )
    improvement_data["Grounding Improvement"].append(
        mode_df["rag_grounding_score"].mean() -␣
 ↪mode_df["baseline_grounding_score"].mean()
    )
    improvement_data["Hallucination Risk Reduction"].append(
        mode_df["baseline_hallucination_risk"].mean() -␣
 ↪mode_df["rag_hallucination_risk"].mean()
    )

improvement_df = pd.DataFrame(improvement_data)
print("\n")
print(improvement_df.to_string(index=False, float_format=lambda x: f"{x:+.2f}"))
```

```
====================================================================================
====================
RAG IMPROVEMENT OVER BASELINE BY MODE
====================================================================================
====================


                Mode  Style Score Change  Grounding Improvement  Hallucination
Risk Reduction
```

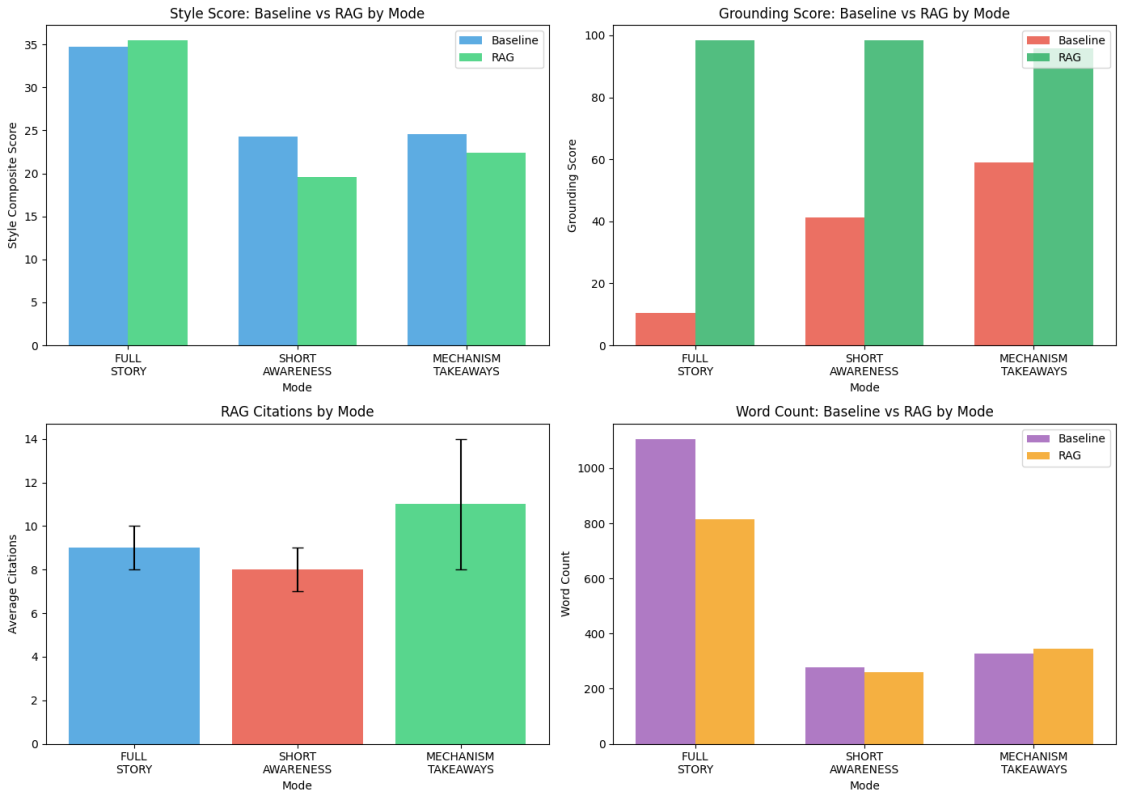
```
        FULL_STORY                +0.70                +87.99
+87.99
    SHORT_AWARENESS               -4.75                +57.19
+57.19
MECHANISM_TAKEAWAYS               -2.15                +36.76
+36.76
```

1.12 12. Visualizations

[35]:
```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Style Score by Mode (Baseline vs RAG)
ax1 = axes[0, 0]
mode_means_baseline = results_df.groupby("mode")["baseline_style_composite"].
 ↪mean()
mode_means_rag = results_df.groupby("mode")["rag_style_composite"].mean()
x = np.arange(len(ALL_MODES))
width = 0.35
ax1.bar(x - width/2, [mode_means_baseline[m] for m in ALL_MODES], width,␣
 ↪label="Baseline", color="#3498db", alpha=0.8)
ax1.bar(x + width/2, [mode_means_rag[m] for m in ALL_MODES], width,␣
 ↪label="RAG", color="#2ecc71", alpha=0.8)
ax1.set_xlabel("Mode")
ax1.set_ylabel("Style Composite Score")
ax1.set_title("Style Score: Baseline vs RAG by Mode")
ax1.set_xticks(x)
ax1.set_xticklabels([m.replace("_", "\n") for m in ALL_MODES])
ax1.legend()

# 2. Grounding Score by Mode
ax2 = axes[0, 1]
mode_means_baseline_grounding = results_df.
 ↪groupby("mode")["baseline_grounding_score"].mean()
mode_means_rag_grounding = results_df.groupby("mode")["rag_grounding_score"].
 ↪mean()
ax2.bar(x - width/2, [mode_means_baseline_grounding[m] for m in ALL_MODES],␣
 ↪width, label="Baseline", color="#e74c3c", alpha=0.8)
ax2.bar(x + width/2, [mode_means_rag_grounding[m] for m in ALL_MODES], width,␣
 ↪label="RAG", color="#27ae60", alpha=0.8)
ax2.set_xlabel("Mode")
ax2.set_ylabel("Grounding Score")
ax2.set_title("Grounding Score: Baseline vs RAG by Mode")
ax2.set_xticks(x)
ax2.set_xticklabels([m.replace("_", "\n") for m in ALL_MODES])
ax2.legend()

# 3. RAG Citations by Mode
```

```
ax3 = axes[1, 0]
mode_means_citations = results_df.groupby("mode")["rag_total_citations"].mean()
mode_std_citations = results_df.groupby("mode")["rag_total_citations"].std()
ax3.bar(ALL_MODES, [mode_means_citations[m] for m in ALL_MODES],
        yerr=[mode_std_citations[m] for m in ALL_MODES],
        color=["#3498db", "#e74c3c", "#2ecc71"], alpha=0.8, capsize=5)
ax3.set_xlabel("Mode")
ax3.set_ylabel("Average Citations")
ax3.set_title("RAG Citations by Mode")
ax3.set_xticklabels([m.replace("_", "\n") for m in ALL_MODES])

# 4. Word Count by Mode
ax4 = axes[1, 1]
mode_means_wc_baseline = results_df.groupby("mode")["baseline_word_count"].
 ↪mean()
mode_means_wc_rag = results_df.groupby("mode")["rag_word_count"].mean()
ax4.bar(x - width/2, [mode_means_wc_baseline[m] for m in ALL_MODES], width,␣
 ↪label="Baseline", color="#9b59b6", alpha=0.8)
ax4.bar(x + width/2, [mode_means_wc_rag[m] for m in ALL_MODES], width,␣
 ↪label="RAG", color="#f39c12", alpha=0.8)
ax4.set_xlabel("Mode")
ax4.set_ylabel("Word Count")
ax4.set_title("Word Count: Baseline vs RAG by Mode")
ax4.set_xticks(x)
ax4.set_xticklabels([m.replace("_", "\n") for m in ALL_MODES])
ax4.legend()

plt.tight_layout()
plt.savefig(PATHS["eval_charts"] / "cross_mode_comparison.png", dpi=150,␣
 ↪bbox_inches="tight")
plt.show()
print("Saved cross-mode comparison chart")
```

```
/tmp/ipython-input-3781817455.py:41: UserWarning: set_ticklabels() should only
be used with a fixed number of ticks, i.e. after set_ticks() or using a
FixedLocator.
  ax3.set_xticklabels([m.replace("_", "\n") for m in ALL_MODES])
```

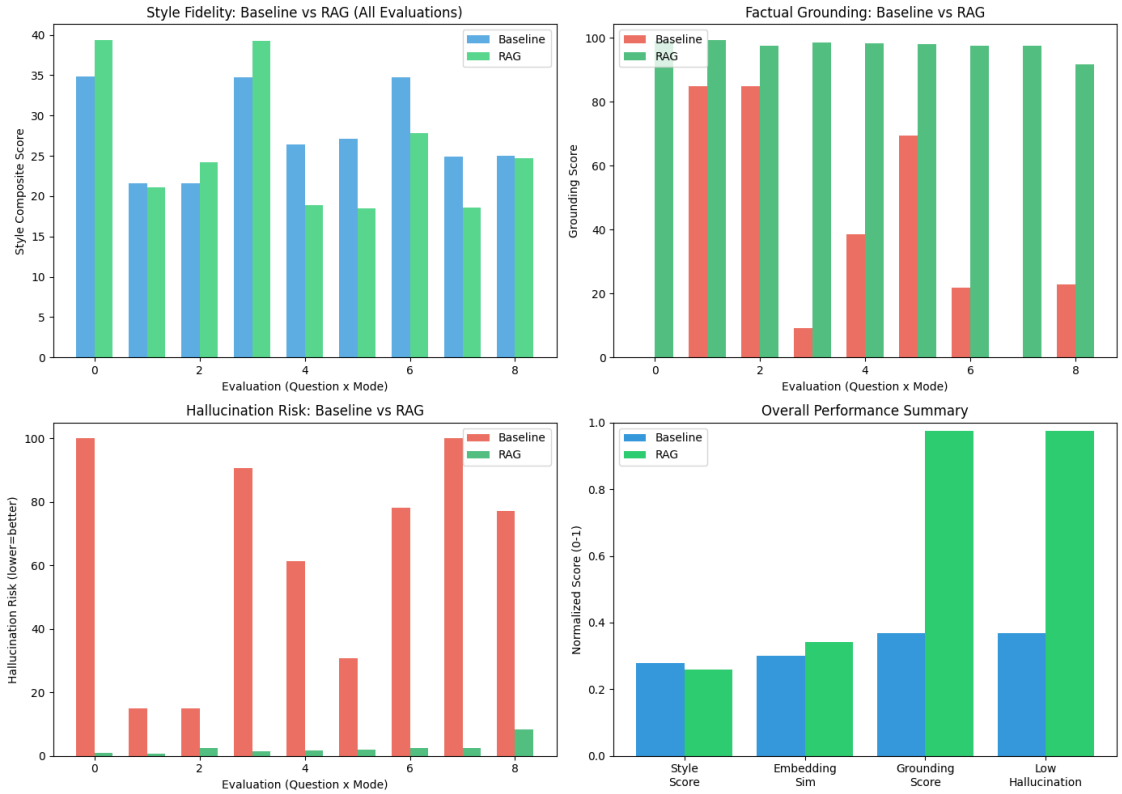


Saved cross-mode comparison chart

[37]:
```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Overall Style Score Comparison
ax1 = axes[0, 0]
question_ids = results_df["question_id"].unique()
x = np.arange(len(results_df))
width = 0.35
ax1.bar(x - width/2, results_df["baseline_style_composite"], width,␣
 ↪label="Baseline", color="#3498db", alpha=0.8)
ax1.bar(x + width/2, results_df["rag_style_composite"], width, label="RAG",␣
 ↪color="#2ecc71", alpha=0.8)
ax1.set_xlabel("Evaluation (Question x Mode)")
ax1.set_ylabel("Style Composite Score")
ax1.set_title("Style Fidelity: Baseline vs RAG (All Evaluations)")
ax1.legend()

# 2. Grounding Score Comparison
ax2 = axes[0, 1]
ax2.bar(x - width/2, results_df["baseline_grounding_score"], width,␣
 ↪label="Baseline", color="#e74c3c", alpha=0.8)
```

```
ax2.bar(x + width/2, results_df["rag_grounding_score"], width, label="RAG",␣
 ↪color="#27ae60", alpha=0.8)
ax2.set_xlabel("Evaluation (Question x Mode)")
ax2.set_ylabel("Grounding Score")
ax2.set_title("Factual Grounding: Baseline vs RAG")
ax2.legend()

# 3. Hallucination Risk
ax3 = axes[1, 0]
ax3.bar(x - width/2, results_df["baseline_hallucination_risk"], width,␣
 ↪label="Baseline", color="#e74c3c", alpha=0.8)
ax3.bar(x + width/2, results_df["rag_hallucination_risk"], width, label="RAG",␣
 ↪color="#27ae60", alpha=0.8)
ax3.set_xlabel("Evaluation (Question x Mode)")
ax3.set_ylabel("Hallucination Risk (lower=better)")
ax3.set_title("Hallucination Risk: Baseline vs RAG")
ax3.legend()

# 4. Summary Radar-like comparison
ax4 = axes[1, 1]
categories = ["Style\nScore", "Embedding\nSim", "Grounding\nScore",␣
 ↪"Low\nHallucination"]
baseline_vals = [
    results_df["baseline_style_composite"].mean() / 100,
    results_df["baseline_style_embedding_sim"].mean(),
    results_df["baseline_grounding_score"].mean() / 100,
    1 - (results_df["baseline_hallucination_risk"].mean() / 100)
]
rag_vals = [
    results_df["rag_style_composite"].mean() / 100,
    results_df["rag_style_embedding_sim"].mean(),
    results_df["rag_grounding_score"].mean() / 100,
    1 - (results_df["rag_hallucination_risk"].mean() / 100)
]

x_pos = np.arange(len(categories))
ax4.bar(x_pos - 0.2, baseline_vals, 0.4, label="Baseline", color="#3498db")
ax4.bar(x_pos + 0.2, rag_vals, 0.4, label="RAG", color="#2ecc71")
ax4.set_xticks(x_pos)
ax4.set_xticklabels(categories)
ax4.set_ylabel("Normalized Score (0-1)")
ax4.set_title("Overall Performance Summary")
ax4.legend()
ax4.set_ylim(0, 1)

plt.tight_layout()
```

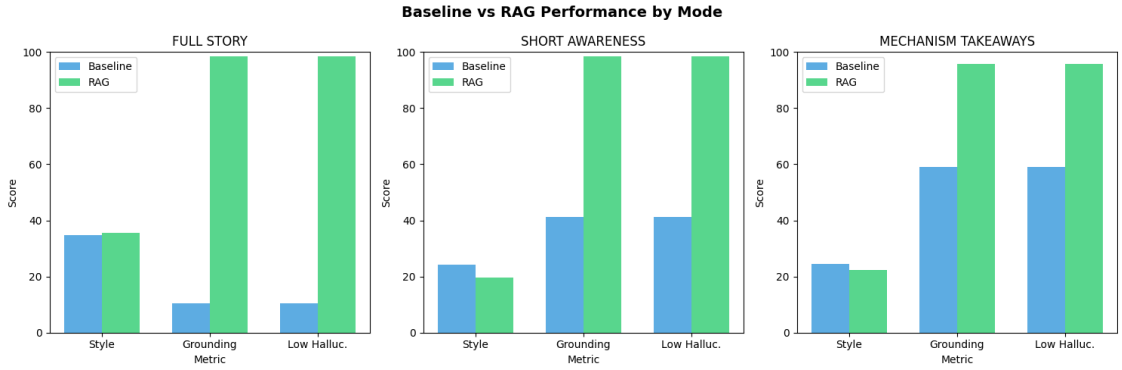
```
plt.savefig(PATHS["eval_charts"] / "baseline_vs_rag_comparison.png", dpi=150,␣
 ↪bbox_inches="tight")
plt.show()
print("Saved baseline vs RAG comparison chart")
```




```
Saved baseline vs RAG comparison chart
```

[38]:
```
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for idx, mode in enumerate(ALL_MODES):
    ax = axes[idx]
    mode_data = results_df[results_df["mode"] == mode]

    metrics = ["Style", "Grounding", "Low Halluc."]
    baseline_vals = [
        mode_data["baseline_style_composite"].mean(),
        mode_data["baseline_grounding_score"].mean(),
        100 - mode_data["baseline_hallucination_risk"].mean()
    ]
    rag_vals = [
        mode_data["rag_style_composite"].mean(),
        mode_data["rag_grounding_score"].mean(),
```

```
        100 - mode_data["rag_hallucination_risk"].mean()
    ]

    x = np.arange(len(metrics))
    width = 0.35

    ax.bar(x - width/2, baseline_vals, width, label="Baseline",␣
 ↪color="#3498db", alpha=0.8)
    ax.bar(x + width/2, rag_vals, width, label="RAG", color="#2ecc71", alpha=0.
 ↪8)

    ax.set_xlabel("Metric")
    ax.set_ylabel("Score")
    ax.set_title(f"{mode.replace('_', ' ')}")
    ax.set_xticks(x)
    ax.set_xticklabels(metrics)
    ax.legend()
    ax.set_ylim(0, 100)

plt.suptitle("Baseline vs RAG Performance by Mode", fontsize=14,␣
 ↪fontweight="bold")
plt.tight_layout()
plt.savefig(PATHS["eval_charts"] / "mode_specific_comparison.png", dpi=150,␣
 ↪bbox_inches="tight")
plt.show()
print("Saved mode-specific comparison chart")
```




```
Saved mode-specific comparison chart
```

1.13 13. Save Final Results

[39]:
```
# Save comprehensive summary
final_summary = {
    "overall": summary,
    "by_mode": {},
    "cross_mode_comparison": cross_mode_df.to_dict("records"),
    "improvement_by_mode": improvement_df.to_dict("records")
}

for mode in ALL_MODES:
    mode_data = results_df[results_df["mode"] == mode]
    final_summary["by_mode"][mode] = {
        "baseline_style_mean": mode_data["baseline_style_composite"].mean(),
        "rag_style_mean": mode_data["rag_style_composite"].mean(),
        "baseline_grounding_mean": mode_data["baseline_grounding_score"].mean(),
        "rag_grounding_mean": mode_data["rag_grounding_score"].mean(),
        "rag_citations_mean": mode_data["rag_total_citations"].mean(),
        "baseline_word_count_mean": mode_data["baseline_word_count"].mean(),
        "rag_word_count_mean": mode_data["rag_word_count"].mean()
    }

with open(PATHS["eval"] / "comprehensive_summary.json", "w", encoding="utf-8")␣
 ↪as f:
    json.dump(final_summary, f, indent=2, default=float)
print(f"Saved comprehensive summary to {PATHS['eval'] / 'comprehensive_summary.
 ↪json'}")

comparison_df.to_csv(PATHS["eval"] / "baseline_vs_rag_table.csv", index=False)
cross_mode_df.to_csv(PATHS["eval"] / "cross_mode_comparison_table.csv",␣
 ↪index=False)
improvement_df.to_csv(PATHS["eval"] / "improvement_by_mode_table.csv",␣
 ↪index=False)
print("Saved all comparison tables")
```

```
Saved comprehensive summary to eval/comprehensive_summary.json
Saved all comparison tables
```

[40]:
```
print("\n" + "=" * 100)
print("EVALUATION COMPLETE")
print("=" * 100)
print(f"\nTotal evaluations completed: {len(results_df)}")
print(f"Questions tested: {len(EVAL_QUESTIONS)}")
print(f"Modes tested: {len(ALL_MODES)} ({', '.join(ALL_MODES)})")
print(f"\nOutput files:")
print(f"  - {PATHS['eval'] / 'detailed_results.csv'}")
print(f"  - {PATHS['eval'] / 'comprehensive_summary.json'}")
```

```
print(f"  - {PATHS['eval'] / 'baseline_vs_rag_table.csv'}")
print(f"  - {PATHS['eval'] / 'cross_mode_comparison_table.csv'}")
print(f"  - {PATHS['eval'] / 'improvement_by_mode_table.csv'}")
print(f"  - {PATHS['eval_outputs'] / 'all_outputs.json'}")
print(f"  - {PATHS['eval_charts'] / 'cross_mode_comparison.png'}")
print(f"  - {PATHS['eval_charts'] / 'baseline_vs_rag_comparison.png'}")
print(f"  - {PATHS['eval_charts'] / 'mode_specific_comparison.png'}")
```

```
================================================================================
====================
EVALUATION COMPLETE
================================================================================
====================

Total evaluations completed: 9
Questions tested: 3
Modes tested: 3 (FULL_STORY, SHORT_AWARENESS, MECHANISM_TAKEAWAYS)

Output files:
  - eval/detailed_results.csv
  - eval/comprehensive_summary.json
  - eval/baseline_vs_rag_table.csv
  - eval/cross_mode_comparison_table.csv
  - eval/improvement_by_mode_table.csv
  - eval/outputs/all_outputs.json
  - eval/charts/cross_mode_comparison.png
  - eval/charts/baseline_vs_rag_comparison.png
  - eval/charts/mode_specific_comparison.png
```

[ ]: