Materials, Methods and Results

Multi-Agent System for Automated Fact-Checking of YouTube Videos

Begoña Echavarren Sánchez

Tutor: Josep-Anton Mir Tutusaus

Master's Degree in Data Science
Universitat Oberta de Catalunya

PEC 3 - Implementation

December 2025

Contents

1 Materials and Methods **3**

1.1 System Architecture Overview . . . 3
1.1.1 High-Level Architecture . . . 3
1.1.2 Design Principles . . . 5
1.1.3 Component Isolation Pattern . . . 5
1.2 Technology Stack . . . 6
1.2.1 Core Technologies . . . 6
1.2.2 Large Language Models . . . 6
1.2.3 External Services . . . 7
1.3 Pipeline Components . . . 7
1.3.1 Transcriptor . . . 7
1.3.2 Claim Extractor . . . 8
1.3.3 Query Generator . . . 10
1.3.4 Online Search . . . 11
1.3.5 Output Generator . . . 14
1.4 LLM Configuration and Model Management . . . 15
1.4.1 Model Abstraction Layer . . . 15
1.4.2 Model Instantiation with Caching . . . 16
1.4.3 Per-Component Model Configuration . . . 16
1.4.4 Common Model Settings . . . 16
1.5 Latency Optimization Strategies . . . 17
1.5.1 Process Tokens Faster: Model Selection . . . 17
1.5.2 Generate Fewer Tokens: Output Constraints . . . 17
1.5.3 Use Fewer Input Tokens: Content Trimming . . . 17
1.5.4 Make Fewer Requests: Combined Operations . . . 18
1.5.5 Parallelize: 4-Level Async Architecture . . . 18
1.5.6 Real-Time Streaming . . . 19
1.5.7 Classical Methods for Non-Reasoning Tasks . . . 19
1.6 Data Schemas and Structured Outputs . . . 19
1.6.1 Pydantic AI Integration . . . 19
1.6.2 Schema Design Principles . . . 20
1.7 Evaluation Framework . . . 20
1.7.1 Experiment Tracker (Singleton Pattern) . . . 20
1.7.2 Pydantic AI Monitoring (Monkey-Patching) . . . 21
1.7.3 Metrics Aggregation . . . 21
1.7.4 Output Directory Structure . . . 21
1.8 Experiment Configuration and Runner . . . 21

1.8.1 YAML Configuration 21
1.8.2 Test Video Corpus 22
1.8.3 Experiment Types 22
1.8.4 Experiment Runner CLI 22
1.9 Analysis and Visualization Pipeline 23
1.9.1 Data Loading and Enrichment 23
1.9.2 Generated Visualizations 23
1.10 API Layer and Real-Time Streaming 23
1.10.1 FastAPI Setup 23
1.10.2 Streaming Endpoint with SSE 23
1.10.3 Progress Events 24
1.10.4 Request/Response Schemas 24
1.11 Code Quality and Engineering Practices 24
1.11.1 Type Safety 24
1.11.2 Logging 25
1.11.3 Error Handling Patterns 25
1.11.4 Configuration Management 25
1.11.5 Pre-commit Hooks 26
1.12 Design Patterns and Software Engineering 26
1.12.1 Patterns Summary 26
1.12.2 Async Patterns 26
1.12.3 Key Engineering Decisions 27

2 Results **27**

1 Materials and Methods

This section presents the comprehensive technical implementation of Factible, a multi-agent system for automated fact-checking of YouTube videos. The system implements an end-to-end pipeline that processes video content through five specialized components, leveraging large language models (LLMs) for reasoning tasks while employing classical algorithms for deterministic operations. The implementation follows design-science principles [6, 11], emphasizing the creation of artifacts that extend human capabilities through systematic evaluation and iterative refinement.

1.1 System Architecture Overview

1.1.1 High-Level Architecture

Factible implements an end-to-end automated fact-checking pipeline for YouTube videos using a multi-agent architecture. Recent research on LLM agents demonstrates that multi-agent collaboration can enhance factuality and reasoning by allowing specialized agents to converse and coordinate on tasks [17]. FactAgent further shows that decomposing fact-checking into dedicated agents for input ingestion, query generation, evidence retrieval, and verdict prediction yields higher accuracy and transparency [19]. The Factible architecture follows this line of work by processing video content through five specialized, modular components that operate sequentially with four levels of internal parallelization.

The system processes a YouTube video URL through five sequential stages, each with specialized responsibilities. The pipeline begins with transcript extraction, proceeds through claim and query generation, conducts online evidence retrieval, and culminates in structured verdict synthesis. This modular design enables independent optimization of each component while maintaining clear data contracts between stages.

The five stages are:

1. **Transcriptor**: Extracts video transcripts via YouTube Transcript API, preserving timestamped segments for claim localization. The component includes automatic fallback to proxy service when rate-limited, ensuring robust transcript retrieval across different access conditions.

2. **Claim Extractor** (LLM Agent): Employs thesis-first reasoning to infer the video's central argument before extracting factual and verifiable claims. Each claim receives an importance score based on its impact on the video's thesis. Post-processing uses fuzzy string matching to locate claims within the transcript for timestamp mapping.

3. **Query Generator** (LLM Agent): Generates diverse search queries across four strategic types—direct, alternative, source-seeking, and contextual. Each query receives a priority score (1–5) based on evidence likelihood, enabling budget-conscious filtering of low-priority queries.

4. **Online Search**: Executes a four-step evidence retrieval pipeline for each query: (i) Google Search via Serper API, (ii) website reliability assessment using Media Bias/Fact Check (MBFC) data combined with domain heuristics [10], (iii) content fetching via Selenium WebDriver with JavaScript rendering support, and (iv) LLM-based evidence extraction with stance classification (supports, refutes, mixed, unclear).

5. **Output Generator** (LLM Agent): Synthesizes evidence into structured verdicts by building evidence bundles grouped by stance, generating natural language summaries with confidence levels, calculating algorithmic evidence quality scores, and mapping claims to video timestamps for interactive navigation.

Figure 1 illustrates the complete pipeline architecture with data flow and parallelization points across all five stages.

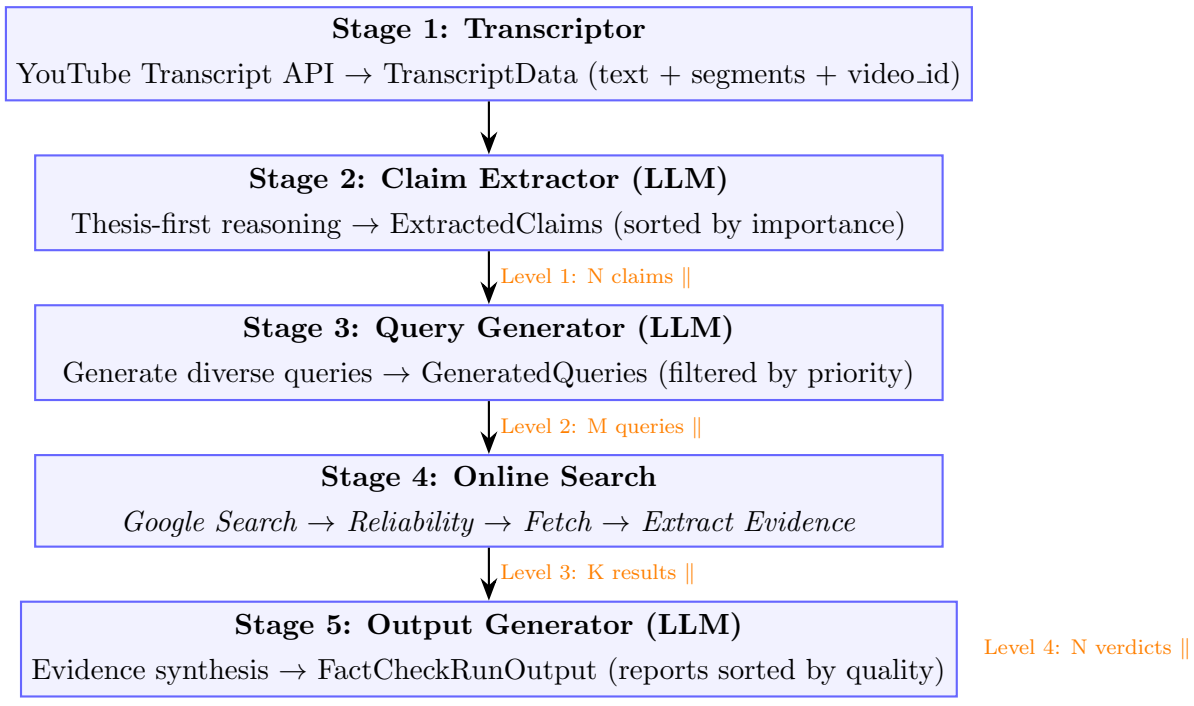

Figure 1: High-level pipeline architecture showing the five main stages and four parallelization levels. Parallelization occurs at claims (Level 1), queries per claim (Level 2), search results per query (Level 3), and final verdicts (Level 4).

1.1.2 Design Principles

The system adheres to several key design principles derived from software engineering best practices and GenAI application development, aligned with the design-science paradigm in information systems research [6]:

1. **Modularity**: Each component is isolated with well-defined inputs and outputs using Pydantic schemas, enabling independent optimization, testing, and replacement.

2. **Structured Outputs**: All LLM interactions use Pydantic AI with typed output schemas, ensuring type safety, automatic validation, and consistent data structures across the pipeline.

3. **Transparency**: The full evidence chain is preserved and exposed to users—sources, reliability ratings, stances, and reasoning are all traceable from final verdict back to original source.

4. **Progressive Enhancement**: The pipeline operates with graceful degradation (e.g., fallback to snippet if scraping fails, fallback to proxy if rate-limited) rather than failing entirely.

5. **Cost-Conscious Design**: Configurable limits (`max_claims`, `max_queries`, `max_results`) prevent runaway API costs during development and production.

6. **Reproducibility**: Deterministic LLM outputs (`temperature=0.0`), structured YAML configurations, and comprehensive experiment tracking enable reproducible research.

7. **Separation of Concerns**: Classical algorithms handle tasks like reliability scoring, deduplication, and quality calculation, reserving LLM calls for tasks requiring reasoning and language understanding.

1.1.3 Component Isolation Pattern

Each component follows a consistent directory structure that promotes modularity and maintainability. This standardized organization enables independent testing of each component in isolation, seamless swapping of LLM models for experimentation, automated metrics collection via decorators, and clear interface contracts through Pydantic schemas. The typical structure includes a public exports file (`__init__.py`), main logic file with tracking decorators (`component_name.py`), and schema definitions file (`schemas.py`) containing all input and output Pydantic models. This separation of concerns facilitates parallel development and reduces coupling between pipeline stages.

1.2 Technology Stack

1.2.1 Core Technologies

Table 1 presents the core technologies employed in the implementation.

Table 1: Core technology stack

Category	Technology	Version	Purpose
Language	Python	3.12	Core implementation with type hints
LLM Framework	Pydantic AI	$\geq$1.0.0	Agent orchestration, structured outputs
Data Validation	Pydantic	$\geq$2.0.0	Schema definitions, runtime validation
Web Framework	FastAPI	$\geq$0.115.0	REST API with SSE streaming
HTTP Server	Uvicorn	$\geq$0.32.0	High-performance ASGI server
Async HTTP	httpx	$\geq$0.28.1	Async HTTP client
Web Scraping	Selenium	$\geq$4.15.2	JavaScript-rendered content extraction
YouTube	youtube-transcript-api	$\geq$1.2.2	Transcript extraction
Domain Info	python-whois	$\geq$0.8.0	Domain age lookup
CLI	Typer	$\geq$0.15.0	Experiment runner CLI
Analysis	pandas, matplotlib	-	Data analysis and visualization

1.2.2 Large Language Models

The system supports multiple LLM providers to enable comparison of cost-quality trade-offs. Table 2 shows the available models and their configurations.

Table 2: LLM providers and pricing

Provider	Model	Context	Pricing (per 1M tokens)	Use Case
OpenAI	gpt-4o-mini	128K	\$0.15 / \$0.60	Default
OpenAI	gpt-4o	128K	\$5.00 / \$15.00	High-quality
OpenAI	gpt-4-turbo	128K	\$10.00 / \$30.00	Premium
Ollama	qwen3:8b	40K	Free (local)	Budget/offline
Ollama	qwen3:4b	256K	Free (local)	Small footprint

Large language models such as GPT-4 offer multimodal capabilities and demonstrate human-level performance across diverse benchmarks [12]. Despite these advances, models still suffer from hallucinations and are constrained by limited context windows, underscoring the need for careful configuration and reliability safeguards [12]. Our model management layer therefore emphasizes deterministic outputs, context-aware trimming, and tool-assisted generation.

1.2.3 External Services

The system integrates with external services for search and transcript extraction:

- **Serper API**: Google Search wrapper providing organic search results with approximately 2,500 queries per month on the free tier.
- **YouTube oEmbed API**: Video metadata retrieval without authentication.
- **Webshare Proxy**: Rate limit bypass for transcript extraction with configurable proxy locations.

1.3 Pipeline Components

1.3.1 Transcriptor

The Transcriptor component extracts YouTube video transcripts with precise timestamp information for later claim-to-video mapping.

Implementation Details The transcriptor uses the `youtube-transcript-api` library to fetch available transcripts, with preference for English (`["en", "en-US"]`). When rate-limited by YouTube, it automatically falls back to a proxy service (Webshare). Key features include:

- **Timestamped Segments**: Each segment preserves `start` time and `duration` in seconds.
- **Character Position Mapping**: Enables mapping claim text positions back to video timestamps.
- **Title Fetching**: Uses YouTube oEmbed API to retrieve video title for context.
- **Proxy Fallback**: Automatic retry through Webshare proxy when rate-limited.

Output Schema Listing 1: Transcriptor output schemas

```python
class TranscriptSegment(BaseModel):
    """A single timestamped segment from a YouTube transcript."""
    text: str = Field(description="The text content of this segment")
    start: float = Field(ge=0.0, description="Start time in seconds")
    duration: float = Field(ge=0.0, description="Duration in seconds")
```

```python

class TranscriptData(BaseModel):
    """Complete transcript data with timestamped segments."""
    text: str = Field(description="Full transcript as plain
        text")
    segments: list[TranscriptSegment]
    video_id: str = Field(description="YouTube video ID")
```

Timestamp Mapping Algorithm The timestamp mapping function enables the system to locate where in the video each claim originates. The algorithm iterates through transcript segments sequentially, maintaining a cumulative character counter. When a character position falls within a segment's range, the function returns the corresponding video timestamp (start time and duration). This mapping is crucial for the user interface, allowing users to jump directly to the video moment where a specific claim was made.

1.3.2 Claim Extractor

The Claim Extractor identifies factual, verifiable claims from video transcripts using LLM-based extraction with thesis-relative importance ranking. Our approach builds on prior work in automated claim detection: supervised models trained on annotated political debates have been used to detect check-worthy claims [5], and end-to-end systems like ClaimBuster monitor public discourse and prioritize factual statements for manual fact-checking [8]. These systems show that focusing on salient, verifiable claims improves the efficiency of fact-checking pipelines.

LLM Configuration The claim extractor uses deterministic settings for reproducibility: temperature set to 0.0 to ensure consistent outputs across multiple runs, max tokens limited to 1,200 to control response length and latency, and automatic retry logic (3 attempts) to handle transient LLM failures gracefully.

Prompt Engineering Strategy: Thesis-First Approach The claim extractor employs a novel *thesis-first approach* with multi-step reasoning designed to prioritize claims most critical to the video's central argument:

Step 1: Thesis Inference — Before listing claims, the LLM infers the video's central thesis in no more than 25 words (e.g., "Climate change alarmism is driven more by politics and media than by settled science").

Step 2: Importance Ranking with Thesis Impact Test — Claims are scored based on their impact on the video's thesis using the question: "If this claim were

proven false, would the thesis collapse or materially weaken?" Table 3 presents the scoring guidelines.

Table 3: Claim importance scoring guidelines

Score Range	Description	Examples
0.85–1.0	Prescriptive/causal claims undermining thesis	Policy proposals, causal mechanis
0.60–0.80	Quantitative/historical evidence tied to thesis	Statistics, dates, expert citations
0.30–0.55	Context/supporting background	Definitions, general facts
0.0–0.25	Peripheral/anecdotal details	Personal stories, credentials

Step 3: Relevance Guardrails — Pure credential facts are capped at 0.30 unless the thesis questions expertise; statements not affecting the thesis are capped at 0.25; pure opinions are excluded; and paraphrases and duplicate numbers are removed.

Dynamic Instructions via Pydantic AI Dependencies The agent uses Pydantic AI's dependency injection mechanism to inject runtime constraints into the system prompt. This design pattern enables dynamic instruction generation based on runtime parameters: when a `max_claims` limit is specified, the instruction explicitly constrains the output size; otherwise, it defaults to requesting only the highest-impact claims. This approach provides flexibility for experimentation while maintaining type safety through Pydantic validation.

Post-Processing: Fuzzy Claim Localization After LLM extraction, each claim is located in the original transcript using fuzzy string matching. The algorithm normalizes text, applies a sliding window with ±2 words around the claim length, computes similarity using `difflib.SequenceMatcher`, and requires a minimum score of 0.5 for a match. This produces `transcript_char_start`, `transcript_char_end`, and `transcript_match_score` fields for timestamp mapping.

Output Schema

Listing 2: Claim extractor output schemas

```python
class Claim(BaseModel):
    """A single claim or fact extracted from text."""
    text: str  # Claim text (<=40 words recommended)
    confidence: float = Field(ge=0.0, le=1.0)
    category: str  # historical, scientific, statistical, etc.
    importance: float = Field(default=0.5, ge=0.0, le=1.0)
    context: str | None = Field(default=None)
    # Post-processing fields
```

```
    transcript_char_start: int | None = None
    transcript_char_end: int | None = None
    transcript_match_score: float | None = None

class ExtractedClaims(BaseModel):
    """Collection of claims extracted from a transcript."""
    claims: list[Claim]  # Sorted by importance (descending)
    total_count: int
```

Standard fact-checking datasets such as FEVER [16] and FEVEROUS [1] are used as external references to evaluate claim extraction performance. FEVER contains 185,445 claims derived from Wikipedia sentences labeled as Supported, Refuted, or NotEnoughInfo, while FEVEROUS extends this to 87,026 claims with both unstructured text and structured table evidence.

1.3.3 Query Generator

The Query Generator produces diverse, prioritized search queries optimized for evidence retrieval. Research on LLMs shows that interleaving reasoning and acting enables models to plan and perform external searches more effectively; the ReAct prompting technique encourages models to produce intermediate reasoning steps and task-specific actions, leading to improved factuality and reduced hallucination [18]. Retrieval-augmented generation methods combine parametric language models with non-parametric memory to retrieve relevant documents [7], while frameworks like RARR perform post-generation research and revision to align outputs with supporting evidence [3].

Query Type Taxonomy The system generates four types of queries with different search strategies, as shown in Table 4.

Table 4: Query type taxonomy

Type	Description	Strategy	Example
DIRECT	Exact claim phrasing	Verbatim search	"unemployment rose 15% Q
ALTERNATIVE	Rephrased with synonyms	Semantic variation	"jobless rate increase third
SOURCE	Target authoritative sources	Source-seeking	"BLS unemployment statist
CONTEXT	Broader context	Background search	"economic indicators fall 20

Priority System Queries are prioritized 1–5 based on likelihood of finding reliable, definitive information: priority 1 queries are always included, priority 2 by default, priority 3 if budget allows, and priorities 4–5 rarely or only for completeness.

Listing 3: Query generator output schemas

Output Schema

```python
class SearchQuery(BaseModel):
    query: str            # Search query text
    query_type: str       # "direct", "alternative", "source", "context"
    priority: int         # 1-5 (1 = highest priority)

class GeneratedQueries(BaseModel):
    original_claim: str           # Reference to source claim
    queries: list[SearchQuery]    # Filtered and sorted queries
    total_count: int              # Original count before filtering
```

1.3.4 Online Search

The Online Search component implements a multi-step pipeline to retrieve, assess, and extract evidence from web sources with adaptive quality filtering. Unlike the other LLM-based components, Online Search orchestrates multiple classical algorithms alongside a single LLM call for evidence extraction. This hybrid approach balances speed, reliability, and reasoning capabilities.

The reliability assessment combines domain-level heuristics with the Media Bias/-Fact Check (MBFC) methodology, which employs a comprehensive weighted scoring system to evaluate media outlets' ideological bias and factual reliability [10]. To mitigate hallucinations and ensure evidence quality, we draw on research like SelfCheck-GPT, which detects hallucinations by comparing multiple sampled responses and ranks passages by factuality [9]. Our multi-agent retrieval strategy is further informed by systems such as FactAgent [19] and LoCal [2], where decomposing, reasoning, and evaluating agents iteratively refine answers and outperform baselines.

Figure 2 illustrates the four-step Online Search pipeline executed for each query.

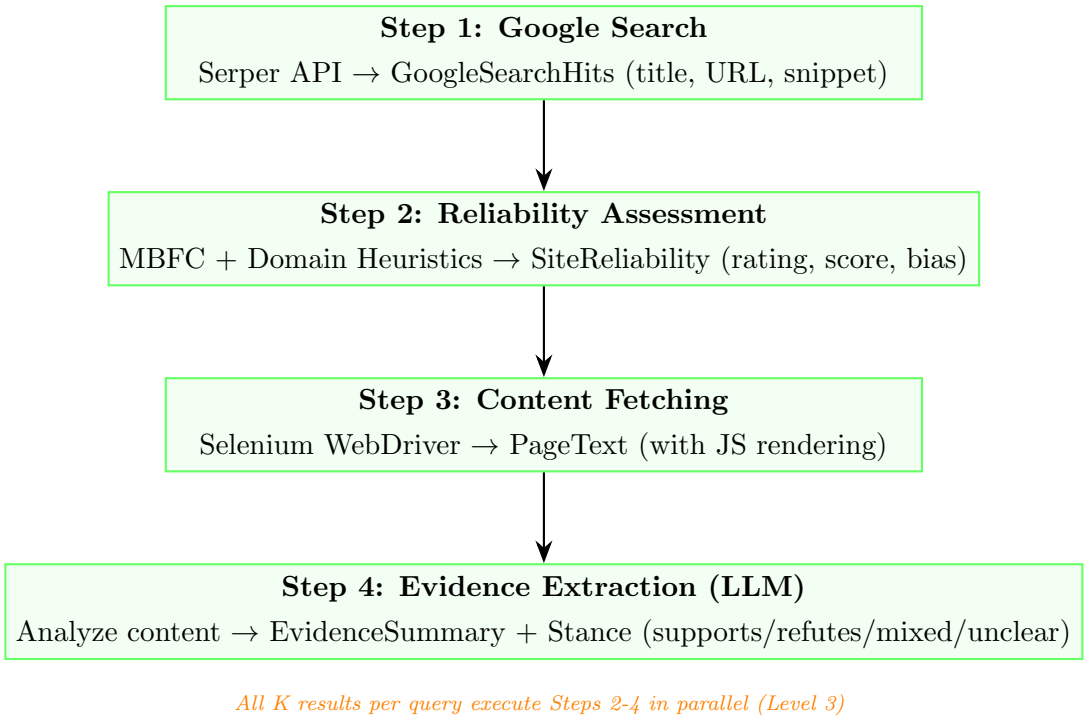

Figure 2: Online Search pipeline showing the four sequential steps executed for each search result. Steps 2–4 run in parallel across all K results per query (Level 3 parallelization).

Step 1: Google Search (Serper API) The Google search client wraps the Serper API for asynchronous search execution. The implementation uses persistent HTTP connections for performance and returns structured search hits containing title, URL, and snippet fields. Each query can retrieve up to 10 results, with the limit parameter controlling the exact number returned. The async design enables parallel query execution across multiple claims simultaneously.

Step 2: Website Reliability Assessment The reliability checker uses a multi-factor scoring system with first-match priority, combining external datasets with algorithmic heuristics:

- **Media Bias/Fact Check (MBFC) Dataset**: The system loads timestamped JSON snapshots containing over 10,000 news sources with credibility ratings. Credibility mappings are: high → 0.85, medium → 0.60, low → 0.30, very low → 0.15.
- **TLD Reputation**: High-trust top-level domains (`.gov`, `.edu`, `.int`) receive a base score of 0.90, reflecting their institutional authority.
- **Domain Age via WHOIS**: Domains ≥10 years old receive a +0.10 bonus (established presence), while domains <1 year old receive a –0.15 penalty (recent creation may indicate lower trust).

The output includes a categorical rating (high, medium, low, unknown), numerical score (0.0–1.0), reasoning for the assessment, and political bias classification when available from MBFC data.

Step 3: Content Fetching (Selenium) Content fetching uses Selenium WebDriver in headless Chrome mode with smart wait strategies to handle JavaScript-heavy sites. The implementation employs a two-phase extraction strategy: first attempting quick extraction of paragraph elements, then waiting for JavaScript rendering if initial content is insufficient (<100 characters). This adaptive approach balances speed for static sites with completeness for dynamic content.

Configuration optimizations include disabling images to reduce load time, setting page load timeouts (20 seconds), and limiting wait times for dynamic content (12 seconds). Content is trimmed to a maximum of 8,000 characters to control LLM input costs while preserving sufficient context for evidence extraction. Async integration wraps blocking Selenium calls in `asyncio.to_thread()` for non-blocking operation, enabling parallel processing of multiple search results.

Step 4: Evidence Extraction (LLM) The evidence extractor analyzes retrieved content against claims using structured stance definitions:

- **SUPPORTS**: Evidence confirms or validates the claim through direct statements, semantic equivalents, or mechanism descriptions.
- **REFUTES**: Evidence contradicts or disproves the claim through counter-evidence or statements that evidence is unproven/disproven.
- **MIXED**: Both supporting and refuting elements present.
- **UNCLEAR**: Genuinely ambiguous content that discusses related topics without addressing the specific claim.

Critical prompt instructions ensure that mere discussion equals UNCLEAR, that mechanisms are recognized even without exact terminology, and that both Google snippets and page content are considered with better evidence prioritized.

Adaptive Credibility Filtering The search orchestrator implements adaptive credibility filtering as a key innovation for ensuring evidence quality. The algorithm operates in three phases:

1. **Initial Batch**: Fetch 2× the desired limit to provide filtering headroom
2. **Quality Check**: If >50% of results are unreliable, fetch an additional batch to increase the pool of high-quality sources

3. **Intelligent Filtering**: Sort all results by reliability score and select the top reliable sources, with a minimum guarantee ensuring at least some results are returned even if reliability is universally low

Additional filtering mechanisms include stance filtering (removing unclear results if >50% have definitive stances) and URL deduplication using hash sets to prevent duplicate sources across different queries for the same claim.

1.3.5 Output Generator

The Output Generator synthesizes evidence into coherent verdicts with confidence levels, quality scoring, and timestamp mapping.

Two-Step Process The Output Generator employs a hybrid approach combining algorithmic evidence organization with LLM-based synthesis:

Step 1: Build Evidence Bundle (Algorithmic) — The system groups evidence by stance (supports, refutes, mixed, unclear), deduplicates sources by URL, and sorts within each group by reliability rating (high first), then numerical score, with alphabetic tie-breaking for consistency. This deterministic organization ensures reproducible output ordering.

Step 2: Generate Verdict (LLM) — Organized evidence is formatted into a structured prompt containing stance labels, source counts, reliability ratings, and evidence summaries. The system prompt instructs the LLM to synthesize concise verdicts, naming sources explicitly only when clarifying contrasting perspectives or when evidence directly conflicts. This reduces verbosity while maintaining attribution transparency.

Evidence Quality Score (Algorithmic) The quality score is calculated algorithmically without LLM involvement for consistency and speed. The scoring formula combines three components with different weights: a base score of 0.3 for having any evidence, an actionable stance bonus of up to 0.3 (scaled by the number of supports/refutes/mixed sources, saturating at 3 sources), and a reliability bonus of up to 0.4 (scaled by the number of high/medium reliability sources, saturating at 3 sources). This design prioritizes both actionable stances and source reliability, with the maximum achievable score of 1.0 indicating high-quality, decisive evidence from multiple reliable sources.

Table 5 summarizes the quality score components.

Table 5: Evidence quality score breakdown

Component	Weight	Criteria
Base	0.3	Having any evidence
Actionable	0.3	Up to 3 supports/refutes/mixed sources
Reliability	0.4	Up to 3 high/medium reliability sources
Maximum	**1.0**	

Parallel Verdict Generation (Level 4) All verdicts are generated concurrently using asynchronous programming primitives. The implementation creates verdict generation tasks for each claim-evidence pair, executes them in parallel using `asyncio.gather()`, and sorts the resulting reports by evidence quality score (descending) to prioritize high-confidence verdicts in the user interface. This parallel generation represents the fourth and final level of parallelization in the pipeline, significantly reducing total latency when processing multiple claims.

Output Schema Listing 4: Output generator schemas

```python
VerdictConfidence = Literal["low", "medium", "high"]

class ClaimFactCheckReport(BaseModel):
    """Structured report ready to present to end users."""
    claim_text: str
    claim_confidence: float
    claim_category: str
    overall_stance: EvidenceStance
    verdict_confidence: VerdictConfidence
    verdict_summary: str
    evidence_by_stance: dict[EvidenceStance, list[
       EvidenceSourceSummary]]
    total_sources: int = Field(ge=0)
    evidence_quality_score: float = Field(ge=0.0, le=1.0)
    timestamp_hint: float | None = None
    timestamp_confidence: float | None = None
```

1.4 LLM Configuration and Model Management

1.4.1 Model Abstraction Layer

The system uses an enum-based model configuration with Pydantic validation for type-safe model management. Each model is defined through a `ModelConfig` schema con-

taining provider name (OpenAI or Ollama), model identifier, input and output pricing per million tokens, and context window size. The `ModelChoice` enumeration defines available models as named constants (e.g., `OPENAI_GPT4O_MINI`, `OLLAMA_QWEN3_8B`), each associated with its configuration. This abstraction enables centralized model definitions, compile-time checking of model names, automatic price tracking for cost estimation, and easy addition of new models through enum extension.

1.4.2 Model Instantiation with Caching

Ollama model instances are cached using Python's `@lru_cache` decorator to enable connection reuse across multiple agent invocations. Without caching, each agent run would create a new provider connection, incurring initialization overhead. The cache is unbounded (`maxsize=None`), ensuring that once a model is instantiated, subsequent requests reuse the existing connection. This optimization is particularly important for Ollama models where the provider connection establishes communication with the local inference server at `http://127.0.0.1:11434/v1`.

1.4.3 Per-Component Model Configuration

A centralized configuration file defines which model each component uses, enabling easy model swapping for experiments. By default, all components use `OPENAI_GPT4O_MINI` for consistency, but this can be overridden on a per-component basis. For example, the Evidence Extractor could be downgraded to a local Ollama model to reduce costs while keeping the Claim Extractor on GPT-4o-mini for quality. This granular control enables component-specific optimization, A/B testing of model choices, and precise cost-quality trade-off analysis.

1.4.4 Common Model Settings

All components use `temperature=0.0` for deterministic outputs, enabling reproducibility, consistency in structured outputs, and meaningful A/B comparisons. Table 6 shows the token limits per component.

Table 6: Component model settings

Component	**Temperature**	**Max Tokens**	**Rationale**
Claim Extractor	0.0	1,200	Deterministic, structured claims
Query Generator	0.0	600	Concise queries, no explanations
Evidence Extractor	0.0	1,100	Summary + key quote
Output Generator	0.0	900	Concise verdict synthesis

1.5 Latency Optimization Strategies

The system implements multiple latency optimization strategies following established principles for LLM applications [13]. Research into LLM latency shows that prompt size, completion length, and model size are the dominant factors in inference time; reducing input and output token counts directly lowers compute and memory overhead [4, 13]. Additional techniques such as defining clear output boundaries, setting token limits, and adjusting sampling temperature can reduce generated tokens, and caching prompts allows reuse of computations for identical prefixes [13]. Choosing smaller model weights yields faster inference speeds. These optimizations can be grouped into seven core principles: processing tokens faster, generating fewer tokens, using fewer input tokens, making fewer requests, parallelizing operations, reducing perceived wait time, and using classical methods where LLMs are not required [13].

1.5.1 Process Tokens Faster: Model Selection

The default model (gpt-4o-mini) is selected for its balanced performance across speed, cost-effectiveness, and large context window (128K tokens). This model provides sufficient reasoning capabilities for fact-checking tasks while maintaining low latency and competitive pricing ($0.15 per million input tokens, $0.60 per million output tokens). For budget-conscious deployments or offline operation, local Ollama models (qwen3:8b, qwen3:4b) offer zero-cost inference at the expense of potential quality degradation. The modular model abstraction layer enables easy comparison of these trade-offs through experiment configuration.

1.5.2 Generate Fewer Tokens: Output Constraints

Each component has carefully tuned `max_tokens` limits to minimize generation latency and API costs without sacrificing information quality. Claims are limited to approximately 40 words (sufficient for most factual assertions), context descriptions to 20 words (brief background), and evidence summaries to 1–2 sentences (key findings only). These constraints are enforced through explicit prompt instructions and validated against output token limits. By preventing verbose outputs, the system reduces both generation time and downstream processing costs for subsequent pipeline stages.

1.5.3 Use Fewer Input Tokens: Content Trimming

Input token counts are minimized through aggressive content trimming strategies. Web content is trimmed to 6,000–8,000 characters before being passed to the Evidence Extractor, removing excessive context while retaining the most relevant portions (typically the first several paragraphs of an article). Evidence prompts include only Google

snippets and extracted page text, explicitly excluding raw HTML, JavaScript, CSS, and other non-content elements that would inflate token counts without improving extraction quality. This targeted trimming reduces LLM input costs by an order of magnitude compared to naive full-page submission.

1.5.4 Make Fewer Requests: Combined Operations

The pipeline minimizes LLM API calls by combining operations into single requests wherever possible. Each component makes exactly one LLM call per input unit (one call for claim extraction, one per claim for query generation, one per search result for evidence extraction, and one per claim for verdict synthesis), with no multi-turn conversations that would multiply request counts. The Query Generator produces all queries for a claim in a single batch call rather than generating queries iteratively. Similarly, the Output Generator synthesizes verdicts with all available evidence in a single call. This design reduces API overhead, improves latency, and simplifies cost tracking.

1.5.5 Parallelize: 4-Level Async Architecture

The system implements four levels of nested parallelization to maximize throughput while maintaining dependency ordering. This architecture enables the pipeline to process multiple claims, queries, and search results simultaneously, dramatically reducing total execution time compared to sequential processing.

The parallelization hierarchy operates as follows:

- **Level 1 (Claims)**: After extracting N claims from the transcript, all claims are processed in parallel. Each claim independently proceeds through query generation and search.
- **Level 2 (Queries per Claim)**: For each claim, the Query Generator produces M queries. These queries are executed in parallel, enabling simultaneous search across different query formulations (direct, alternative, source-seeking, contextual).
- **Level 3 (Search Results per Query)**: For each query, the Online Search component retrieves K results from Google. The four-step pipeline (reliability assessment, content fetching, and evidence extraction) runs in parallel for all K results, with each result processed independently.
- **Level 4 (Verdicts)**: After all evidence collection completes, the Output Generator synthesizes verdicts for all N claims in parallel, producing the final fact-check reports simultaneously.

This design achieves maximum theoretical parallelization of $N \times M \times K$ operations during the search phase, bounded only by system resources and API rate limits. Blocking I/O operations (Selenium WebDriver for content fetching, WHOIS for domain age lookup) are wrapped in `asyncio.to_thread()` to avoid blocking the event loop, ensuring that CPU-bound and I/O-bound operations can execute concurrently.

1.5.6 Real-Time Streaming

Server-Sent Events (SSE) provide progressive updates as the pipeline executes, improving perceived responsiveness for users. The streaming endpoint emits progress updates at key milestones, ranging from 5% (transcript extraction initiated) through 100% (fact-checking complete). Importantly, extracted claims are streamed to the user interface immediately after the Claim Extractor completes, allowing users to preview claims and begin reviewing the video's assertions before the time-intensive search phase completes. This progressive disclosure pattern reduces perceived latency and enables users to provide early feedback or cancellation if the extracted claims are not aligned with their expectations.

1.5.7 Classical Methods for Non-Reasoning Tasks

Table 7 shows operations handled by classical algorithms rather than LLMs.

Table 7: Operations using classical methods

Operation	**Method**	**Rationale**
Reliability scoring	Rule-based + MBFC lookup	Faster, deterministic, no API cost
Claim localization	Fuzzy string matching	No LLM needed for text search
Evidence quality score	Algorithmic calculation	Consistent, fast, reproducible
URL deduplication	Hash set	O(1) lookup
Stance filtering	Threshold-based	Simple percentage check

1.6 Data Schemas and Structured Outputs

1.6.1 Pydantic AI Integration

All LLM agents use Pydantic models as their `output_type`, providing comprehensive type safety throughout the pipeline. This integration ensures automatic JSON parsing and validation of LLM outputs, graceful error handling with automatic retries on validation failures, full IDE support with autocomplete for all schema fields, and consistent serialization via `model_dump()` for logging and persistence. The agent configuration includes the model selection, output schema type, runtime dependencies type, model

settings (temperature, token limits), system prompt, and retry count. This declarative configuration style reduces boilerplate while maintaining explicit control over agent behavior.

1.6.2 Schema Design Principles

The schema design follows five guiding principles to balance flexibility, safety, and maintainability. First, schemas prefer flat structures over deeply nested objects to simplify validation and reduce complexity. Second, optional fields use Python's union type syntax (`str | None`) for clarity. Third, `Literal` types constrain string fields to predefined values (e.g., stance must be "supports", "refutes", "mixed", or "unclear"). Fourth, all fields include descriptive names and `Field` descriptions for documentation and IDE hints. Fifth, numeric fields include validation constraints (e.g., `Field(ge=0.0, le=1.0)` for scores) to catch invalid data at runtime. These principles ensure robust data contracts across component boundaries.

1.7 Evaluation Framework

Evaluating fact-checking systems remains an open challenge because existing benchmarks are narrow and risk overfitting. Commentators have noted a "benchmark crisis" where canonical datasets no longer represent broad language understanding [15]. Practical evaluation strategies include multiple-choice benchmarks, verifiers, public leaderboards, and LLM judges, each with distinct trade-offs [14]. Our evaluation framework therefore combines quantitative metrics with qualitative assessments using external datasets such as FEVER [16] and FEVEROUS [1].

1.7.1 Experiment Tracker (Singleton Pattern)

The `ExperimentTracker` implements a singleton pattern with context manager support for global access during pipeline execution. This design enables any component to log metrics, timing, and LLM calls without explicit parameter passing. When initialized, the tracker creates a timestamped run directory, initializes data structures for timing, metrics, and LLM call logs, and registers itself as the current global tracker. The context manager protocol ensures automatic saving on exit and cleanup of the global reference.

A complementary `timer` context manager provides automatic timing instrumentation. When a code block executes within a timer context, the elapsed duration is automatically logged to the current tracker with a descriptive label. This declarative timing approach eliminates manual timing boilerplate and ensures consistent timing collection across all pipeline stages.

1.7.2 Pydantic AI Monitoring (Monkey-Patching)

The `@track_pydantic(component_name)` decorator enables automatic tracking of all LLM calls via runtime monkey-patching of `Agent.run()` and `Agent.run_sync()`. Each call records component name, model, timestamp, latency, input prompt, estimated tokens, output, and calculated cost in USD.

1.7.3 Metrics Aggregation

The `metrics.json` file provides aggregated experiment metrics for quantitative analysis and comparison across runs. Metrics include total pipeline execution time, per-step timing breakdowns (transcript, claims, queries, search, output), total LLM call count, estimated total cost in USD, and success/failure status. This structured format enables automated analysis scripts to compare different parameter configurations, identify performance bottlenecks, and track cost-quality trade-offs across the experiment corpus.

1.7.4 Output Directory Structure

Each experiment run creates a timestamped directory containing structured artifacts for analysis. The directory includes `config.json` (run ID, parameters, model configuration), `llm_calls.json` (all Pydantic AI calls with full input/output), `outputs.json` (final claims and fact-check reports), `metrics.json` (aggregated timing, cost, success status), and `transcript.txt` (original video transcript for reference). This organization enables reproducible analysis, post-hoc debugging of individual LLM calls, and batch processing of experiment results across parameter sweeps.

1.8 Experiment Configuration and Runner

1.8.1 YAML Configuration

The YAML configuration defines test videos and experiment parameters with automatic expansion:

Listing 5: Example YAML configuration

```yaml
videos:
  - id: "climate_what_scientists_say"
    url: "https://www.youtube.com/watch?v=OwqIy8Ikv-c"
    description: "Climate Change: What Do Scientists Say?"
    tags: ["climate", "climate_skepticism", "short", "selected"
       ]

experiments:
  - name: "vary_claims"
```

```
    description: "OFAT: Impact of max_claims on quality/cost/time"
    max_claims: [1, 3, 5, 7, 10]  # Auto-expands to 5 experiments
    max_queries_per_claim: 2
    max_results_per_query: 3
    video_filter: ["selected"]
```

1.8.2 Test Video Corpus

The evaluation dataset includes 19 videos across domains: Climate (10 videos covering climate skepticism, renewable energy, CO2, Texas freeze), Technology (7 videos on 5G health concerns, AI job displacement), and Health (2 videos on EMF radiation, alkaline water). Selection criteria include verifiable factual claims, range of difficulty levels, mix of viewpoints, available English transcripts, and durations from 2–25 minutes.

1.8.3 Experiment Types

Three experiment types are supported:

Baseline: Standard configuration (`max_claims=5`, `max_queries=2`, `max_results=3`).

OFAT (One-Factor-At-A-Time) Analysis: Sensitivity analysis varying one parameter while holding others constant—`vary_claims` [1, 3, 5, 7, 10], `vary_queries` [1, 2, 3, 4, 5], `vary_results` [1, 2, 3, 5, 7].

Strategic Combinations: `minimal` (1/1/1 for fast prototyping), `deep` (3/4/5 for comprehensive evidence), `broad` (10/1/2 for coverage-focused).

1.8.4 Experiment Runner CLI

The experiment runner provides command-line options:

Listing 6: Experiment runner CLI usage

```
factible-experiments run                    # Run all experiments
factible-experiments run --experiment vary_claims  # Specific group
factible-experiments run --video fossil_fuels      # Specific video
factible-experiments run --dry-run          # Preview without executing
factible-experiments analyze                # Analyze completed runs
```

1.9 Analysis and Visualization Pipeline

1.9.1 Data Loading and Enrichment

The analysis script loads all experiment runs into a pandas DataFrame, extracting configuration, metrics, and outputs from JSON files. Enrichment adds categorization including experiment type, OFAT parameter, and strategy type.

1.9.2 Generated Visualizations

The analysis pipeline generates visualizations including run summary overviews, component breakdowns showing latency and tokens by component, claims analysis with stance distribution, pipeline timing tables, scalability plots (tokens vs. time), OFAT sensitivity analysis, and strategy comparisons.

1.10 API Layer and Real-Time Streaming

1.10.1 FastAPI Setup

The API uses FastAPI with CORS middleware for frontend development and API versioning:

Listing 7: FastAPI setup

```
app = FastAPI(
    title="Factible API",
    version="0.1.0",
    description="Automated fact-checking for YouTube videos"
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "http://localhost
       :5173"],
    allow_methods=["*"], allow_headers=["*"],
)

app.include_router(fact_check_router, prefix="/api/v1")
```

1.10.2 Streaming Endpoint with SSE

The streaming endpoint (`POST /api/v1/fact-check/stream`) returns a `StreamingResponse` with `text/event-stream` media type. A callback-based progress handler collects updates into an asyncio queue, which are then yielded as SSE events.

1.10.3 Progress Events

Table 8 shows the progress events emitted during pipeline execution.

Table 8: SSE progress events

Stage	Progress	Event Name	Data Payload
1	5%	transcript_extraction	–
2	15%	transcript_complete	transcript_length
3	20%	claim_extraction	–
4	35%	claims_extracted	claims[], total_claims
5	90%	generating_report	–
6	100%	complete	result (full output)
Error	100%	error	error message

1.10.4 Request/Response Schemas

Listing 8: API request/response schemas

```python
class FactCheckRequest(BaseModel):
    """Request schema for fact-checking a YouTube video."""
    video_url: HttpUrl = Field(..., description="YouTube video
       URL")
    experiment_name: str = Field(default="default")
    max_claims: int | None = Field(default=5, ge=1, le=20)
    max_queries_per_claim: int = Field(default=2, ge=1, le=5)
    max_results_per_query: int = Field(default=3, ge=1, le=10)

class ProgressUpdate(BaseModel):
    """SSE response schema for progress updates."""
    step: str
    message: str
    progress: int  # 0-100
    data: dict | None
```

1.11 Code Quality and Engineering Practices

1.11.1 Type Safety

The codebase enforces strong type safety through multiple mechanisms. All functions use Python 3.12 type annotations with modern union syntax (`str | None`) and generic

types (`list[T]`), enabling static analysis and IDE assistance. Pydantic models provide runtime validation for all data structures crossing component boundaries, catching type mismatches and constraint violations at execution time. Additionally, static type checking via mypy with strict mode configuration runs in continuous integration, preventing type errors from reaching production. This layered approach combines the benefits of gradual typing with runtime safety nets.

1.11.2 Logging

Module-level loggers provide structured logging throughout the pipeline execution. Each component initializes a logger using Python's standard logging module, configured with consistent formatting for timestamp, level, and message. Log messages use semantic levels (INFO for normal progress, WARNING for recoverable issues, ERROR for failures) and include contextual information such as claim indices, query counts, and reliability scores. The logging output supports both real-time monitoring during development and post-hoc analysis from experiment run files.

1.11.3 Error Handling Patterns

Table 9 summarizes error handling strategies.

Table 9: Error handling patterns

Error Type	**Handling**	**Fallback**
No transcript	Return empty claims	Continue with empty pipeline
LLM failure	Retry 3x	Return empty/unclear
Search API failure	Log and continue	Empty results for query
Scraping failure	Fall back to snippet	Use Google snippet
Reliability failure	Default to "unknown"	Conservative rating
Verdict failure	Error message in summary	Unclear stance

1.11.4 Configuration Management

Configuration is managed through multiple layers optimized for different use cases. Sensitive credentials (API keys for Serper, OpenAI, Webshare) are stored in environment variables loaded from a `.env` file, following security best practices by keeping secrets out of version control. Experiment parameters (max claims, queries per claim, results per query, video corpus) are defined in YAML files for human readability and version control tracking. Model settings (temperature, token limits, retry counts) are specified as Python constants for type safety and inline documentation. Finally, API server configuration (CORS origins, port, host) uses Pydantic Settings with automatic

environment variable override support. This layered approach balances security, flexibility, and maintainability.

1.11.5 Pre-commit Hooks

Code quality is enforced automatically through pre-commit hooks that run before each git commit. The ruff linter performs fast Python linting with automatic fixes for common issues (unused imports, trailing whitespace, line length violations) and consistent code formatting following PEP 8 style guidelines. The mypy static type checker validates type annotations across the entire codebase in strict mode, catching potential type errors before runtime. Additional standard hooks validate YAML and TOML file syntax, preventing configuration errors. These automated checks ensure code quality standards are maintained consistently across all contributions without requiring manual review for style issues.

1.12 Design Patterns and Software Engineering

1.12.1 Patterns Summary

Table 10 summarizes the design patterns employed.

Table 10: Design patterns used in the implementation

Pattern	**Usage**	**Benefit**
Singleton	ExperimentTracker	Global access during pipeline
Context Manager	Tracker, Timer, Fetcher	Resource cleanup, timing
Decorator	@track_pydantic	Cross-cutting concerns
Monkey-Patching	Agent monitoring	Non-invasive instrumentation
Factory	get_model()	Model instantiation
Strategy	Adaptive filtering	Runtime algorithm selection
Observer	Progress callbacks	Decoupled progress reporting
Builder	Evidence bundle	Complex object construction

1.12.2 Async Patterns

The implementation employs four asynchronous programming patterns to maximize concurrency. First, parallel independent tasks use `asyncio.gather()` to execute multiple coroutines concurrently and collect their results. Second, thread pools handle blocking I/O operations (Selenium, WHOIS) via `asyncio.to_thread()`, preventing them from blocking the event loop. Third, queue-based producer/consumer patterns

enable streaming results from background workers to the API endpoint. Fourth, background task spawning via `asyncio.create_task()` allows fire-and-forget operations that don't block the main execution flow. These patterns combine to create a highly concurrent pipeline that efficiently utilizes system resources.

1.12.3 Key Engineering Decisions

Several architectural decisions significantly shaped the implementation. Pydantic AI was chosen over LangChain for its simpler API surface and native support for type-safe structured outputs through Pydantic models. Selenium was selected over lightweight HTTP libraries like requests specifically to handle JavaScript-rendered content that requires browser execution. The MBFC dataset provides authoritative reliability ratings based on established fact-checking methodology, offering more credible assessments than simple domain heuristics alone. The `asyncio.to_thread` pattern enables non-blocking integration of synchronous libraries (Selenium, WHOIS) within the async pipeline. Temperature 0.0 ensures reproducible experiments by eliminating sampling randomness. Finally, token estimation using character count divided by 4 provides a computationally cheap approximation for cost tracking without requiring exact tokenization.

2 Results

[Results section to be completed with experimental data]

References

[1] Rami Aly, Zhijiang Guo, Michael Schlichtkrull, James Thorne, Andreas Vlachos, Christos Christodoulopoulos, Oana Cocarascu, and Arpit Mittal. Feverous: Fact extraction and verification over unstructured and structured information. *arXiv preprint arXiv:2106.05707*, 2021. URL https://arxiv.org/abs/2106.05707.

[2] Yifan Chen et al. Local: Logical and causal fact-checking with llm based multi-agents. *OpenReview*, 2024. URL https://openreview.net/pdf/9ddca198ed6f1db6d975787e879eced0a2b0d342.pdf.

[3] Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Y. Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, et al. Rarr: Researching and revising what language models say, using language models. *arXiv preprint arXiv:2210.08726*, 2022. URL https://arxiv.org/pdf/2210.08726.pdf.

[4] Graphsignal. Llm api latency optimization explained, 2024. URL https://graphsignal.com/blog/llm-api-latency-optimization-explained/. Accessed: December 2025.

[5] Naeemul Hassan, Bill Adair, James T. Hamilton, Chengkai Li, Mark Tremayne, Jun Yang, and Cong Yu. Detecting check-worthy factual claims in presidential debates. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1835–1838, 2015. URL https://ranger.uta.edu/~cli/pubs/2015/claimbuster-cikm15-hassan.pdf.

[6] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004. URL https://wise.vub.ac.be/sites/default/files/thesis_info/design_science.pdf.

[7] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020. URL https://nlp.cs.ucl.ac.uk/publications/2020-05-retrieval-augmented-generation-for-knowledge-intensive-nlp-tasks/.

[8] Chengkai Li et al. A platform for live and on-demand monitoring of public discourse. *Proceedings of the VLDB Endowment*, 10(12):1945–1948, 2017. URL https://vldb.org/pvldb/vol10/p1945-li.pdf.

[9] Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models. *arXiv preprint arXiv:2303.08896*, 2023. URL https://arxiv.org/abs/2303.08896.

[10] Media Bias/Fact Check. Methodology - media bias/fact check, 2024. URL https://mediabiasfactcheck.com/methodology/. Accessed: December 2025.

[11] Briony J. Oates. *Researching Information Systems and Computing*. SAGE Publications, 2006.

[12] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2024. URL https://arxiv.org/pdf/2303.08774.pdf.

[13] OpenAI. Latency optimization - openai platform documentation, 2024. URL https://platform.openai.com/docs/guides/latency-optimization. Accessed: December 2025.

[14] Sebastian Raschka. Understanding the 4 main approaches to llm evaluation (from scratch). *Ahead of AI*, 2025. URL https://magazine.sebastianraschka.com/p/llm-evaluation-4-approaches.

[15] Sebastian Ruder. The evolving landscape of llm evaluation. *NLP Newsletter*, 2025. URL https://www.ruder.io/the-evolving-landscape-of-llm-evaluation/.

[16] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. Fever: A large-scale dataset for fact extraction and verification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 809–819, 2018. URL https://aclanthology.org/N18-1074/.

[17] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023. URL https://arxiv.org/abs/2308.08155.

[18] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023. URL https://arxiv.org/abs/2210.03629.

[19] Xuan Zhang, Wei Wei, Yuxiao Wen, Yang Shi, and Bowen Zou. Factagent: Towards agentic multi-hop fact-checking via large language models. *arXiv preprint arXiv:2506.17878*, 2025. URL https://arxiv.org/abs/2506.17878. Available at: https://github.com/HySonLab/FactAgent.