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

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 [16]. 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 [18]. 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 the following stages:

1. **Transcriptor**: Extracts video transcript via YouTube Transcript API, preserving timestamped segments for claim localization, with automatic fallback to proxy service when rate-limited.

2. **Claim Extractor** (LLM Agent): Employs thesis-first reasoning to infer the video's central argument, extracts factual and verifiable claims with importance scoring, and performs post-processing through fuzzy matching to locate claims in transcript positions.

3. **Query Generator** (LLM Agent): Generates diverse search queries across four types (direct, alternative, source-seeking, and contextual), assigns priority scores based on evidence likelihood, and filters by priority threshold and budget constraints.

4. **Online Search**: Executes a multi-step pipeline including Google Search via Serper API, website reliability assessment using Media Bias/Fact Check (MBFC) data combined with heuristics [10], content fetching via Selenium WebDriver, and LLM-based evidence extraction with stance classification.

5. **Output Generator** (LLM Agent): Builds evidence bundles grouped by stance, generates verdicts with confidence levels, calculates algorithmic evidence quality scores, and maps claims to video timestamps for user interface navigation.

Figure 1 illustrates the complete pipeline architecture with data flow and parallelization points.

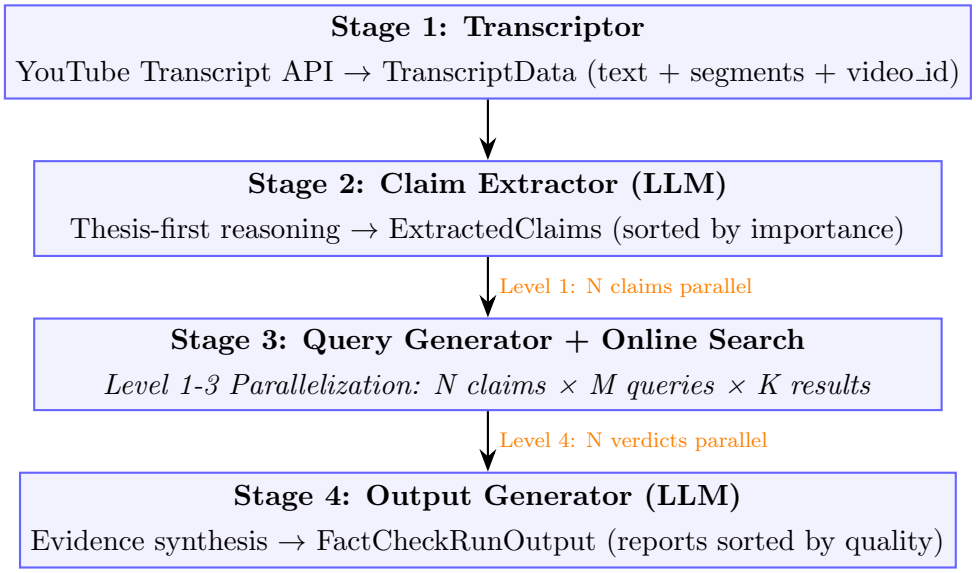

Figure 1: High-level pipeline architecture showing the four main stages and parallelization levels.

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 enables independent testing, model swapping, metrics collection, and clear interface contracts:

Listing 1: Component directory structure

```
component/
    __init__.py           # Public exports
    component_name.py     # Main logic with @track_pydantic
       decorator
    schemas.py            # Pydantic models for input/output
```

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	≥1.0.0	Agent orchestration, structured outputs
Data Validation	Pydantic	≥2.0.0	Schema definitions, runtime validation
Web Framework	FastAPI	≥0.115.0	REST API with SSE streaming
HTTP Server	Uvicorn	≥0.32.0	High-performance ASGI server
Async HTTP	httpx	≥0.28.1	Async HTTP client
Web Scraping	Selenium	≥4.15.2	JavaScript-rendered content extraction
YouTube	youtube-transcript-api	≥1.2.2	Transcript extraction
Domain Info	python-whois	≥0.8.0	Domain age lookup
CLI	Typer	≥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 2: 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")

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 `map_char_position_to_timestamp()` function enables the system to locate where in the video each claim originates by iterating through segments and tracking character positions:

Listing 3: Timestamp mapping algorithm

```
def map_char_position_to_timestamp(
    char_position: int, transcript_data: TranscriptData
) -> dict | None:
    """Map character position in transcript to video timestamp.
        """
    current_char = 0
    for idx, segment in enumerate(transcript_data.segments):
        segment_text = segment.text + " "
        segment_length = len(segment_text)
        if current_char <= char_position < current_char +
            segment_length:
            return {
                "segment_index": idx,
                "start": segment.start,
                "duration": segment.duration,
            }
        current_char += segment_length
    return {"start": transcript_data.segments[-1].start, ...}
```

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:

Listing 4: Claim extractor model settings

```
CLAIM_EXTRACTOR_MODEL_SETTINGS: ModelSettings = {
    "temperature": 0.0,     # Deterministic outputs for
       reproducibility
    "max_tokens": 1200,     # Limit response length to control
       latency
}
retries = 3  # Automatic retries on LLM failures
```

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 and `@agent.instructions` decorator to inject runtime constraints:

Listing 5: Dynamic instruction injection

```
class ClaimExtractorDeps(BaseModel):
    max_claims: int | None = None

```

```python
@agent.instructions
def _limit_instruction(ctx: RunContext[ClaimExtractorDeps]) -> str:
    """Dynamically inject max_claims constraint into system prompt."""
    max_claims = ctx.deps.max_claims
    if max_claims is not None and max_claims >= 0:
        return f"Never output more than {max_claims} claims."
    return "Output only the highest-impact claims."
```

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 6: 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 [15] 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 [17]. 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.

Output Schema Listing 7: Query generator output schemas

```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. 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 SelfCheckGPT, 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 [18] and LoCal [2], where decomposing, reasoning, and evaluating agents iteratively refine answers and outperform baselines.

Step 1: Google Search (Serper API) The Google search client wraps the Serper API for asynchronous search execution:

Listing 8: Google search client implementation

```
class GoogleSearchClient:
    """Wrapper around the Google Serper search API."""

    async def search(self, query: str, limit: int = 5) -> list[
        GoogleSearchHit]:
        """Execute async search with connection reuse."""
        client = await self._get_client()
        response = await client.post(
            "https://google.serper.dev/search",
            headers={"X-API-KEY": self.api_key},
            json={"q": query, "num": min(limit, 10)},
        )
        raw_results = response.json().get("organic", [])
        return [GoogleSearchHit(title=r["title"], url=r["link"]
            ],
                                snippet=r["snippet"])
                for r in raw_results[:limit]]
```

Step 2: Website Reliability Assessment The reliability checker uses a multi-factor scoring system with first-match priority:

Factor 1: 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 $\rightarrow$ 0.85, medium $\rightarrow$ 0.60, low $\rightarrow$ 0.30, very low $\rightarrow$ 0.15.

Factor 2: TLD Reputation — High-trust top-level domains (`.gov`, `.edu`, `.int`) receive a 0.90 score.

Factor 3: Domain Age via WHOIS — Domains ≥10 years old receive a +0.10 bonus, while domains <1 year old receive a –0.15 penalty.

Listing 9: Reliability assessment output schema

```python
class SiteReliability(BaseModel):
    """Assessment of a website's reliability."""
    rating: Literal["high", "medium", "low", "unknown"]
    score: float = Field(ge=0.0, le=1.0)
    reasons: list[str] = Field(default_factory=list)
    bias: str | None = None  # Political bias if available
```

Step 3: Content Fetching (Selenium) Content fetching uses Selenium WebDriver in headless Chrome mode with smart wait strategies to handle JavaScript-heavy sites:

Listing 10: Selenium content fetcher configuration

```python
class SeleniumContentFetcher:
    def __init__(self, headless: bool = True, wait_timeout: int
        = 12,
                 page_load_timeout: int = 20, max_characters:
                    int = 8000):
        chrome_options = Options()
        if headless:
            chrome_options.add_argument("--headless=new")
        chrome_options.add_argument("--disable-gpu")
        chrome_options.add_argument("--no-sandbox")
        chrome_options.add_argument("--blink-settings=
           imagesEnabled=false")
```

The fetcher first attempts a quick extraction of paragraph elements; if content is insufficient (<100 characters), it waits for JavaScript rendering before retrying. Content is trimmed to a maximum of 8,000 characters. Async integration wraps blocking Selenium calls in `asyncio.to_thread()` for non-blocking operation.

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:

Listing 11: Adaptive credibility filtering algorithm

```
async def search_online_async(claim: str, query: str, limit:
    int = 5,
                                min_credibility: str = "medium")
                                    -> SearchResults:
    # Step 1: Fetch initial batch (2x limit for filtering
        headroom)
    batch_1 = await search_client.search(query, limit=limit *
        2)

    # Step 2: Quality check - if >50% unreliable, fetch more
    reliable_count = sum(1 for _, score, _ in all_assessed if
        score >= min_score)
    if reliable_count < len(all_assessed) / 2:
        batch_2 = await search_client.search(query, limit=limit
            * 2)

    # Step 3: Intelligent filtering with minimum guarantee
    all_assessed.sort(key=lambda x: x[1], reverse=True)
    if len(reliable) >= min_guaranteed_sources:
        filtered_results = reliable[:limit * 2]
    else:
        needed = min_guaranteed_sources - len(reliable)
        filtered_results = reliable + unreliable[:needed]
```

Additional filtering includes stance filtering (removing unclear results if >50% have definitive stances) and URL deduplication using hash sets.

1.3.5 Output Generator

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

Two-Step Process **Step 1: Build Evidence Bundle (Algorithmic)** — The system groups, deduplicates, and sorts evidence by stance. Within each stance group, sources are sorted by reliability (high first), then by score, with alphabetic tie-breaking.

Step 2: Generate Verdict (LLM) — Evidence is formatted for the LLM prompt with stance, source count, reliability, and summaries. The system prompt instructs concise synthesis, naming sources only when clarifying contrasting perspectives, and explicitly naming supporting and refuting sources when evidence conflicts.

Evidence Quality Score (Algorithmic) The quality score is calculated algorithmically without LLM involvement for consistency and speed:

Listing 12: Evidence quality score calculation

```
def _calculate_evidence_quality_score(bundle: ClaimEvidenceBundle) -> float:
    """Quality score from 0.0 (no evidence) to 1.0 (high-quality evidence)."""
    if bundle.total_sources == 0:
        return 0.0

    score = 0.3  # Base score for having any evidence

    # Bonus for actionable stances (supports/refutes/mixed)
    actionable_count = sum(len(sources) for stance, sources
                           in bundle.stance_groups.items()
                           if stance in ("supports", "refutes",
                              "mixed"))
    score += 0.3 * min(1.0, actionable_count / 3.0)

    # Bonus for high/medium reliability sources
    high_reliability_count = sum(1 for sources in bundle.
       stance_groups.values()
                                 for source in sources
                                 if source.reliability.rating
                                    in ("high", "medium"))
    score += 0.4 * min(1.0, high_reliability_count / 3.0)

    return min(1.0, score)
```

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 `asyncio.gather()`:

Listing 13: Parallel verdict generation

```
async def generate_run_output(...) -> FactCheckRunOutput:
    """Generate all verdict reports in parallel."""
    report_tasks = [
        _generate_report_with_context(idx, claim, results)
        for idx, (claim, results) in enumerate(claim_results,
           1)
    ]
    reports = await asyncio.gather(*report_tasks)
    sorted_reports = sorted(reports,
                            key=lambda r: r.
                               evidence_quality_score,
                            reverse=True)
    return FactCheckRunOutput(claim_reports=sorted_reports,
       ...)
```

Output Schema

Listing 14: Output generator schemas

```
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:

Listing 15: Model configuration abstraction

```
class ModelConfig(BaseModel):
    """Configuration for a language model including provider
        and pricing."""
    provider: str  # "openai" or "ollama"
    model_name: str
    price_input_per_million: float
    price_output_per_million: float
    context_window: int

class ModelChoice(Enum):
    """Available language models with their configurations."""
    OPENAI_GPT4O_MINI = ModelConfig(
        provider="openai", model_name="gpt-4o-mini",
        price_input_per_million=0.150,
        price_output_per_million=0.600, context_window=128_000)
    # ... additional models
```

1.4.2 Model Instantiation with Caching

Ollama model instances are cached using `@lru_cache` for connection reuse:

Listing 16: Model instantiation with caching

```
@lru_cache(maxsize=None)
def _get_ollama_model(model_name: str) -> OpenAIChatModel:
```

```
    """Get cached Ollama model instance for connection reuse.""
    "
    provider = OllamaProvider(base_url="http://127.0.0.1:11434/
    v1")
    return OpenAIChatModel(model_name=model_name, provider=
    provider)
```

1.4.3 Per-Component Model Configuration

A centralized configuration enables easy model swapping, per-component optimization, experiment flexibility, and cost tracking:

Listing 17: Per-component model configuration

```
CLAIM_EXTRACTOR_MODEL = ModelChoice.OPENAI_GPT4O_MINI
QUERY_GENERATOR_MODEL = ModelChoice.OPENAI_GPT4O_MINI
EVIDENCE_EXTRACTOR_MODEL = ModelChoice.OPENAI_GPT4O_MINI
OUTPUT_GENERATOR_MODEL = ModelChoice.OPENAI_GPT4O_MINI
```

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. 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]. Additional techniques such as defining clear output boundaries, setting token limits, and adjusting sampling temperature can reduce generated tokens [4], and caching prompts allows reuse of computations for identical prefixes. Choosing smaller model weights yields faster inference speeds.

1.5.1 Process Tokens Faster: Model Selection

The default model (gpt-4o-mini) balances speed, cost-effectiveness, and 128K context. Local Ollama models provide budget and offline options.

1.5.2 Generate Fewer Tokens: Output Constraints

Each component has carefully tuned `max_tokens` limits: claims limited to 40 words, context to 20 words, evidence summaries to 1–2 sentences.

1.5.3 Use Fewer Input Tokens: Content Trimming

Web content is trimmed to 6,000–8,000 characters; evidence prompts include only snippets and page content (not raw HTML).

1.5.4 Make Fewer Requests: Combined Operations

Single LLM calls per component with no multi-turn conversations; batch generation produces all queries in one call; single synthesis calls with all evidence per claim.

1.5.5 Parallelize: 4-Level Async Architecture

The system implements four levels of parallelization:

- **Level 1**: Process N claims in parallel using `asyncio.gather`
- **Level 2**: For each claim, process M queries in parallel
- **Level 3**: For each query, process K results in parallel (reliability assessment, content fetching, and evidence extraction)
- **Level 4**: Generate all N verdicts in parallel

Blocking I/O operations (Selenium, WHOIS) are wrapped in `asyncio.to_thread()` to avoid blocking the event loop.

1.5.6 Real-Time Streaming

Server-Sent Events (SSE) provide progressive updates as the pipeline executes, with progress stages from 5% (transcript extraction) through 100% (complete). Claims are shown immediately after extraction, before search completes.

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 `output_type`, ensuring type safety, automatic parsing, error handling with automatic retries, IDE support with full autocomplete, and serialization via `model_dump()`.

Listing 18: Pydantic AI agent pattern

```
agent = Agent(
    model=get_model(MODEL_CHOICE),
    output_type=ExtractedClaims,  # Pydantic model
    deps_type=ClaimExtractorDeps,  # Runtime dependencies
    model_settings={"temperature": 0.0, "max_tokens": 1200},
    system_prompt="...",
    retries=3,
)
```

1.6.2 Schema Design Principles

The schema design follows five principles: flat structures to avoid deep nesting; optional fields using `str | None`; `Literal` types for constraints; descriptive field names with `Field` descriptions; and validation constraints such as `Field(ge=0.0, le=1.0)`.

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 [14]. Practical evaluation strategies include multiple-choice benchmarks, verifiers, public leaderboards, and LLM judges, each with distinct trade-offs [13]. Our evaluation framework therefore combines quantitative metrics with qualitative assessments using external datasets such as FEVER [15] 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:

Listing 19: Experiment tracker implementation

```
class ExperimentTracker:
    """Track experiments with structured output to disk."""
    _current: Optional["ExperimentTracker"] = None  # Singleton

    def __init__(self, component: str, experiment_name: str,
       config: dict):
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        self.run_id = f"{timestamp}_{component}_{
           experiment_name}"
        self.run_dir = base_dir / self.run_id
        self.run_dir.mkdir(parents=True, exist_ok=True)

        self.timing_data: dict[str, float] = {}
        self.metrics: dict[str, Any] = {}
        self.pydantic_calls: list[dict] = []

    def __enter__(self):
        ExperimentTracker.set_current(self)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.save()  # Auto-save on exit
        ExperimentTracker.set_current(None)
```

A `timer` context manager automatically logs timing to the current tracker:

Listing 20: Timer context manager

```
@contextmanager
def timer(label: str):
    """Time a code block and auto-log to current tracker."""
    start = time.time()
```

```
    try:
        yield
    finally:
        duration = time.time() - start
        tracker = ExperimentTracker.get_current()
        if tracker:
            tracker.log_timing(label, duration)
```

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 including total time, per-step timing breakdown, LLM call count, total cost, and success status.

1.7.4 Output Directory Structure

Each experiment run creates a directory with:

Listing 21: Experiment output directory structure

```
factible/experiments/runs/
  20251208_123456_end_to_end_EXPERIMENT_NAME/
    config.json          # Run ID, parameters, model config
    llm_calls.json       # All Pydantic AI calls with full I/O
    outputs.json         # Final results (claims, reports)
    metrics.json         # Aggregated timing, cost, success
    transcript.txt       # Original transcript for reference
```

1.8 Experiment Configuration and Runner

1.8.1 YAML Configuration

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

Listing 22: Example YAML configuration

```
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 23: 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 24: 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 25: API request/response schemas

```
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

All functions use Python 3.12 type annotations (`str | None`, `list[T]`). Pydantic models provide runtime validation for all data structures. Static type checking uses mypy with strict mode.

1.11.2 Logging

Module-level loggers with emoji indicators enable quick scanning of logs:

Listing 26: Logging with emoji indicators

```
_logger.info("Processing %d claims in PARALLEL", len(claims))
_logger.warning(">50% sources unreliable -> Fetching additional
    batch")
_logger.error("Failed to fetch content: %s", exc)
```

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 environment variables (`.env`) for API keys, YAML files for experiment parameters, Python constants for model settings, and Pydantic Settings for API configuration.

1.11.5 Pre-commit Hooks

Code quality is enforced through pre-commit hooks: ruff for linting with auto-fix and formatting, mypy for static type checking, and standard hooks for YAML/TOML validation.

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 uses four async patterns: parallel independent tasks via `asyncio.gather()`, thread pools for blocking I/O via `asyncio.to_thread()`, queue-based producer/consumer patterns, and background task spawning via `asyncio.create_task()`.

1.12.3 Key Engineering Decisions

Key engineering decisions include: Pydantic AI over LangChain for simpler, type-safe structured outputs; Selenium over requests for JavaScript-rendered content support; MBFC dataset for authoritative reliability data; `asyncio.to_thread` for non-blocking I/O with synchronous libraries; temperature 0.0 for reproducible experiments; and token estimation (len//4) as a good-enough approximation for cost tracking.

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] 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.

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

[15] 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/.

[16] 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.

[17] 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.

[18] Wei Zhang et al. Towards robust fact-checking: A multi-agent system with advanced evidence retrieval. *arXiv preprint arXiv:2506.17878*, 2025. URL https://arxiv.org/abs/2506.17878.