Pre-Execution – High-Level System Explanation

1. Architecture and System Design

The system is built as a modular pipeline with three clearly separate stages: ingestion, classification, and storage. This design supports adaptability, maintainability, and independent scaling of each stage.

Key Components (from implementation)

- Ingestion Layer: The ScraperEngine uses Playwright with a headless Chromium browser to navigate and render target pages before extracting content.
- Classification Layer: The HeadlineClassifier uses the HuggingFace facebook/bart-large-mnli transformer model for zero-shot topic classification.
- Storage Layer: The Database class uses SQLite with connection pooling and batch inserts for efficient structured storage.
- Configuration System: A JSON file (config.json) defines the websites, CSS selectors, and candidate categories, allowing the system to run without changing the code.
- Orchestration: The scraper.py script ties together configuration loading, scraping, classification, and database operations.

Flow Structure

The system's pipeline is illustrated in the attached Lucidchart diagram (**Link**)**.**
In summary, the process flows as follow:
Config.json → **ScraperEngine** (*Playwright headless chromium*) → **HeadlineClassifier** (*Transformer-based-zero-shot-model*) → **Database** (*SQLite*)**.**

Scalability Preference and Rationale

The preferred scaling method for this system is to use a message queue architecture with containerized workers.

- A message queue (such as RabbitMQ or AWS SQS) decouples the ingestion stage from classification and storage.
- Each stage can be scaled independently. For example, adding more scraper workers without changing the classification layer.
- The queue acts as a buffer, applying backpressure when downstream services are slower, preventing overload.
- Built-in retry logic and dead-letter queues (DLQ) increase reliability by safely handling failures.

- Horizontal scaling is straightforward: add more worker containers to process jobs in parallel.
- Per-domain rate limits can be enforced at the worker level to avoid IP blocks and rate limits.
- The design remains flexible to integrate a pub/sub model later if multiple services need the same data, or to use serverless functions for specific stateless tasks such as classification bursts. Serverless functions are ideal for elastic, bursty workloads, allowing classification surges to be handled without maintaining idle infrastructure during low-traffic periods.

Example of scaled setup
In production, scraper.py would become a queue producer, pushing {URL, selector, categories} jobs into the message queue. Multiple worker containers would run ScraperEngine to process jobs concurrently, classify headlines, and store results in batches.

2. Web Scraping Approach

Preferred Method for Dynamic HTML Content
The system uses a rendering engine (Playwright with Chromium) combined with CSS selectors to handle dynamic, JavaScript-rendered content. This ensures that the DOM is fully rendered before data extraction. CSS selectors are chosen over XPath because they are generally faster in browser contexts, easier to read, and less prone to breaking when the page structure changes slightly.

Dynamic Content Handling

- Playwright launches Chromium to render the full DOM, including JavaScript-driven elements.
- The scraper waits for page load using wait_until="domcontentloaded" and then waits for the target selector to appear.
- Primary selectors come from config.json. If not found, fallback selectors (h1, h2, a) are used to improve resilience.

Hybrid Playwright + ZenRows Option
In production, the system could combine Playwright with ZenRows to gain both flexibility and stronger anti-bot handling.

- Normal Operation: Use Playwright locally for full control over navigation, interaction, and debugging, keeping the flow identical to the current implementation.
- High-Risk Targets or Blocked Requests: Switch to ZenRows for those specific URLs. ZenRows handles proxy rotation, CAPTCHA solving, and advanced anti-bot detection bypass automatically.

- Configuration-Driven Switching: A flag in config.json could indicate which sites should use ZenRows instead of local Playwright rendering, allowing rapid adaptation without code changes.
- Benefits: Maintain Playwright's deep control for most sources while leveraging ZenRows' infrastructure when facing aggressive bot protections. This hybrid approach also enables scaling without re-engineering the scraping logic.

Limitations and Planned Improvements

- Rate Limits and IP Blocking: Currently not an issue in the happy flow due to low volume. At scale, a token-bucket rate limiter, randomized delays, proxy rotation, and concurrency caps per domain will be added.
- CAPTCHAs: Not triggered in the current flow. At scale, solutions may include human-like browsing patterns, residential proxies, or CAPTCHA-solving APIs (ZenRows could be used here automatically).
- Structural Drift: Selectors are stored in the configuration file for quick updates. Future improvement: automated selector testing and alerts when extraction volume changes unexpectedly.

3. Data Classification Strategy

Chosen Model

The classifier uses HuggingFace facebook/bart-large-mnli, a transformer-based model trained on the Multi-Genre Natural Language Inference (MNLI) dataset. This enables zero-shot classification by reframing labels as hypotheses, allowing classification into categories not seen during training.

Why It Fits This Task

- Handles short-form text like video titles effectively.
- Works immediately without fine-tuning, making it ideal for a dynamic label set defined in configuration.
- Runs locally, avoiding API quota restrictions and external dependencies.
- Can adapt to new categories simply by editing config.json.

Alternative Models

- DeBERTa-MNLI or RoBERTa-MNLI: Similar zero-shot capabilities, potentially with different performance trade-offs.
- GPT models via API: Highly flexible but incur usage quotas, cost, and potential privacy issues.

- spaCy textcat: Strong when there is labeled training data for a fixed set of categories, but weaker for open-topic classification because it cannot classify unseen categories without retraining.

Prompt Engineering, Context, and Efficiency

- Managing context: Keep category labels concise, unambiguous, and domain-relevant to avoid confusion. Avoid synonyms in the same set. For example, "Marketing Strategy" vs "Growth Strategy".
- Reducing cost: Use the local BART-MNLI model for most classifications; only route low-confidence predictions to more expensive API-based models.
  This cuts label count per pass, improves accuracy and latency.
- Maintaining efficiency: Batch multiple headlines in a single inference call, truncate excessively long text, and cache results for duplicates to avoid reprocessing.
- Confidence thresholding ensures that only predictions above a set certainty level are stored without review.

Future improvements - managing context and reduce cost:

- Context management: keep candidate labels concise and unambiguous. use a stable hypothesis template ("This title is about {label}."). For breadth with precision, apply a hierarchical pass (coarse then fine labels) and optional one-line label hints. Titles are normalized (remove emojis/URLs, trim) to protect token budget and stabilize scores.
- Cost control: run a local zero-shot model for all items and escalate only low-confidence cases to a stronger API model. Enable caching via a hash of the normalized title to avoid repeat work. Add light rules for obviously domain-specific terms to cut unnecessary API calls.
- Efficiency: perform batched inference (16–64 items), keep label sets small per pass, and use an async worker pool. Monitor throughput, low-confidence rate, and label drift; auto-tune batch size and thresholds as needed.

4. Data Modeling and Storage

Database Type Preference

The task requires structured, relational storage with schema enforcement and ACID guarantees, making an SQL database the preferred choice. SQLite is used in the prototype for its simplicity, portability, and developer-friendliness. For production scalability, PostgreSQL is the natural next step because it supports concurrent writes, advanced indexing, and analytical queries.

Data Model

- Current schema: one headlines table with source, headline, category, raw_label, confidence, and scraped_at.
- Future enhancements: normalization into sources and categories tables, adding indexes for frequent query patterns such as (source, scraped_at) and (category, scraped_at).

Why This Model Fits the Task

This normalized, relational model ensures structured, queryable storage with enforced schema integrity. It supports downstream analytics such as trend detection, time-series analysis, and cross-source comparisons while remaining easy to extend if new attributes are needed.

Performance Optimizations in Current Implementation

- Connection pooling to reuse open database connections.
- Batch inserts of 50 records to minimize commit overhead.

5. Future Scalability & Innovation

- Domain-Aware Scheduling – Adjust scraping frequency per site's update rate to save resources and improve efficiency.
- Distributed Headless Browser Pool – Deploy Playwright workers across multiple regions/containers for higher concurrency and lower latency.
- Confidence-Based Dual Model Classification – Keep BART-MNLI for most cases, route low-confidence cases to a more powerful fallback model like GPT.
- Hybrid SQL + NoSQL Storage – PostgreSQL for structured storage, Elasticsearch for full-text search and analytics.
- Real-Time Stream Processing – Move from batch to streaming pipelines using Kafka or AWS Kinesis for near-instant classification and storage.
- Content Deduplication – Hash headlines before classification to skip previously seen items.
- Adaptive Scraping Strategy – Automatically switch between Playwright, ZenRows, or other methods based on site health metrics and block detection.
- Machine Learning for Selector Recovery – Train a model to locate target elements visually or semantically when selectors break.
- Online Learning for Classification – Incorporate human feedback to continuously improve the local classifier without full retraining.

- Explainable Classification Reports – Store multiple top predictions and token importance scores to make classification decisions more transparent.