VeriFact — Automated Neural Claim Verification System

Detailed Presentation Script (Expert Panel)

SECTION 1: Introduction & Problem Statement

[~3 minutes]

Good morning/afternoon. My name is Adetya Jamwal, and today I'll be presenting **VeriFact** — an automated neural claim verification system that I've designed and engineered end-to-end.

The Problem

The misinformation crisis is not a content problem — it's a **velocity problem**. Roughly 4.5 billion pieces of content are created daily across digital platforms. Manual fact-checking organizations — Snopes, PolitiFact, Full Fact — have editorial teams of 20–50 people. They can verify perhaps 10–15 claims per day. The asymmetry is staggering: content creation outpaces verification by several orders of magnitude.

Now, the naive solution is: "just train a classifier on fake vs. real news." And that's exactly what most academic literature proposes — supervised models trained on datasets like LIAR, FakeNewsNet, or ISOT. The problem? **Dataset obsolescence.** A model trained on 2020 political misinformation cannot detect 2024 health misinformation or 2025 geopolitical claims. The topics shift, the language evolves, and the classifier's decision boundary becomes meaningless.

My Approach

VeriFact takes a fundamentally different approach. Instead of learning *what* is fake, it learns *how to verify*. The system is a **retrieval-augmented verification pipeline** — it retrieves live evidence from the open web and uses a **cross-encoder NLI model** to determine the logical relationship between the claim and retrieved evidence. It's inference-only — no custom training, no static fact databases, no obsolescence.

Scope Boundaries

Let me be precise about what this system does and does not do:

- **Does:** Verify English-language textual claims against web-sourced evidence, with provenance tracking and semantic reasoning.
- **Does not:** Verify images, video (deepfakes), audio, or generate new knowledge. It is strictly bounded to what it can retrieve and reason about.

SECTION 2: System Architecture

[~5 minutes]

Design Philosophy

Before I walk through components, let me state the design philosophy clearly because it dictates every architectural decision:

Provenance over generation. Explainability over accuracy optimization.

Unlike an LLM-based fact-checker that might hallucinate "facts," every signal in VeriFact's output is traceable to a specific URL, a specific sentence within that URL, and a specific NLI score. The system never invents knowledge — it only retrieves and reasons.

High-Level Data Flow

The pipeline has six stages. Let me walk through each:

```
User Input → Claim Extraction → Query Generation → Evidence Retrieval →
    Semantic Filtering & Stance Detection → Weighted Aggregation → Verdict +
Explanation
```

Layered Architecture

The system is organized in four layers:

1. **Interface Layer** — Flask REST API with Pydantic input validation, rate limiting (5/min per IP, 100/hour), and CORS. Served via Gunicorn with threaded workers for concurrent I/O alongside blocking CPU inference.

2. **Controller Layer** — The [check_claim()](file:///c:/Fake_News_Detector/app_flask.py#184-274) function in [app_flask.py](file:///c:/Fake_News_Detector/app_flask.py) orchestrates the end-to-end pipeline. It's synchronous at the request level but internally parallelizes I/O-bound work.

3. **Core Services Layer** — 11 specialized modules, each with a single responsibility:

    - `claim_extractor` — Input normalization
    - `query_generator` — Search query expansion
    - [web_search](file:///c:/Fake_News_Detector/app/core/web_search.py#220-253) — Multi-API retrieval with fallback
    - `scraper` — Content extraction
    - `embedder` — Semantic similarity
    - `stance_detector` — NLI-based stance classification
    - `evidence_aggregator` — Parallel scraping, embedding, and stance aggregation
    - `verdict_engine` — Weighted scoring and verdict computation
    - `source_scorer` — Domain credibility weighting
    - `llm_helper` — Optional Groq-powered enhancement
    - `model_registry` — Thread-safe singleton model management

4. **External Interfaces** — Wrappers for Tavily, Brave, DuckDuckGo search APIs, Groq LLM API, and HuggingFace model hub.

SECTION 3: The Pipeline — Stage by Stage

[~12 minutes]

Stage 1: Claim Extraction ([claim_extractor.py](file:///c:/Fake_News_Detector/app/core/claim_extractor.py))

When a user submits input, it can be:

- A raw text string
- A URL (from which we extract the article body using `trafilatura`)
- A direct claim statement

For text inputs, I don't simply take the first sentence. I implemented a **sentence importance scoring** heuristic:

- **Named entity density:** +2 points per unique entity type (PERSON, ORG, GPE, etc.)
- **Statistical content:** +1 if the sentence contains numeric tokens
- **Length constraints:** +1 for sentences between 30–200 characters (optimal for verification)
- **Quotation marks:** +1.5 (direct quotes are strong claim indicators)
- **Position bonus:** Earlier sentences get a decaying bonus (first sentence +1.5, second +1.0, third +0.5)

The top-scoring sentence is selected as the primary claim. This is critical because articles often bury the verifiable claim deep within context.

Additionally, the system runs **embedding-based keyword extraction** — functionally equivalent to KeyBERT but using my local SBERT model. It generates n-gram candidates from noun chunks and named entities, embeds them alongside the full text, ranks by cosine similarity, and applies **Maximal Marginal Relevance (MMR)** with λ=0.7 to balance relevance and diversity. These keywords feed downstream query generation.

Stage 2: Query Generation ([query_generator.py](file:///c:/Fake_News_Detector/app/core/query_generator.py))

A single claim needs multiple search queries to avoid **keyword myopia** — the tendency of search engines to fixate on literal keywords and miss the semantic intent.

The system uses a **dual-strategy approach**:

Strategy A — LLM Decomposition (Primary): When Groq is available, the claim is sent to Llama-3.3-70b with a structured prompt requesting 4–5 precise, verifiable search queries. The LLM is instructed to include queries that could *disprove* the claim — this is critical for avoiding confirmation bias in evidence retrieval.

Strategy B — NER-based Expansion (Fallback): Using spaCy's `en_core_web_sm`, named entities are extracted. The system then generates:

- The exact claim + verification suffixes ("fact check", "true or false", "hoax")
- Entity-centric queries ("*entity* controversy", "*entity* news verification")
- Keyword-based queries from the MMR-selected keywords

In both cases, queries are capped at 10 and deduplicated. The LLM path preserves order via `dict.fromkeys()`; the NER fallback deduplicates via `set()`.

Stage 3: Evidence Retrieval ([web_search.py](file:///c:/Fake_News_Detector/app/core/web_search.py))

This is where I implemented a **3-tier search fallback chain**:

1. **Tavily** (primary) — Uses their `advanced` search depth, best quality results, fastest. Requires API key.
2. **Brave Search** (secondary) — 2,000 free requests/month, reliable. 0.3-second delay between queries to avoid throttling.
3. **DuckDuckGo** (tertiary) — Free, no API key. Uses `lite` backend to avoid rate limiting, with 3 retries and exponential backoff. Also falls back to DDG's news search if web search returns empty.

Each tier is wrapped in exception handling. If Tavily returns no results or throws an error, Brave is tried. If Brave fails, DuckDuckGo activates. The system URL-deduplicates across all tiers.

API clients are **cached as singletons** — the Tavily client and Brave session are initialized once and reused, avoiding TCP connection overhead on repeated calls.

Stage 4: Evidence Processing ([evidence_aggregator.py](file:///c:/Fake_News_Detector/app/core/evidence_aggregator.py) + [scraper.py](file:///c:/Fake_News_Detector/app/core/scraper.py) + [embedder.py](file:///c:/Fake_News_Detector/app/core/embedder.py))

This stage has three sub-processes running in parallel using a `ThreadPoolExecutor` (3 workers, down from 5 to manage memory on my EC2 instance):

4a. Scraping: Each URL is scrapped via `trafilatura` with an 8-second timeout. Trafilatura performs intelligent content extraction — strips ads, navigation bars, sidebars, and boilerplate. The scraper uses a `requests.Session` with connection pooling (10 connections), automatic retries (2 retries with 0.5s backoff for 429/5xx errors), and a custom User-Agent.

If a URL returns 403 (Cloudflare, paywalled sites like NYT), it's gracefully skipped — no pipeline halt.

4b. Sentence Segmentation: Extracted text (capped at 10,000 characters) is segmented into sentences using spaCy's sentence boundary detection, filtered to >20 characters, and capped at 50 sentences per source.

4c. Semantic Filtering: Here's where the SBERT model becomes critical. The claim and all candidate sentences are mapped to a **384-dimensional vector space** using `all-MiniLM-L6-v2`. For each source, the sentence with the highest cosine similarity to the claim is selected:

$$Sim(C, S_i) = \frac{C \cdot S_i}{|C| |S_i|}$$

The claim embedding is **LRU-cached** (32 entries), so if multiple sources are being compared against the same claim, the claim is only embedded once.

Sentences below a **similarity threshold of 0.25** are early-exited — they skip stance detection entirely and are marked as "discusses" with 0 confidence. This saves expensive Transformer inference on irrelevant content.

Stage 5: Stance Detection ([stance_detector.py](file:///c:/Fake_News_Detector/app/core/stance_detector.py))

This is the intellectual core of the system. I use a **cross-encoder NLI model:** `nli-deberta-v3-small` running locally.

The cross-encoder takes the evidence sentence as the **premise** and the claim as the **hypothesis**, and outputs three logits: *entailment*, *contradiction*, *neutral*. These are softmaxed into probabilities.

Why a Cross-Encoder and Not a Bi-Encoder?

A bi-encoder (like SBERT) encodes premise and hypothesis separately — fast but loses cross-attention between the two texts. A cross-encoder processes the pair jointly through all Transformer layers, enabling it to capture fine-grained logical relationships. For NLI, cross-encoders consistently outperform bi-encoders by 5–8% on benchmarks.

Why DeBERTa-v3-small and Not BART-MNLI?

I originally used `facebook/bart-large-mnli` via HuggingFace's Inference API as a **zero-shot classifier** — it re-framed NLI as classification with user-defined labels. I migrated to DeBERTa-v3 as a **direct cross-encoder NLI model** for three reasons:

1. **Architectural correctness:** Zero-shot classification repurposes NLI by constructing synthetic hypotheses like "This text is about support." Direct NLI, by contrast, takes the actual premise-hypothesis pair and outputs entailment/contradiction/neutral — this is what we actually need.
2. **Reliability:** External API calls introduced rate limits, 503 errors, and unpredictable latency. Local inference eliminates all of that.
3. **Latency control:** Local inference at ~50ms/call on CPU is deterministic, versus 200ms–2s from an API. DeBERTa-v3-small achieves 88% on MNLI — excellent for its size class.

Confidence Calibration

Raw softmax outputs from neural networks are notoriously overconfident. I apply **temperature scaling** with T=1.3:

$$calibrated = raw^{1/T}$$

This pulls high-confidence scores slightly downward, improving calibration without changing the ranking of labels.

Minimum Confidence Thresholds

Entailment ("supports") requires ≥0.50 confidence; contradiction ("refutes") requires ≥0.40. Below these thresholds, the label is downgraded to "discusses." The asymmetry is intentional — false support (topic overlap) is more common than false refutation, so we require more evidence to say "supports."

High-Stakes Claim Handling

Claims containing keywords like "killed," "assassinated," "nuclear," "war" activate a stricter threshold: support requires ≥0.70 confidence. This prevents the system from confidently validating dangerous claims based on topical overlap.

Outcome Mismatch Detection (v2.2)

This was a direct response to a critical failure case where the system incorrectly validated claims like "Trump was assassinated" because evidence about the "assassination attempt" had high topical similarity but opposite semantic meaning.

The module now runs **before** NLI inference and checks for:

- **Negating modifiers:** "attempted," "survived," "failed," "thwarted," "prevented" in evidence but not in claim → override to "refutes" (0.75 confidence)
- **Action changes:** "repositioned," "planned," "threatened" → override to "discusses" (0.60 confidence)
- **Explicit negation patterns:** Regex patterns for "not dead," "is alive," "hoax," "debunked," "misinformation," "conspiracy theory"

Relationship Claim Verification (v2.3)

Claims asserting familial relationships (e.g., "Rahul Gandhi is Modi's son") required special handling because NLI models are poor at entity-level factual verification — they can detect semantic relatedness but not factual correctness of specific relationships.

The pipeline:

1. [is_relationship_claim()](file:///c:/Fake_News_Detector/app/core/stance_detector.py#79-83) — Keyword scan for "son of," "daughter of," "married to," etc.
2. [extract_relationship_entities()](file:///c:/Fake_News_Detector/app/core/stance_detector.py#85-117) — Regex-based extraction of (subject, relationship, object) tuples
3. [verify_relationship_claim()](file:///c:/Fake_News_Detector/app/core/stance_detector.py#119-161) — If evidence mentions the same subject with the same relationship type but a *different* object → "refutes" with 0.85 confidence

Stage 6: Verdict Computation ([verdict_engine.py](file:///c:/Fake_News_Detector/app/core/verdict_engine.py))

Evidence Scoring

Each piece of evidence contributes a scalar score combining four factors:

$$Score_i = Sim_i \times Conf_i \times Dir_i \times W_{src,i} \times Boost_i$$

Where:

- **Sim** = cosine similarity (0–1)
- **Conf** = calibrated NLI confidence (0–1)
- **Dir** = +1 (supports), −1 (refutes), 0 (discusses)
- **W_src** = source credibility weight (0.3–1.5)
- **Boost** = similarity boost: $1.0 + (Sim - 0.5) \times 0.5$ — amplifies high-similarity evidence, dampens low-similarity

Source Credibility Weighting ([source_scorer.py](file:///c:/Fake_News_Detector/app/core/source_scorer.py))

I maintain a three-tier domain lookup:

Tier	Weight	Examples
Tier 1 — Wire services, fact-checkers	1.5	Reuters, AP, Snopes, PolitiFact
Tier 1.5 — Major newspapers, gov/edu	1.3–1.4	BBC, NYT, Guardian, .gov, .edu
Tier 2 — Standard/unknown	1.0	Any unlisted domain
Tier 3 — Social media	0.4–0.6	Twitter, Facebook, Reddit, TikTok

This is a static lookup table — a known limitation I'll address later.

Net Score Aggregation

$$NetScore = \sum_{i=1}^{N} Score_i$$

Verdict Thresholds

Verdict	Condition
LIKELY TRUE	NetScore > +0.35
LIKELY FALSE	NetScore < −0.35
MIXED / MISLEADING	−0.35 ≤ NetScore ≤ +0.35
UNVERIFIED	No evidence found

The thresholds were tuned from the original ±0.4 to ±0.35 for better recall without significantly hurting precision.

Confidence is derived via sigmoid: $confidence = \sigma(|NetScore|) = \frac{1}{1 + e^{-|NetScore|}}$

LLM Tiebreaker (v2.0)

When the NLI pipeline produces a MIXED verdict, the system optionally invokes Groq's Llama-3.3-70b as a **tiebreaker**. The LLM receives the claim and up to 8 pieces of evidence with their stances, and is asked to synthesize a verdict with reasoning.

The LLM's verdict is only accepted if:

- Confidence ≥ 0.60
- Verdict is either LIKELY TRUE, LIKELY FALSE, or UNVERIFIABLE

This is not an LLM-first system — the LLM is a fallback for when the NLI pipeline is genuinely inconclusive. The NLI verdict always takes priority when it's decisive.

AI-Generated Summary

Finally, the system generates a 2–3 sentence plain-language summary explaining *why* the verdict was reached, referencing specific sources and evidence. This is generated by Groq with a carefully crafted prompt that forbids technical jargon, scoring references, or system-centric language. It's designed as if a journalist is explaining the finding.

If Groq is unavailable, the system falls back to a rule-based decision reason from the explanation builder.

SECTION 4: Model Selection & Trade-offs

[~3 minutes]

Model	Purpose	Size	Latency (CPU)	Why This Model?
nli-deberta-v3-small	Stance detection (NLI)	~180 MB	~50ms/pair	88% MNLI accuracy, cross-encoder architecture trained for direct NLI (entailment/contradiction/neutral), runs on CPU
all-MiniLM-L6-v2	Sentence embeddings	~90 MB	~10ms/sent	Best STS benchmark performance in its size class, 384-dim embeddings are compact yet expressive
en_core_web_sm	Tokenization + NER	~12 MB	<5ms	Lightweight CPU model, sufficient for sentence segmentation and named entity extraction
Llama-3.3-70b	LLM reasoning (optional)	API-based	~1–2s	Hosted on Groq for fast inference; used only for query decomposition, tiebreaking, and summaries

Key Design Decision: Pre-trained NLI vs. Supervised Fine-tuning

I deliberately chose a **pre-trained NLI model used in inference-only mode** over supervised fine-tuning on fact-checking datasets. A supervised model trained on LIAR or FakeNewsNet would perform well on those specific distributions but degrade rapidly on novel domains. The DeBERTa-v3 NLI model was trained on the Multi-Genre NLI (MNLI) corpus, which teaches it generalized *logical reasoning* — it understands entailment and contradiction at a linguistic level, not at a topic level. This makes the system **domain-agnostic** and future-proof without any custom training.

Thread-Safe Model Management ([model_registry.py](file:///c:/Fake_News_Detector/app/core/model_registry.py))

All models are managed through a centralized registry with:

- **Double-checked locking** — Thread-safe singleton initialization using `threading.Lock`
- **Lazy loading** — Models are only loaded when first needed (saves ~3 minutes of startup time)
- **Warmup endpoint** — `/warmup` POST endpoint triggers pre-loading of all models with verification tests
- **Gradient disabled** — `torch.set_grad_enabled(False)` globally for inference, reducing memory footprint

---

SECTION 5: Deployment & Infrastructure

[~2 minutes]

Containerization

- **Docker image** based on `python:3.10-slim`
- Non-root user (`appuser`) for security
- ML models are lazy-loaded at runtime (downloaded on first request or via `/warmup` endpoint), keeping the base image lightweight

Runtime Environment

- **AWS EC2** (c7i-flex.large, 4 GB RAM)
- **Gunicorn** with threaded workers: 2 workers × 4 threads
- 120-second timeout to accommodate worst-case inference
- **WhiteNoise** for static file serving in production

CI/CD Pipeline

Fully automated via **GitHub Actions**:

1. Run `pytest` suite
2. Build Docker image → push to DockerHub
3. SSH into EC2 → prune old containers/images → pull new image → restart container

---

SECTION 6: Performance Assessment

[~2 minutes]

How to Evaluate a Retrieval-Augmented Verification System

This is not a standard classifier where you compute accuracy on a held-out test set. The system's performance depends on:

1. **Retrieval quality** — Are we finding relevant, authoritative sources?
2. **Semantic filtering precision** — Is the similarity threshold filtering noise correctly?
3. **Stance detection accuracy** — Is the NLI model correctly classifying entailment/contradiction?
4. **End-to-end verdict correctness** — Does the weighted aggregation produce the right verdict?

Evaluation Methodology

- **Unit tests** across all modules (pytest) — verdict computation, scoring, thresholds, edge cases
- **Integration tests** — End-to-end pipeline tests with known claims
- **Ablation testing** — Testing with and without LLM features to measure enhancement value
- **Manual testing** on adversarial claims — relationship claims, high-stakes claims, sarcastic claims, partial truths

Observed Latency Profile

Stage	Typical Time
Claim extraction	<1s

Stage	Typical Time
Query generation (LLM)	1–2s
Web search (Tavily)	2–4s
Scraping (parallel, 3 sources)	3–8s
Embedding + Stance (per source)	~0.5s
Verdict computation	<0.1s
LLM summary	1–2s
Total (end-to-end)	**8–18s**

SECTION 7: Key Challenges Overcome

[~3 minutes]

1. Semantic Gap / Action Conflation

Problem: Evidence about an "assassination attempt" was being matched to a claim about "assassination" with high similarity — same people, same event, opposite outcome.

Solution: Built an outcome mismatch detection module that runs *before* NLI. It scans for 30+ negating modifiers and action change keywords, plus regex patterns for explicit negations like "not dead," "survived," "debunked." This pre-filter catches cases where the SBERT similarity is misleadingly high.

2. Relationship Claim False Positives

Problem: "Rahul Gandhi is Modi's son" would get "supports" because evidence discussing both individuals in political contexts had high topical overlap.

Solution: Built a specialized verification pipeline that extracts (subject, relationship, object) tuples from both claim and evidence. If the subject and relationship match but the object differs ("son of Modi" vs. "son of Rajiv Gandhi"), it overrides to "refutes" with 85% confidence.

3. API Reliability & Cold Start

Problem: Original system depended on HuggingFace Inference API for both NLI and embeddings — rate limits, 503 errors, unpredictable latency.

Solution: Migrated all core ML to local inference: DeBERTa-v3-small for NLI and MiniLM for SBERT. This eliminated external dependencies for the critical path. Groq LLM is the only remaining external ML dependency, and the system works completely without it.

4. Memory Constraints on EC2 (4 GB RAM)

Problem: Loading DeBERTa + MiniLM + spaCy simultaneously approached the 4 GB limit, causing OOM kills.

Solution: Reduced ThreadPoolExecutor workers from 5 to 3, capped evidence sources, reduced spaCy processing limit from 50k to 10k characters, capped sentences at 50 per source, and disabled gradient

computation globally.

5. Overconfident Models

Problem: Raw NLI softmax outputs often showed >90% confidence on topically similar but logically different claims.

Solution: Applied temperature scaling (T=1.3) for confidence calibration, added asymmetric minimum confidence thresholds (0.50 for support, 0.40 for refutation), and imposed stricter thresholds for high-stakes claims (0.70) and relationship claims (0.80).

SECTION 8: Practical Significance

[~1 minute]

VeriFact demonstrates that **production-viable fact-checking** doesn't require massive infrastructure. The entire system — including three Transformer models and a full NLP pipeline — runs on a single 4 GB EC2 instance with purely CPU inference.

The system's value propositions:

1. **No training data dependency** — Works on any topic without retraining
2. **Full provenance** — Every verdict traces to source URLs and specific sentences
3. **Graceful degradation** — Works without Groq LLM; works with any one search API; handles scraping failures
4. **Extensible architecture** — Adding a new search provider, a new scoring heuristic, or a new model is a single-module change

The broader insight: combining **retrieval-augmented generation patterns** with **NLI models** creates a verification system that is both transparent and generalizable — two properties that are typically in tension.

POTENTIAL QUESTIONS FROM THE PANEL

Q: "Why not fine-tune DeBERTa on a fact-checking dataset?"

A: Fine-tuning on LIAR or FakeNewsNet would boost benchmark accuracy on those distributions but reduce generalization. The claim domains shift rapidly — a model trained on political misinformation from 2020 won't catch health misinformation in 2025. The pre-trained NLI model provides domain-agnostic *logical* reasoning: it understands entailment and contradiction regardless of topic. Additionally, fine-tuning risks making the model confidently wrong on out-of-distribution topics, while the NLI model naturally defaults to "neutral" when uncertain — a much safer failure mode.

Q: "How do you handle adversarial inputs?"

A: The system has several defense layers — Pydantic input validation (claim length, URL format, max_results bounds), rate limiting, and the similarity threshold acts as a natural filter against irrelevant noise. However, adversarial *semantic* attacks (e.g., carefully crafted claims designed to exploit the NLI model) are a known open problem in the NLI literature, and this system does not claim to be adversarially robust.

Q: "What about multilingual claims?"

A: Currently English-only. spaCy, DeBERTa-v3-small, and MiniLM are all English-optimized. Multilingual extension would require swapping to multilingual variants (mDeBERTa, multilingual-MiniLM) and multilingual search APIs. The architecture supports this — it would be model swaps in the registry, not architectural changes.

Q: "How does your source credibility scoring compare to ClaimBuster or Google's Fact Check Tools API?"

A: My source scoring is simpler — a static domain lookup table rather than a dynamic credibility assessment. ClaimBuster uses supervised learning to rank check-worthiness; Google's API indexes existing fact-checks. VeriFact's strength is that it doesn't depend on prior fact-checks existing — it verifies against primary sources. The source scorer is the weakest module and would benefit from integration with external credibility APIs.

Q: "Can the LLM tiebreaker be manipulated?"

A: Yes, LLMs are susceptible to prompt injection via evidence text. This is mitigated by the fact that the LLM is never the primary decision-maker — it only activates when the NLI pipeline produces a MIXED score, and its verdict is only accepted above 0.60 confidence. The NLI pipeline is the ground truth; the LLM is an auxiliary signal.

Q: "What's the theoretical upper bound on your system's accuracy?"

A: The accuracy is bounded by three factors: (1) search coverage — claims about events not indexed by search engines will return UNVERIFIED, (2) scraping success rate — paywalled/Cloudflare-protected sites reduce evidence pool, (3) NLI model accuracy — DeBERTa-v3-small has 88% MNLI accuracy, which means roughly 12% of premise-hypothesis pairs will be misclassified at the individual level, though aggregation across multiple sources mitigates this.

Q: "Why cosine similarity for semantic matching? Why not a learned retriever?"

A: Cosine similarity over dense SBERT embeddings is computationally efficient and performs well for sentence-level STS tasks — MiniLM-L6-v2 achieves ~78% Spearman correlation on STS benchmarks. A learned retriever (like ColBERT or DPR) would improve retrieval quality but adds training complexity and another model to manage. Given the memory constraints of a 4 GB EC2 instance, SBERT was the optimal choice.

Q: "How do you prevent echo chamber effects in evidence retrieval?"

A: Two mechanisms: (1) The query generator produces queries designed to *disprove* the claim (e.g., "X hoax," "X false") alongside queries that could confirm it. (2) The source credibility weighting amplifies diverse authoritative sources and dampens social media echo chambers. However, if all top search results happen to come from the same editorial perspective, the system cannot correct for that — it has no way to measure editorial diversity.

[End of Script — Total Duration: ~30 minutes]