The Agentic Historian — Technical Design Document

Generated on 2025-12-05

The Agentic Historian

An Agentic, Evidence-Grounded Historical Fact Verification Chat Application (LangGraph + Google Search API + Fine-Tuned LoRA LLM)

Course: Deep Learning & Applied LLMs

Due: 8 days

Repo Codename: `agentic-historian`

1. Project Overview

The Agentic Historian is an AI-powered chat application that helps users **verify historical facts** by combining:

1. **Live evidence retrieval** via the **Google Custom Search JSON API** (no scraping; we only use the official API responses).
2. An **agentic workflow** orchestrated with **LangGraph** to enforce a structured, auditable reasoning pipeline.
3. A **custom fine-tuned LLM** (Mistral/Llama family) using **PEFT/LoRA** trained on **TruthfulQA**-style supervision to improve truthfulness/refusal behavior (and reduce unsupported claims).
4. A **Streamlit** UI that presents: the final answer, the evidence used, source citations, and model uncertainty.

Core product promise:

> Answers must be **evidence-grounded**, **citation-backed**, and **explicit about uncertainty** when evidence is insufficient.

2. System Architecture

2.1 High-Level Flow

We implement a **deterministic agent pipeline** using LangGraph:

```
Router → Researcher (Google Tool) → Fact Analyst → Writer (Fine-Tuned Model)
```

- **Router:** classifies the user input and selects the appropriate path (historical fact-check vs. out-of-domain vs. ambiguous).
- **Researcher:** issues a Google Search API call and returns structured evidence candidates.
- **Fact Analyst:** evaluates source credibility signals, extracts candidate claims, detects contradictions, and builds a compact, citation-indexed “Evidence Bundle”.

- **Writer:** a fine-tuned LoRA model generates the final answer strictly from the Evidence Bundle and must cite sources.

2.2 Architecture Diagram (Logical)

```
■■■■■■■■■■■■■■■■■■■■■■
■   Streamlit    ■  User chat UI + evidence viewer
■■■■■■■■■■■■■■■■■■■■■■
        ■ user_query
        ▼
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ Router (LangGraph)   ■  intent + route decision
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        ■ if historical verification needed
        ▼
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ Researcher Agent (Tool Node) ■  Google Custom Search JSON API
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        ■ SearchResults[]
        ▼
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ Fact Analyst (Verifier Node) ■  scoring, claim extraction, contradictions
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        ■ EvidenceBundle
        ▼
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■ Writer (Fine-Tuned LoRA LLM) ■  evidence-grounded response + citations
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
        ■ FinalAnswer + Citations + Confidence
        ▼
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■   Streamlit UI      ■  show answer + sources + uncertainty
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

2.3 LangGraph as the Orchestration Backbone

We use LangGraph because it models agent workflows as graphs with explicit **state**, **nodes**, and **edges**, supporting durable execution and structured routing.

Critically, LangGraph state can be defined as a **Pydantic model**, making the workflow type-safe and debuggable end-to-end.

3. Detailed Component Design

3.1 Shared State (Pydantic)

We keep the entire run auditable using a typed Pydantic state, e.g.:

- `user_query: str`
- `route: Literal["fact_check","clarify","out_of_scope"]`
- `search_queries: list[SearchQuery]`
- `search_results: list[SearchResult]`
- `evidence_bundle: EvidenceBundle`
- `final_answer: str`

- `citations: list[Citation]`
- `confidence: float`
- `run_metadata: dict` (latency, token usage, API quota info)

Design rule: every node must be a pure transformation over state:

  input state → output state delta

3.2 Router Node

Goal: decide if we run the full fact-check pipeline.

Heuristics (fast + reliable):

- If query is not historical (“write me a poem”), route to `out_of_scope`.
- If query is underspecified (“Did it happen?” with no subject), route to `clarify`.
- Else route to `fact_check`.

Optional: use a lightweight classifier prompt (small base model) but we keep it deterministic when possible.

3.3 Researcher Agent (Google Tool Node)

Tool: Google **Custom Search JSON API** (Programmable Search Engine).

Constraints: No scraping. We only use:

- title
- snippet
- display link / URL
- optional metadata returned by API

Query strategy:

- Expand user query into 1–3 tightly scoped search queries:
- include dates, names, and disambiguators
- optionally restrict to trusted domains (`site:.edu`, `site:.gov`, museums, encyclopedias) depending on claim type

Output: `SearchResult[]` normalized to a strict schema.

Caching: store results keyed by `(query, date_bucket)` to reduce cost and improve demo reliability.

3.4 Fact Analyst Node (Verification & Evidence Synthesis)

Goal: convert noisy search snippets into a structured evidence pack suitable for a constrained writer.

Steps:

1. **Source scoring** (heuristic):

- Domain type signals (e.g., `.gov`, `.edu`, major encyclopedias, museums)
- Result consistency across multiple sources
- Recency relevance (for “as of” queries)

2. **Claim extraction**:

- Identify atomic claims (date, person, event, location).

3. **Contradiction detection**:

- If multiple high-ranked sources disagree, mark as “contested”.

4. **Evidence bundle construction**:

- Build a compact, numbered set of evidence items:
- `E1: [source title] — snippet — URL`
- `E2: ...`

Output: `EvidenceBundle` containing:

- `evidence_items: list[EvidenceItem]`
- `findings: list[Finding]` (each finding references evidence IDs)
- `verdict: Literal["supported","not_supported","contested","insufficient"]`

3.5 Writer Node (Fine-Tuned LoRA Model)

Goal: produce a final response **only from the Evidence Bundle**.

Writer constraints:

- Must cite evidence by ID (`[E1]`, `[E2]`).
- Must explicitly state uncertainty:
- “Evidence is insufficient to confirm…”
- “Sources disagree…”
- Must refuse to invent missing facts.

Prompt design: a strict instruction template + evidence pack + formatting policy:

- Output format:
- **Answer**
- **Confidence** (0–1)
- **Evidence** (IDs used)
- **Notes / Limitations**

Model: base LLM (Mistral/Llama) + LoRA adapter trained with TruthfulQA-style supervision to reinforce truthful/refusal behavior.

4. Tech Stack Rationale

4.1 `uv` *for Project & Dependency Management*

We use `uv` because it provides:

- fast dependency resolution and installation
- a **project workflow** with `uv init`, `uv add`, and automatic lock/sync mechanics
- a universal lockfile and reproducible environments suited for team projects under deadline.

Key behaviors:

- `uv init` scaffolds the project.
- Locking/syncing can be automatic (e.g., `uv run` can ensure environment is synced before execution).

4.2 LangGraph for Agentic Workflow Orchestration

We chose LangGraph because:

- It models the application as a stateful graph with explicit nodes and edges
- It supports common workflow vs. agent patterns and structured routing
- It supports state definitions via Pydantic, improving correctness in multi-agent pipelines

4.3 Pydantic for Schemas & Validation

Pydantic ensures:

- strict typing and validation for:
- search results
- evidence bundles
- outputs shown in the UI
- fewer hidden runtime failures (critical for demos and grading)

4.4 PyTorch + Hugging Face + PEFT/LoRA for Fine-Tuning

We fine-tune a base model with LoRA to:

- efficiently adapt behavior with limited compute
- produce a Writer model that prefers “I don’t know” over hallucination (to be measured in evaluation)

4.5 Streamlit for UI

Streamlit gives:

- fast iteration for a polished demo
- simple UX for showing:
- chat transcript
- “evidence panel”
- run metadata (latency, caching hits, #sources)

5. Repository Structure (Standard, Complete)

```
agentic-historian/
■■ README.md
■■ LICENSE
■■ pyproject.toml
■■ uv.lock
■■ .python-version
■■ .gitignore
■■ .env.example
■■ ruff.toml
■■ mypy.ini
■■ pytest.ini
■■ docs/
■  ■■ technical_design.md
■  ■■ evaluation_plan.md
■  ■■ demo_script.md
■■ data/
■  ■■ raw/
■  ■■ processed/
■  ■■ README.md
■■ models/
■  ■■ base/
■  ■■ lora_adapters/
■■ notebooks/
■  ■■ 01_truthfulqa_eda.ipynb
■  ■■ 02_lora_sft_experiments.ipynb
■  ■■ 03_eval_analysis.ipynb
■■ scripts/
■  ■■ run_app.sh
■  ■■ run_graph_cli.sh
■  ■■ train_lora.sh
■  ■■ eval.sh
■■ src/
■  ■■ agentic_historian/
■     ■■ __init__.py
■     ■■ config.py
■     ■■ app/
■     ■  ■■ streamlit_app.py
■     ■■ graph/
■     ■  ■■ state.py
■     ■  ■■ graph.py
■     ■  ■■ nodes/
■     ■     ■■ router.py
■     ■     ■■ researcher.py
■     ■     ■■ fact_analyst.py
■     ■     ■■ writer.py
■     ■■ tools/
■     ■  ■■ google_search.py
■     ■■ llm/
■     ■  ■■ prompts.py
■     ■  ■■ loader.py
■     ■  ■■ generation.py
■     ■■ training/
■     ■  ■■ data_prep.py
■     ■  ■■ train_lora.py
■     ■  ■■ eval_truthfulqa.py
■     ■■ utils/
■     ■  ■■ logging.py
■     ■  ■■ cache.py
■     ■  ■■ text.py
■     ■■ types/
■        ■■ schemas.py
■■ tests/
■  ■■ test_router.py
■  ■■ test_google_tool.py
■  ■■ test_fact_analyst.py
■  ■■ test_writer_constraints.py
■  ■■ test_graph_smoke.py
```

```
■■ .github/
   ■■ workflows/
      ■■ ci.yml
```

6. MLOps & Evaluation Plan (Minimum Viable, Grade-Friendly)

6.1 Offline Evaluation (Model Behavior)

- Use TruthfulQA validation split to evaluate:
- truthfulness-aligned answer preference
- refusal correctness for unanswerable questions
- Metrics:
- exact/substring match for known answers (when applicable)
- “refusal rate” on unanswerable prompts
- qualitative rubric: “evidence grounded / not grounded”

6.2 Online Evaluation (End-to-End App)

- Create 20–30 historical fact queries:
- famous events/dates
- contested claims
- ambiguous questions
- Score:
- evidence quality
- citation correctness (IDs map to displayed sources)
- uncertainty handling in contested/insufficient cases

7. Risk Register & Mitigations

1. **No-scraping limitation reduces depth**
   - Mitigation: explicit “Evidence Limitation” note; rely on multiple sources; domain-restricted search.
2. **Search results may conflict**
   - Mitigation: Fact Analyst labels “contested”; Writer must present both sides with citations.
3. **Hallucinations despite fine-tuning**
   - Mitigation: strict Writer prompt + schema validation + “must cite evidence IDs” check.
4. **Latency / quota limits**
   - Mitigation: caching, query minimization, and demo-safe fallback examples.

8. Run Instructions (Target UX)

- Install dependencies: `uv sync`
- Run app: `uv run streamlit run src/agentic_historian/app/streamlit_app.py`
- Train LoRA: `uv run python -m agentic_historian.training.train_lora ...`
- Evaluate: `uv run python -m agentic_historian.training.eval_truthfulqa ...`