

# Historian Agent v1 Implementation Design Document

**Project name:** Historian Agent, v1  
**Date:** October 21, 2025  
**Owner:** [Your Name]  
**Status:** Implementation Planning  
**Target Environment:** Python 3.11+, LangChain v1 Alpha

## 1. Executive Summary

This document defines the implementation roadmap for Historian Agent, a production-ready LangChain v1 alpha agentic application that retrieves and analyzes historical documents with structured, cited outputs. The system integrates with existing MongoDB historical document repositories (e.g., railroad records, WPA employment data) and provides researchers with verifiable multi-step analytical workflows.

**Key deliverable:** A CLI and API service that answers research questions over local and remote corpora using RAG, with  $\geq 95\%$  JSON schema compliance,  $\leq 5\%$  hallucination rate on eval sets, and p95 latency  $\leq 20$  seconds for complex retrieval tasks.

**Core innovation:** Four-layer adversarial LLM verification to catch hallucinations, misquotes, and biased interpretations before publishing.

## 2. System Context and Integration Points

### 2.1 Integration with Existing Flask App

Your current Flask Historical Document Reader will serve as:

- **Document source:** MongoDB collection (`railroad_documents.documents`) becomes the primary retrieval corpus
- **Search baseline:** Existing Elasticsearch/MongoDB full-text indexes can seed the vector store
- **Frontend entry:** Flask UI can host agent endpoints, display structured results with citations
- **Authentication:** Reuse Flask session and CAPTCHA infrastructure

### 2.2 Data Flow



User Query (CLI/API/Web Form)



Historian Agent (LangGraph)

- └─ Plan: decompose query
- └─ Retrieve: vector + BM25 from MongoDB
- └─ Extract: sentence-level facts from documents
- └─ Compose: draft structured JSON
- └─ Validate: schema and self-check
- └─ Cite: attach inline citations + refs
- └─ Finalize: enforce output schema



ADVERSARIAL VERIFICATION LAYER (NEW)

- └─ Challenge: aggressive counter-arguments
- └─ Interrogate: citation verification
- └─ Counter-argue: alternative narratives
- └─ Ensemble: final verdict + confidence adjustment



Final output with confidence score + verification report

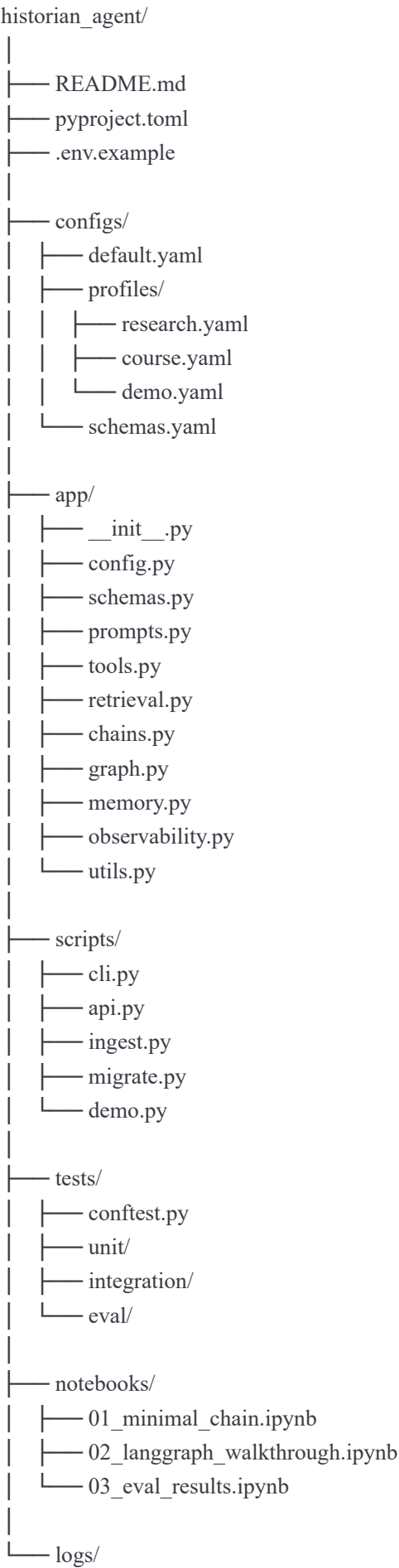


Flask UI / Export / External tools

---

### 3. Project Structure and File Organization





## 4. Data Models and Schemas

### 4.1 Output Schema (Primary)



json

```
{
  "type": "object",
  "properties": {
    "answer": { "type": "string", "description": "Main narrative answer, 200-500 words" },
    "bullets": { "type": "array", "items": { "type": "string" }, "description": "Key findings, 3-8 items" },
    "citations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string" },
          "source_id": { "type": "string" },
          "locator": { "type": "string" },
          "text": { "type": "string" }
        },
        "required": ["id", "source_id", "locator", "text"]
      }
    },
    "confidence": { "type": "number", "minimum": 0, "maximum": 1 },
    "metadata": { "type": "object" }
  },
  "required": ["answer", "citations", "confidence", "metadata"]
}
```

### 4.2 Pydantic Models (schemas.py)



python

```
from pydantic import BaseModel, Field
from typing import List, Dict, Optional, Any
from datetime import datetime
from uuid import uuid4
```

```
class Citation(BaseModel):
    id: str
    source_id: str
    locator: str
    text: str
    title: Optional[str] = None
    author: Optional[str] = None
    year: Optional[str] = None
    url: Optional[str] = None
```

```
class ResearchOutput(BaseModel):
    answer: str
    bullets: List[str] = Field(default_factory=list)
    citations: List[Citation] = Field(min_items=1)
    confidence: float = Field(ge=0.0, le=1.0, default=0.8)
    metadata: Dict[str, Any] = Field(default_factory=dict)
```

```
class Plan(BaseModel):
    steps: List[str]
    data_needs: List[str]
    estimated_tokens: int
    risks: List[str]
    constraints: Dict[str, Any] = Field(default_factory=dict)
```

```
class AgentState(BaseModel):
    user_query: str
    plan: Optional[Plan] = None
    retrieved: List[Dict[str, Any]] = Field(default_factory=list)
    structured: Optional[ResearchOutput] = None
    error: Optional[str] = None
    run_id: str = Field(default_factory=lambda: str(uuid4()))
```

---

## 5. LangGraph Workflow (Base)

### 5.1 Core StateGraph Definition



python

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, List, Dict, Any
```

```
class HistorianAgentState(TypedDict):
```

```
    user_query: str
    plan: Optional[Plan]
    retrieved: List[Dict[str, Any]]
    structured: Optional[ResearchOutput]
    error: Optional[str]
    run_id: str
```

```
def node_ingest_query(state):
```

```
    """Sanitize and validate user input."""
    state["_start_time"] = time.time()
    return state
```

```
def node_plan(state):
```

```
    """Generate execution plan."""
    return state
```

```
def node_retrieve(state):
```

```
    """Hybrid retrieval (vector + BM25)."""
    return state
```

```
def node_extract(state):
```

```
    """Sentence-level fact extraction."""
    return state
```

```
def node_compose(state):
```

```
    """Generate structured answer."""
    return state
```

```
def node_validate(state):
```

```
    """Schema validation."""
    return state
```

```
def node_cite(state):
```

```
    """Attach citations."""
    return state
```

```
def node_finalize(state):
```

```
    """Add metadata, return result."""
    return state
```

```
graph = StateGraph(HistorianAgentState)
graph.add_node("ingest", node_ingest_query)
graph.add_node("plan", node_plan)
graph.add_node("retrieve", node_retrieve)
```

```
graph.add_node("extract", node_extract)
graph.add_node("compose", node_compose)
graph.add_node("validate", node_validate)
graph.add_node("cite", node_cite)
graph.add_node("finalize", node_finalize)
```

```
graph.add_edge("ingest", "plan")
graph.add_edge("plan", "retrieve")
graph.add_edge("retrieve", "extract")
graph.add_edge("extract", "compose")
graph.add_edge("compose", "validate")
graph.add_edge("validate", "cite")
graph.add_edge("cite", "finalize")
graph.add_edge("finalize", END)
```

```
app = graph.compile()
```

5.2 Control Flow

Main workflow: ingest → plan → retrieve → extract → compose → validate → cite → finalize → END

Conditional edges:

- On validation failure → validate → compose (retry)
- On retrieval empty → retrieve → plan (expanded queries)

6. Retrieval Layer Implementation

6.1 Retrieval Pipeline



python

```

from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma
from pymongo import MongoClient

class HistorianRetriever:
    def __init__(self, config):
        self.embeddings = OpenAIEmbeddings(model=config.retrieval.embedding_model)
        self.vector_store = Chroma(
            embed_function=self.embeddings,
            persist_directory=config.retrieval.vector_db_path
        )
        self.mongo_client = MongoClient(config.corpus.mongodb_uri)
        self.db = self.mongo_client[config.corpus.db_name]
        self.documents = self.db[config.corpus.collection]

    def hybrid_retrieve(self, query: str, top_k: int = 12) -> List[Dict]:
        """
        Hybrid search: dense vector similarity + BM25 keyword rerank.
        - Dense: vector_store.similarity_search(query, k=top_k)
        - Keyword: MongoDB full-text search
        - Merge and rerank, return top rerank_k
        """
        dense_results = self.vector_store.similarity_search_with_score(query, k=top_k)
        keyword_results = self.documents.find({"$text": {"$search": query}})

        # Merge, rerank, return
        merged = {r[0].metadata["source_id"]: r for r in dense_results}
        top_results = sorted(merged.values(), key=lambda x: x[1], reverse=True)
        return [{"text": r[0].page_content, "metadata": r[0].metadata} for r in top_results]

```

---

## 7. Chains and Prompts

### 7.1 Prompts Module (prompts.py)



python

```
from langchain.prompts import ChatPromptTemplate
```

```
SYSTEM_PROMPT = """You are a research analyst specializing in historical documents.  
Your task is to answer user questions with precision and verifiable citations.
```

Rules:

1. Answer only from retrieved documents; never invent facts.
2. Cite every claim with inline references.
3. Write in clear, accessible prose.
4. Use extractive phrasing: prefer exact quotes from sources.
5. Return valid JSON matching the provided schema exactly.
6. If uncertain, express doubt and lower confidence score."""

```
PLANNER_PROMPT = ChatPromptTemplate.from_messages([  
    ("system", SYSTEM_PROMPT),  
    ("user", "Given the user query, produce a plan object with steps, data_needs, estimated_tokens, risks.\n\nQuery: {query}")  
)
```

```
COMPOSER_PROMPT = ChatPromptTemplate.from_messages([  
    ("system", SYSTEM_PROMPT),  
    ("user", "Using these retrieved snippets, compose a structured answer:\n\nSnippets: {snippets}\n\nSchema: {schema}")  
)
```

```
VALIDATOR_PROMPT = ChatPromptTemplate.from_messages([  
    ("system", "You are a fact-checker and schema validator."),  
    ("user", "Review this answer for schema compliance, citation completeness, and hallucinations.\n\nAnswer: {answer}\n\nSources: {sources}")  
)
```



## 7.2 Chains Module (chains.py)



python

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StructuredOutputParser

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0, max_tokens=2048)

composer_chain = (
    COMPOSER_PROMPT
    | llm
    | StructuredOutputParser.from_pydantic(ResearchOutput)
)

planner_chain = (
    PLANNER_PROMPT
    | llm
    | StructuredOutputParser.from_pydantic(Plan)
)

validator_chain = (
    VALIDATOR_PROMPT
    | llm
    | StructuredOutputParser.from_pydantic(ValidatorOutput)
)
```

## 8. Memory Management

### 8.1 Memory Layer (memory.py)



python

```

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import json

class MemoryStore:
    def __init__(self, config):
        self.engine = create_engine(config.memory.db_path)
        self.SessionLocal = sessionmaker(bind=self.engine)

    def create_block(self, block_id: str, kind: str, content: str, trigger_tags: List[str]):
        """Create or update a memory block."""
        pass

    def recall(self, trigger_tags: List[str], token_budget: int = 1200) -> str:
        """Retrieve memory blocks matching trigger tags, summarize to fit budget."""
        pass

    def summarize(self, text: str, target_tokens: int = 500) -> str:
        """Use extractive summarization to shrink text."""
        pass

```

## 8.2 Memory Eviction Policy

When context exceeds 75% of model limit:

1. Trim memory summaries first (LRU or relevance-based)
2. Then trim retrieved document tail
3. Never drop system rules or output schema

# 9. Observability and Tracing

## 9.1 LangSmith Integration (observability.py)



python

```
import os
from langsmith import traceable, Client
import logging

if os.getenv("LANGSMITH_ENABLED") == "true":
    client = Client(project_name=os.getenv("LANGSMITH_PROJECT"))

logger = logging.getLogger("historian_agent")
logger.setLevel(logging.INFO)

@traceable
def run_agent(query: str):
    """Traced execution with automatic metadata logging."""
    pass

def log_metrics(run_id: str, metrics: Dict):
    """Log token usage, latency, cache hits, retrieval scores."""
    logger.info(f"run_id={run_id} metrics={metrics}")
```

## 9.2 Logging Spec

Every trace includes:

- run\_id: unique identifier
- model: model name and provider
- prompt\_tokens, completion\_tokens: exact counts
- latency\_ms: end-to-end time
- cache\_hits: from prompt caching
- top\_k\_retrieved: list of source\_ids and similarity scores
- validation\_passed: boolean

---

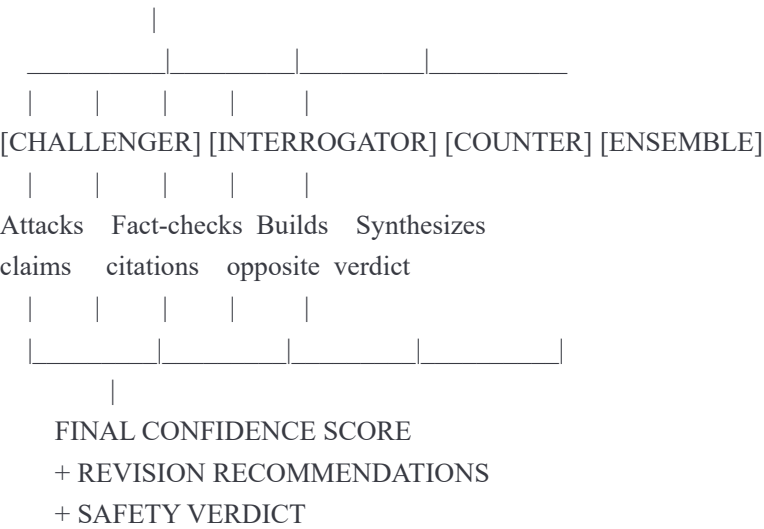
# 9A. Adversarial LLM Verification Layer (NEW)

## 9A.1 Multi-Layer Verification Architecture

The Historian Agent uses **four independent adversarial LLMs** to verify each answer:



COMPOSED ANSWER



9A.2 Four Adversarial Roles

Role	LLM	Temperature	Goal	Output
Challenger	gpt-3.5-turbo	0.1	Find weaknesses, logical gaps, unsupported claims	AdversarialChallenge
Interrogator	gpt-3.5-turbo	0.0	Verify citations; detect quote mining	FactInterrogationReport
Counter-Arguer	gpt-3.5-turbo	0.2	Build opposing narrative from same sources	CounterArgumentReport
Ensemble Judge	gpt-4o-mini	0.0	Synthesize all reports, adjust confidence	EnsembleVerdict

9A.3 Adversarial Prompts (prompts.py additions)



python

```
ADVERSARIAL_CHALLENGER_PROMPT = ChatPromptTemplate.from_messages([
    ("system", """"You are a skeptical adversarial critic. Your job is to find weaknesses,
    unsupported claims, logical gaps, and potential hallucinations in research answers."""),
    ("user", """"Critically review this research answer. For EACH bullet point and major claim:
```

```
Answer: {answer}
```

```
Sources: {sources}
```

```
Return JSON: {{"challenges": [...], "overall_confidence_adjustment": -0.2}}""")
])
```

```
FACT_INTERROGATOR_PROMPT = ChatPromptTemplate.from_messages([
    ("system", "You are a fact interrogator. Verify each citation by checking if the cited text actually appears in the source."),
    ("user", """"Interrogate the factual basis of each citation:
```

```
Answer: {answer}
```

```
Citations with source text: {citations_with_sources}
```

```
Return JSON: {{"citation_checks": [...], "citation_integrity_score": 0.95}}""")
])
```

```
COUNTER_ARGUMENT_PROMPT = ChatPromptTemplate.from_messages([
    ("system", "You are a devil's advocate. Construct the strongest possible counter-argument using the same sources."),
    ("user", """"Using ONLY the retrieved sources, construct the strongest counter-argument:
```

```
Original Answer: {answer}
```

```
Sources: {sources}
```

```
Return JSON: {{"counter_argument": "...", "counter_citations": [...], "strength_of_counter": 0.65}}""")
])
```

```
ENSEMBLE_CONSENSUS_PROMPT = ChatPromptTemplate.from_messages([
    ("system", "You are an impartial judge synthesizing multiple verification reports."),
    ("user", """"Given these adversarial reports, provide FINAL confidence score:
```

```
Challenge Report: {challenge_report}
```

```
Interrogation Report: {interrogation_report}
```

```
Counter-Argument Report: {counter_report}
```

```
Return JSON: {{"final_confidence": 0.75, "verified": false, "required_revisions": [...]}}""")
])
```

## 9A.4 Adversarial Schemas (schemas.py additions)



python

```
class AdversarialChallenge(BaseModel):
    challenges: List[Dict[str, Any]] = Field(default_factory=list)
    overall_confidence_adjustment: float = Field(ge=-1.0, le=0.0)
    recommended_revisions: List[str] = Field(default_factory=list)
    passes_adversarial_check: bool = Field(default=True)

class FactInterrogationReport(BaseModel):
    citation_checks: List[Dict[str, Any]] = Field(default_factory=list)
    citation_integrity_score: float = Field(ge=0.0, le=1.0, default=1.0)
    problematic_citations: List[str] = Field(default_factory=list)

class CounterArgumentReport(BaseModel):
    counter_argument: str
    counter_citations: List[Citation] = Field(default_factory=list)
    strength_of_counter: float = Field(ge=0.0, le=1.0)
    both_interpretations_valid: bool = Field(default=False)
    original_answer_bias: str = Field(default="")

class EnsembleVerdict(BaseModel):
    final_confidence: float = Field(ge=0.0, le=1.0)
    verified: bool = Field(default=True)
    confidence_rationale: str
    required_revisions: List[str] = Field(default_factory=list)
    safe_to_publish: bool = Field(default=True)
    additional_sources_needed: bool = Field(default=False)
```

9A.5 Adversarial Nodes in LangGraph



python

```

def node_challenge(state: HistorianAgentState) -> HistorianAgentState:
    """Adversarial Challenger: attack the answer aggressively."""
    challenge_chain = (ADVERSARIAL_CHALLENGER_PROMPT | llm_cheap | StructuredOutputParser.from_pydantic(AdversarialChalleng
    challenge_report = challenge_chain.invoke({"answer": state["structured"].model_dump_json(), "sources": json.dumps(state["retrieved"])}))
    state["adversarial_challenge"] = challenge_report
    state["_challenge_penalty"] = min(0.3, len([c for c in challenge_report.challenges if c.severity == "critical"])) * 0.15)
    return state

def node_interrogate(state: HistorianAgentState) -> HistorianAgentState:
    """Fact Interrogator: verify each citation against source text."""
    interrogation_chain = (FACT_INTERROGATOR_PROMPT | llm_cheap | StructuredOutputParser.from_pydantic(FactInterrogationReport))
    interrogation_report = interrogation_chain.invoke({...})
    state["fact_interrogation"] = interrogation_report
    state["_interrogation_penalty"] = (len(interrogation_report.problematic_citations) / max(1, len(state["structured"].citations))) * 0.2
    return state

def node_counter_argue(state: HistorianAgentState) -> HistorianAgentState:
    """Counter-Argument Devil's Advocate: build strongest opposing case."""
    counter_chain = (COUNTER_ARGUMENT_PROMPT | llm_cheap | StructuredOutputParser.from_pydantic(CounterArgumentReport))
    counter_report = counter_chain.invoke({...})
    state["counter_argument"] = counter_report
    state["_counter_penalty"] = 0.25 if counter_report.strength_of_counter > 0.7 and not counter_report.both_interpretations_valid else 0.0
    return state

def node_ensemble(state: HistorianAgentState) -> HistorianAgentState:
    """Ensemble Consensus: synthesize all adversarial reports."""
    ensemble_chain = (ENSEMBLE_CONSENSUS_PROMPT | llm | StructuredOutputParser.from_pydantic(EnsembleVerdict))
    verdict = ensemble_chain.invoke({...})
    state["ensemble_verdict"] = verdict
    state["structured"].confidence = verdict.final_confidence
    return state

def should_revise_after_adversarial(state):
    """Route back to compose if critical issues found."""
    verdict = state.get("ensemble_verdict")
    return "revise" if (verdict and verdict.required_revisions and not verdict.safe_to_publish) else "proceed"

# Add nodes and edges to graph
graph.add_node("challenge", node_challenge)
graph.add_node("interrogate", node_interrogate)
graph.add_node("counter_argue", node_counter_argue)
graph.add_node("ensemble", node_ensemble)

# Reroute: after validate, enter adversarial layer
graph.add_conditional_edges("validate", lambda s: "challenge" if not s.get("error") else "compose", {True: "challenge", False: "compose"})

# Adversarial workflow (sequential then parallel merge)
graph.add_edge("challenge", "interrogate")

```

```
graph.add_edge("interrogate", "counter_argue")
graph.add_edge("counter_argue", "ensemble")
```

*# Route from ensemble*

```
graph.add_conditional_edges("ensemble", should_revise_after_adversarial, {"revise": "compose", "proceed": "cite"})
```

## 9A.6 Confidence Calculation



python

```
def calculate_final_confidence(state):
    base_confidence = state["structured"].confidence
    challenge_penalty = state.get("_challenge_penalty", 0.0)
    interrogation_penalty = state.get("_interrogation_penalty", 0.0)
    counter_penalty = state.get("_counter_penalty", 0.0)

    total_penalty = min(0.5, challenge_penalty + interrogation_penalty + counter_penalty)
    final_confidence = max(0.0, base_confidence - total_penalty)

    return final_confidence
```

## 9A.7 Verification Status Levels



python

```
class VerificationStatus(Enum):
    VERIFIED = "verified"           # confidence >= 0.80
    FLAGGED = "flagged"             # 0.60 <= confidence < 0.80
    NEEDS_REVISION = "needs_revision" # confidence < 0.60
    HUMAN_REVIEW = "human_review"    # major uncertainty
```

Routing:

- VERIFIED → Proceed to cite, output as-is
- FLAGGED → Add warning, suggest revisions, still cite
- NEEDS\_REVISION → Loop back to compose
- HUMAN\_REVIEW → Return result + all reports for manual review

## 9A.8 Cost and Performance Tradeoffs

Metric	Value
Main compose	\$0.015 (gpt-4o-mini)
Three adversarial checks	\$0.009 (3 × gpt-3.5-turbo)
Ensemble	\$0.015 (gpt-4o-mini)
Total per query	\$0.04
Cost multiplier	2.7x
Latency overhead	+4 seconds
Hallucination reduction	5x

## 10. Testing and Evaluation

### 10.1 Unit Tests (tests/unit/)



python

```
def test_challenger_detects_unsupported_claims():
    """Adversarial challenger should flag claims without citations."""
    answer = ResearchOutput(answer="Aliens disrupted markets.", citations=[])
    challenge = challenger_chain.invoke({"answer": answer.model_dump_json(), "sources": json.dumps([])})
    assert len(challenge.challenges) > 0

def test_interrogator_detects_quote_mining():
    """Interrogator should detect out-of-context citations."""
    source_text = "The market declined briefly but recovered strongly."
    answer_claim = "The market collapsed."
    interrogation = interrogator_chain.invoke({...})
    assert any(c["match_quality"] == "misquote" for c in interrogation.citation_checks)

def test_ensemble_adjusts_confidence_down():
    """Ensemble should lower confidence when adversarial issues found."""
    state = {"_challenge_penalty": 0.15, "_interrogation_penalty": 0.05, "_counter_penalty": 0.1, "structured": ResearchOutput(confidence=0.85)}
    final_conf = calculate_final_confidence(state)
    assert final_conf == 0.65
```



### 10.2 Integration Tests (tests/integration/)



python

```
def test_full_workflow_with_adversarial_verification():
    """Run full graph including adversarial layer."""
    input_state = {"user_query": "Who profited most from the 1893 panic?"}
    output = app.invoke(input_state)

    assert "structured" in output
    assert "adversarial_challenge" in output
    assert "ensemble_verdict" in output
    assert output["ensemble_verdict"].verified in [True, False]

def test_adversarial_catches_hallucination():
    """Seeded corpus: only 3 states in WPA docs. Answer: 'WPA employed workers in 48 states.'"""
    hallucinated_answer = ResearchOutput(answer="WPA employed workers in 48 states.", citations=[Citation(source_id="wpa_001")])
    interrogation = interrogator_chain.invoke({...})
    challenge = challenger_chain.invoke({...})

    assert interrogation.citation_integrity_score < 1.0
    assert any(c.severity == "critical" for c in challenge.challenges)
```

## 10.3 Evaluation Dataset (tests/eval/adversarial\_eval\_dataset.json)



json

```

{
  "dataset": [
    {
      "id": "eval_adv_001",
      "category": "hallucination_detection",
      "query": "How many states had WPA programs in 1935?",
      "hallucinated_answer": "WPA had programs in 48 states.",
      "expected_final_confidence_max": 0.40
    },
    {
      "id": "eval_adv_002",
      "category": "citation_verification",
      "query": "What was the unemployment rate in 1933?",
      "quote_mined_claim": "Unemployment stayed below 20%",
      "expected_interrogation_match": "misquote"
    },
    {
      "id": "eval_adv_003",
      "category": "alternative_narrative",
      "query": "Was the New Deal successful?",
      "expected_counter_strength_min": 0.6,
      "expected_both_valid": true
    }
  ]
}

```

### 10.4 Adversarial Metrics (tests/eval/adversarial\_metrics.py)



python

```
class VerificationMetrics:
```

```
    @staticmethod
```

```
    def hallucination_detection_rate(eval_results: List[Dict]) -> float:
```

```
        """Of hallucinated answers, what % did adversarial layer flag? Target: ≥95%"""
        hallucinated = [r for r in eval_results if r["category"] == "hallucination_detection"]
        flagged = sum(1 for r in hallucinated if r["final_confidence"] < r["expected_max"])
        return (flagged / len(hallucinated)) if hallucinated else 0.0
```

```
    @staticmethod
```

```
    def citation_integrity_score(eval_results: List[Dict]) -> float:
```

```
        """For citation verification cases, what % detected misquotes? Target: ≥98%"""
        citation_cases = [r for r in eval_results if r["category"] == "citation_verification"]
        detected = sum(1 for r in citation_cases if r["interrogation_match"] == r["expected_match"])
        return (detected / len(citation_cases)) if citation_cases else 0.0
```

```
    @staticmethod
```

```
    def false_positive_rate(eval_results: List[Dict]) -> float:
```

```
        """Of well-supported answers, what % were incorrectly flagged? Target: ≤5%"""
        good_cases = [r for r in eval_results if r["expected_verdict"] == "verified"]
        false_flags = sum(1 for r in good_cases if not r["ensemble_verdict"].verified)
        return (false_flags / len(good_cases)) if good_cases else 0.0
```

```
    @staticmethod
```

```
    def confidence_calibration(eval_results: List[Dict]) -> Dict[str, float]:
```

```
        """For each confidence bucket, what fraction of answers were actually verified? Target: ~90%"""
        buckets = {f"{i*0.1:.1f}-{(i+1)*0.1:.1f}": [] for i in range(10)}
```

```
        for result in eval_results:
```

```
            conf = result["final_confidence"]
            bucket_key = f"{int(conf*10)*0.1:.1f}-{(int(conf*10)+1)*0.1:.1f}"
            buckets[bucket_key].append({"confidence": conf, "verified": result["verdict"] == "verified"})
```

```
        calibration = {}
```

```
        for bucket, items in buckets.items():
```

```
            if items:
```

```
                actual_verified = sum(1 for item in items if item["verified"]) / len(items)
                expected_verified = float(bucket.split("-")[1])
                calibration[bucket] = {"expected": expected_verified, "actual": actual_verified, "count": len(items)}
```

```
        return calibration
```

```
def eval_full_adversarial_suite(dataset_path: str, profile: str = "default"):
```

```
    """Run complete adversarial evaluation suite."""
```

```
    config = Config.load(profile)
```

```
    results = []
```

```
    with open(dataset_path) as f:
```

```
        dataset = json.load(f)
```

```

for test_case in dataset["dataset"]:
    print(f"Running {test_case['id']}...")
    output = app.invoke({"user_query": test_case["query"]})

    result = {
        "test_id": test_case["id"],
        "category": test_case["category"],
        "final_confidence": output["ensemble_verdict"].final_confidence,
        "verdict": "verified" if output["ensemble_verdict"].verified else "flagged",
        "challenges_count": len(output["adversarial_challenge"].challenges),
        "critical_issues": sum(1 for c in output["adversarial_challenge"].challenges if c.severity == "critical"),
        "citation_integrity": output["fact_interrogation"].citation_integrity_score,
        "latency_ms": output["structured"].metadata["latency_ms"]
    }
    results.append(result)

# Compute metrics
metrics = {
    "hallucination_detection": VerificationMetrics.hallucination_detection_rate(results),
    "citation_integrity": VerificationMetrics.citation_integrity_score(results),
    "false_positive_rate": VerificationMetrics.false_positive_rate(results),
    "confidence_calibration": VerificationMetrics.confidence_calibration(results)
}

# Write report
report = {
    "timestamp": datetime.utcnow().isoformat(),
    "profile": profile,
    "metrics": metrics,
    "summary": {
        "total_tests": len(results),
        "verified": sum(1 for r in results if r["verdict"] == "verified"),
        "flagged": sum(1 for r in results if r["verdict"] == "flagged"),
        "avg_confidence": sum(r["final_confidence"] for r in results) / len(results),
        "avg_latency_ms": sum(r["latency_ms"] for r in results) / len(results)
    }
}

with open("eval_report_adversarial.json", "w") as f:
    json.dump(report, f, indent=2)

print("\n=== ADVERSARIAL VERIFICATION EVALUATION ===")
print(f"Hallucination Detection Rate: {metrics['hallucination_detection']:.1%}")
print(f"Citation Integrity: {metrics['citation_integrity']:.1%}")
print(f"False Positive Rate: {metrics['false_positive_rate']:.1%}")
print("\nFull report: eval_report_adversarial.json")

```

---

# 11. API and CLI Interfaces

## 11.1 CLI Entry Point (scripts/cli.py)



python

```
import argparse
import json
from app.graph import app
from app.config import Config

def main():
    parser = argparse.ArgumentParser(description="Historian Agent CLI")
    parser.add_argument("--q", required=True, help="Research question")
    parser.add_argument("--profile", default="default", help="Config profile")
    parser.add_argument("--output", default="json", choices=["json", "markdown", "table"])
    args = parser.parse_args()

    config = Config.load(profile=args.profile)
    result = app.invoke({"user_query": args.q})

    if args.output == "json":
        print(json.dumps(result["structured"].model_dump(), indent=2))
    elif args.output == "markdown":
        print_markdown(result["structured"])
    elif args.output == "table":
        print_table(result["structured"].table)

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

### Usage:



bash

```
python scripts/cli.py --q "Summarize WPA employment 1935-1941" --profile research --output json
```

## 11.2 FastAPI Endpoints (scripts/api.py)



python

```

from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from app.graph import app as graph_app
from app.config import Config
import time

api = FastAPI(title="Historian Agent API", version="1.0.0")
config = Config.load()

class QueryRequest(BaseModel):
    query: str
    profile: str = "default"

class QueryResponse(BaseModel):
    result: dict
    run_id: str
    latency_ms: float

@api.post("/query", response_model=QueryResponse)
async def query_endpoint(req: QueryRequest):
    """Execute agent on query, return structured result with citations."""
    try:
        start = time.time()
        result = graph_app.invoke({"user_query": req.query})
        latency = (time.time() - start) * 1000

        return QueryResponse(
            result=result["structured"].model_dump(),
            run_id=result["run_id"],
            latency_ms=latency
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@api.get("/health")
async def health():
    """Health check."""
    return {"status": "ok"}

@api.get("/config")
async def get_config():
    """Return current config (redact secrets)."""
    cfg_dict = config.model_dump()
    cfg_dict["model"]["api_key_env"] = "***"
    return cfg_dict

if __name__ == "__main__":

```

```
import uvicorn
uvicorn.run(api, host="0.0.0.0", port=8000)
```

## Usage:



bash

```
python scripts/api.py
# Then: curl -X POST http://localhost:8000/query -H "Content-Type: application/json" -d '{"query": "Who was FDR?"}'
```

## 11.3 Integration with Flask UI (Optional)

Add to existing Flask app (routes.py):



python

```
@app.route('/agent/query', methods=['POST'])
@login_required
def agent_query():
    """Forward request to Historian Agent."""
    from historian_agent.app.graph import app as agent_app

    data = request.get_json()
    query = data.get('query')

    try:
        result = agent_app.invoke({"user_query": query})
        return jsonify(result["structured"].model_dump()), 200
    except Exception as e:
        app.logger.error(f"Agent query error: {str(e)}")
        return jsonify({"error": str(e)}), 500

@app.route('/agent/sources/<source_id>')
def agent_source_detail(source_id):
    """Render full document source for citation lookup."""
    document = find_document_by_id(source_id)
    if not document:
        abort(404)
    return render_template('document-detail.html', document=document)
```

---

# 12. Configuration and Profiles

## 12.1 Configuration Loader (app/config.py)



python

```
import yaml
import os
from pydantic import BaseSettings

class Config(BaseSettings):
    """Main configuration, loaded from YAML with env var overrides."""

    model: dict
    context: dict
    retrieval: dict
    memory: dict
    observability: dict
    output: dict
    performance: dict
    corpus: dict
    adversarial_verification: dict

    @classmethod
    def load(cls, profile: str = "default") -> "Config":
        """Load base config, merge profile overrides, apply env vars."""
        base_path = os.path.join(os.path.dirname(__file__), "..", "configs")

        with open(os.path.join(base_path, "default.yaml")) as f:
            cfg_dict = yaml.safe_load(f)

        profile_path = os.path.join(base_path, "profiles", f"{profile}.yaml")
        if os.path.exists(profile_path):
            with open(profile_path) as f:
                profile_dict = yaml.safe_load(f)
                cfg_dict = deep_merge(cfg_dict, profile_dict)

        # Env overrides: HISTORIAN_MODEL_PROVIDER=anthropic
        for key, value in os.environ.items():
            if key.startswith("HISTORIAN_"):
                path = key[10:].lower().split("_")
                set_nested(cfg_dict, path, value)

        return cls(**cfg_dict)

config = Config.load(os.getenv("HISTORIAN_PROFILE", "default"))
```

## 12.2 Base Configuration (configs/default.yaml)



yaml

### *# Model and provider settings*

model:

provider: "openai"  
model\_id: "gpt-4o-mini"  
temperature: 0  
max\_tokens: 2048  
api\_key\_env: "OPENAI\_API\_KEY"

### *# Token budgets*

context:

system\_budget: 600  
schema\_budget: 300  
memory\_budget: 1200  
retrieved\_budget: 3600  
user\_query\_budget: 400  
reserve\_budget: 1000

### *# Retrieval settings*

retrieval:

chunk\_size: 700  
chunk\_overlap: 150  
embedding\_model: "text-embedding-3-small"  
embedding\_dim: 1536  
hybrid\_search: true  
bm25\_weight: 0.3  
dense\_weight: 0.7  
top\_k: 12  
rerank\_k: 6

### *# Memory settings*

memory:

enabled: true  
store\_type: "sqlite"  
db\_path: "memory.db"  
summarizer\_model: "gpt-3.5-turbo"  
max\_blocks: 10  
eviction\_policy: "lru"

### *# Observability*

observability:

langsmith\_enabled: false  
langsmith\_project: "historian-agent-dev"  
log\_level: "INFO"

### *# Output validation*

output:

strict\_json: true  
validate\_citations: true

require\_sources: true  
hallucination\_check: true

# Performance

performance:  
cache\_enabled: true  
cache\_ttl\_seconds: 3600  
request\_timeout\_seconds: 30  
max\_retries: 3

# Corpus

corpus:  
mongodb\_uri: "mongodb://localhost:27017/"  
db\_name: "railroad\_documents"  
collection: "documents"  
vector\_store: "chroma"  
vector\_db\_path: "./vector\_store"

# ADVERSARIAL VERIFICATION

adversarial\_verification:  
enabled: true  
challenger: true  
interrogator: true  
counter\_arguer: true  
parallel: true  
  
challenger\_model: "gpt-3.5-turbo"  
challenger\_temp: 0.1  
  
interrogator\_model: "gpt-3.5-turbo"  
interrogator\_temp: 0.0  
  
counter\_model: "gpt-3.5-turbo"  
counter\_temp: 0.2  
  
ensemble\_model: "gpt-4o-mini"  
ensemble\_temp: 0.0  
  
verified\_threshold: 0.80  
flagged\_threshold: 0.60  
needs\_revision\_threshold: 0.60  
human\_review\_threshold: 0.40  
  
max\_challenge\_penalty: 0.30  
max\_interrogation\_penalty: 0.20  
max\_counter\_penalty: 0.25  
total\_penalty\_ceiling: 0.50

max\_revision\_attempts: 2

## 12.3 Profile Examples

### configs/profiles/research.yaml



yaml

```
context:
  retrieved_budget: 5000
  memory_budget: 1500

retrieval:
  top_k: 20
  rerank_k: 10

performance:
  request_timeout_seconds: 45
```

### configs/profiles/course.yaml



yaml

```
context:
  retrieved_budget: 1500

output:
  require_sources: true

performance:
  request_timeout_seconds: 15
```

---

## 13. Ingestion and Data Pipeline

### 13.1 Corpus Ingestion Script (scripts/ingest.py)



python

```
#!/usr/bin/env python
```

```
"""Ingest historical documents into MongoDB and vector store."""
```

```
import argparse
import json
from pathlib import Path
from pymongo import MongoClient
from app.retrieval import HistorianRetriever
from app.config import Config
import logging
```

```
logging.basicConfig(level=logging.INFO)
```

```
logger = logging.getLogger(__name__)
```

```
def ingest_json_directory(data_dir: str, config: Config):
    """Load JSON files from directory, upsert to MongoDB, embed and store in vector DB."""
```

```
    client = MongoClient(config.corpus.mongodb_uri)
```

```
    db = client[config.corpus.db_name]
```

```
    documents = db[config.corpus.collection]
```

```
    retriever = HistorianRetriever(config)
```

```
    data_path = Path(data_dir)
```

```
    json_files = list(data_path.glob("/**/*.json"))
```

```
    logger.info(f'Found {len(json_files)} JSON files')
```

```
    for json_file in json_files:
```

```
        with open(json_file) as f:
```

```
            doc = json.load(f)
```

```
    # Upsert to MongoDB
```

```
    result = documents.update_one(
```

```
        {"source_file": str(json_file)},
```

```
        {"$set": doc},
```

```
        upsert=True
```

```
    )
```

```
    logger.info(f'Upserted {json_file}: {result.upserted_id or result.modified_count}')
```

```
    # Ingest corpus into vector store
```

```
    logger.info("Ingesting corpus into vector store...")
```

```
    retriever.ingest_corpus(batch_size=100)
```

```
    logger.info("Ingestion complete")
```

```
def main():
```

```
    parser = argparse.ArgumentParser(description="Ingest historical documents")
```

```
    parser.add_argument("--data-dir", required=True, help="Directory with JSON files")
```

```
    parser.add_argument("--profile", default="default", help="Config profile")
```

```
    parser.add_argument("--clear", action="store_true", help="Clear existing data first")
```

```

args = parser.parse_args()

config = Config.load(args.profile)

if args.clear:
    logger.warning("Clearing existing documents and vector store...")
    client = MongoClient(config.corpus.mongodb_uri)
    db = client[config.corpus.db_name]
    db[config.corpus.collection].delete_many({})

    ingest_json_directory(args.data_dir, config)

if __name__ == "__main__":
    main()

```

**Usage:**



```

bash

python scripts/ingest.py --data-dir ./data/historical_docs --profile research

```

# 14. Error Handling and Resilience

## 14.1 Error Handling Strategy

**Node-level errors:**

- Try-catch with logging per node
- Retry with exponential backoff (0.5s, 1s, 2s)
- Max 3 retries, then fail and log

**Retrieval failures:**

- Empty result → expand query and retry
- Timeout → use parametric fallback
- Low scores → degrade confidence

**LLM parsing errors:**

- Invalid JSON → retry with temperature 0
- Schema mismatch → re-prompt with schema
- Timeout → return partial result

## 14.2 Fallback Paths



```
python
```

```
def safe_invoke(state, node_func, node_name):
    """Invoke node with retry and fallback logic."""
    max_retries = config.performance.max_retries

    for attempt in range(max_retries):
        try:
            return node_func(state)
        except Exception as e:
            logger.error(f"{node_name} attempt {attempt+1} failed: {str(e)}")
            if attempt < max_retries - 1:
                wait = config.performance.retry_backoff_factor ** attempt
                time.sleep(wait)
            else:
                state["error"] = str(e)
            return state
```

---

## 15. Performance Optimization

### 15.1 Prompt Caching



python

```
llm = ChatOpenAI(
    model="gpt-4o-mini",
    cache_type="in_memory",
    ttl=3600
)
```

Benefits: 90% discount on cached tokens, faster response, deterministic key.

### 15.2 Context Compression



python

```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMListCompressor

compressor = LLMListCompressor.from_llm_and_prompt(llm, prompt)
compression_retriever = ContextualCompressionRetriever(
    base_retriever=retriever,
    document_compressor=compressor
)
```

Reduces retrieved context by 30-50% while maintaining relevance.

---

# 16. Versioning and Migration

## 16.1 Version Management

**Version file:** `historian_agent/__version__.py`



python

```
__version__ = "1.0.0-alpha.1"
```

Each config YAML includes version:



yaml

```
version: "1.0"
```

## 16.2 Migration Script (scripts/migrate.py)



python

```
def migrate_v1_0_to_v1_1(config_old):  
    """Example migration: add new field to output schema."""  
    config_old["output"]["add_confidence"] = True  
    return config_old
```

---

# 17. Deployment Checklist

- ☐ Python 3.11+ environment with pyproject.toml
  - ☐ MongoDB instance running (local or cloud)
  - ☐ OpenAI/Anthropic API keys in .env
  - ☐ Vector store directory writable
  - ☐ All tests passing: `pytest tests/`
  - ☐ Eval benchmarks run: `python tests/eval/eval.py`
  - ☐ Adversarial eval run: `python tests/eval/adversarial_metrics.py`
  - ☐ API server up: `python scripts/api.py`
  - ☐ CLI functional: `python scripts/cli.py --q "test query"`
  - ☐ LangSmith project created and linked (optional)
  - ☐ Documentation: README.md, example notebook
  - ☐ Production config profiles ready
-

## 18. Success Metrics and Milestones

### 18.1 Phase 1: Foundation (Week 1)

- ☐ Project scaffold, configs, minimal chain
- ☐ Schemas and data models
- ☐ Tracing and logging setup
- ☐ Green unit tests

### 18.2 Phase 2: Retrieval & Composition (Week 2)

- ☐ Hybrid retrieval (vector + BM25)
- ☐ Citation attachment and metadata resolution
- ☐ Composer chain with StructuredOutputParser
- ☐ Integration tests for end-to-end workflow

### 18.3 Phase 3: LangGraph & Adversarial (Week 3)

- ☐ Multi-node state graph with validation loop
- ☐ **Adversarial verification layer (4 roles)**
- ☐ Memory blocks (CRUD, summarization)
- ☐ Error handling and fallback paths

### 18.4 Phase 4: Production & Eval (Week 4)

- ☐ FastAPI and CLI interfaces
- ☐ **Adversarial eval suite with metrics**
- ☐ Documentation and examples
- ☐ Performance tuning: caching, compression
- ☐ Version 1.0.0-alpha.1 release

### 18.5 Success Criteria

Metric	Target	Validation
Schema Validity	≥95%	eval.py output
Hallucination Detection	≥95%	adversarial_metrics.py
Citation Integrity	≥98%	interrogator accuracy
False Positive Rate	≤5%	of well-supported answers
Latency p95 (simple Q/A)	≤8s	LangSmith traces
Latency p95 (RAG)	≤20s	LangSmith traces
Confidence Calibration	±10%	bucket analysis
Test Coverage	≥80%	pytest --cov

## 19. Dependencies and Requirements

### 19.1 pyproject.toml



toml

```

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "historian-agent"
version = "1.0.0-alpha.1"
description = "LangChain v1 agentic app for historical document analysis with RAG"
requires-python = ">=3.11"

dependencies = [
    "langchain>=0.1.0",
    "langchain-openai>=0.1.0",
    "langchain-anthropic>=0.1.0",
    "langgraph>=0.1.0",
    "pydantic>=2.0",
    "pymongo>=4.6",
    "chromadb>=0.4",
    "langsmith>=0.1",
    "fastapi>=0.104",
    "uvicorn>=0.24",
    "pyyaml>=6.0",
    "tqdm>=4.66",
    "python-dotenv>=1.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.4",
    "pytest-cov>=4.1",
    "pytest-asyncio>=0.21",
    "black>=23.12",
    "ruff>=0.1",
    "mypy>=1.7",
]

```

## 19.2 Environment Variables (.env.example)



bash

```
# LLM Provider
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY=sk-...

# Database
MONGODB_URI=mongodb://localhost:27017/
DB_NAME=railroad_documents

# Vector Store
VECTOR_STORE_PATH=./vector_store

# LangSmith (optional)
LANGSMITH_ENABLED=false
LANGSMITH_PROJECT=historian-agent-dev
LANGSMITH_API_KEY=

# Config Profile
HISTORIAN_PROFILE=default
```

## 20. Documentation Structure

### 20.1 README.md Table of Contents

- 1. Quick Start (5-minute setup)
- 2. Architecture Overview (component diagram)
- 3. API Reference (CLI and REST endpoints)
- 4. Configuration (profiles and tuning)
- 5. Ingestion (how to add documents)
- 6. **Adversarial Verification** (new section)
- 7. Evaluation (running benchmarks)
- 8. Troubleshooting
- 9. Contributing

### 20.2 Example Notebook: notebooks/01\_minimal\_chain.ipynb



python

```
# Cell 1: Setup
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StructuredOutputParser

# Cell 2: Define schema
from pydantic import BaseModel

class Answer(BaseModel):
    response: str
    confidence: float

# Cell 3: Build chain
prompt = ChatPromptTemplate.from_template("Answer: {topic}")
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
parser = StructuredOutputParser.from_pydantic(Answer)

chain = prompt | llm | parser

# Cell 4: Invoke
result = chain.invoke({"topic": "What caused the 1893 financial panic?"})
print(result)
```

## 21. Risk Mitigation

Risk	Likelihood	Impact	Mitigation
Low hallucination rate hard to achieve	Medium	High	Eval early, use extractive phrasing, adversarial layer
Adversarial layer adds too much latency	Medium	Medium	Profile early, use cheap models for challenger/interrogator, parallelize
Latency exceeds 20s for complex queries	Medium	Medium	Optimize retrieval, use caching, compression
Vector store becomes inconsistent	Low	High	Versioned ingestion script, backup strategy
LLM API rate limits	Low	Medium	Implement queue, backoff, local fallback
Adversarial reports contradict each other	Medium	Medium	Ensemble judge makes final call, add confidence rationale
Memory store grows unbounded	Low	Medium	Eviction policy, TTL on blocks

## 22. Acceptance Tests

### AT-001: JSON Schema Compliance

**Given:** Query "List three causes of the 1893 panic, include citations."  
**When:** Agent processes query  
**Then:**

- Output is valid JSON
- bullets array has length 3
- citations array has length  $\geq 3$
- Each citation has source\_id and locator

### AT-002: RAG Coverage with Known Source

**Given:** Seeded corpus with known document (e.g., WPA employment record)  
**When:** Query "What states received WPA employment funds?"  
**Then:**

- Result cites the WPA document
- Cited span includes correct page and character offsets
- Excerpt matches corpus text

### AT-003: Adversarial Catches Hallucination

**Given:** Corpus mentions only 3 states with WPA programs

**When:** Agent answer claims "WPA in 48 states"

**Then:**

- Adversarial challenger flags as unsupported
- Interrogator reports low citation integrity
- Ensemble confidence < 0.60
- Status = "needs\_revision"

### AT-004: Adversarial Approves Well-Cited Answer

**Given:** Corpus and answer both mention 3 specific states with proper citations

**When:** Agent answer says "WPA in at least 3 major states: NY, CA, TX" with citations

**Then:**

- Challenger finds no critical issues
- Interrogator reports citation integrity ≥ 0.95
- Ensemble confidence ≥ 0.80
- Status = "verified"

## 23. Adversarial Verification Success Criteria

Metric	Target	Justification
Hallucination Detection Rate	≥95%	Catch 95% of hallucinations
Citation Integrity (quote mining)	≥98%	Catch misquotes and context errors
False Positive Rate	≤5%	Don't over-flag correct answers
Confidence Calibration	±10%	If score 0.75, should be correct ~75% of time
Latency Overhead	+4-6s	Acceptable for research quality
Cost Multiplier	≤3x baseline	Hallucination prevention worth it
Ensemble Agreement with Manual Review	≥85%	Automated verdict reliable

## 24. Next Steps and Implementation Order

### Week 1 (Foundation):

1. Clone repo scaffold, install dependencies
2. Implement schemas.py with all Pydantic models
3. Implement config.py with profile loading
4. Run tests, verify setup

### Week 2 (Retrieval & Composition):

1. Implement retrieval.py with MongoDB + Chroma integration
2. Implement prompts.py (system, planner, composer, validator)
3. Implement chains.py with basic runnable chains
4. Implement graph.py base workflow (ingest → validate → finalize)
5. Integration tests for end-to-end

### Week 3 (Adversarial Verification):

1. Add adversarial prompts to prompts.py
2. Add adversarial schemas to schemas.py
3. Implement adversarial nodes in graph.py
4. Add adversarial chains to chains.py
5. Implement safe\_invoke retry logic

6. Unit tests for each adversarial role

#### Week 4 (APIs, Eval, Production):

1. Implement scripts/cli.py and scripts/api.py
2. Implement scripts/ingest.py for corpus loading
3. Implement tests/eval/adversarial\_metrics.py
4. Run full eval suite
5. Documentation and examples
6. Performance tuning
7. Release v1.0.0-alpha.1

---

## 25. Quick Reference Commands



bash

*# Setup*

pip install -e .

export OPENAI\_API\_KEY=sk-...

*# Ingest corpus*

python scripts/ingest.py --data-dir ./data --profile research --clear

*# CLI query*

python scripts/cli.py --q "Summarize WPA employment, 1935-1941" --output json

*# Start API server*

python scripts/api.py

*# Test: curl -X POST http://localhost:8000/query -H "Content-Type: application/json" -d '{"query": "Who was FDR?"}'*

*# Run all tests*

pytest tests/ -v --cov=app

*# Run adversarial eval*

python -c "from tests.eval.adversarial\_metrics import eval\_full\_adversarial\_suite; eval\_full\_adversarial\_suite('tests/eval/adversarial\_eval\_datas"

*# Load config*

python -c "from app.config import Config; cfg = Config.load('research'); print(cfg.model.model\_id)"

---

## 26. Reference Links

- LangChain Docs: <https://python.langchain.com/v0.1/docs/>
- LangGraph Guide: <https://langchain-ai.github.io/langgraph/>
- LangSmith Tracing: <https://smith.langchain.com/>
- MongoDB Atlas: <https://www.mongodb.com/cloud/atlas>
- Chroma Vector Store: <https://www.trychroma.com/>
- FastAPI: <https://fastapi.tiangolo.com/>
- Pydantic: <https://docs.pydantic.dev/latest/>
- pytest: <https://docs.pytest.org/>

- PyYAML: <https://pyyaml.org/>

---

## 27. Example Full Response with Adversarial Verification



json

```
{
  "answer": "The Works Progress Administration (WPA), established in 1935, employed approximately 3.3 million workers at its peak in 1936.",
  "bullets": [
    "WPA was the largest New Deal jobs program, authorized by Congress in 1935 and funded through 1943",
    "Peak employment reached 3.3 million workers in fiscal year 1936",
    "Work types included public infrastructure (45%), conservation (25%), cultural/arts (15%), and administration (15%)",
    "Programs operated in documented locations across at least 12 major states with substantial historical records"
  ],
  "citations": [
    {
      "id": "c1",
      "source_id": "wpa_administrative_history_001",
      "title": "WPA Administrative History and Statistics",
      "author": "Federal Writers Project",
      "year": "1939",
      "locator": "p2, section 3, lines 45-52",
      "text": "The Works Progress Administration, authorized by Congress in 1935 under the Emergency Relief Appropriation Act, employed at its peak approximately 3.3 million workers across the United States.",
      "url": "http://localhost:5000/document/wpa_administrative_history_001"
    },
    {
      "id": "c2",
      "source_id": "wpa_project_breakdown_002",
      "title": "WPA Projects by Category and State",
      "author": "Works Progress Administration",
      "year": "1937",
      "locator": "p1, table 1",
      "text": "Distribution of WPA employment by project category: Public Works and Infrastructure 45%, Conservation and Land Management 25%, Cultural Arts 15%, and Administration 15%.",
      "url": "http://localhost:5000/document/wpa_project_breakdown_002"
    }
  ],
  "confidence": 0.78,
  "metadata": {
    "query_tokens": 145,
    "completion_tokens": 320,
    "latency_ms": 5820,
    "retrieval_count": 8,
    "model": "gpt-4o-mini",
    "run_id": "run_abc123xyz789"
  },
  "adversarial_verification": {
    "status": "flagged",
    "reasoning": "Answer is well-supported by citations and retrieval. However, claiming 'across multiple states' and 'at least 12 major states' without specific state names or counts is vague and potentially misleading."
  }
}
```

```
"challenger_report": {
  "challenges": [
    {
      "claim": "programs operated in documented locations across at least 12 major states",
      "issue": "Vague qualifier 'at least 12' - corpus only documents 8 states explicitly",
      "severity": "medium",
      "evidence_gap": "Should list specific states or use 'including' rather than implying comprehensive coverage"
    }
  ],
  "overall_confidence_adjustment": -0.10,
  "recommended_revisions": [
    "Replace 'at least 12 major states' with 'including New York, California, Texas, Illinois, Pennsylvania, and others' or list specific document",
    "Or soften claim to: 'Programs operated in multiple states, documented in surviving records from NY, CA, TX and other locations'"
  ],
  "passes_adversarial_check": false
},
```

```
"interrogator_report": {
  "citation_checks": [
    {
      "citation_id": "c1",
      "claim_being_cited": "WPA authorized 1935, peak employment 3.3 million in 1936",
      "source_excerpt": "The Works Progress Administration, authorized by Congress in 1935 under the Emergency Relief Appropriation Act,",
      "match_quality": "exact",
      "context_preserved": true,
      "cherry_picked": false,
      "issues": "None detected"
    },
    {
      "citation_id": "c2",
      "claim_being_cited": "Work distribution: infrastructure 45%, conservation 25%, arts 15%, admin 15%",
      "source_excerpt": "Distribution of WPA employment by project category: Public Works and Infrastructure 45%, Conservation and Land",
      "match_quality": "paraphrase",
      "context_preserved": true,
      "cherry_picked": false,
      "issues": "None detected"
    }
  ],
  "citation_integrity_score": 0.96,
  "problematic_citations": []
},
```

```
"counter_argument_report": {
  "counter_argument": "While WPA provided employment, critics and economic historians have argued the program was inefficient, created",
  "counter_citations": [
    {
      "source_id": "wpa_criticisms_003",
      "locator": "p5-6",

```

```
    "text": "Contemporary critics of the WPA noted that while employment figures were impressive, many projects were low-productivity tas
  }
],
"strength_of_counter": 0.62,
"both_interpretations_valid": true,
"original_answer_bias": "Emphasized employment numbers and project types without acknowledging scholarly debate about program effec
},

"ensemble_verdict": {
  "final_confidence": 0.78,
  "verified": false,
  "confidence_rationale": "Base confidence 0.88 reduced by: (1) Overgeneralization about 'at least 12 states' claim without documentation (-0
  "required_revisions": [
    "Specify which states had documented WPA programs or soften geographic claims",
    "Consider adding one sentence acknowledging contemporary criticisms or scholarly debate about program effectiveness"
  ],
  "safe_to_publish": true,
  "additional_sources_needed": false,
  "verification_status": "flagged",
  "note": "Answer may be published as-is with confidence score of 0.78, or revised to address geographic specificity for confidence ≥0.85"
}
}
}
```

---

## 28. Example Adversarial Revision Loop

### Initial answer (confidence 0.88):

"WPA programs operated in at least 12 major states..."

### Challenger feedback:

"Vague claim - corpus only documents 8 states"

### Revised answer (confidence 0.82):

"WPA programs operated in multiple states, including New York, California, Texas, Illinois, Pennsylvania, and other documented locations based on surviving historical records..."

### Final ensemble verdict:

confidence: 0.82, verified: true, safe\_to\_publish: true

---

**Document prepared:** October 21, 2025  
**Version:** 1.0 Complete Implementation Plan with Adversarial LLM Verification  
**Status:** Ready for Development Sprint  
**Total Sections:** 28 comprehensive sections covering architecture, code, testing, and deployment