ChatGPT

Roadmap for a Multi-Engine News/Text Analysis Pipeline in Go

Introduction and Goals

This roadmap outlines a comprehensive plan to develop a **multi-engine text analysis pipeline** in Go. The goal is to build a system that can ingest news articles or arbitrary text (open-ended content), process and analyze the content through a sequence of specialized "engines," and finally produce a structured report of the findings. Key considerations include: working in **pure Go** (no Python dependencies), supporting web content ingestion (web search or file reading), and designing with future **multi-language** support in mind. Each engine in the pipeline will have a **single responsibility (SRP)** to keep modules simple and maintainable. The engines will be chained in a logical order, passing structured data between them (Engine-to-Engine data dependencies) to incrementally build up analysis results. Finally, a dedicated **reporting phase** will orchestrate specific engines to compile all analysis outputs into a comprehensive report – the "cherry on the cake" of the pipeline's results.

Requirements and Considerations

- **Open-Ended Content:** The system must handle a wide variety of news or text content. This means the analysis engines (e.g., summarization, entity extraction) should be domain-agnostic and robust to different topics or writing styles. Prompts for LLM-based engines should be crafted to handle broad inputs.
- **Pure Go Implementation:** All components should be implemented in Go (no reliance on Python scripts or external pipelines). This involves using Go libraries for NLP tasks and calling external AI services (like OpenAI) via Go SDKs. For example, OpenAI provides an official Go SDK (released in 2024) for calling its APIs [1]. We will leverage such Go libraries or REST API calls directly, ensuring the core orchestration and data processing is done in Go.
- **Web Search and File Input:** The pipeline should ingest content either from web sources (e.g., news websites or search results) or local files. This requires an ingestion engine capable of fetching HTML pages or reading documents. We must account for network variability (timeouts, errors) and possibly use caching for repeated content. Reading from files should support common formats (plain text, maybe HTML or PDF with appropriate parsing if needed).
- **Multi-Language Support:** The system should be designed to handle multiple languages in the future. Initially, we might focus on English content, but the architecture will include language detection and translation steps. This ensures that if input text is in another language (French, Spanish, etc.), we can detect it and either translate it to a target language (e.g., English) for analysis or process it directly if our analysis engines can handle it. Go has libraries like **lingua-go** for language detection (which supports short or long text and even mixed-language input) [2]. We will integrate such a library to identify the language of each article. For translation, we can either use an external API or prompt an LLM to translate the text as a preprocessing step. The design will keep the translation logic modular so that it can be added or adjusted easily in the future.

- **Structured Outputs and Data Flow:** Each engine will output data in a well-defined structured format (Go structs), which will serve as input to subsequent engines. Using structured data ensures clarity in what each engine produces and consumes, and makes the engine chaining explicit. We will define Go types for these outputs (e.g., `NormalizedArticle`, `Summary`, `Entities`, `Classification`, `Report`) and make sure engines share data via these types rather than through unstructured text wherever possible. This not only helps with type-safety but also makes testing easier (we can validate the fields of these structs).

With these requirements in mind, we proceed to the architecture and task breakdown.

Architecture Overview

The system is organized as a **pipeline of engines** connected in sequence, with some parallel branches where appropriate. Each engine addresses one aspect of the processing, following SRP (Single Responsibility Principle) – meaning each engine has one well-defined job [3]. Engines communicate by passing their structured outputs to the next engine(s) in the chain. We will implement this pipeline possibly using Go's concurrency features (goroutines and channels) to maximize performance where tasks can be parallelized, while maintaining a clear data flow.

Planned Engines and Workflow:
1. **Ingestion Engine** – Fetches or reads raw content. If from web, it might take a URL or query, retrieve the article (using an HTTP client or search API), and output the raw text or HTML plus metadata (e.g., source info). If from file, it reads the file content.
2. **Normalization Engine** – Cleans and normalizes the text. This includes stripping HTML tags (if any), removing boilerplate or noise (ads, navigation text), normalizing Unicode characters, fixing encodings, and standardizing whitespace and punctuation. The result is a cleaned, canonical text form ready for analysis. *Normalization* is critical: it transforms text into a standard form (e.g., converting "gooood" or "gud" to "good") [4], which helps ensure consistent downstream analysis.
3. **Language Detection Engine** – Determines the language of the normalized text. If content is not in the desired language (e.g., not English), this engine can trigger a translation step. We plan to use a library like `lingua-go` for accurate detection [2]. This engine outputs a language code (ISO language tag) and possibly a confidence score.
4. **Translation Engine (Optional, Conditional)** – If the article is in a foreign language, this engine will produce a translated version of the text (likely into English, if we standardize on English for analysis). We can integrate with an external translation API or use an LLM (by prompting it to translate). This engine's output is the text in the target language. (If the text is already in English or a supported analysis language, this engine can be skipped.)
5. **Summarization Engine** – Generates a concise summary of the article. Likely implemented via an LLM call (e.g., using GPT-4 through OpenAI's Go SDK) with a carefully crafted prompt instructing the model to summarize the given text. The output is a `Summary` struct containing a short summary (a few sentences or a paragraph) and possibly key bullet points. We will ensure the prompt limits the length and focuses on factual accuracy.
6. **Entity Extraction Engine** – Identifies key entities in the text, such as people, organizations, locations, and dates. We have options here: use an NLP library (e.g., the Go `prose` library for English NER, or spaGO) or use an LLM with a prompt to extract entities. Given multi-language needs and flexibility, using an LLM might be effective: we can prompt the model to output JSON with lists of entities. For example, ask the model to list `"people", "organizations", "locations", "dates"` from the text and return a JSON object.

The output will be an `Entities` struct (with fields like `People []string`, `Organizations []string`, etc.).

7. **Topic Classification Engine** – Determines the main topic or category of the article (e.g. Politics, Sports, Technology, Finance, etc.). This provides higher-level context. Implementation can be via a prompt to an LLM: *"What is the primary topic of this article?"*, possibly constrained to a set of categories if we have a predefined taxonomy. Alternatively, a simple keyword-based or ML classifier could be used, but an LLM will likely handle nuanced topics better (and adapt to multi-language inputs). The output is a `Classification` struct (e.g. with fields like `PrimaryCategory string` and maybe `SubCategory string`).

8. **(Optional) Sentiment Analysis Engine** – (If needed for reporting) Analyzes the tone or sentiment of the article (positive, negative, neutral, or perhaps emotional tone). This can be done with a lexicon-based library (e.g., `govader` for English sentiment [5]) or by asking an LLM *"What is the tone of this article?"*. Output could be a `Sentiment` struct (with fields like `Polarity` or `Score`). This is not strictly required, but if the report should include the article's sentiment or tone, it's a useful addition. 9. **Reporting Engine** – This is the final phase that compiles all the results into a coherent report. It takes inputs from all previous analysis engines (summary, entities, category, etc.) and produces a human-readable report. The report can be structured (for example, a markdown or HTML document) summarizing the article and listing key findings. We might implement this by prompting an LLM to generate a nicely formatted report given the collected data (for a more natural narrative), or we could programmatically construct the report from templates. Because this phase is the culmination of all analysis, it's treated as a separate **standalone phase**, orchestrated after all other engines have completed. The reporting phase is the "cherry on the cake," ensuring the user sees a polished output after all the heavy analysis is done.

Engine Chaining and Data Dependencies:

The order above reflects logical dependencies. Early engines (ingestion, normalization, language detection/translation) prepare the text for analysis. Once we have a clean English (or chosen target language) text, we can branch out to multiple analysis engines in parallel, since summarization, entity extraction, classification, and sentiment analysis all depend only on the processed text. They do not depend on each other's results (for example, summarization doesn't strictly need the entities or vice versa). This means after text normalization/translation, we can **fan-out** to run several engines concurrently to save time. In Go, a pipeline can have multiple goroutines reading from the same input channel (fan-out) and then their results can be merged (fan-in) [6]. We will utilize this pattern: one stage produces the final text, then multiple goroutines perform analysis, and finally the results are gathered for reporting. If concurrency is used, we must manage synchronization (e.g., using `sync.WaitGroup` or channels to collect outputs). The orchestrator (main controller) will trigger the **Reporting Engine** only after all required analysis results are ready, ensuring nothing is missed in the final report.

To manage data between engines, we will define data structures for each engine's output and potentially use an in-memory context or state object that accumulates results. For instance, the orchestrator could maintain a struct like:

```
type ArticleAnalysis struct {
    Source          string
    OriginalText    string
    Language        string
    NormalizedText  string
```

```
    TranslatedText   string  // if applicable
    Summary          string
    Entities         Entities  // custom struct with lists
    Category         string
    Sentiment        string  // or a custom type
    Report           string  // final report text
}
```

This is one way (a composite struct) to hold everything. However, to keep engines loosely coupled, we'll more likely use distinct output structs per engine and only combine at the end. Each engine function can return its own struct, and the orchestrator will pass what's needed to the next engine. For example, `NormalizationEngine` returns a `NormalizedArticle` struct with cleaned text (and perhaps detected title, author if parsed from HTML), then `LanguageDetectionEngine` consumes `NormalizedArticle.Content` and returns a `LanguageResult` (with language code and maybe probability). The orchestrator then decides if a `TranslationEngine` is invoked (based on `LanguageResult.lang`). When translation is done (or skipped), we then supply the text to all downstream analyzers.

By planning structured data exchange, we ensure each engine **shares data via well-defined contracts**. This will make our code easier to test and reason about, since we can feed a known struct to any engine and verify the output struct.

Next, we break down the development tasks engine by engine, including testing strategy for each.

Task Breakdown and Implementation Plan

Below is a **detailed roadmap** divided into concrete tasks. Each task includes development steps and testing strategy. The tasks are ordered logically in phases: design, implementing each engine in the pipeline, then orchestration and final touches like prompt management and optimization.

1. Project Setup and High-Level Design

Development Tasks:
- Set up a Go module for the project. Decide on package structure (e.g., a package for each engine or all under one `analysis` package with sub-packages). Initialize version control.
- Define a common `Engine` interface (if useful) that all engines implement. For example, `type Engine interface { Run(input any) (output any, err error) }` or a more specific generic interface. This is optional but can formalize how engines are invoked.
- Design and document the data flow between engines. Create a flow diagram or outline illustrating engine chaining and parallelization points. Ensure that engines and their data dependencies are identified (e.g., SummarizationEngine needs normalized text (and possibly language info to know what language it's in), ReportingEngine needs outputs from Summary, Entities, etc.).
- **Define StructuredOutput types for each engine** in Go. For example:

```
type NormalizedArticle struct { Content string; Title string; /* ... */ }
type LanguageResult struct { LangCode string; Confidence float64; }
type Summary struct { Text string; KeyPoints []string; }
type Entities struct { People []string; Organizations []string; Locations
[]string; Dates []string; }
type Category struct { Label string; Confidence float64; }
type Sentiment struct { Polarity string; Score float64; }
type Report struct { Content string /* maybe other metadata */ }
```

These are illustrative; we will refine fields as needed. The key is each engine will output one of these types (or a simple type like string for the summary if we don't need a struct). Having explicit types enforces SRP and clarity.
- Choose libraries for external needs: - HTTP fetching (standard `net/http` is fine for ingestion). - HTML parsing (maybe Go's `net/html` or a library like `goquery` for easy HTML query if needed to extract article bodies). - Text normalization (for Unicode normalization, use `golang.org/x/text/unicode/norm` if needed to NFC normalize characters [7] ; for general cleaning like removing markup or non-text content, either do manual regex/strings or use an existing package if any suits). - Language detection (decide on `lingua-go` [2] vs `whatlanggo` or others; lingua-go is high accuracy but ensure license and performance acceptable). - If planning on non-LLM NLP (like using `prose` for NER or sentiment lexicons), list those. If primarily using LLM for analysis, list OpenAI Go SDK (e.g. `github.com/openai/openai-go` or community `go-openai`) and/or LangChainGo/Eino if we plan to use them. - Prompts management (note if using LangChainGo or Eino for easier prompt templating and output parsing). We will evaluate these after implementing basic pipeline. - Establish configuration management: e.g., how to supply API keys (for OpenAI, etc.), any settings (like choose model names, prompt templates). Possibly use environment variables or a config file for such parameters. - Plan error handling and logging: decide how engines propagate errors (e.g., return error up the chain, possibly wrap with context of which engine failed). Logging at each stage for debug (e.g., log when text is fetched, when summary is generated, etc.). Consider using Go's `context.Context` to carry deadlines or cancellation signals, especially for long operations (like network calls or LLM API calls). This way, if the process needs to be canceled or time out, all goroutines can be notified [8] . - **Testing Strategy:** For this initial setup/design, create unit tests or at least pseudo-tests for the data models and any utility function (like a function that strips HTML). However, most testing here is reviewing design. Verify that the structured types cover all data we intend to capture. We can write a test that manually populates these structs to ensure they can represent a sample scenario. If using an interface for engines, we might write a dummy engine implementation to test the interface works as expected.
- Additionally, if we have a CI pipeline, set that up to run `go test` and perhaps linters (`golangci-lint`) to enforce code quality from the start.

2. Implement Ingestion Engine (Task: Data Fetching)

Development Tasks:
- **Functionality:** Implement the engine that obtains the raw text for analysis. This could be split into two modes: - *Web Ingestion:* Accept a URL or search query. If a URL, fetch the page via HTTP. Use a user agent string that mimics a browser (some sites block default Go user agent). If a search query is allowed, possibly integrate a news API or a web search API (depending on scope) to get relevant articles, then fetch them. For now, focusing on direct URL or raw HTML input might be simpler. - *File Ingestion:* Accept a file path (or file

content) to ingest. Read the file (assuming UTF-8 text or HTML). Possibly support PDF via an external library if needed, but that can be an extension. - **Parsing (for HTML):** If the input is HTML (from web or file), parse it to extract the main content. Use `net/html` or a library like `goquery` to find article tags. Many news sites have a main `<article>` tag or specific div ids. For a general solution, consider using an open-source "readability" library or algorithm to extract the core text of an article (boilerplate removal). If none in Go, a heuristic approach: remove `<script>`, `<style>`, etc., then extract text within `<p>`, `<h1>` (for title), etc. We could also let the LLM do summarization without strict parsing, but it's better to remove clutter to avoid irrelevant text in summary. - **Metadata:** Extract or set metadata like `Title` of the article (maybe from `<title>` tag or `<h1>` header), publish date (if available in meta tags), and source (e.g., domain name). These can be stored in the `NormalizedArticle` or a metadata struct. While not explicitly asked, title and date might be useful in reporting. - **Output:** The ingestion engine should output a struct, e.g. `RawArticle` with fields: `Content` (string, possibly HTML or raw text), `Source` (URL or filename), and maybe `Title` if easily available. This goes as input to normalization. - **Testing Strategy:** - Write unit tests using a sample HTML string. For example, embed a small HTML snippet representing a news article (with some known structure) and test that the ingestion engine's parsing function correctly extracts the expected text and title. - Use a local HTTP test server or recorded HTTP response to test the HTTP fetch logic without hitting the network. For file reading, create a temporary file in test with sample content and verify reading works. - Include edge cases: missing title tag, or content inside multiple nested divs. Ensure the engine handles empty or very large content gracefully (maybe set a size limit to avoid extremely long pages). - If possible, test with an actual known URL of a news article (this might be more of an integration test). We must be mindful of not making network calls in unit tests (to keep tests deterministic), but we can simulate by saving an article HTML to a file and using that. - Confirm that errors (like HTTP 404 or file not found) are properly returned and do not crash the program. Use Go's error wrapping to add context (e.g., "ingestion: failed to fetch URL ...").

3. Implement Normalization Engine (Task: Text Cleanup)

Development Tasks:

- **Cleaning HTML/Text:** If the ingestion engine provided HTML content, this engine will strip out all HTML tags and scripts (if not already done). If the ingestion already produced raw text, this step might be simpler (just identity transform, but likely we'll still do some normalization on the text). We can use regex replacements or an HTML-to-text library. A simple approach: iterate through the parsed HTML node tree and extract only text nodes (this avoids regexing HTML which can be error-prone). Libraries like `goquery` allow selecting text directly. Another alternative is using the `html/template` or `text/template` sanitizer, but that's more for escaping HTML, not extracting text. - **Unicode Normalization:** Use `golang.org/x/text/unicode/norm` to normalize the Unicode form (NFC or NFD). This ensures characters like "é" are consistently one codepoint or a combination as needed. Go doesn't guarantee string normalization by default [9], so doing this can prevent subtle issues (especially if we compare or search text). Probably use NFC (canonical composition) which is standard on the web [10]. - **Lowercasing / Casing:** Decide if we want to lowercase all text. For analysis like classification or summarization, case usually isn't an issue (LLMs and modern NLP handle case). Lowercasing might be useful for certain keyword matching or if we were doing traditional ML. We might skip lowercasing to preserve proper nouns for readability in outputs (especially since we plan to output entities and summary, which should maintain original capitalization for names). - **Removing noise:** This includes trimming extra whitespace (collapse multiple spaces/newlines into one), removing non-text characters (e.g., strange Unicode symbols or advertisements like "Subscribe now"). If we know some patterns (like common boilerplate phrases), we can filter them. Since news articles might include captions, author bio, etc., we might consider removing those if identifiable.

However, it might be safer initially to keep everything and let summarizer ignore less relevant parts. - **Normalization of punctuation:** e.g., convert fancy quotes or dashes to standard ones, if needed for consistency. (This is minor, but can help if downstream we needed to match quotes, etc.) - **Output:** A `NormalizedArticle` struct containing at least `Content` (cleaned text). We might also carry over the `Title` and `Source` from the raw input (so they are available to later stages like reporting). If the input had clear paragraph breaks, we might keep those (maybe store content as a slice of paragraphs or a single string with newline separators, depending on how we feed it to LLM). - **Implementation detail:** For performance on large text, avoid unnecessary copying. Using streaming normalization could be good if text is huge (but typically news articles are not extremely large). Still, ensure our method can handle a few thousand words efficiently. Using bytes vs runes carefully: since we deal with Unicode, some processing may require converting to `[]rune` or using the `x/text` package iterators. - **Testing Strategy:** - Prepare a sample of unnormalized text. For example: a string with HTML tags (`<p>Some text</p>`), HTML entities (`&amp;`), irregular spacing (double spaces, newlines), and mixed Unicode forms (like an “é” in both composed and decomposed form to ensure our normalization unifies them). - Write tests that feed this into the Normalization engine and assert that the output string meets expectations: HTML tags gone, entities decoded (`&amp;` -> `&`), multi-spaces collapsed, newline trimming done, Unicode normalized (we might test by comparing to a pre-normalized string or ensuring no combining characters remain if that’s our chosen normalization). - Test that the `Title` and other metadata are preserved if they should be simply carried through. - Include edge cases: empty input (should output empty content), content that is already plain text (should remain unchanged aside from trivial trims), content that is extremely large (performance test, not necessarily unit test, but ensure no algorithmic bottleneck like O(n^2) loops). - Test that no important content is accidentally stripped. For example, if the article text contains an email or a URL, ensure our regex doesn’t inadvertently remove parts of it considering them “non-word”. We may decide what’s considered noise carefully or provide a config for normalization rules.

4. Implement Language Detection & Translation Engine (Task: Multi-language support)

Development Tasks:

- **Language Detection:** Using the normalized text, detect the language. Integrate a library like `lingua-go` or `whatlanggo`. For `lingua-go`, instantiate the detector with all languages (or a set of expected languages if we want to limit). Detect the language of the `NormalizedArticle.Content`. This should return a language code (like "en", "fr", etc.) and possibly a confidence or score. We create a `LanguageResult` struct with these.
- If the confidence is below a threshold or it detects multiple languages, we should handle that. For simplicity, if a clear single language is detected, proceed. If the text is mixed or low confidence, perhaps default to treating as the most likely language or allow the LLM engines to handle it (since models like GPT-4 can process many languages directly). - **Deciding on Translation:** Based on the detected language and our target language (say English), decide whether to translate. If `LangCode` is "en" (English) or another language we plan to handle directly, we can skip translation. If not, and if we want all downstream analysis in English, mark that translation is needed. (We could also allow summarization in the original language if we wanted, but likely we want uniform output in English). - **Translation:** If needed, there are a few approaches: - Use an external translation API (Google Translate API, DeepL API, etc.). These would be additional dependencies and costs, but are straightforward: call API with text and get English text. However, since we already use LLMs, we could just as well use the LLM to translate. - Use an LLM prompt: e.g., “Translate the following French text to English. Text: <...>”. GPT-4 or similar will do a good translation while preserving meaning. This is a single prompt call. The downside is using some token quota for text that we will again summarize or analyze, but it might be acceptable for pipeline simplicity. Alternatively, we could

skip a separate translation step and directly prompt the summarizer with a multilingual prompt (GPT-4 can summarize French text in English if asked). That is a design choice: *Engine chaining vs prompt cleverness.* For clarity, an explicit translation engine is cleaner: first translate, then feed the English text to all analysis engines. This way those engines don't have to worry about language differences. - **Output:** The engine outputs either the original text (if no translation) or the translated text (English). We may update the `NormalizedArticle.Content` to this new text or have a separate field `TranslatedText`. To keep things clear, we can create a `PreparedText` struct that has `Content` and a field indicating language (and maybe OriginalContent if we want to keep that). But it might be simpler to say: after this engine, we treat `Content` as English text for downstream. We should still keep the original language code around for the report (e.g., might mention "Originally in French"). - **Testing Strategy:** - For language detection: prepare known text samples in a few languages (short paragraphs in English, French, Spanish, etc.). Test that our detection correctly identifies them. If using `lingua-go`, test a multilingual sentence as well to see how it behaves (though analysis might not handle truly mixed-language content well, but at least ensure no crash). - We can simulate translation by a stub in tests to avoid calling external service. For example, if our translation engine uses an interface for translators, in test we plug a fake translator that just appends "[Translated]" to the text or uses a small dictionary map for a couple of words. This way, the engine's flow can be tested without external API. If we do use a real API in code, for tests we should not call it (to keep tests offline and deterministic). - If we use an LLM for translation in the actual implementation, in unit tests we might skip actually calling it (since that's external and possibly nondeterministic). Instead, consider designing the engine to allow dependency injection of a translation function. In production, that function calls the API; in tests, it's a dummy that returns a preset string (e.g., returns `"Hello World"` when given `"Bonjour le monde"`). - Test that when input is already English, the engine correctly passes it through without changes (and doesn't call translation). Also test that the language code is recorded properly. - If using concurrency (though likely detection and translation will be sequential in one engine), ensure any potential race conditions are tested (e.g., if we made it asynchronous, but probably not necessary here). - Also test error paths: if translation API fails or times out, ensure we handle that (maybe try an alternative approach or abort with error). If detection fails (shouldn't usually, but if text is too short or gibberish), decide on a default (perhaps assume English or skip translation).

5. Implement Summarization Engine (Task: Summarize Content)

Development Tasks:
- **LLM Integration:** Set up integration with an LLM (likely OpenAI GPT-4 or GPT-3.5). Use the Go SDK [11] to call the Chat Completions API. Alternatively, use **LangChainGo** which can simplify prompt management and LLM calls. For a first implementation, directly calling the SDK might be straightforward: construct the prompt, call the API, get the response. Ensure to handle API keys securely (probably via env var). - **Prompt Design:** Craft a prompt for summarization. This prompt should instruct the model to produce a concise summary of the text. Key details for a good prompt: - Instruct the model to focus on main points and facts (to ensure important info is captured). - Possibly specify a length or format (e.g., "in 3-4 sentences" or "one paragraph"). - Mention the language if needed (e.g., "Respond in English" if there's a chance the model might respond in the source language, but if we translated already, it should be fine). - Because this is an automated pipeline, we want determinism in format: perhaps just plain text summary. We might avoid asking for bullet points here (unless we specifically want them) because parsing them later is extra work – but including key point bullets could be useful for the report. We can decide: maybe have the summary as a short paragraph, and optionally also a list of 2-3 key bullet points of insight. We can instruct that in the prompt. - **Model parameters:** Use temperature low (like 0.2 or 0) for factual consistency (we don't want creative deviations). Limit max tokens of output to say 150 or so, to get a brief summary. - **Output:** The

engine returns a `Summary` struct, containing at least the summary text. If we also extract bullet points, it could have a `KeyPoints []string` field. If the model returns them in some structured way (like a JSON or a delimiter-separated list), we would parse that. Alternatively, we can prompt it to produce a JSON with keys `"summary"` and `"key_points"`, but doing JSON parsing of a possibly long text might risk truncation. It might be easier to let it produce markdown (like a short paragraph followed by a bullet list). However, since this is internal, a simpler approach: just get the summary text and later if we want bullet points, we can either prompt for them separately or derive from summary. Given time constraints in pipeline, probably one prompt for summary is enough. - **Implementation:** Write the code to call OpenAI API. For example, using the SDK:

```
req := openai.ChatCompletionRequest{
    Model: "gpt-4",
    Messages: []openai.ChatMessage{
       {Role: "system", Content: "You are a helpful assistant that summarizes
news."},
       {Role: "user", Content:
fmt.Sprintf("Summarize the following article in a few sentences:\n\n%s",
articleText)},
    },
    Temperature: 0.2,
    MaxTokens: 150,
}
resp, err := client.CreateChatCompletion(ctx, req)
summaryText := resp.Choices[0].Message.Content
```

(The above is conceptual; actual code will handle errors and possibly retry on rate limit.) - **Error handling:** If the API fails (network issue or API returns an error), the engine should return an error up the chain. Possibly implement a retry mechanism for transient errors or rate limits (maybe a short exponential backoff and retry a couple times). - **Testing Strategy:** - **Unit testing with mocking:** Since calling the real OpenAI API in a test is not ideal (cost, variability), we should abstract the LLM call behind an interface. For example, define an interface `LLMSummarizer` with a method `Summarize(text string) (string, error)`. In production, implement it using OpenAI; in tests, implement a fake that returns a canned summary (for a given input, maybe use a simple deterministic algorithm or a map of known inputs to outputs). This allows testing the SummarizationEngine logic (like that it calls the interface and wraps the result in a struct) without hitting the real API. - Test that the engine properly uses the translation (i.e., if we provide text in a non-English language but translation engine should have given English by now; perhaps test that summarizer does produce output in English for known foreign input by simulating translation happened). - If not using an interface, another approach is dependency injection: pass a function pointer to the engine for the API call, which in tests can be replaced. - Test prompt correctness to some extent: We can't fully test the real model's compliance, but we can test that our prompt string is constructed as expected (maybe by unit testing the function that builds the prompt, ensuring it includes the article text and instruction). - If we decide to parse any structured output (like expecting JSON from the model), test the parser on some sample model outputs (which we can store). For instance, if we prompt the model to output `"Summary: ... KeyPoints: ..."` format, ensure our parsing splits it correctly. - Integration test: As a later integration test (not unit), one could call the real API with a short known text to see if the summary is sensible. But this might be part of a final end-to-end test rather than automated in CI.

6. Implement Entity Extraction Engine (Task: Extract Entities)

Development Tasks:
- **LLM vs Library:** Decide if using the LLM for extraction or a native library. A native solution in Go for named entity recognition (NER) could be the `prose` library (which supports extracting persons, places, etc., but only for English). Given multi-language and the pattern of using LLM, using the LLM is attractive for consistency. We will proceed with an LLM prompt approach, as it's flexible and handles arbitrary content. - **Prompt Design:** Construct a prompt that asks the model to identify key entities. For example: *"Extract the main people, organizations, locations, and dates mentioned in the following text. Return the results in JSON with the structure:* `{ "people": [...], "organizations": [...], "locations": [...], "dates": [...] }`. *Include each item only once (no duplicates) and use their full names as given in the text."* This prompt directs the model to output a JSON string. We take advantage of OpenAI's ability to follow formatting instructions. (OpenAI's structured output feature or function calling could also be used here to ensure JSON, but for now a prompt with explicit JSON format should suffice.) We should also caution the model not to hallucinate entities not in text, maybe by saying "only include entities explicitly mentioned in the text."
- **Parsing JSON:** After the model responds, we will likely get a JSON (or something close to JSON). We can use Go's `encoding/json` to parse it into our `Entities` struct. We need to be prepared to handle minor format issues (the model might not always perfectly format JSON, but with a good prompt and GPT-4, it should in most cases). If parsing fails, we might log the raw output for debugging. Alternatively, to be safer, we could ask for a more easily parseable format (like each category on a separate line prefixed by a label). However, since the question emphasizes structured output types, JSON is a natural choice. - **Alternate approach:** If not using LLM, integrate `prose` or spaGO for NER. But each has limitations (e.g., spaGO requires models to be loaded, `prose` is English-only). LLM approach will handle multilingual content (if we translated to English, it's fine; if we decided to skip translation and just do multilingual in future, GPT can handle many languages too). - **Output:** An `Entities` struct populated with the lists from the JSON. If the model didn't find any in a category, the list would be empty. (Our struct should be initialized with empty slices to avoid nil issues.) - **Rate & Complexity:** Entity extraction prompt may require the model to output quite a bit if the article is full of names. But typically it's limited. Should still use a low temperature (0) for accuracy, and perhaps a moderate token limit for output (maybe 200 tokens, as lists can grow if article is long). - **Testing Strategy:** - As with summarization, abstract the LLM call for unit tests. Perhaps have an interface `EntityExtractor` or reuse a generic LLM interface but with a method for extraction. In tests, simulate the model's JSON output. For instance, take a sample text and craft a fake model response JSON (as a string). Test that the engine's JSON parsing correctly populates the `Entities` struct. We can have a test where the fake LLM returns a known JSON and ensure our code maps it to the struct properly. - Test the prompt building function: ensure the text is inserted properly and the prompt includes the instructions for JSON format. - If we choose to incorporate OpenAI's **function calling** feature (which returns structured data natively), that could simplify parsing – but that may complicate our integration. We might note it but implement straightforward prompt+parse for now. In tests, we'd then test the fallback parsing logic (for example, if model returned some text with quotes, etc.). - If using a library method (for backup or comparison), test that it identifies at least some entities from a sample text and compare with expected. This could be used as a baseline to verify the LLM approach isn't missing obvious ones. - Include edge cases in tests: text with no recognizable entities (the output JSON should have empty lists, ensure parsing yields empty slices not errors). Text with tricky entities (e.g., "Apple" could be org or just fruit – the model might list it under org if context suggests company). We can't fully control correctness via tests, but we can at least check format and that it doesn't crash on such content. - Also test handling of duplicates: If "John Smith" appears twice in text, our prompt said no duplicates, but ensure if model does give duplicates how our parsing handles it. Possibly we might deduplicate after parsing as a safety measure in code.

7. Implement Topic Classification Engine (Task: Categorize the Article)

Development Tasks:
- **Classification via LLM:** Use the LLM to identify a high-level category of the text. Define a set of possible categories (maybe based on typical news sections: Politics, Business, Technology, Sports, Entertainment, Health, etc.). We can list these in the prompt to guide the model, or leave it open-ended for it to name a category. However, for consistency, providing a set or asking for a concise label is better. - **Prompt Design:** For example: *"Determine the primary topic category of the following article. Choose one of the following categories (whichever is most appropriate): Politics, Business, Technology, Science, Health, Sports, Entertainment, Other. Respond with only the category name."* This ensures we get a single-word/phrase answer. If we want a subcategory, we could prompt for a second level too, but that might complicate things. We might stick to one category for now. - **Alternate approach:** If this needed to be done without an LLM (to save API calls), one could implement a simple keyword-based classifier or use a pre-trained classifier model. But given the dynamic range of news and the fact we already use LLM, leveraging it here is simplest and likely accurate. - **Output:** A `Category` struct, e.g. `{ Label string; Confidence float64 }`. The confidence could be implied or we could ask the model how confident it is. But quantifying confidence from the model's perspective is tricky. Alternatively, we could not use a struct and just keep a string category (maybe it's enough). We include confidence if we might use it to decide whether to label as "Other/Unknown" if confidence low. Since GPT won't give a numeric confidence normally (unless we define a mapping), we might skip that or just set Confidence = 1.0 by assumption if we trust the model. (Or parse if it says "maybe X", but better to force a firm single answer.) - **Testing Strategy:** - Similar to previous LLM engines: use a mock model in tests. For a given input text, have the fake model return, say, "Politics". Test that our engine returns Category.Label = "Politics". - Test that if the text is clearly about sports (we supply a sports article snippet), the engine categorizes it correctly (in a controlled test, we might manually simulate what we expect the model to do). - We might also test that our prompt restricts output properly. For instance, if our fake model tries to return a sentence like "The article is about politics", ensure our code either takes just the keyword or we refine the prompt to avoid that (likely by saying "respond with only the category"). - Edge case: article that spans two categories (e.g., "business politics"). The model might return a combined or ambiguous answer. We should decide how to handle that (maybe still output one or choose the dominant topic). For test, we could simulate an ambiguous scenario and see if our engine might just output one category or if the prompt needs to emphasize one. - If we were to implement any fallback (like if model returns something not in our list), test that logic too. For instance, model might output "Political" instead of "Politics" – we could normalize by mapping synonyms or partial matches to our known set. A simple approach: take model output, check if it matches one of our allowed categories (case-insensitive, maybe singular vs plural issues). If not, categorize as "Other" or attempt a secondary prompt. We can include tests for a couple of variations ("Technology" vs "Tech") to ensure we handle them if needed.

8. (Optional) Implement Sentiment Analysis Engine (Task: Determine Tone)

This task is optional depending on whether the final report should include sentiment/tone of the article. We include it for completeness:

Development Tasks:
- **Sentiment via LLM or Library:** One could use a lexicon-based approach (like VADER – which we have in Go as `govader` [5]). For English text, VADER can output a sentiment score and category. This is fast and no API needed. If we trust it, that might be a good non-LLM solution. Alternatively, ask the LLM: *"What is the overall tone or sentiment of this article (positive, negative, or neutral)?"* or a more nuanced description like

“analytical, critical, celebratory,” etc. For simplicity, maybe just positive/negative/neutral if needed. - **Implementation:** If using VADER or similar: - Integrate `govader`: feed the text and get a compound score. Then threshold that to get Pos/Neg/Neutral. - If using LLM: prompt and parse the answer (which should be just one word ideally). - **Output:** `Sentiment` struct with e.g. `Sentiment: "Positive"` (or an enum). If a numeric score is present, include it. - **Testing Strategy:** - If using VADER, test it on known sentences ("I love this!" -> positive, "This is awful." -> negative, etc.). - If using LLM, test with the mock as before, ensuring the prompt yields one of the categories and our code captures it. - Sentiment might not be crucial for the first version, so this engine could be implemented later or toggled off in the pipeline if not needed.

9. Implement the Reporting Engine (Task: Generate Final Report)

Development Tasks:
- **Purpose:** The reporting phase takes all analysis results and produces a final human-readable report. This report should synthesize the findings: it typically will include the summary of the article, mention key entities (e.g., notable people or organizations involved), possibly the category of the news, and any other insights (like sentiment). The report is the **standalone output** for end users, separate from the raw analysis data. - **Approach 1 – LLM-Generated Report:** Given that we have multiple pieces of data (summary text, entity lists, category, etc.), one effective approach is to prompt an LLM to compose the report. This can result in a nicely flowing narrative. We would supply the LLM with the pieces in a prompt, and ask it to format it in a structured way (maybe as a little article or bullet points). - **Prompt Design:** For example: *“You are a reporting assistant. Using the following analysis results of a news article, produce a concise report:\n- Summary: <summary text>\n- Key Entities: <list of people, orgs, etc.>\n- Category: <Category>\n\nWrite a report that first presents the summary in your own words, then lists the key people/organizations and their roles, and mentions the topic category. The report should be in a neutral tone.”* This prompt gives the model all the pieces. We need to ensure the prompt is not too large (should be fine since summary and lists are relatively small compared to full text). - We can instruct format: maybe the output should be markdown with sections, or just a couple of paragraphs. We should avoid the model just regurgitating the summary verbatim – by saying "in your own words" or "based on the summary and entities". - Also, since this is final output to user, we might want no obviously structured markers like JSON here – just nice text. - **Approach 2 – Template-based assembly:** Alternatively, we could generate the report with code: e.g., start with the summary text, then append a section "Key Entities: ..." listing entities, and "Category: ...". This is deterministic and no additional API call. It ensures we include everything. However, it might read more like a raw data dump than a narrative. It could be acceptable depending on requirement. Because the user mentioned reporting as a standalone phase and likely expects something polished, using the LLM might give a more refined output. - **Decision:** We will likely go with the LLM approach for a more fluent report, but we must then pay for one more API call and ensure the model doesn’t hallucinate new info. We mitigate hallucination by only giving it the analysis results (which themselves derive from the text). It might still infer or embellish slightly. To control that, we instruct it to stick to the given info. - **Engine Implementation:** If using LLM: - Prepare the prompt with all inputs. Possibly use the Chat API with roles, e.g., provide a system message with instructions on style, then a user message containing the bullet list of results (summary, entities, etc.) and asking to compose the report. - Call the API, get the result text. - Output a `Report` struct containing the `Content` (which could be markdown text). - This engine depends on the outputs of multiple previous engines, so the orchestrator needs to collect those and feed them in. The engine function signature might accept all needed pieces (or we have a composite struct as shown earlier). - **Orchestration Note:** If we treat reporting as another Engine in the pipeline, it’s a special one that needs multiple inputs. We might circumvent the Engine interface here and just call a function in the orchestrator once we have all results. Alternatively, define the input to reporting engine as that big `ArticleAnalysis` struct containing

everything. - **Testing Strategy:** - **Unit Test with Mock LLM:** Similar to others, simulate the final LLM output. For given fake inputs (a fake summary, some entities, etc.), have the mock LLM return a pretend report string (like "In summary, … Key people include … The article is categorized as …"). Test that our reporting engine properly constructs the prompt and returns the string in the `Report` struct. - If using template approach, test that function with sample inputs produces the expected formatted string. - Ensure that if some analysis result is missing (say entity list is empty or sentiment engine was not used), the report generation can handle it (maybe by omitting that section gracefully). We can simulate missing data by leaving those fields empty and see if the prompt or template can still produce a coherent report. - We should also do an integration-style test: take a fully analyzed sample (maybe from a real article analysis we prepared manually or from running the pipeline on a known input) and feed it to the report generator, then manually check if the output includes all the main points. This ensures our prompt isn't causing any loss of info. - Test length: if summary is long or there are many entities, does the report still read well? (Potentially instruct the LLM to maybe limit how many entities to mention if the list is huge – e.g., mention top 5 entities for brevity. We can decide such policy.) - Since the final output is user-facing, we might consider a manual quality test: actually run the pipeline on a real news article and read the final report to judge correctness and readability. That can inform prompt tweaks.

10. Engine Orchestration and Pipeline Optimization

With all individual engines implemented and tested in isolation, we integrate them into the full pipeline.

Development Tasks:
- **Sequential vs Parallel Execution:** As discussed, some parts of the pipeline must be sequential (e.g., ingestion -> normalization -> possibly translation happen in order). Once we have the final text ready, we can run summarization, entity extraction, classification, and sentiment in parallel since they are independent. Then, we join results and call reporting. Implement the orchestrator to reflect this: - Pseudocode:

```
raw := IngestionEngine.Run(input)
norm := NormalizationEngine.Run(raw.Content)
langRes := LanguageDetectEngine.Run(norm.Content)
textForAnalysis := norm.Content
if langRes.Lang != "en" {
    translated := TranslationEngine.Run(norm.Content, langRes.Lang, "en")
    textForAnalysis = translated.Content
}
// Now parallel analysis:
var summary Summary
var entities Entities
var category Category
var sentiment Sentiment
var wg sync.WaitGroup
wg.Add(4)
go func(){ defer wg.Done(); summary = SummarizationEngine.Run(textForAnalysis) }
()
go func(){ defer wg.Done(); entities = EntityEngine.Run(textForAnalysis) }()
```

```
go func(){ defer wg.Done(); category = ClassifyEngine.Run(textForAnalysis) }()
go func(){ defer wg.Done(); sentiment = SentimentEngine.Run(textForAnalysis) }()
wg.Wait()
report := ReportingEngine.Run(summary, entities, category, sentiment, /* maybe
original language or title for context */)
```

(Proper error handling omitted in pseudocode for brevity – in reality, we'd capture errors from each and possibly cancel others if one fails critically.) - Using goroutines and WaitGroup (or channels) to collect results speeds up the analysis step, especially if using external API calls which can be done concurrently. However, be mindful of rate limits: OpenAI API might rate-limit if we fire off many requests simultaneously. If that's a concern, we might orchestrate slight staggering or at least handle 429 responses with retries. Also, the number of parallel calls (4 in this case) should be okay for a single article, but if we were processing many articles concurrently, we'd need a more global rate control. - If needed, we can configure the orchestrator to optionally run in sequential mode (for easier debugging) or parallel mode (for production) via a flag. - **Error Propagation:** Design how errors in any engine affect the pipeline. A robust approach: if a critical engine fails (e.g., we cannot get content or the normalization fails), abort the pipeline and return an error to the user or calling context. For engines in parallel, if one fails, you might choose to still wait for others and then decide to either include what succeeded in the report or to omit the report entirely. Possibly, if summarization fails but others succeed, we could still produce a partial report (since summary is quite central, maybe we wouldn't). For now, simplest: if any of the parallel engines errors out, cancel the others (using context cancellation) and report failure. We can refine this later. - **Shared Data Dependencies:** The orchestrator ensures each engine gets the data it needs. We should avoid global variables; instead pass data explicitly. If some data is needed by multiple engines, we've either already computed it (like `textForAnalysis` string) or we can compute shared things in advance. For example, maybe both summarization and entity extraction need the text and also perhaps the language (though by the time we pass English text, they can assume English). We might store the final chosen text and its language in a struct and give a copy to each goroutine. Since string is immutable in Go and we aren't modifying it, sharing the same string reference is fine (no copying on write). - **Performance Optimizations:** - Making parallel calls as above is the main optimization for throughput on a single article. - If processing multiple articles in a batch, we could also parallelize at the article level (with caution to not overwhelm APIs). This would require concurrency control (like a semaphore to limit total concurrent API calls). - The pipeline itself isn't heavy on CPU except possibly ingestion parsing and normalization, which are fast. The slow parts are network calls. We should ensure our HTTP calls (ingestion, API calls) use efficient clients (reuse TCP connections, etc.). - Caching: If the same article might be processed multiple times, consider caching results of engines (probably not a main requirement now, but something to note for future). - Orchestration overhead: calling engines as functions vs separate processes – since we are in one process, function calls are cheap. If we had designed each engine as a microservice or CLI command, orchestration would call them via OS (which is slower). We stick to in-memory calls for efficiency. (In case initially there was an idea of separate command execution, consolidating into one process is a major optimization in orchestration.) - If using a framework like **CloudWeGo Eino**, we could define this pipeline as a Workflow DAG to automatically handle concurrency and data mapping [12]. Eino can map struct fields from one node to another and manage concurrency, which aligns with our design of structured outputs. We could consider using it to simplify orchestration logic. (For example, Eino's workflow could take our `NormalizedArticle` as input and fan-out to LLM calls, then fan-in to a final node for reporting, with type-safe edges.) - **Testing Strategy:** - **Integration Test (Single Article):** Write a test that uses stubbed engines (or real ones with stubbed LLM calls) to run through the full pipeline on a sample input. For instance, provide a short text or use a dummy ingestion that just returns a static paragraph (to avoid external fetch in test), then run through normalization, detection, etc

(we can use the actual code since we have fake LLM for summary/entity). Verify at the end that `report.Content` contains expected pieces (like it mentions the summary or category we know from the dummy data). - **Simulated Failure Test:** Create scenarios where an engine fails to ensure the orchestrator handles it. For example, make the SummarizationEngine return an error in a test; check that the orchestrator cancels others (maybe via a context) and that no report is produced (or an error is propagated). This might require ability to inject errors or a special mode in the fake engines. - **Parallel Execution Order:** Ensure that even with concurrency, the final outcome is deterministic given the same inputs (the order of completion of goroutines shouldn't matter for result assembly). We might have to gather results in a map or struct without relying on completion order. Test by checking all outputs are present. Also test that no race conditions occur (run the race detector in Go tests to ensure no concurrent writes to shared memory without lock). - **Performance Test (Basic):** As a simple benchmark, measure the pipeline execution time for a moderate text (maybe 1000 words) using a fake fast LLM (that returns instantly). Ensure it logically runs faster when parallelizing than if we forced sequential. This indicates our concurrency is effective. (Real performance with actual API calls is harder to test reliably due to external factors, but we could do a dry run and ensure it's within acceptable range, e.g., hopefully dominated by the slowest single API call rather than sum of all.)

11. Prompt Management and Redaction

Managing prompts across multiple engines is an important concern for maintainability and security.

Development Tasks:
- **Centralize Prompt Definitions:** We should avoid scattering raw prompt strings throughout the code for each engine. Instead, define them as constants or in a configuration (possibly external files or at least constants at top of each engine file). This makes it easier to review and tweak prompts. For example, have `const SummarizePromptTemplate = "...%s..."` where `%s` will be replaced with the article text. For more complex prompts with multiple insertions, consider Go's `text/template` or simple string formatting. - **Use of Prompt Libraries:** Investigate libraries to manage prompts: - **LangChainGo:** LangChain's Go port provides prompt templates, output parsers, etc. It specifically has a prompts package with types and utilities for prompt templates [13]. This could be useful for complex prompts and ensures consistency (for example, we can define a template with placeholders and easily fill them, or use few-shot examples if needed). - **CloudWeGo Eino:** Eino also likely has some support for prompt templates (as it mentions composition and flows). But Eino is more about orchestration; prompt formatting we might still do ourselves or via LangChainGo integrated. - **Prompt customization libraries:** There are community libraries like `lexlapax/go-llms` which unify different providers; some have their own prompt helpers. We might not need heavy frameworks if our use-case is straightforward, but it's good to be aware of them. - **Versioning and Testing Prompts:** If prompts are externalized (say in JSON or YAML files or a database), we could implement a system to load them by name. However, that might be overkill for now. Simpler: keep them in code constants, but clearly marked and documented. Possibly write tests for prompts (at least to ensure placeholders exist and maybe that the text fits within token guidelines). - **Prompt Redaction:** Ensure that any sensitive data or internal instructions in prompts are not leaked in outputs. "Redaction" here can mean: - If we include any internal notes in system prompts (like "you are an LLM, do X..."), ensure the model is instructed not to include that in the answer. This is usually handled by the API (the user only sees assistant responses). But we should double-check that in final outputs, we're not accidentally printing or storing the prompts in logs where end-users can see them. - Another angle: If the input text has PII or sensitive info that should not appear in the final report, we might need a step to redact or anonymize it. The question's wording suggests focusing on prompt content, but it's wise to consider privacy. For example, if

an article contains a phone number or personal address, maybe the final summary should omit that if not crucial. We can enforce this via prompt instructions (like telling the summarizer to exclude personally identifiable info if appropriate). This might not be a requirement, but it's a consideration for a robust system. - **Libraries for Redaction:** There are libraries for PII detection (not sure if in Go) or we could implement simple regex removal of phone/email if needed. But likely not a core ask now. - **Prompts Tuning and Management Process:** We might adopt a convention to clearly label parts of the prompt that may change. Possibly maintain an array of example prompts for testing or use prompt ids if using a prompt versioning system (some use cases use something like PromptHub or storing prompts in a database for dynamic updates). - **Prompt Examples in Documentation:** It's helpful for development and future maintainers to have examples of each prompt and expected output format in the repo. We should include that in our project README or docs. In fact, below we will enumerate each prompt template and its usage (as requested).

- **Testing Strategy:**
- **Prompt Unit Tests:** Even though prompts are mostly static strings, we can write tests that inject a small dummy text into each prompt template to ensure the formatting doesn't break (e.g., if we expect `%s` replacement, test that formatting works and the final string contains the text and required keywords like "JSON"). This catches issues like missing placeholders or accidental newlines.
- If using LangChainGo prompt templates, write a test to ensure the template compiles and fills correctly. For example, use their `prompt.NewPromptTemplate` with our template and verify it returns a correct string when given sample variables.
- **Redaction Tests:** If we implement any redaction function (say remove emails), test it on strings with known sensitive info to ensure it masks or removes them. Also test that it doesn't remove legitimate content accidentally.
- Since prompt effectiveness is hard to unit test, a lot will be iterative manual testing (observing outputs). We should prepare a few example articles and run the pipeline with actual LLM calls in a controlled way (perhaps a dev script rather than automated test) to see how well the prompts perform. Then adjust prompts as needed (this is an ongoing tuning process).
- Ensure that no prompt includes API keys or secrets inadvertently (they shouldn't, but just a security check).
- Logging: If we log prompts for debugging, ensure in production mode we either sanitize them or disable logging of full text (especially if text is sensitive or if prompt contains instructions that could be exploited if seen). This is more of an operational security test.

12. Prompt Templates for Each Engine (Detailed Examples)

Based on the **StructuredOutput** types and the roles of each engine, we design specific prompts. Below are the prompts we intend to use for the LLM-driven engines. These are written as template examples, where `<<...>>` indicates places we insert data:

- **Summarization Engine Prompt:**

```
Summarize the following article in 3-4 sentences, focusing on the key facts
and outcomes. Keep the summary concise and factual.
```

```
Article:
"<<INSERT ARTICLE TEXT HERE>>"
```

Rationale: This prompt instructs the model to produce a short summary, emphasizing facts. We explicitly limit to 3-4 sentences to ensure brevity. The article text will replace the placeholder. The model's response (assistant message) is expected to be a short paragraph summarizing the content.

- **Entity Extraction Engine Prompt:**

```
Identify the main entities mentioned in the text below, including:
- People (individuals' names)
- Organizations (companies, institutions, etc.)
- Locations (cities, countries, landmarks)
- Dates (specific dates or years)

Provide the output in a JSON format with keys "people", "organizations",
"locations", "dates". Include each entity only once, and use the exact
names as mentioned in the text.

Text:
"<<INSERT ARTICLE TEXT HERE>>"
```

Rationale: We clearly list what types of entities we want and specify JSON output. By enumerating the categories, we help the model not to miss any. We also tell it to avoid duplicates and to use exact names from text (to prevent it from, say, expanding acronyms or adding titles). We expect an answer like:

```
{"people": ["Alice", "Bob"], "organizations": ["Acme Corp"], "locations":
["Paris"], "dates": ["January 5, 2023"]}
```

The code will parse this JSON.

- **Topic Classification Engine Prompt:**

```
Determine the primary topic of the following article. Choose one category
that best fits the content from the list: Politics, Business, Technology,
Science, Health, Sports, Entertainment, or Other.

Only respond with the single category name.

Article:
"<<INSERT ARTICLE TEXT HERE>>"
```

Rationale: This prompt guides the model to pick one of the provided categories. We include "Other" for anything that doesn't fit well. The instruction "Only respond with the single category name." is to prevent it from giving a sentence or multiple categories. For example, it might answer simply: `Technology`. Our code will take the raw answer as the category label.

- **Sentiment Analysis Engine Prompt:** (if using LLM)

```
What is the overall sentiment of the following article? Please answer with
one of: Positive, Negative, or Neutral, based on the tone of the writing.

Article:
"<<INSERT ARTICLE TEXT HERE>>"
```

Rationale: This is straightforward. The model should respond with one word. If the article is mixed or just factual, likely "Neutral". We explicitly restrict answers to those three to avoid anything too fancy.

- **Reporting Engine Prompt:**

```
You are a report-writing assistant. Compile a brief report using the
analysis results of a news article provided below.

Analysis Results:
- Summary: <<INSERT SUMMARY TEXT>>
- Key Entities: People: <<INSERT people list>>; Organizations: <<INSERT
orgs list>>; Locations: <<INSERT locs list>>; Dates: <<INSERT dates list>>.
- Topic Category: <<INSERT CATEGORY>>
- Overall Sentiment: <<INSERT SENTIMENT>>.

Requirements:
1. Begin with a concise rephrasing of the summary, capturing the main point
of the article.
2. Mention the most important people, organizations, or locations involved,
explaining their relevance (in one or two sentences).
3. State the topic category of the article (e.g., politics, sports) as
context.
4. If relevant, comment on the tone of the article (e.g., whether it's
positive, negative, or neutral in outlook).
5. Write in a neutral, professional tone. Do not introduce new information
not in the analysis results.

Now, draft the report:
```

Rationale: We supply the summary and entities extracted, as well as category and sentiment. The instructions tell the model how to structure the report: starting with the summary (rephrased to avoid just copying verbatim), then incorporating entities, then category and sentiment. Numbering

the requirements in the prompt helps ensure the model covers each point. The final line invites it to begin drafting. We expect a few paragraphs from this. For example, the model might produce something like:

"The article reports that XYZ Corp's quarterly profits have fallen 10%, marking the first decline in two years. CEO Jane Doe stated that... (Summary rephrased)
Key figures in the story include Jane Doe, the CEO of XYZ Corp, and John Smith, an industry analyst who provided commentary. XYZ Corp is at the center of the report, and the events take place in London (Locations). The piece is categorized as Business news.
The tone of the article is neutral, presenting facts and statements without clear positive or negative bias."

This output would then be captured in the `Report.Content`. If the model mistakenly omits something or adds extra, we might refine the prompt further during testing.

- **Note:** We have carefully designed these prompts based on the StructuredOutput expectations. For example, the Entities prompt explicitly uses JSON keys that match our `Entities` struct fields, making parsing direct. The final report prompt uses a structured bullet input to reduce ambiguity.

We will manage these prompt templates possibly in a separate file or section in code for easy editing. Using a library like LangChainGo, we could formalize them as `PromptTemplate` objects and even attach an `OutputParser` for the JSON parts. LangChainGo's prompt utilities can help ensure no prompt injection issues and provide easier formatting [13], but it's optional.

13. Testing and Quality Assurance Phase

After implementing and unit-testing each component, a thorough integration testing and QA phase is needed:

- **Integration Tests on Real Data:** Run the entire pipeline on a few real news articles (covering different domains: politics, sports, etc. and different languages if possible). Examine the outputs:
- Is the summary accurate and sufficiently brief?
- Are the entities correct and complete? (No false entities and none important missed.)
- Is the category correct?
- Does the final report read well and include all major info? Is it free of hallucinations?
- **Iterate on Prompts:** If any outputs are unsatisfactory (e.g., the summary is too long or misses a key point, or the report has awkward phrasing), refine the prompt wording or format. This iterative prompt engineering is expected.
- **Performance Testing:** If possible, simulate processing multiple articles in parallel to see how the system scales. (This might involve writing a script to feed, say, 5 URLs at once and see if our concurrency is an issue with rate limits. We might implement a simple queue or limit concurrent LLM calls if needed).
- **Multi-language Test:** Take a non-English article (or translate one manually) and run it through. Verify that language detection triggers translation and that the rest of the pipeline works on the translated text. Confirm that the final report still notes the content correctly. (We might include in the report something like "Originally in French" if needed for context, though the prompt we gave does not explicitly do that. We could add that if desired: e.g., feed the original language into the prompt so it can say "(This article was originally published in French.)" in the report).

- **Robustness Tests:** Deliberately test some tricky cases:
- Very short articles or texts (just a headline or one sentence news). Does summarizer handle it (maybe the "summary" will just be the text itself).
- Very long article (to see if we hit token limits). If an article is longer than model input limit, we'd need a strategy (like summarizing in chunks then summarizing summaries). This is advanced; for now, we assume input fits in prompt. But note as a future improvement in the roadmap if needed.
- Articles with lists or bullet points in them (should still summarize fine).
- Content with potentially sensitive info (to see if we need any additional redaction or handling).
- **User Acceptance Testing:** If there are end-users or stakeholders, present them with the final reports and gather feedback. Perhaps the format needs tweaking (maybe they want a different ordering of info, or to exclude sentiment). Adjust accordingly.
- **Documentation:** Finalize documentation for the project. This includes:
- A README explaining how to run the pipeline, what each engine does.
- Documenting how to add support for new languages or new analysis engines in the future (for example, if later they want to add an engine for bias detection in articles, etc., the pipeline is extensible).
- Note any limitations (like reliance on external API, cost considerations, etc.).
- Instructions for obtaining API keys and configuring environment variables.
- **Deployment Considerations:** If this will run as a service or a CLI tool, consider how to package it. As a standalone CLI, ensure flags or config for input source. As a service (API endpoint), ensure it can handle requests sequentially or concurrently with the orchestrator logic.

Conclusion

This roadmap provided a **granular breakdown** of tasks to develop a multi-engine analysis pipeline for news/text in Go. We covered implementation details for text normalization (transforming text to a canonical form and cleaning HTML) [4], the ordering and data dependency of engines, and the structured output types each engine produces. The engines were kept focused and modular (adhering to SRP), and we planned for a robust orchestration mechanism to tie them together efficiently (leveraging Go's concurrency to parallelize independent steps) [14]. A separate final reporting phase uses orchestration and possibly an LLM to produce the polished output of all analysis.

We also addressed **prompt management**: using libraries like LangChainGo for prompt templates and output parsing [13], and ensuring prompts are handled carefully (stored clearly, and not leaking internal info). We listed concrete prompt templates for each engine, aligning with the structured outputs expected. Going forward, one should keep prompts under version control and refine them as the system learns from more examples.

By following this roadmap – implementing each engine with thorough testing and then optimizing the pipeline – we will achieve a maintainable, extensible, and efficient analysis system. It will be capable of ingesting open-ended news content, performing multi-faceted analysis, and delivering a coherent report to end-users, with support for multiple languages and further enhancements in the future.

---

[1] [11] Using the OpenAI Responses API in Go (Golang) – Intro Guide
https://chris.sotherden.io/openai-responses-api-using-go/

2 5 Language Detection - Awesome Go / Golang
https://awesome-go.com/language-detection/

3 Single Responsibility Principle: Everything You Need to ... - Alooba
https://www.alooba.com/skills/concepts/programming/object-oriented-programming/single-responsibility-principle/

4 Text Processing Techniques in NLP - DEV Community
https://dev.to/chinhh/text-processing-techniques-in-nlp-1034

6 8 14 Go Concurrency Patterns: Pipelines and cancellation - The Go Programming Language
https://go.dev/blog/pipelines

7 9 10 Text normalization in Go - The Go Programming Language
https://go.dev/blog/normalization

12 GitHub - cloudwego/eino: The ultimate LLM/AI application development framework in Golang.
https://github.com/cloudwego/eino

13 langchaingo package - github.com/tmc/langchaingo - Go Packages
https://pkg.go.dev/github.com/tmc/langchaingo