Quick notes

# Overall Workflow

Read URLs file

Check if new URLs need to be processed

Read/parse webpages; save json

Extract requirements; save json

Iter 0:

Flatten requirements

Flatten resume; save json

Calculate sim scores; save csv

Prune and save json

Iter 1:

Modify requirements; save flat json

Calculate sim scores; save csv

Prune and save json

Iter 2:

Modify requirements; save flat json

Calculate sim scores; save csv files

Prune and save each json

# Prompting Tricks

Use JSON or not?

(from GPT-o1, the newest model) From Using JSON format in prompts can indeed help guide language models like GPT to produce structured outputs that are easier to parse. Including a clear JSON schema or example in the prompt can encourage the model to respond in the desired format.

However, in the specific case of your prompt, embedding the tasks and instructions themselves within a JSON-like structure can potentially confuse the model. This is because the model might interpret the entire JSON structure as something to process or reproduce, rather than focusing on the specific tasks and the expected output.

Here's why including JSON in the prompt can be both helpful and tricky:

* Helpful: Providing a clear example of the desired JSON output format can guide the model to produce responses that are easier to parse programmatically.
* Tricky: If the prompt includes complex JSON structures for instructions or tasks, the model might misinterpret the instructions or include unwanted elements in the output.

Recommendations:

* Use JSON Format for Output Examples, Not Instructions:
* Keep your instructions in plain text for clarity.
* Provide the desired output format as a JSON example.
* Present tasks and instructions in bullet points or numbered lists.
* Use clear and direct language.
* Include a Clear JSON Schema or Example for the Expected Output:
* Show the exact JSON structure you expect in the response.
* Emphasize that the model should follow this format strictly.

## Pipeline Structure

Each process is a pipeline or mini pipeline. A pipeline manager, run\_pipeline function calls each pipeline using pipeline\_config.

### Pipeline Manager: run\_pipeline function to run all the pipelines

#### Code

def run\_pipeline(

    pipeline\_id: str, llm\_provider: str = "openai", model\_id: Optional[str] = None

):

    """

    \*Thi is the sync version

    Executes a dynamically specified pipeline based on its configuration in `PIPELINE\_CONFIG`.

    This function facilitates running various pipeline functions with unique configurations,

    dynamically handling both the function selection and required arguments:

    - Based on the 'pipeline\_id', the function retrieves a specific function name

    and configuration details from `PIPELINE\_CONFIG`.

    - Using 'globals()', the function name (stored as a string) is dynamically converted to

    a callable function.

    - Additionally, `run\_pipeline` leverages Python's `inspect` library to examine the function's

    signature, ensuring only the necessary arguments are passed.

    Note:

        \*'globals()' is used here as a lookup tool to dynamically access function objects

        \*based on their names stored as strings, not to make a function accessible outside

        \*of a local function.

    This approach supports flexible configurations, enabling `run\_pipeline` to handle

    multiple pipelines with distinct argument requirements without the need to hardcode

    specific functions or argument lists. This makes `run\_pipeline` adaptable to different

    processes and configurations, promoting modularity and reusability.

    Args:

        pipeline\_id (str): The identifier of the pipeline to execute. This ID is used to look up

            the configuration and function name in `PIPELINE\_CONFIG`.

        llm\_provider (str): The LLM provider to use, typically `"openai"` or `"claude"`.

            Defaults to `"openai"`.

        model\_id (str, optional): The specific model ID to use. If `None`, the function uses a default

            based on `llm\_provider` or the pipeline configuration.

    Raises:

        FileNotFoundError: If required input/output configurations are missing for the specified

            `llm\_provider` in `PIPELINE\_CONFIG`.

        ValueError: If a pipeline function is specified but is not found in the global namespace.

    Example:

        Suppose `PIPELINE\_CONFIG` has a pipeline with ID `"3b"` configured as follows:

            PIPELINE\_CONFIG = {

                "3b": {

                    "function": "run\_resume\_editing\_pipeline",

                    "io": {

                        "openai": {

                            "mapping\_file\_prev": "path/to/prev/file",

                            "mapping\_file\_curr": "path/to/curr/file"

                        }

                    }

                }

            }

        Then, to run this pipeline with OpenAI as the provider and a specific model:

            run\_pipeline("3b", llm\_provider="openai", model\_id="gpt-4-turbo")

        This will execute `run\_resume\_editing\_pipeline` with the following arguments:

        - `mapping\_file\_prev` (from `io\_config`)

        - `mapping\_file\_curr` (from `io\_config`)

        - `llm\_provider="openai"`

        - `model\_id="gpt-4-turbo"`

    How It Works:

    1. The function retrieves the specific configuration for `pipeline\_id` from

    `PIPELINE\_CONFIG`. This includes the function name and the relevant I/O configurations for

    the given `llm\_provider`.

    2. The `func\_name` is retrieved as a string and converted to a function object

    using 'globals()'.

    3. Using `inspect.signature`, the function examines the parameters required by 'func\_name'.

    4. A dictionary `kwargs` is initialized with values from `io\_config`.

    5. The `llm\_provider` and `model\_id` arguments are conditionally added to `kwargs`

    only if they are expected by the function, preventing unexpected argument errors.

    6. Finally, the dynamically selected function is executed with `func(\*\*kwargs)`.

    This dynamic approach ensures that `run\_pipeline` can adapt to different functions,

    configurations, and argument requirements, promoting flexibility and reducing

    the need for code duplication.

    """

    config = PIPELINE\_CONFIG[pipeline\_id]

    func\_name = config["function"]

    io\_config = config["io"][llm\_provider]

    # Use the provided model\_id or fallback to default if not provided

    model\_id = model\_id or config.get("model\_id", DEFAULT\_MODEL\_IDS.get(llm\_provider))

    logger.info(

        f"Running pipeline '{config['description']}' for provider '{llm\_provider}' with model ID '{model\_id}'"

    )

    # Prepare arguments dynamically based on the function's signature

    func = globals()[func\_name]  # Retrieve the function object from globals

    func\_signature = inspect.signature(func)  # Get function's parameters

    kwargs = io\_config.copy()  # Start with io\_config as the base arguments

    # Add llm\_provider and model\_id only if the function accepts them

    if "llm\_provider" in func\_signature.parameters:

        kwargs["llm\_provider"] = llm\_provider

    if "model\_id" in func\_signature.parameters:

        kwargs["model\_id"] = model\_id

    # Log for debugging purposes

    logger.info(f"Calling function '{func\_name}' with kwargs: {kwargs}")

    # Call the function with the dynamically prepared kwargs

    func(\*\*kwargs)

#### Code Explanation:

In this setup, we build a function, `run\_pipeline`, to handle multiple pipeline processes with varying requirements in a dynamic, flexible way. The function needs to decide which specific pipeline function to call based on an input parameter (`pipeline\_id`). Each pipeline function might require a different set of arguments, so instead of hardcoding specific function names and arguments, we create a general method that can adapt to different configurations.

To begin, `run\_pipeline` accesses the appropriate function name from a configuration dictionary called `PIPELINE\_CONFIG`, which maps each `pipeline\_id` to a particular function and its input/output files. The function name is retrieved as a string, and `globals()[func\_name]` uses this string to access the actual function in the global namespace.

By using `globals()`, we’re able to call the right function dynamically without hardcoding it, making our code more modular and flexible. This approach allows `run\_pipeline` to call any function simply by specifying its name in the configuration.

Globals() access the internal Symbols table that contains function references (each time you create a function or class, it is saved in the Symbols table.

Once we have the function, we also need to decide which arguments to pass to it, as different functions may require different arguments. To achieve this, we use Python’s `inspect` library to examine the function’s signature and determine the names of the parameters it expects. This step lets us conditionally add parameters like `llm\_provider` or `model\_id` to the argument dictionary only if the function actually requires them. We start by copying `io\_config`, which contains general input/output configurations, into `kwargs`, the dictionary of arguments that will eventually be passed to the function. Using `inspect`, we check if `llm\_provider` and `model\_id` are parameters in the function’s signature; if they are, we add them to `kwargs`. This ensures that each function only receives the arguments it needs, preventing errors that would occur if we passed unexpected arguments.

* io\_config contains pre-defined input/output files for each provider (openai, claude, etc.) that are specific to each pipeline.
* kwargs = io\_config.copy() initializes kwargs with the arguments from io\_config so that they’re always included.

Finally, we call the function with ‘func(\*\*kwargs)’, using `kwargs` to dynamically supply the arguments. This approach allows `run\_pipeline` to remain highly flexible, as it can now support any pipeline function by adjusting to the required arguments at runtime. The result is a single, adaptable function that can handle different pipelines with unique configurations, reducing redundancy and enhancing code reusability. This design makes it easy to add or modify pipelines with minimal changes to the codebase, as `run\_pipeline` adjusts automatically based on the configuration and the specific requirements of each pipeline.

# Similarity Scores and Related Metrics

## Directory Organization

Suggested by GPT

src/

└── evaluation\_optimization/

├── \_\_init\_\_.py # Makes this directory a package

├── metrics/

│ ├── \_\_init\_\_.py # Exposes metric calculation functions/classes

│ ├── metric\_definitions.py # Defines various metrics

│ ├── metric\_calculator.py # Classes/functions for finding and calculating metrics

├── indices/

│ ├── \_\_init\_\_.py # Exposes index calculation functions/classes

│ ├── multivariate\_indexer.py # Class for handling multivariate index creation

├── changes/

│ ├── \_\_init\_\_.py # Exposes change calculation functions/classes

│ ├── change\_calculator.py # Methods for calculating changes (absolute/percentage)

├── analysis/

│ ├── \_\_init\_\_.py # Exposes analysis-related methods

│ ├── descriptive\_stats.py # Descriptive statistics calculation

│ ├── plotting.py # Plotting functions for quick visualization

└── utils/

├── \_\_init\_\_.py # Utility functions

├── data\_merger.py # Merge DataFrames across iterations

├── data\_loader.py # Functions for loading data

I will not go this detailed - I don’t have that much code.

## Current Similarity Scores

Based on the provided similarity scores for the responsibilities against the job requirements, here's a quick summary and analysis:

### Summary of Similarity Scores:

1. \*\*Similarity Metrics Used:\*\*

- Several metrics have been used to measure the similarity between each responsibility and the reference job requirements text:

- \*\*Self Attention Similarity (`self\_attention\_similarity`)\*\*

- \*\*Layer-wise Attention Similarity (`layer\_wise\_attention\_similarity`)\*\*

- \*\*Self Hidden State Similarity (`self\_hidden\_state\_similarity`)\*\*

- \*\*CLS Embedding Similarity (`cls\_embedding\_similarity`)\*\*

- \*\*SBERT Similarity (`sbert\_similarity`)\*\*

- \*\*STS Similarity (`sts\_similarity`)\*\*

- An \*\*Average\*\* score is provided to give an overall similarity measure for each responsibility.

2. \*\*Highest Similarity Scores:\*\*

- The responsibility \*\*"Co-authored an industry-recognized report on M&A in the engineering services sector..."\*\* has the highest \*\*Average similarity score (0.390)\*\*. It also scores highly across multiple metrics, especially \*\*`cls\_embedding\_similarity (0.810)`\*\* and \*\*`sts\_similarity (0.663)`\*\*. This indicates that this responsibility has the strongest alignment with the job requirements.

3. \*\*Moderate Similarity Scores:\*\*

- The responsibilities \*\*"Provided strategic insights to a major global IT vendor..."\*\* and \*\*"Assisted a U.S.-based international services provider in its growth strategy..."\*\* both have moderate \*\*Average similarity scores (0.313)\*\*. These scores are boosted by relatively high \*\*`cls\_embedding\_similarity`\*\* values (0.808 and 0.728, respectively), indicating that while these responsibilities have some relevance, their overall alignment is moderate.

4. \*\*Lower Similarity Scores:\*\*

- The responsibilities \*\*"Achieved over 40% centralization of tasks to an offshore team..."\*\* and \*\*"Developed Python tools to automate and accelerate internal processes..."\*\* have the lowest \*\*Average similarity scores (0.268 and 0.266, respectively)\*\*. Their lower scores are reflected across all metrics, with relatively low \*\*`self\_attention\_similarity`\*\*, \*\*`layer\_wise\_attention\_similarity`\*\*, and \*\*`self\_hidden\_state\_similarity`\*\* values. This suggests these responsibilities are less aligned with the job requirements.

5. \*\*Key Observations:\*\*

- \*\*`cls\_embedding\_similarity`\*\* tends to have the highest values across all responsibilities, indicating that the \*\*CLS token embedding\*\* similarity (a standard method of representing sentence-level embeddings in BERT models) captures a significant amount of the semantic alignment.

- \*\*`sbert\_similarity`\*\* and \*\*`sts\_similarity`\*\* are also relatively high for certain responsibilities, indicating semantic similarity from sentence embeddings and sentence similarity models.

- The metrics that seem to contribute most to the \*\*Average\*\* similarity score are \*\*`cls\_embedding\_similarity`\*\*, \*\*`sbert\_similarity`\*\*, and \*\*`sts\_similarity`\*\*, while the \*\*attention-based similarities\*\* (`self\_attention\_similarity`, `layer\_wise\_attention\_similarity`, `self\_hidden\_state\_similarity`) tend to be much lower.

### Analysis:

1. \*\*Responsibility Alignment with Job Requirements:\*\*

- The most aligned responsibility appears to be related to strategic insights and market analysis, which matches closely with several job requirements such as "performing market analysis and developing competitive intelligence" and "management consulting, product management, and strategy."

- Less aligned responsibilities, such as "Developed Python tools..." or "Achieved over 40% centralization of tasks...", may not directly match the core requirements like "knowledge of AI and ML" or "ability to form and refine hypotheses."

2. \*\*Diverse Requirement Coverage:\*\*

- Responsibilities that involve \*\*strategic decision-making, market analysis, and stakeholder management\*\* have generally higher similarity scores, aligning well with the provided job requirements which emphasize \*\*consulting, management, strategy, and technical expertise.\*\*

3. \*\*Potential Gaps:\*\*

- Technical responsibilities such as developing Python tools or managing offshore teams show weaker alignment, suggesting they may not be the primary focus of the job requirements provided, which focus more on strategic, analytical, and consulting skills.

### Conclusion:

- The provided similarity scores effectively highlight which responsibilities align most closely with the job requirements and which are less relevant. This kind of analysis helps identify the most valuable experiences on a resume that match the expectations for a specific role.

If you need further detailed insights or additional analyses, feel free to ask!

## Better Approaches for Asymmetric Matching

For cases like this, where alignment of responsibilities to requirements matters more in one direction, consider using asymmetric similarity or matching techniques:

### Soft Cosine Measure (SCM):

How It Works: Soft Cosine Measure takes into account the similarity between words that are not identical but related. For example, it can measure the similarity between "managing" and "leading" responsibilities.

Applicability: This can be more effective than cosine similarity because it considers the semantic closeness of words across the entire text. However, it would still be limited in representing the true asymmetric relationship you're looking for.

### Word Mover's Distance (WMD):

How It Works: WMD computes the minimal distance that words from one text need to travel to match the words in another text. It accounts for the semantic distance between words using pre-trained embeddings.

Applicability: WMD can capture the cost of aligning each word in "responsibilities" to a word in "requirements," potentially reflecting the asymmetry more naturally. However, this approach could still miss specific context, such as the order or relationship among responsibilities.

### Jaccard or Containment-Based Similarity Measures:

How It Works: You can calculate the containment ratio—the proportion of overlapping elements in the "responsibilities" found in "requirements." This is asymmetric since it measures how well one set (responsibilities) is contained in another set (requirements).

Applicability: This is more directly relevant when you want to assess how well one text (responsibilities) fits into another (requirements). However, it requires tokenization and does not inherently handle deep semantic meanings.

### Textual Entailment or Alignment Models:

How It Works: Models designed for textual entailment assess whether one text (hypothesis) logically follows from another text (premise). Here, "responsibilities" would be the hypothesis and "requirements" the premise.

Applicability: This could be a robust approach to verify if all responsibilities are entailed by the requirements, thereby capturing an asymmetric relationship.

### Custom Asymmetric Metric:

How It Works: Create a custom metric that calculates the coverage of terms or concepts from "responsibilities" in "requirements."

This could involve:

* Tokenizing both texts.
* Calculating the number of tokens or concepts in "responsibilities" that are covered by "requirements."
* Measuring how much of the "responsibilities" content is present in "requirements" (as a percentage or ratio).
* Applicability: This can be specifically tailored to ensure full coverage and provide a more meaningful measure of alignment in your context.

## Suggested Scoring Framework for Asymmetric Relationship (By ChatGPT)

Here’s how you could assign asymmetric similarity scores:

### Direct Matching Score (DMS):

Calculate the proportion of key terms or concepts in the requirement that are covered by the responsibility.

Example: If a requirement has 5 key terms and the responsibility covers 4 of them, the score could be 4/5 = 0.80.

### Partial Coverage Score (PCS):

Go beyond just term matching by considering synonyms and related concepts. This involves using a semantic similarity model to determine if terms in the requirement are represented in the responsibility, even if not verbatim.

Weighting can be added to different terms in the requirement based on their importance. For example, "project management" might be more critical than "basic programming skills."

### Entailment Probability Score (EPS):

Use a textual entailment model to predict the likelihood that the content of the requirement is entailed by the responsibility. The output probability can serve as the similarity score.

This is especially useful for nuanced requirements that are not straightforward or binary but involve complex conditions or criteria.

### Semantic Coverage Ratio (SCR):

For each requirement, compute the semantic overlap with the responsibility using a method like the Soft Cosine Similarity which accounts for semantic similarity between words.

This ratio could be defined as the sum of the similarities of matching terms divided by the total number of unique terms in the requirement.

## New Custom Similarity Scores to Consider for the Project

BertScore Precision, Soft Similarity, Word Mover's Distance, NLI models, Jaccard Similarity

### Scores to consider:

Asymmetrical:

* BertScore Precision (semantic coverage)
* Soft Similarity (semantic coverage): Soft Similarity (e.g., Soft Cosine Similarity) considers the similarity between words that are not identical but are semantically related. It enhances semantic coverage by recognizing related but not identical terms.
* Word Mover's Distance (WMD) (for a mix of semantic and partial coverage)
* NLI models (for entailment)

Symmetrical:

* Jaccard Similarity (partial coverage): Calculates the size of the intersection divided by the size of the union of two sets. It's suitable for text data, especially when dealing with keywords or phrases.

### Comprehensive Framework for Asymmetric Text Similarity Analysis:

1. BERTScore Precision:

* Focus: Measures how well the semantic content of one text (e.g., a responsibility) aligns with another text (e.g., a requirement), focusing on token-level precision using contextual embeddings.
* Use Case: Effective for capturing semantic coverage by identifying how much of the requirement's meaning is present in the responsibility. It is asymmetric in nature because it focuses on how much of the requirement is covered by the responsibility.
* Strength: Captures semantic nuances and is robust to word order changes, making it effective for understanding the alignment of texts where semantic similarity is critical.

1. Soft Similarity:

* Focus: Computes similarity by considering semantically related terms, not just exact matches. It accounts for the similarity between words that are related but not identical.
* Use Case: Enhances semantic coverage by recognizing partial matches, synonyms, and related terms. It is useful when a requirement can be met in different ways that involve related but not identical terminology.
* Strength: Provides more flexibility than strict semantic measures like BERTScore Precision by allowing for partial matches and relatedness.

1. Word Mover's Distance (WMD):

* Focus: Measures the minimum cumulative distance that words from one text need to "travel" in the embedding space to match the words in another text, capturing both semantic similarity and partial coverage.
* Use Case: Useful for a mix of semantic alignment and content alignment when there are significant overlaps or partial matches between texts.
* Strength: Provides a nuanced similarity measure that accounts for both semantic distance and partial coverage, capturing the degree to which one text can be transformed into another.

1. NLI Models (e.g., BERT-NLI, RoBERTa-NLI, DeBERTa-NLI):

* Focus: Predicts whether one text (e.g., a responsibility) entails, contradicts, or is neutral to another text (e.g., a requirement). NLI models are specifically designed to handle asymmetric entailment tasks.
* Use Case: Ideal for detecting whether a responsibility logically entails a requirement or supports it directionally. This is crucial for asymmetric relationships where you want to know if a responsibility fulfills a requirement without assuming the reverse.
* Strength: Provides a binary or probabilistic output for entailment, making it ideal for fine-tuning decisions where logical entailment is required.

1. Jaccard Similarity:

* Focus: Measures the exact overlap between sets of terms from two texts, focusing on partial coverage. It calculates the ratio of the intersection of terms to the union.
* Use Case: Useful for cases where exact term overlap is important or where a requirement must be explicitly mentioned in the responsibility.
* Strength: Simple, intuitive, and provides a clear measure of overlap. It is most effective when the goal is to quantify how much of the content from one text is present in another, especially in more rigid matching scenarios.

### What Ranges Are Considered High or Low

### a) Interpreting High vs. Low Scores for Each Metric:

1. \*\*BERTScore Precision:\*\*

- \*\*Range:\*\* 0 to 1

- \*\*High Scores:\*\* Generally above 0.85. A high score indicates that the tokens in the candidate text (e.g., responsibility) are well-aligned with the reference text (e.g., requirement) in terms of semantic similarity.

- \*\*Low Scores:\*\* Below 0.70. A low score suggests that the candidate and reference texts do not share much semantic similarity.

- \*\*Typical Use:\*\* Used for evaluating precision when the goal is to see how much of the candidate content is relevant to the reference.

2. \*\*Soft Similarity (SBERT Cosine Similarity):\*\*

- \*\*Range:\*\* -1 to 1

- \*\*High Scores:\*\* Above 0.7. High scores indicate that the embeddings of the two texts are close, meaning they are semantically similar.

- \*\*Low Scores:\*\* Below 0.4. Low scores suggest that the texts are semantically different.

- \*\*Typical Use:\*\* Measures overall semantic similarity, often used in tasks like paraphrase detection and information retrieval.

3. \*\*Word Mover's Distance (WMD):\*\*

- \*\*Range:\*\* 0 to infinity (practically within 0 to ~20 for most text)

- \*\*High Scores:\*\* (i.e., smaller distance) Below 5 is considered a good match. A smaller distance means that the texts are closer semantically.

- \*\*Low Scores:\*\* Above 15 is considered a poor match. A larger distance indicates that the texts are semantically different.

- \*\*Typical Use:\*\* Used in text retrieval and text similarity tasks where the goal is to measure how far the meaning of one text is from another.

4. \*\*NLI Entailment Score:\*\*

- \*\*Range:\*\* 0 to 1

- \*\*High Scores:\*\* Above 0.7. Indicates a strong likelihood that the hypothesis (responsibility) is entailed by the premise (requirement).

- \*\*Low Scores:\*\* Below 0.3. Suggests that the hypothesis is not supported or entailed by the premise.

- \*\*Typical Use:\*\* Captures directional semantic entailment; helpful in tasks like textual entailment and natural language understanding.

5. \*\*Jaccard Similarity:\*\*

- \*\*Range:\*\* 0 to 1

- \*\*High Scores:\*\* Above 0.5. Indicates a significant overlap in the tokens or words between the texts.

- \*\*Low Scores:\*\* Below 0.2. Suggests minimal overlap between the texts.

- \*\*Typical Use:\*\* Measures lexical similarity based on shared words or tokens. It does not capture semantic similarity as effectively as other methods.

### b) Variations in Scores and Applicability in ML Models:

To make these scores "applicable" in ML models, variations in scores should capture meaningful differences in relevance, similarity, or entailment between text pairs. Here’s how variations in each score type can be leveraged:

1. \*\*BERTScore Precision:\*\*

- \*\*Small Variations (e.g., 0.85 vs. 0.88):\*\* May indicate minor differences in precision; not always significant. In ML models, such variations can help fine-tune thresholds for classification.

- \*\*Large Variations (e.g., 0.65 vs. 0.90):\*\* Indicative of substantial differences in text relevance. Useful for ranking, filtering, or classification tasks.

2. \*\*Soft Similarity:\*\*

- \*\*Small Variations (e.g., 0.65 vs. 0.68):\*\* Generally not impactful in ML models. Variations need to be more pronounced (e.g., 0.65 vs. 0.80) to influence model decisions.

- \*\*Large Variations (e.g., 0.35 vs. 0.80):\*\* Strongly suggest semantic differences. Good for tasks like paraphrase detection or semantic search.

3. \*\*Word Mover's Distance (WMD):\*\*

- \*\*Small Variations (e.g., 7 vs. 8):\*\* Generally minor; a model might treat them similarly.

- \*\*Large Variations (e.g., 3 vs. 15):\*\* Indicate a strong difference in semantic content. Useful for clustering, anomaly detection, and retrieval tasks where semantic distance matters.

4. \*\*NLI Entailment Score:\*\*

- \*\*Small Variations (e.g., 0.6 vs. 0.65):\*\* May not indicate meaningful entailment differences; could be treated similarly.

- \*\*Large Variations (e.g., 0.2 vs. 0.8):\*\* Clearly distinguishes between entailment and non-entailment. Crucial for entailment-based classification tasks or filtering relevant text.

5. \*\*Jaccard Similarity:\*\*

- \*\*Small Variations (e.g., 0.15 vs. 0.20):\*\* Usually not significant unless coupled with other features.

- \*\*Large Variations (e.g., 0.10 vs. 0.60):\*\* Clear indication of token overlap. Effective for text matching tasks where token presence is key (e.g., keyword matching).

### Combining Scores in ML Models:

- \*\*Feature Engineering:\*\* Combining these metrics as features in an ML model can provide a robust representation of text similarity. Each metric captures different aspects of text similarity (e.g., semantic, lexical, entailment).

- \*\*Thresholding:\*\* Models can be trained to learn optimal thresholds for each metric to determine relevance or similarity.

- \*\*Weighted Averaging:\*\* Some applications may benefit from weighted combinations of these metrics to emphasize certain types of similarity (e.g., semantic over lexical).

- \*\*Ensemble Models:\*\* Use ensemble methods (e.g., stacking, bagging) to leverage multiple similarity metrics to make more accurate predictions.

By understanding how each score works and the implications of their variations, you can design more effective text similarity models tailored to specific applications in NLP.

BERTScore Precision:

Low Scores: If the BERTScore Precision is low (e.g., around 0.82 or lower), it means there is less overlap between the tokens (words or phrases) in the responsibility text and the tokens in the reference text. In this case, the responsibility might lack key phrases or terms that are explicitly stated in the job requirements.

Implication: Such responsibilities may not directly speak to the required qualifications or experiences. For instance, "Oversaw two Appian implementations that increased team productivity by 20 to 30%" (0.8297) may not directly align with the specifics of AI, machine learning, or the other strategic management and analytical skills required by the job.

Soft Similarity:

Low Scores: A low soft similarity score (e.g., below 0.30) indicates that the semantic meaning of the responsibility diverges from the meaning of the requirements. This metric is more sensitive to the overall context and concepts rather than exact word matches.

Implication: Responsibilities with low soft similarity scores likely do not cover the core topics or themes emphasized in the job description. For example, "First to implement API (application programming interface)" (0.1577) is quite low, suggesting it doesn't capture broader themes like managing cross-functional teams or strategic AI development.

Word Mover's Distance (WMD):

High Scores: Since WMD measures distance (where a lower distance is better), higher values indicate that more "movement" (word conversion or alignment) is needed to match the two texts.

Implication: Responsibilities with higher WMD scores (closer to 15.9) are more distanced from the reference text in terms of the words and their meanings. For example, "Collaborated with the engineering services research team..." (15.91) suggests it is less lexically aligned with the reference compared to others with lower scores.

NLI Entailment Score:

Low Scores: A low entailment score (e.g., below 0.05) implies that the requirement text (premise) is unlikely to be inferred from the responsibility text (hypothesis).

Implication: This might mean that the experience or responsibility described doesn't logically lead to the conclusion that the candidate meets specific requirements. For example, "Developed Python tools to automate and accelerate internal processes..." (0.0025) has a very low entailment score, indicating it may not provide evidence for the ability to "manage executive stakeholders" or "shape the future of AI."

Jaccard Similarity:

Low Scores: Jaccard Similarity scores that are low (e.g., below 0.05) indicate minimal direct word overlap between the texts.

Implication: Low Jaccard scores, such as "First to launch a chatbot" (0.0267), suggest that there are not many shared terms between the responsibility and the requirements, making it less likely that this experience is directly relevant to the role.

### Outputs

#### Responsibility w/t Requirements as a Whole



##### Interpretation (by GPT)

Thank you for providing the full reference text. With this reference, the similarity metrics can be interpreted more specifically in the context of how well the responsibilities align with this detailed set of requirements for a role. Here's a refined interpretation with the given reference in mind:

### Contextual Interpretation:

1. \*\*BERTScore Precision\*\*:

- The BERTScore Precision metric indicates how much the content of the responsibility overlaps with the content of the reference text (the job requirements). Given the reference emphasizes a variety of skills—ranging from technical knowledge, management, market analysis, data handling, problem-solving, and communication—higher scores imply that the responsibility has keywords or phrases that match the job requirements closely.

- For example:

- "First to build an ontology" has a high precision score (0.8695), suggesting it aligns well with the technical and possibly the strategic elements in the reference, such as "knowledge of the Machine Learning and Artificial Intelligence market landscape" and "building assets and programs that surface valuable insights."

- "Led the external software development team to build and implement new tools" also scores highly (0.8562), which aligns with requirements like "Work across Program Management teams" and "manage executive stakeholders and communicate with a highly technical management team."

2. \*\*Soft Similarity\*\*:

- This metric captures the semantic similarity and is sensitive to the meaning of words rather than their exact match. Responsibilities with higher soft similarity scores suggest they capture broader themes or underlying concepts present in the requirements.

- For example: "Championed new technology projects using ML, NLP, chatbot, ontology, web-scraping, API, UX (User Experience)" scores the highest (0.4332). This reflects its strong conceptual overlap with the reference's focus on advanced AI/ML concepts and strategic planning, such as "Help shape the future of AI" and "build assets and programs that surface valuable insights."

3. \*\*Word Mover's Distance (WMD)\*\*:

- WMD values reflect how much transformation is needed to convert the words in the responsibility to those in the reference. Lower WMD indicates a closer match.

- Responsibilities with WMD values closer to 15.4-15.5 are more semantically aligned:

- "Managed three major data integration projects critical to the launch of the new platform" (15.43) and "First to build an ontology" (15.46) have among the lowest WMD values, indicating they contain words and concepts closely related to those in the reference text.

4. \*\*NLI Entailment Score\*\*:

- This metric indicates the likelihood that the "premise" (requirement) can be inferred from the "hypothesis" (responsibility). Higher entailment scores suggest a stronger inferential relationship.

- For instance:

- "First to build an ontology" (0.1715) and "Advised services firms on deal pursuit and sales orchestration strategies" (0.0870) have higher entailment scores. These responsibilities may suggest an ability to "form and refine hypotheses, gather supporting data, and make recommendations," which is a key requirement in the reference text.

5. \*\*Jaccard Similarity\*\*:

- This metric is less influenced by semantic meaning and more by direct word overlap. Responsibilities with a slightly higher Jaccard similarity, like "Collaborated with the engineering services research team to pioneer the engineering services tracker..." (0.0802), suggest there are more common words between the responsibility and the reference text, though it does not guarantee semantic alignment.

Summary

The provided metrics allow us to measure alignment between the responsibilities and a comprehensive set of job requirements that span technical skills, strategic capabilities, and management expertise. Different metrics emphasize different aspects of similarity:

* \*\*BERTScore Precision\*\* and \*\*NLI Entailment Score\*\* are useful for understanding direct and inferential content alignment.
* \*\*Soft Similarity\*\* and \*\*Word Mover's Distance\*\* provide insights into broader semantic relationships.
* \*\*Jaccard Similarity\*\* indicates basic lexical overlap.

By combining these metrics, we can holistically assess how well a given responsibility matches a set of complex requirements, providing a nuanced understanding of alignment beyond just word matching.

Conclusion: Comparing responsibility to the entire requirements is not a good measure.

### Responsibility vs Requirement (Seg by Seg)

A screenshot of a computer screen

Description automatically generated

**Much better results.**

#### Interpretation:

BERTScore Precision:

* The mean score is around 0.83, with a narrow standard deviation of 0.017, indicating that most values are closely clustered around the mean.
* Scores range from 0.78 to 0.88, suggesting a relatively high level of precision in matching between texts.
* A low range and high mean indicate that most text pairs have relatively good semantic precision.

Soft Similarity (SBERT):

* The mean score is 0.26, with a standard deviation of 0.11. This suggests more variability in soft similarity compared to BERTScore Precision.
* Scores range from 0.03 to 0.56, showing that there is a broader range of semantic similarity among the text pairs.
* The distribution suggests that while some pairs are semantically similar, many have low similarity.

Word Mover's Distance (WMD):

* The mean score is around 5.07, with a standard deviation of 0.90. WMD values range from 3.61 to 8.19, which is within a reasonable range for text comparisons.
* Lower WMD scores indicate closer semantic similarity. Since the 25th percentile is 4.36 and the 75th percentile is 5.57, most text pairs fall into a moderate distance range.

DeBERTa Entailment Score:

* The mean score is 0.079, with a high standard deviation of 0.14, indicating a broad variation in entailment probabilities.
* Scores range from 0.0004 to 0.88, suggesting that some pairs have very high entailment, while others have almost none.
* The median score is 0.0277, indicating that over half of the pairs have low entailment, which aligns with the typical usage of entailment in detecting strong directional relationships.

Conclusion:

* BERTScore Precision is consistently high across text pairs, indicating strong semantic overlap or matching.
* Soft Similarity (SBERT) shows more variation, suggesting mixed levels of semantic similarity.
* Word Mover's Distance (WMD) is generally moderate, indicating the semantic distances are neither too close nor too far apart.
* DeBERTa Entailment Score shows considerable variability, highlighting that entailment is detected in only some text pairs, which aligns with the directional nature of this metric.

These statistics look appropriate and can be utilized for various downstream ML tasks, such as classification, ranking, or filtering.

## What to Filter Out

Not all 4 metrics can be low

BERTScore precision should not be low;

# Editing Responsibilities

Temperature

When comparing candidate vs reference text and modifying the candidate, temperature affects the modification process:

Low Temperature (e.g., 0.1-0.3)

1. Minimal changes: Candidate text modifications will be subtle, preserving the original meaning and structure.

2. Localized edits: Changes will focus on specific words or phrases, rather than rephrasing entire sentences.

3. Conservative rewording: Model will prioritize preserving the original text's tone, style, and syntax.

4. Less semantic drift: Candidate text will remain closer to the original meaning.

High Temperature (e.g., 0.7-1.0)

1. More significant changes: Candidate text modifications will be more substantial, potentially altering sentence structure and meaning.

2. Global revisions: Changes may rephrase entire sentences or paragraphs, introducing new ideas.

3. Creative rewording: Model will take more liberties with tone, style, and syntax.

4. Increased semantic drift: Candidate text may deviate further from the original meaning.

Comparison highlights

| Temperature | Modification style | Semantic drift |

| --- | --- | --- |

| Low | Subtle, localized | Minimal |

| High | Significant, global | Increased |

Implications for comparison

1. Low temperature: Suitable for evaluating minor variations, grammatical corrections, or subtle rewording.
2. High temperature: Useful for assessing more significant changes, creative rewrites, or paraphrasing.

When modifying the candidate text, consider:

1. Your specific comparison goals.

2. The reference text's complexity and style.

3. The desired level of semantic preservation.

Adjust the temperature accordingly to achieve the desired modification style.

Would you like more information on fine-tuning temperature or exploring other comparison metrics?

Optimal Temperatures to Set:

* text\_editor.edit\_for\_semantics(temperature=0.5) (if you set this too high, the end sentence till be too long)
* text\_editor.edit\_for\_entailment(temperature=0.6)
* text\_editor.edit\_for\_dp(temperature=0.9)

# Evaluate Results

## Composite Scores

## PCA Scores

Below is the code that incorporates the PCA approach into a method within your existing class. This method is modeled after your `calculate\_composite\_score` method and handles the DataFrame with columns that have indices (e.g., `soft\_similarity\_0`, `soft\_similarity\_1`, etc.). It standardizes the necessary columns, normalizes and reverses the polarity of `word\_movers\_distance`, and applies PCA to reduce the metrics to a single composite score.

```python

def calculate\_pca\_composite\_score(self, df):

"""

Calculate the composite score using PCA based on four metrics.

Parameters:

- df: DataFrame containing the metrics.

Returns:

- df: DataFrame with the composite score added.

"""

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler, StandardScaler

from sklearn.decomposition import PCA

# Step 1: Identify the columns for each metric

bert\_columns = [col for col in df.columns if col.startswith('bert\_score\_precision')]

soft\_similarity\_columns = [col for col in df.columns if col.startswith('soft\_similarity')]

word\_movers\_columns = [col for col in df.columns if col.startswith('word\_movers\_distance')]

deberta\_entailment\_columns = [col for col in df.columns if col.startswith('deberta\_entailment\_score')]

# Step 2: Calculate the mean of the metrics if there are multiple columns

df['bert\_score\_precision\_mean'] = df[bert\_columns].mean(axis=1)

df['soft\_similarity\_mean'] = df[soft\_similarity\_columns].mean(axis=1)

df['word\_movers\_distance\_mean'] = df[word\_movers\_columns].mean(axis=1)

df['deberta\_entailment\_score\_mean'] = df[deberta\_entailment\_columns].mean(axis=1)

# Step 3: Normalize word\_movers\_distance and reverse polarity

scaler\_wmd = MinMaxScaler()

word\_movers\_distance\_normalized = scaler\_wmd.fit\_transform(

df['word\_movers\_distance\_mean'].values.reshape(-1, 1)

).flatten()

word\_movers\_distance\_adjusted = 1 - word\_movers\_distance\_normalized # Lower distance is better

# Step 4: Create a DataFrame with the metrics

metrics\_df = pd.DataFrame({

'bert\_score\_precision': df['bert\_score\_precision\_mean'],

'soft\_similarity': df['soft\_similarity\_mean'],

'word\_movers\_distance': word\_movers\_distance\_adjusted,

'deberta\_entailment\_score': df['deberta\_entailment\_score\_mean']

})

# Step 5: Standardize the metrics

scaler\_standard = StandardScaler()

metrics\_standardized = scaler\_standard.fit\_transform(metrics\_df)

# Step 6: Apply PCA to reduce to a single composite score

pca = PCA(n\_components=1)

principal\_component = pca.fit\_transform(metrics\_standardized)

# Step 7: Add the composite score back to the DataFrame

df['composite\_score'] = principal\_component.flatten()

return df

```

### \*\*Explanation of the Method:\*\*

#### \*\*Imports:\*\*

- \*\*`numpy` and `pandas`\*\*: For data manipulation.

- \*\*`MinMaxScaler` and `StandardScaler`\*\*: For normalization and standardization.

- \*\*`PCA`\*\*: For principal component analysis.

#### \*\*Step 1: Identify the Columns for Each Metric\*\*

- \*\*Metric Column Selection\*\*: Use list comprehensions to find all columns that start with each metric's prefix.

```python

bert\_columns = [col for col in df.columns if col.startswith('bert\_score\_precision')]

soft\_similarity\_columns = [col for col in df.columns if col.startswith('soft\_similarity')]

word\_movers\_columns = [col for col in df.columns if col.startswith('word\_movers\_distance')]

deberta\_entailment\_columns = [col for col in df.columns if col.startswith('deberta\_entailment\_score')]

```

#### \*\*Step 2: Calculate the Mean of the Metrics\*\*

- \*\*Handle Multiple Columns\*\*: If there are multiple columns for a metric (e.g., `soft\_similarity\_0`, `soft\_similarity\_1`), calculate their mean for each row.

```python

df['soft\_similarity\_mean'] = df[soft\_similarity\_columns].mean(axis=1)

```

- \*\*Repeat for All Metrics\*\*: Do the same for `bert\_score\_precision`, `word\_movers\_distance`, and `deberta\_entailment\_score`.

#### \*\*Step 3: Normalize `word\_movers\_distance` and Reverse Polarity\*\*

- \*\*Normalize\*\*: Scale `word\_movers\_distance\_mean` to a range between 0 and 1.

```python

scaler\_wmd = MinMaxScaler()

word\_movers\_distance\_normalized = scaler\_wmd.fit\_transform(

df['word\_movers\_distance\_mean'].values.reshape(-1, 1)

).flatten()

```

- \*\*Reverse Polarity\*\*: Since a lower distance is better, subtract from 1.

```python

word\_movers\_distance\_adjusted = 1 - word\_movers\_distance\_normalized

```

#### \*\*Step 4: Create a DataFrame with the Metrics\*\*

- \*\*Combine Metrics\*\*: Create a new DataFrame `metrics\_df` with the mean values of each metric.

```python

metrics\_df = pd.DataFrame({

'bert\_score\_precision': df['bert\_score\_precision\_mean'],

'soft\_similarity': df['soft\_similarity\_mean'],

'word\_movers\_distance': word\_movers\_distance\_adjusted,

'deberta\_entailment\_score': df['deberta\_entailment\_score\_mean']

})

```

#### \*\*Step 5: Standardize the Metrics\*\*

- \*\*Standardization\*\*: Transform the metrics to have a mean of zero and a standard deviation of one.

```python

scaler\_standard = StandardScaler()

metrics\_standardized = scaler\_standard.fit\_transform(metrics\_df)

```

#### \*\*Step 6: Apply PCA\*\*

- \*\*PCA Transformation\*\*: Reduce the four standardized metrics to one principal component.

```python

pca = PCA(n\_components=1)

principal\_component = pca.fit\_transform(metrics\_standardized)

```

- \*\*Principal Component\*\*: The resulting `principal\_component` is an array with one value per row.

#### \*\*Step 7: Add the Composite Score to the Original DataFrame\*\*

- \*\*Add to DataFrame\*\*: Add the composite score as a new column in `df`.

```python

df['composite\_score'] = principal\_component.flatten()

```

### \*\*Usage Example:\*\*

Assuming you have an instance of your class and your DataFrame is named `df`, you can call the method as follows:

```python

# Calculate the composite score and update the DataFrame

df = self.calculate\_pca\_composite\_score(df)

```

### \*\*Notes and Considerations:\*\*

- \*\*Adjust Column Prefixes\*\*: Ensure that the prefixes used in `col.startswith('metric\_prefix')` match your DataFrame's column naming convention.

- \*\*Missing Values\*\*: Handle any missing values in your data before applying PCA (e.g., fill with mean or median).

- \*\*Multiple Columns per Metric\*\*: The method averages all columns for each metric. If you need to select specific columns, adjust the method accordingly.

- \*\*Explained Variance\*\*: You can access the explained variance of the principal component if needed:

```python

explained\_variance = pca.explained\_variance\_ratio\_[0]

print(f"Explained Variance: {explained\_variance:.2%}")

```

### \*\*Example Output:\*\*

After running the method, your DataFrame `df` will have a new column `composite\_score`:

```plaintext

soft\_similarity\_0 soft\_similarity\_1 ... deberta\_entailment\_score\_mean composite\_score

0 0.75 0.80 ... 0.85 1.234567

1 0.60 0.65 ... 0.70 0.987654

...

```

### \*\*Additional Options:\*\*

- \*\*Using Specific Columns\*\*: If you prefer to use specific columns instead of averaging, you can modify Step 2:

```python

df['soft\_similarity\_mean'] = df['soft\_similarity\_0'] # Use only 'soft\_similarity\_0'

```

- \*\*Including Weights\*\*: PCA does not incorporate weights directly. If you need to apply specific weights to the metrics, consider using a weighted sum approach instead.

### \*\*Final Thoughts:\*\*

- \*\*PCA vs. Weighted Sum\*\*: PCA is suitable when you want to combine correlated metrics into a single score based on their variance structure. If you have predefined weights for each metric, a weighted sum might be more appropriate.

- \*\*Interpretability\*\*: The composite score from PCA is a linear combination of the standardized metrics. The weights (loadings) can be accessed via `pca.components\_` if you wish to interpret the contribution of each metric.

```python

loadings = pca.components\_[0]

metric\_contributions = pd.Series(loadings, index=metrics\_df.columns)

print(metric\_contributions)

```

### \*\*Alternative: Weighted Sum Method Using Standardization\*\*

If you decide to stick with the weighted sum approach but want to standardize the metrics, you can modify your existing method:

```python

def calculate\_weighted\_composite\_score(self, df):

"""

Calculate the composite score based on weighted contributions from 4 metrics after standardization.

Parameters:

- df: DataFrame containing the metrics.

Returns:

- df: DataFrame with the composite score added.

"""

from sklearn.preprocessing import MinMaxScaler, StandardScaler

# Identify and process columns as before (Steps 1-3)

# Standardize the metrics

scaler\_standard = StandardScaler()

metrics\_standardized = scaler\_standard.fit\_transform(metrics\_df)

# Apply weights

weights = np.array([0.45, 0.35, 0.15, 0.05])

composite\_scores = metrics\_standardized.dot(weights)

# Add the composite score to the DataFrame

df['composite\_score'] = composite\_scores

return df

```

### \*\*Conclusion:\*\*

By integrating the `calculate\_pca\_composite\_score` method into your class, you can effectively compute a composite score using PCA, handling columns with indices, and ensuring proper data preprocessing. Remember to adjust the method to fit your specific data structure and requirements.

## Output Analysis

### Range and Variance of Composite Scores and PCA Scores

Range: The range of both scores gives an idea of the spread of the data. For \*\*composite scores\*\*, the range will depend on the weights assigned to different metrics. Ideally, these scores are scaled between 0 and 1 (since the metrics are normalized). The \*\*PCA scores\*\* will depend on the variance captured by each principal component. A wider range in PCA scores typically indicates greater variability in the data along that principal component.

Variance:

Composite Scores: Variance in composite scores represents how spread out the "fit" or "alignment" is between responsibilities and requirements. Low variance suggests that most responsibilities have similar levels of fit, while high variance indicates some responsibilities are significantly better (or worse) matches than others.

- \*\*PCA Scores\*\*: Variance in PCA scores shows how much of the total variance in the dataset is explained by each principal component. A high variance in PCA scores along one component indicates that this component captures significant variability, implying that responsibilities or requirements exhibit substantial differences along this axis.

### 2. \*\*Analysis of Composite and PCA Score Combinations\*\*

By analyzing different combinations of \*\*composite scores\*\* and \*\*PCA scores\*\*, we can categorize responsibilities and requirements into four cases:

#### \*\*Case 1: Low Composite Score / Low PCA Score\*\*

- \*\*Interpretation\*\*: A responsibility or requirement with both low composite and PCA scores is likely not very distinctive and doesn't align well with the required metrics.

- \*\*Characteristics\*\*:

- \*\*Composite Score\*\*: Low alignment across the weighted metrics.

- \*\*PCA Score\*\*: This responsibility or requirement doesn't explain much variance in the dataset, meaning it’s not particularly unique or important.

- \*\*Action\*\*: Candidates in this category might be \*\*pruned\*\* as they are neither well-aligned nor structurally significant.

#### \*\*Case 2: High Composite Score / Low PCA Score\*\*

- \*\*Interpretation\*\*: A responsibility or requirement with a high composite score but a low PCA score aligns well with the metrics but doesn’t contribute significantly to the overall variance in the dataset.

- \*\*Characteristics\*\*:

- \*\*Composite Score\*\*: High fit or alignment across multiple metrics.

- \*\*PCA Score\*\*: Doesn’t represent much variability in the dataset.

- \*\*Action\*\*: These responsibilities are \*\*good fits\*\* based on metrics but \*\*not unique or important\*\* from a variance perspective. If pruning based solely on alignment, keep them; if uniqueness is a priority, these could be deprioritized.

#### \*\*Case 3: Low Composite Score / High PCA Score\*\*

- \*\*Interpretation\*\*: A low composite score but high PCA score indicates that the responsibility or requirement is structurally important but doesn’t align well with the predefined metrics.

- \*\*Characteristics\*\*:

- \*\*Composite Score\*\*: Poor alignment based on the weighted metrics.

- \*\*PCA Score\*\*: High variance, meaning it captures important structural information in the dataset.

- \*\*Action\*\*: These cases are \*\*unique or distinctive\*\* but don’t fit the requirements well. They may warrant closer examination to understand if they represent outliers, exceptions, or unexplored areas worth investigating.

#### \*\*Case 4: High Composite Score / High PCA Score\*\*

- \*\*Interpretation\*\*: Responsibilities or requirements with both high composite and PCA scores are \*\*highly aligned\*\* with the metrics and are \*\*structurally significant\*\*.

- \*\*Characteristics\*\*:

- \*\*Composite Score\*\*: High fit with multiple metrics.

- \*\*PCA Score\*\*: These responsibilities or requirements capture substantial variability in the dataset.

- \*\*Action\*\*: These are \*\*key data points\*\* that are both well-aligned with the requirements and structurally important. These should be prioritized and retained.

### 3. \*\*Which Score is More Appropriate for Pruning?\*\*

- \*\*PCA Scores for Pruning\*\*:

- \*\*Insight\*\*: PCA scores give insights into the \*\*underlying structure\*\* of the data and help to identify which responsibilities or requirements capture the most variability.

- \*\*Pruning Approach\*\*: If your goal is to reduce redundancy and keep only the most important (distinctive or unique) data points, \*\*PCA scores\*\* are ideal. Low PCA scores can indicate that certain responsibilities or requirements are not contributing much to the overall structure and can be pruned.

- \*\*Composite Scores for Pruning\*\*:

- \*\*Insight\*\*: Composite scores provide a \*\*domain-specific summary\*\* of how well responsibilities align with requirements based on weighted metrics.

- \*\*Pruning Approach\*\*: If your goal is to \*\*remove poorly aligned responsibilities\*\* (i.e., those that don’t meet the key metrics), use \*\*composite scores\*\* for pruning. Low composite scores can be pruned to focus on responsibilities that meet your alignment criteria.

### \*\*Recommendation:\*\*

- \*\*Combined Pruning\*\*: A more balanced approach could involve using \*\*both scores\*\* together. Responsibilities or requirements that are \*\*low in both composite and PCA scores\*\* are strong candidates for pruning because they are neither well-aligned with your metrics nor structurally significant.

- \*\*Low Composite Score/Low PCA Score\*\*: These are the best candidates for removal.

- \*\*High Composite Score/Low PCA Score\*\*: Retain if alignment is more important than uniqueness.

- \*\*Low Composite Score/High PCA Score\*\*: Consider keeping for their structural value, or prune if alignment is critical.

- \*\*High Composite Score/High PCA Score\*\*: Prioritize these for retention.

# Pruning Results Based on Scores

## Claude Recommendations

Outline provided by Claude

```python

import pandas as pd

import numpy as np

def load\_data(file\_path):

# Load data from the provided file format

# This is a placeholder function - you'll need to implement the actual loading logic

pass

def calculate\_alignment\_score(row):

# Calculate a single alignment score from the provided metrics

# You can adjust the weights based on your priorities

weights = {

'scaled\_bert\_score\_precision': 0.3,

'scaled\_soft\_similarity': 0.3,

'scaled\_word\_movers\_distance': 0.2,

'scaled\_deberta\_entailment\_score': 0.2

}

return sum(row[metric] \* weight for metric, weight in weights.items())

def prune\_texts(data, alignment\_threshold=0.5, composite\_threshold=0.3, pca\_threshold=-0.5, target\_ratio=0.8):

# Calculate alignment scores

data['alignment\_score'] = data.apply(calculate\_alignment\_score, axis=1)

# Filter based on thresholds

filtered\_data = data[

(data['alignment\_score'] >= alignment\_threshold) &

(data['composite\_score'] >= composite\_threshold) &

(data['pca\_score'] >= pca\_threshold)

]

# Sort by alignment score

sorted\_data = filtered\_data.sort\_values('alignment\_score', ascending=False)

# Select top texts to meet target ratio

target\_count = int(len(data['Requirement Key'].unique()) \* target\_ratio)

selected\_data = sorted\_data.groupby('Responsibility Key').head(target\_count)

return selected\_data['Responsibility']

# Example usage

data = load\_data('your\_data\_file.csv')

pruned\_responsibilities = prune\_texts(data)

print(pruned\_responsibilities)

```

Here's a breakdown of the updated pruning pipeline:

1. Data Loading:

- Load the data from your provided format. You'll need to implement the `load\_data` function to parse your specific file structure.

2. Filtering:

- Apply thresholds on composite score and PCA score.

- These thresholds can be adjusted as needed.

3. Sorting and Selection:

- Sort the filtered data by alignment score.

- Select the top responsibilities to meet your target ratio (e.g., 80% of the number of unique requirements).

5. Return Pruned List:

- Return the final list of selected responsibilities.

Key Points:

- This pipeline uses the pre-calculated metrics you've provided, including the composite score and PCA score.

- It balances multiple factors: alignment (using various similarity metrics), overall quality (composite score), and dimensionality reduction insights (PCA score).

- The pipeline is flexible: you can adjust thresholds and weights as needed.

- It aims to achieve your target ratio while maintaining quality across multiple dimensions.

Recommendations:

1. Metric Utilization:

- Leverage all available metrics. The composite\_score and pca\_score are particularly useful as they already combine multiple factors.

- Pay special attention to the scaled metrics as they're likely more comparable across different scales.

2. Threshold Tuning:

- Start with moderate thresholds and adjust based on results. For example:

* + composite\_threshold: 0.3 (adjust based on the distribution in your full dataset)
  + pca\_threshold: -0.5 (since PCA scores can be negative, this might need significant tuning)

3. Responsibility Selection:

- The pipeline selects responsibilities rather than requirements. Ensure this aligns with your goal of pruning List A.

4. Data Exploration:

- Before finalizing thresholds, explore the distributions of your scores to ensure you're not cutting off too much or too little data.

5. Iterative Approach:

- Run the pipeline multiple times with different parameters to find the best balance between pruning and maintaining relevance.

To use this pipeline:

1. Prepare your data in a format that can be loaded into a pandas DataFrame.

2. Adjust the thresholds and weights in the `prune\_texts` function as needed.

3. Call the `prune\_texts` function with your data.

4. The function will return your pruned list of responsibilities.

## GPT Recommendations

### Prompt to GPT

My data:

1. Two lists of texts: list A is much longer than list B;

2. A matching file with each element of A comparing to each element of B;

3. each row has 4 metric scores-Bert score precision, soft similarity, word mover distance, and Alberta entailment scores; the directional scores, especially entailment and soft similarity are more important.

4. Each row also has two indices derived from the metrics, PCA and weighted scaled average. 5. A new list of A, after using LLM to modify each element of A to achieve better aligned with each element of List B;

6. A new matching file like the previous one but based on modified list A;

7 & 8, other iterations (the logic is the same).

See example data.

Once I am done with a couple iterations, what is the best course, or courses of, action give my goal below?

<goal>short list List A by prioritizing texts that are more aligned with List B and reduce texts that are less relevant. There final list should have optimal overlap and alignment with B, but not perfect alignment; list A should still be larger than B, I.e., 70-90 percent but not 95%+. This is to preserve authenticity of the original list. The final list A will be re-insert back into JSON to form the edited document A</goal >

<example data>

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Responsibility Key | Responsibility | Requirement Key | Requirement | bert\_score\_precision | soft\_similarity | word\_movers\_distance | deberta\_entailment\_score | bert\_score\_precision\_cat | soft\_similarity\_cat | word\_movers\_distance\_cat | deberta\_entailment\_score\_cat | scaled\_bert\_score\_precision | scaled\_deberta\_entailment\_score | scaled\_soft\_similarity | scaled\_word\_movers\_distance | composite\_score | pca\_score |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 0.pie\_in\_the\_sky.0 | MBA or graduate degree in a management, technical, or engineering field | 0.8172221183776855 | 0.1528222411870956 | 0.740102 | 0.050603 | Medium | Low | Medium | Low | 0.3374518106179494 | 0.056978 | 0.22941442256170863 | 0.740102 | 0.23382285561302601 | -0.98042 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 0.pie\_in\_the\_sky.1 | Knowledge of the Machine Learning and Artificial Intelligence market landscape, ideally with a focus on developer tooling | 0.8406854867935181 | 0.2081294506788253 | 0.6526921202829397 | 0.001224 | Medium | Medium | Medium | Low | 0.5705670209931011 | 0.000911 | 0.33437339200803734 | 0.6526921202829397 | 0.2438729721983386 | -0.34797 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.0 | 11 years of experience in management consulting, product management and strategy, or analytics in a technology company | 0.8380018472671509 | 0.2966089248657226 | 0.6114212519765081 | 0.000421 | Medium | Medium | Medium | Low | 0.5439043022532797 | 0 | 0.5022848544223263 | 0.6114212519765081 | 0.2947081019569544 | 0.39193560638356206 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.1 | Experience working with and analyzing data, and managing multiple cross-functional programs or projects | 0.8287657499313354 | 0.2934322357177734 | 0.6739010982681253 | 0.048156 | Medium | Medium | Medium | Low | 0.4521410594261681 | 0.054199338953625574 | 0.49625630909732765 | 0.6739010982681253 | 0.32177162842472334 | 0.20475685573616237 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.2 | Experience with performing market analysis and developing competitive intelligence | 0.8258360028266907 | 0.2834358811378479 | 0.740102 | 0.1734191924333572 | Medium | Medium | Medium | Low | 0.42303319219494817 | 0.19642616390312853 | 0.47728577913258674 | 0.740102 | 0.3876087888630489 | 0.050617 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.3 | Ability to manage executive stakeholders and communicate with a highly technical management team | 0.8464140295982361 | 0.2675277888774872 | 0.6955220164826892 | 0.005443 | Medium | Medium | Medium | Low | 0.6274817161637998 | 0.005702 | 0.4470962797011563 | 0.6955220164826892 | 0.29475202460644345 | -0.08474 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.4 | Ability to form and refine hypotheses, gather supporting data, and make recommendations | 0.8294726610183716 | 0.1206346675753593 | 0.7175798533808311 | 0.036552 | Medium | Low | Medium | Low | 0.4591644212832744 | 0.041024 | 0.1683306219810472 | 0.7175798533808311 | 0.20797177583955775 | -1.13058 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 1.down\_to\_earth.5 | Excellent problem solving and analysis skills, including opportunity identification, market segmentation, and framing of complex/ambiguous problems | 0.8162370920181274 | 0.1965780556201934 | 0.5913197764017799 | 0.005891 | Medium | Low | Medium | Low | 0.32766529476208817 | 0.00621 | 0.312452 | 0.5913197764017799 | 0.21723384690509767 | -0.22277 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 2.other.0 | English proficiency is a requirement for all roles unless stated otherwise in the job posting | 0.8235018253326416 | 0.1588080525398254 | 0.740102 | 0.001018 | Medium | Low | Medium | Low | 0.39984247771888803 | 0.000677 | 0.24077396495512127 | 0.740102 | 0.21558293275507773 | -0.99315 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 2.other.1 | Work across Program Management teams and our partners (engineering, UX, Customer Experience, TPM, Marketing, Developer Relations, etc.) to help shape the future of AI at Google | 0.8446339964866638 | 0.3389898538589477 | 0.495496 | 0.028476 | Medium | Medium | Medium | Low | 0.6097965830693148 | 0.031854 | 0.5827130422153138 | 0.495496 | 0.3230978702197551 | 1.1144343213563241 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 2.other.2 | Leverage first party and third party market data to build assets and programs that surface valuable insights to our business stakeholders and help inform product roadmaps | 0.8557024002075195 | 0.3567725419998169 | 0.5329467068056377 | 0.013899 | High | Medium | Medium | Low | 0.719764 | 0.015303219900380438 | 0.6164600462389064 | 0.5329467068056377 | 0.33857768659800197 | 1.0953326009593602 |
| 0.responsibilities.0 | Provided strategic insights to a major global IT vendor, optimizing their service partner ecosystem in Asia Pacific for improved local implementation outcomes. | 2.other.3 | Identify gaps in the existing data and engage in original research to fill these gaps, utilizing third party vendors and tooling where appropriate. Create ongoing cadences to enable research distribution and actionable recommendations (e.g. newsletters, dashboards, exec reviews, etc.) | 0.8228855133056641 | 0.3333046436309814 | 0.2903396954025568 | 0.017782 | Medium | Medium | Medium | Low | 0.39371924318242435 | 0.019711306403089116 | 0.5719239640445501 | 0.2903396954025568 | 0.2722803917664874 | 1.758654931728454 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 0.pie\_in\_the\_sky.0 | MBA or graduate degree in a management, technical, or engineering field | 0.8195508718490601 | 0.240227 | 0.76312 | 0.065323 | Medium | Medium | Medium | Low | 0.3605886358925776 | 0.073691 | 0.39528708266298895 | 0.76312 | 0.30400890240952605 | -0.44881 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 0.pie\_in\_the\_sky.1 | Knowledge of the Machine Learning and Artificial Intelligence market landscape, ideally with a focus on developer tooling | 0.8252819776535034 | 0.2498035579919815 | 0.631872 | 0.016338 | Medium | Medium | Medium | Low | 0.4175287951914246 | 0.018071983114664816 | 0.4134602126301961 | 0.631872 | 0.2685007661456086 | 0.015359394646933923 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.0 | 11 years of experience in management consulting, product management and strategy, or analytics in a technology company | 0.8336870670318604 | 0.3040940761566162 | 0.631872 | 0.000955 | Medium | Medium | Medium | Low | 0.5010357386077633 | 0.000605 | 0.5164897613930808 | 0.631872 | 0.3008765162600876 | 0.3705211291815432 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.1 | Experience working with and analyzing data, and managing multiple cross-functional programs or projects | 0.8255743384361267 | 0.1813950538635254 | 0.6526921202829397 | 0.1350825577974319 | Medium | Low | Medium | Low | 0.42043348236757083 | 0.152898 | 0.28363832935905875 | 0.6526921202829397 | 0.28700289675018037 | -0.38432 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.2 | Experience with performing market analysis and developing competitive intelligence | 0.8262200951576233 | 0.3111447691917419 | 0.7175798533808311 | 0.2607548534870147 | Medium | Medium | Medium | Medium | 0.4268492582832444 | 0.295589 | 0.5298701774663209 | 0.7175798533808311 | 0.44744920602843763 | 0.4162668717567929 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.3 | Ability to manage executive stakeholders and communicate with a highly technical management team | 0.8392523527145386 | 0.1618953943252563 | 0.6739010982681253 | 0.008112 | Medium | Low | Medium | Low | 0.5563284280342291 | 0.008732 | 0.246633 | 0.6739010982681253 | 0.21915250363582467 | -0.7275 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.4 | Ability to form and refine hypotheses, gather supporting data, and make recommendations | 0.8247349262237549 | 0.150433 | 0.6955220164826892 | 0.1225656569004058 | Medium | Low | Medium | Low | 0.41209368430402904 | 0.1386857399150469 | 0.2248794963965709 | 0.6955220164826892 | 0.26604939338817574 | -0.75664 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 1.down\_to\_earth.5 | Excellent problem solving and analysis skills, including opportunity identification, market segmentation, and framing of complex/ambiguous problems | 0.8154844045639038 | 0.2296628355979919 | 0.5715506253044318 | 0.015622 | Medium | Medium | Medium | Low | 0.3201871317324496 | 0.017258678846751113 | 0.37523826135467836 | 0.5715506253044318 | 0.24084174733746264 | 0.080452 |
| 0.responsibilities.1 | Assisted a U.S.-based international services provider in its growth strategy by precisely evaluating and scaling new engineering service opportunities in vital emerging markets. | 2.other.0 | English proficiency is a requirement for all roles unless stated otherwise in the job posting | 0.8225706815719604 | 0.1124952435493469 | 0.7175798533808311 | 0.001077 | Medium | Low | Medium | Low | 0.39059130074319803 | 0.000744 | 0.15288407232670792 | 0.7175798533808311 | 0.18101080276725778 | -1.2301 |

</example data>

### Recommendation

Here’s the updated recommendation using the \*\*Max-Based Approach\*\*:

1. \*\*Max-Based Retention of Responsibilities\*\*

- For each responsibility in List A, find the maximum composite score across all its comparisons with List B’s requirements.

- If a responsibility shows \*\*high alignment with at least one requirement\*\* (i.e., it has a high maximum composite score), retain that responsibility.

- Responsibilities with \*\*no strong alignment\*\* (low maximum composite score across all requirements) will be considered for elimination.

2. \*\*Threshold for High Alignment\*\*

- Set a threshold based on the maximum composite score to determine what qualifies as "high alignment." Responsibilities with a maximum composite score above this threshold will be retained, while those below it will be eliminated.

- Adjust this threshold dynamically to ensure that you retain 70-90% of List A while preserving its alignment with List B.

3. \*\*Iterate and Refine\*\*

- After each iteration (modifications to List A using LLM), apply the same max-based filtering method to further refine the list.

- Continue adjusting the threshold based on the max composite score to maintain the target size of List A.

4. \*\*Final Selection\*\*

- Once the process is complete, the final list of responsibilities will have the strongest alignment with at least one requirement from List B.

- The final List A will retain 70-90% of its original size, prioritizing alignment with B while keeping a degree of authenticity.

5. \*\*Reinsertion into JSON\*\*

- Once finalized, reinsert the shortlisted List A back into JSON format to complete the process and generate the final edited document.

This approach ensures that strong matches are preserved, even if some responsibilities show weaker alignment in other areas, giving you a balance between alignment and authenticity.

Final Solution: Claude + GPT

high level -> GPT implementation -> Claude refined (added a more dynamic approach - elbow method -> GPT further code implementation

## Notes on Adaptive Pruning from Claude

Here's how the `adaptive\_pruning` method works:

1. Initial Setup:

- It starts with the ranked DataFrame (sorted by the combined score).

- It calculates the total number of unique responsibilities.

- It determines the target count of responsibilities based on the `target\_percentage` (e.g., if we want to keep 80% of responsibilities, and we start with 100, the target count would be 80).

2. Pruning Loop:

- The method uses a while loop to gradually prune the responsibilities.

- It starts with `current\_percentage = 1.0`, meaning it initially includes all responsibilities.

3. Gradual Reduction:

- In each iteration of the loop, it reduces the `current\_percentage` by a small step (default is 0.05 or 5%).

- It then calculates a threshold score based on this percentage. For example, if `current\_percentage` is 0.95, it finds the score at the 95th percentile.

4. Thresholding:

- It keeps all responsibilities with a combined score less than or equal to this threshold. (Remember, lower scores are better because they represent better rankings.)

5. Check and Repeat:

- After each reduction, it checks if the number of unique responsibilities is still above the target count.

- If it is, the loop continues, further reducing the percentage and recalculating the threshold.

6. Loop Termination:

- The loop stops when the number of unique responsibilities falls below or reaches the target count.

Here's a step-by-step example:

1. Start with 100 unique responsibilities, target is 80% (80 responsibilities).

2. First iteration: Keep 95% (threshold at 95th percentile) - might have 95 responsibilities left.

3. Second iteration: Keep 90% - might have 90 responsibilities left.

4. Third iteration: Keep 85% - might have 85 responsibilities left.

5. Fourth iteration: Keep 80% - reaches 80 responsibilities, stops here.

The key advantages of this method are:

1. Flexibility: It can adapt to different distributions of scores.

2. Controlled Reduction: It reduces the list size gradually, which helps in finding a balance between pruning and maintaining diversity.

3. Target-Oriented: It aims for a specific reduction percentage, aligning with your goal of keeping 70-90% of the original list.

This approach should help you avoid the overly aggressive pruning you were experiencing with the elbow method, while still allowing you to reduce your list to a desired size range.

## Notes on How to Deal with Too Few Results with Pruning Claude

Recommendations

1. Don't prune at each iteration: Your suggestion to not prune at each iteration is a good one. Instead, focus on refining and improving the alignment of responsibilities during each iteration.
2. Use a more flexible pruning approach: Instead of relying solely on the elbow method, which seems to be overly aggressive in your case, I've proposed an adaptive pruning approach in the artifact above. This method allows you to target a specific percentage range for the final list size.
3. Consider all responsibilities: Allow duplicate responsibilities during the iteration process, as you suggested. This can help preserve more nuanced alignments between List A and List B.
4. Combine multiple metrics: The revised approach uses a combined score based on both composite and PCA rankings. This can provide a more balanced view of each responsibility's relevance.
5. Visualize your data: The plot\_score\_distribution method will help you understand the distribution of your scores, which can inform your pruning strategy.
6. Adaptive pruning: The adaptive\_pruning method allows you to set a target percentage (e.g., 80% of original size) and gradually prunes until reaching that target.

# Async Approach

Use async because of playwright.

You have to await the functions; otherwise, you will run into problems b/c of race issue.

# JSON, Pydantic Models, I/O Workflow

## Pydantic Models for JSON Related Validation

For all these models, keep the “data” part under a data field.

Why?

In API design, especially for Large Language Models (LLMs), it's a common convention to wrap the response content under a single top-level field, such as:

* data
* content
* response
* result
* payload

Reasons:

1. Consistency: Maintains consistency across API endpoints so that it’s easier to design for different platforms.
2. Clarity: Clearly distinguishes response content from metadata.
3. Flexibility: saves room for adding new metadata or other properties.

JSON and Tabular data are usually under “data”; text usually under content.

### Model Classes

JSONResponse model

# Generalized JSON Response Model (New Version)

class JSONResponse(BaseModel):

    """Basic and generic model for JSON response"""

    data: Union[

        Dict[str, Any], List[Dict[str, Any]]

    ]  # Allow both dict and list of dicts

    class Config:

        arbitrary\_types\_allowed = True

Editing Models

class EditingData(BaseModel):

    """

    Inner model for editing responses, specifically for optimized text.

    Attributes:

        optimized\_text (str): The optimized text produced by an editing operation.

    """

    optimized\_text: str

class EditingResponseModel(JSONResponse):

    """

    Model for responses involving text editing operations.

    Inherits from JSONResponse to standardize with JSON data structure.

    Attributes:

        data (Dict[str, EditingData]): A dictionary where keys represent identifiers and values are EditingData instances containing `optimized\_text`.

    Config:

        json\_schema\_extra (dict): Provides an example structure for documentation.

    """

    data: Dict[str, EditingData]

    class Config:

        json\_schema\_extra = {

            "example": {

                "status": "success",

                "message": "Text editing processed successfully.",

                "data": {

                    "0.pie\_in\_the\_sky.0": {

                        "optimized\_text": "This is the optimized text after editing."

                    },

                    "1.down\_to\_earth.0": {

                        "optimized\_text": "Another piece of optimized text."

                    },

                },

            }

        }

Job site parsing models

class JobSiteData(BaseModel):

    """

    Inner model containing detailed job site information.

    Attributes:

        url (Optional[str]): The URL of the job posting.

        job\_title (Optional[str]): Title of the job position.

        company (Optional[str]): Name of the company posting the job.

        location (Optional[str]): Job location.

        salary\_info (Optional[str]): Salary information, if available.

        posted\_date (Optional[str]): Date when the job was posted.

        content (Optional[Dict[str, Any]]): Contains the job description, responsibilities, and qualifications as a dictionary.

    """

    url: Optional[str] = Field(None, description="Job posting URL")

    job\_title: Optional[str] = Field(None, description="Job title")

    company: Optional[str] = Field(None, description="Company name")

    location: Optional[str] = Field(None, description="Job location")

    salary\_info: Optional[str] = Field(None, description="Salary information")

    posted\_date: Optional[str] = Field(None, description="Job posting date")

    content: Optional[Dict[str, Any]] = Field(

        None,

        description="Dictionary containing job description, responsibilities, and qualifications",

    )

class JobSiteResponseModel(BaseResponseModel):

    """

    Model for handling job site response data, standardizing job-related information.

    Attributes:

        data (JobSiteData): Holds detailed job site information as a nested JobSiteData instance.

    Config:

        json\_schema\_extra (dict): Provides an example structure for documentation.

    """

    data: JobSiteData

    class Config:

        json\_schema\_extra = {

            "example": {

                "status": "success",

                "message": "Job site data processed successfully.",

                "data": {

                    "url": "https://example.com/job-posting",

                    "job\_title": "Software Engineer",

                    "company": "Tech Corp",

                    "location": "San Francisco, CA",

                    "salary\_info": "$100,000 - $120,000",

                    "posted\_date": "2024-11-08",

                    "content": {

                        "description": "We are looking for a Software Engineer...",

                        "responsibilities": [

                            "Develop software",

                            "Collaborate with team",

                        ],

                        "qualifications": [

                            "BS in Computer Science",

                            "2+ years experience",

                        ],

                    },

                },

            }

        }

## Data Flow for Each Step

1. LLM Response Retrieval

- **Function**: *call\_llm\_async* or *call\_llm*

- Process: The function sends a prompt to the LLM API (e.g., OpenAI or Claude) and waits for a response.

- The response typically arrives as raw JSON, structured based on the ‘expected\_res\_type’ (e.g., ‘json’ for structured JSON data).

* Code

        if llm\_provider == "openai":

            openai\_client = cast(OpenAI, client)  # Cast to OpenAI

            response = openai\_client.chat.completions.create(

                model=model\_id,

                messages=[

                    {

                        "role": "system",

                        "content": "You are a helpful assistant who adheres to instructions.",

                    },

                    {"role": "user", "content": prompt},

                ],

                temperature=temperature,

                max\_tokens=max\_tokens,

            )

            response\_content = response.choices[0].message.content

        elif llm\_provider == "claude":

            claude\_client = cast(Anthropic, client)  # Cast to Anthropic (Claude)

            system\_instruction = (

                "You are a helpful assistant who adheres to instructions."

            )

            response = claude\_client.messages.create(

                model=model\_id,

                max\_tokens=max\_tokens,

                messages=[{"role": "user", "content": system\_instruction + prompt}],

                temperature=temperature,

            )

            # Need to add an extra step to extract content from response object's TextBlocks

            # (Unlike GPT and LlaMA, Claude uses multi-blocks in its responses:

            # The content attribute of Message is a list of TextBlock objects,

            # whereas others wrap everything into a single block.)

            response\_content = (

                response.content[0].text

                if hasattr(response.content[0], "text")

                else str(response.content[0])

            )

        elif llm\_provider == "llama3":

            # Construct an instance of Options

            options = {

                "temperature": temperature,

                "max\_tokens": max\_tokens,

                "batch\_size": 10,

                "retry\_enabled": True,

            }

            response = ollama.generate(model=model\_id, prompt=prompt, options=options)  # type: ignore

            response\_content = response["response"]

* Example Response:

Raw claude Response:

{

"optimized\_text": "Led strategic market analysis and research initiatives for B2B SaaS security solutions, focusing on analytics-driven product positioning and marketing strategy development"

}

1. Response Validation and Initial Parsing

* Function: `validate\_response\_type`
* Process\*\*:
* The function checks if the raw response content matches the expected structure and type.
* For example, if the response type is expected to be `json`, it verifies that the content is a valid JSON structure.
* Outcome:
* If valid, it proceeds to the next step.
* If not valid, an error is logged, and processing stops for this response.
* Example:

validated response content after validate\_json\_type:

JSON

data={'optimized\_text': 'Led strategic market analysis and research initiatives for B2B SaaS security solutions, focusing on analytics-driven product positioning and marketing strategy development'}

1. Pydantic Model Validation (`validate\_json\_type`)

* - \*\*Function\*\*: `validate\_json\_type`
* Purpose\*\*: This function validates the JSON data against a specific Pydantic model, such as `EditingResponseModel`.
* Process:
* Based on `json\_type`, `validate\_json\_type` selects the appropriate Pydantic model from `json\_model\_mapping`.
* For each `json\_type` (e.g., `editing`), the function instantiates a model like `EditingResponseModel` with the LLM’s JSON response to ensure it has the required structure.
* If the structure does not match the expected model (e.g., missing `optimized\_text`), a validation error is raised.
* Example\*\*:

Suppose `json\_type="editing"`, and the response data looks like:

```json

{

"optimized\_text": "Provided strategic insights..."

}

```

- This response would successfully validate against `EditingResponseModel`:

```python

class EditingResponseModel(BaseModel):

optimized\_text: str

```

* Outcome: The validated data is returned as an instance of the model (e.g., `EditingResponseModel`).

1. modify\_resp\_based\_on\_reqs\_async or modify\_resp\_based\_on\_reqs

This function modifies a single responsibility based on multiple job requirements. Each modified responsibility text should be associated with a unique requirement key, nested under optimized\_by\_requirements.

* Expected Output Structure from modify\_resp\_based\_on\_reqs\_async:
* Each call should return a tuple where: resp\_key is the key for the responsibility.

The second element is a ResponsibilityMatch object with optimized\_by\_requirements, containing optimized text responses keyed by requirement IDs.

Example of the Expected Return Value:

python

Copy code

(

"0.responsibilities.0",

ResponsibilityMatch(

optimized\_by\_requirements={

"0.pie\_in\_the\_sky.0": OptimizedText(

optimized\_text="Provided strategic insights to optimize the service partner ecosystem in APAC."

),

"1.down\_to\_earth.0": OptimizedText(

optimized\_text="10+ years experience in business development and technology."

),

}

)

)

### 4. \*\*Structuring and Storing the Validated Data\*\*

- \*\*Process\*\*:

- After `validate\_json\_type`, the validated data is converted back into a dictionary, maintaining the structure specified in the Pydantic model.

- This data is then stored in a dictionary for each responsibility, under a key `optimized\_by\_requirements` (based on the structure needed for `ResponsibilityMatch`).

- \*\*Example Structure\*\*:

```python

{

"0.responsibilities.0": ResponsibilityMatch(

optimized\_by\_requirements={

"0.pie\_in\_the\_sky.0": EditingResponseModel(

optimized\_text="Provided strategic insights..."

)

}

)

}

```

- This data is then assigned as the `responsibilities` field in `ResponsibilityMatches`.

### 5. \*\*Final Validation with `ResponsibilityMatches`\*\*

- \*\*Model\*\*: `ResponsibilityMatches`

- \*\*Process\*\*:

- The entire dictionary structure of responsibilities and their optimized texts is validated against the `ResponsibilityMatches` model.

- This model ensures that each `responsibility\_key` (e.g., `"0.responsibilities.0"`) maps to a `ResponsibilityMatch`, and each `requirement\_key` within maps to an `OptimizedText`.

- \*\*Final Structure\*\*:

```json

{

"responsibilities": {

"0.responsibilities.0": {

"optimized\_by\_requirements": {

"0.pie\_in\_the\_sky.0": {

"optimized\_text": "Provided strategic insights..."

}

}

},

"1.responsibilities.1": {

"optimized\_by\_requirements": {

"1.down\_to\_earth.0": {

"optimized\_text": "Enhanced service partner ecosystem..."

}

}

}

}

}

```

### 6. \*\*Output JSON Generation\*\*

- \*\*Process\*\*:

- Finally, the `ResponsibilityMatches` instance is serialized into JSON format using `model\_dump()`.

- This output can be saved to a file or returned as the API response, maintaining the expected structure.

- \*\*Final JSON Output\*\*:

```json

{

"responsibilities": {

"0.responsibilities.0": {

"optimized\_by\_requirements": {

"0.pie\_in\_the\_sky.0": {

"optimized\_text": "Provided strategic insights..."

}

}

}

}

}

```

By following these steps, you ensure that:

- Each responsibility aligns with its requirements in a nested structure.

- The final output matches the format required by `ResponsibilityMatches`.

# Dynamic Scoring and Transformer Inspired Refinement Extension

Based on the data and your descriptions, here are specific recommendations to integrate **emergent relationships** and **transformer-inspired refinement extensions** into your pipeline, considering the structure of your CSV files and the alignment process:

**1. Emergent Relationships**

**Concept:**

Emergent relationships will adjust alignment scores dynamically, allowing contextual influence between **responsibilities** (resume parts) and **requirements** (job postings). The goal is to ensure the relationships adapt based on alignment quality and contextual dependencies, reflecting the hierarchical and ordered nature of the responsibilities.

**Recommendations:**

1. **Use Dynamic Score Adjustments**:
   * Since **responsibilities** have an inherent order, ensure scores reflect how previous and subsequent responsibilities influence the alignment.
   * Adjust scores dynamically: S′(i,j)=S(i,j)+α∑k=1nSim(responsibilityi,requirementk)S'(i, j) = S(i, j) + \alpha \sum\_{k=1}^n \text{Sim}(\text{responsibility}\_i, \text{requirement}\_k) Where:
     + S(i,j)S(i, j): Original score for ii-th responsibility and jj-th requirement.
     + α\alpha: Weight for contextual influence.
2. **Penalize Over-Alignment**:
   * Avoid over-aligning multiple responsibilities to the same requirement by introducing penalties: S′(i,j)=S′(i,j)−β⋅Overlap(j)S'(i, j) = S'(i, j) - \beta \cdot \text{Overlap}(j) Where Overlap(j)\text{Overlap}(j) is the count of responsibilities already aligned to requirement jj, and β\beta controls the penalty strength.
3. **Incorporate Local Dependencies**:
   * Leverage sequential context in **responsibilities** using attention-like weights: S′(i,j)=S′(i,j)+∑k≠iγi,k⋅S(k,j)S'(i, j) = S'(i, j) + \sum\_{k \neq i} \gamma\_{i, k} \cdot S(k, j)
     + γi,k\gamma\_{i, k}: A weight based on the distance or relationship between responsibilities ii and kk.

**2. Transformer-Inspired Refinement Extensions**

**Concept:**

Transformer-like mechanisms refine alignments iteratively by considering relationships between responsibilities and requirements in a global context. This approach respects the **order of responsibilities** while treating requirements as unordered.

**Recommendations:**

1. **Iterative Refinement**:
   * Use multiple refinement passes where scores are updated based on current alignments: At+1(i,j)=Softmax(At(i,j)+∑kαj,kAt(i,k))A^{t+1}(i, j) = \text{Softmax}\left(A^t(i, j) + \sum\_k \alpha\_{j, k} A^t(i, k)\right) Where At(i,j)A^t(i, j) is the alignment score at iteration tt.
2. **Multi-Head Alignment Scoring**:
   * Incorporate different alignment criteria (e.g., semantic similarity, entailment) into separate scoring heads: Sfinal(i,j)=Concat(S1(i,j),S2(i,j),… )⋅WS\_{\text{final}}(i, j) = \text{Concat}(S\_1(i, j), S\_2(i, j), \dots) \cdot W
     + Each head (S1,S2,…S\_1, S\_2, \dots) focuses on a specific aspect (e.g., semantic match, dependency coherence).
3. **Weighted Aggregation**:
   * Dynamically weight responsibilities based on their significance:
     + Example: If a responsibility aligns poorly with all requirements, reduce its influence in subsequent iterations.

**3. Pipeline Integration**

To integrate these concepts into your pipeline:

**Dynamic Scoring Adjustments**

* Modify composite scores in the CSV to reflect:
  + Contextual influence between responsibilities.
  + Alignment penalties for overuse of requirements.

**Iterative Refinement**

1. **Pass 1**:
   * Use your existing pipeline to generate initial alignments.
2. **Pass 2 and Beyond**:
   * Refine alignments by adjusting scores based on emergent relationships:
     + Incorporate sequential dependencies for responsibilities.
     + Adjust based on global context for requirements.

**Final Selection**

* Use an optimization method (e.g., Hungarian algorithm or ILP) after refinement to select alignments.

**4. Handling Responsibilities vs. Requirements**

1. **Order Matters for Responsibilities**:
   * Incorporate sequential penalties or rewards based on the natural flow of responsibilities.
   * Use dependency parsing to validate coherence.
2. **Order Mostly Doesn’t Matter for Requirements**:
   * Treat requirements as independent entities, focusing only on maximizing alignment quality.

**5. Practical Implementation**

1. **Scoring Adjustments**:
   * Add dynamic scoring logic to your current pipeline based on emergent relationships.
2. **Iterative Updates**:
   * Implement iterative alignment refinement (e.g., multi-pass scoring).
3. **Validation**:
   * Use dependency parsing and LLM-powered editing to ensure alignments are logically consistent and coherent.

Would you like help implementing these changes into your existing pipeline or creating a sample scoring adjustment function?

### Losses Functions

#### CosineEmbeddingLoss?

CosineEmbeddingLoss is a PyTorch loss function used to measure the **similarity or dissimilarity** between two input embeddings (vectors) based on their cosine similarity. It is particularly useful when you want to train a model to predict whether two inputs are **similar** (positive relationship) or **dissimilar** (negative relationship).

**Key Concepts**

1. **Cosine Similarity**:
   * Measures the cosine of the angle between two vectors.
   * Ranges from -1 (completely opposite) to 1 (exactly the same).
   * Formula: Cosine Similarity=x1⋅x2∥x1∥∥x2∥\text{Cosine Similarity} = \frac{\mathbf{x\_1} \cdot \mathbf{x\_2}}{\|\mathbf{x\_1}\| \|\mathbf{x\_2}\|} Where:
     + x1,x2\mathbf{x\_1}, \mathbf{x\_2}: Input vectors.
     + ⋅\cdot: Dot product.
     + ∥x∥\|\mathbf{x}\|: Magnitude of vector x\mathbf{x}.
2. **Purpose**:
   * If the two embeddings are **similar** (positive relationship), the cosine similarity should be high (close to 1).
   * If the embeddings are **dissimilar** (negative relationship), the cosine similarity should be low (close to -1).

**Formula for CosineEmbeddingLoss**

The loss function operates on two vectors (x1\mathbf{x\_1}, x2\mathbf{x\_2}) and a **target label** (yy):

Loss(x1,x2,y)={1−cos(x1,x2),if y=1max⁡(0,cos(x1,x2)−margin),if y=−1\text{Loss}(\mathbf{x\_1}, \mathbf{x\_2}, y) = \begin{cases} 1 - \text{cos}(\mathbf{x\_1}, \mathbf{x\_2}), & \text{if } y = 1 \\ \max(0, \text{cos}(\mathbf{x\_1}, \mathbf{x\_2}) - \text{margin}), & \text{if } y = -1 \end{cases}

* y=1y = 1: The target indicates that the embeddings should be **similar**.
  + Loss is minimized when cosine similarity is 1.
  + 1−cos(x1,x2)1 - \text{cos}(\mathbf{x\_1}, \mathbf{x\_2}) penalizes low similarity.
* y=−1y = -1: The target indicates that the embeddings should be **dissimilar**.
  + Loss is minimized when cosine similarity is less than a specified **margin**.
  + Default margin is 0, meaning embeddings should be orthogonal or less similar than that.

**Key Parameters**

1. **Inputs**:
   * x1,x2\mathbf{x\_1}, \mathbf{x\_2}: Input tensors of shape [batch\_size,embedding\_dim][\text{batch\\_size}, \text{embedding\\_dim}].
   * yy: Tensor of labels, shape [batch\_size][\text{batch\\_size}], with values 1 (similar) or -1 (dissimilar).
2. **margin**:
   * A hyperparameter that defines the threshold for dissimilarity when y=−1y = -1.
   * Default: 0.0.
   * Higher margin values enforce stronger dissimilarity.
3. **Reduction**:
   * mean: Average loss across the batch.
   * sum: Sum of losses across the batch.
   * none: Returns individual losses for each sample.

**Intuition**

* For **similar pairs** (y=1y = 1):
  + The loss encourages the cosine similarity to be close to 1.
  + Example: Refined resume embeddings aligning with job description embeddings.
* For **dissimilar pairs** (y=−1y = -1):
  + The loss penalizes cosine similarity if it is greater than the margin.
  + Example: Embeddings from unrelated job descriptions and resumes.

**Example Usage in PyTorch**

import torch

import torch.nn as nn

# Instantiate the loss function

cosine\_loss = nn.CosineEmbeddingLoss(margin=0.5)

# Example embeddings (batch\_size=3, embedding\_dim=4)

x1 = torch.tensor([[1.0, 0.0, 0.0, 0.0],

[0.0, 1.0, 0.0, 0.0],

[0.0, 0.0, 1.0, 0.0]], requires\_grad=True)

x2 = torch.tensor([[1.0, 0.0, 0.0, 0.0],

[0.0, -1.0, 0.0, 0.0],

[0.0, 0.0, -1.0, 0.0]], requires\_grad=True)

# Similarity labels: 1 = similar, -1 = dissimilar

y = torch.tensor([1, -1, -1])

# Compute the loss

loss = cosine\_loss(x1, x2, y)

print(f"Cosine Embedding Loss: {loss.item()}")

**Example Output:**

If x1x\_1 and x2x\_2 are:

* Perfectly similar for y=1y = 1, the loss will be close to 0.
* Highly dissimilar for y=−1y = -1, the loss will also be minimized.

**Use in Your AlignmentLoss**

In your AlignmentLoss:

* **Inputs**:
  + x1\mathbf{x\_1}: Refined embeddings (refined\_embs from the TransformerModel).
  + x2\mathbf{x\_2}: Job description embeddings (job\_desc\_embs).
  + y=1y = 1: Assumes all pairs are positively aligned (similar).
* Example:

align\_labels = torch.ones(refined\_embs.size(0), device=device) # Positive pairs

align\_loss = self.align\_loss(refined\_embs, job\_desc\_embs, align\_labels)

This encourages the model to produce embeddings for resumes that align closely with job descriptions, measured by cosine similarity.

#### Mean Squared Error Loss.

It measures the average squared difference between the predicted values (model outputs) and the target values (ground truth). This is a common loss function used in regression tasks and situations where you want to minimize the squared difference between two sets of values.

**Formula for MSELoss**

MSELoss=1N∑i=1N(predictioni−targeti)2\text{MSELoss} = \frac{1}{N} \sum\_{i=1}^{N} (\text{prediction}\_i - \text{target}\_i)^2

Where:

* NN: Number of samples (batch size).
* predictioni\text{prediction}\_i: Model's predicted value for the ii-th sample.
* targeti\text{target}\_i: Ground truth value for the ii-th sample.

It computes the **element-wise squared difference**, averages it over all the samples, and returns the result.

**How MSELoss Works**

1. **Penalizes Large Errors**:
   * Since the differences are squared, larger errors are penalized more heavily than smaller ones. For example:
     + A prediction off by 5 contributes 52=255^2 = 25 to the loss.
     + A prediction off by 0.5 contributes 0.52=0.250.5^2 = 0.25.
2. **Smooth Gradient**:
   * The quadratic nature of the loss ensures smooth gradients, which helps optimization algorithms like SGD converge effectively.

**Parameters in MSELoss**

1. **reduction** (default: 'mean'):
   * 'mean': Averages the squared differences across all samples.
   * 'sum': Sums the squared differences without averaging.
   * 'none': Returns the raw squared differences for each sample.
2. **Inputs**:
   * Predictions (input): Tensor of shape [batch\_size, ...].
   * Targets (target): Tensor of the same shape as input.

**Example in PyTorch**

import torch

import torch.nn as nn

# Instantiate MSELoss

mse\_loss = nn.MSELoss()

# Example predictions and targets

predictions = torch.tensor([2.5, 0.0, 2.0], requires\_grad=True)

targets = torch.tensor([3.0, -0.5, 2.0])

# Compute the loss

loss = mse\_loss(predictions, targets)

print(f"MSE Loss: {loss.item()}") # Output: MSE Loss: 0.2916666567325592

**Output Calculation**

For the given example:

1. Compute element-wise squared difference:
   * (2.5−3.0)2=0.25(2.5 - 3.0)^2 = 0.25
   * (0.0−(−0.5))2=0.25(0.0 - (-0.5))^2 = 0.25
   * (2.0−2.0)2=0.0(2.0 - 2.0)^2 = 0.0
2. Take the average:
   * MSE Loss=0.25+0.25+0.03=0.2917\text{MSE Loss} = \frac{0.25 + 0.25 + 0.0}{3} = 0.2917

**Use in Your AlignmentLoss**

In your code, MSELoss is used to compute the **substance loss**, which ensures that the refined embeddings (refined\_embs) are close to the original embeddings (resume\_embs.mean(dim=1)):

substance\_loss = self.substance\_loss(refined\_embs, resume\_embs.mean(dim=1))

Here:

* **Predictions (refined\_embs)**: The refined embeddings output by the TransformerModel.
* **Targets (resume\_embs.mean(dim=1))**: The average of the original resume embeddings across sections.

This penalizes large deviations between the refined embeddings and the original embeddings, helping the model preserve core resume information during refinement.

# DuckDB Design

## Database Schemas

Tables

* job\_posting\_urls
* job\_postings
* extracted\_requirements
* flattened\_responsibilities
* flattened\_requirements
* pruned\_responsibilities
* edited\_responsibilities
* similarity\_metrics (use version to control for original or edited)

~~Stages~~

* ~~preprocessing~~
* ~~staging~~
* ~~evaluation~~
* ~~editing~~
* ~~revaluation~~
* ~~crosstab~~
* ~~llm\_trim~~
* ~~final\_export~~

Use tables as stages (changed to improve workflow)

JOB\_URLS = "job\_urls"

JOB\_POSTINGS = "job\_postings"

EXTRACTED\_REQUIREMENTS = "extracted\_requirements"

FLATTENED\_REQUIREMENTS = "flattened\_requirements"

FLATTENED\_RESPONSIBILITIES = "flattened\_responsibilities"

EDITED\_RESPONSIBILITIES = "edited\_responsibilities"

SIMILARITY\_METRICS = "similarity\_metrics"

Metadata

Each table now includes:

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| stage | TEXT | One of 'preprocessing', 'staging', 'evaluation', 'editing', 'cleanup' |
| llm\_provider | TEXT | 'openai' or 'anthropic' |
| source\_file | TEXT | Optional file path or name from which data was loaded |
| timestamp | TIMESTAMP | Automatically set when data is inserted |

| **Table** | **Model** | **Flattening Function** |
| --- | --- | --- |
| job\_urls | JobPostingUrlsFile | flatten\_job\_urls\_to\_table |
| job\_postings | JobPostingsFile | flatten\_job\_postings\_to\_table |
| extracted\_requirements | ExtractedRequirementsFile | flatten\_extracted\_requirements\_to\_table |
| requirements | Requirements | flatten\_requirements\_to\_table |
| flattened\_responsibilities | Responsibilities | flatten\_responsibilities\_to\_table |
| pruned\_responsibilities | NestedResponsibilities | flatten\_nested\_responsibilities\_to\_table |
| edited\_responsibilities | NestedResponsibilities | flatten\_nested\_responsibilities\_to\_table |

## All Fields

**🧭 1. pipeline\_control**

**Purpose**: Tracks the current pipeline stage for each job URL.

**Fields**:

* url: Job posting URL (primary key)
* llm\_provider: e.g., "openai", "anthropic"
* version: One of "original", "edited", "final"
* status: "new", "in\_progress", "complete", "skipped", "error"
* is\_active: True or False
* stage: Current pipeline stage
* last\_updated: Timestamp
* notes: Optional metadata

**📌 2. job\_urls**

**Purpose**: Registry of job URLs and associated metadata.

**Fields**:

* url: str (primary key)
* company: str
* job\_title: str
* source\_file, stage, timestamp: Metadata

**📄 3. job\_postings**

**Purpose**: Flattened job description content. **Fields**:

* url, status, message
* job\_title, company, location, salary\_info, posted\_date
* content: JSON string (full job description incl. responsibilities, qualifications)
* source\_file, stage, timestamp

**📊 4. extracted\_requirements**

**Purpose**: Nested requirements parsed from job postings. **Fields**:

* url, status, message
* requirement\_category: e.g., "pie\_in\_the\_sky"
* requirement\_category\_idx: integer index of category
* requirement: str
* requirement\_idx: index within list
* source\_file, stage, timestamp

**🧱 5. flattened\_responsibilities**

**Purpose**: Resume responsibilities in key-value form. **Fields**:

* url
* responsibility\_key: e.g., "0.responsibilities.0"
* responsibility: str
* source\_file, stage, timestamp

**🧱 6. flattened\_requirements**

**Purpose**: Job requirements in key-value form. **Fields**:

* url
* requirement\_key: e.g., "1.down\_to\_earth.1"
* requirement: str
* source\_file, stage, timestamp

**✂️ 7. pruned\_responsibilities**

**Purpose**: Manually trimmed responsibilities after human review. **Fields**:

* url, responsibility\_key, responsibility
* pruned\_by: Annotator identifier (e.g., "xfz" or "llm")
* source\_file, stage, timestamp

**🧠 8. edited\_responsibilities**

**Purpose**: LLM-edited text aligning resume responsibility to job requirement. **Fields**:

* url, responsibility\_key, requirement\_key
* optimized\_text: str
* llm\_provider, source\_file, stage, timestamp

**🧮 9. similarity\_metrics**

**Purpose**: Scores for similarity/entailment between responsibilities and requirements. **Fields**:

* Keys: url, responsibility\_key, requirement\_key
* Text: responsibility, requirement
* Raw metrics: bert\_score\_precision, soft\_similarity, word\_movers\_distance, deberta\_entailment\_score, roberta\_entailment\_score
* Categoricals: \*\_cat fields
* Scaled metrics: scaled\_\*
* Composite: composite\_score, pca\_score
* Metadata: version, llm\_provider, source\_file, stage, timestamp

## Metadata

**🧾 Metadata Fields Definitions**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| url | str or HttpUrl | The unique identifier for a job posting. Serves as the primary key across all tables and controls pipeline flow per job. |
| llm\_provider | str | The name of the LLM used for processing. Expected values include "openai" and "anthropic". Used for tracking which model generated or modified the data. |
| version | str (enum) | Indicates the data output version. Options include:• "original": Raw extracted or initial version• "edited": After LLM-based editing• "final": Post-trimmed and human-reviewed |
| status | str (enum) | Lifecycle status of a given record in the pipeline:  • "new": Newly added  • "in\_progress": Currently being processed  • "complete": All steps completed successfully  • "skipped": Intentionally skipped  • "error": Failed due to error |
| is\_active | bool | Boolean flag that determines whether the record is considered in scope for processing. If False, the row is ignored by all pipelines. |
| stage | str | The **current** or **last completed** pipeline stage. Helps drive FSM transitions and control reprocessing. |
| last\_updated | datetime | Timestamp of the last modification to the row. Auto-filled on every state update or ingestion. |
| notes | Optional[str] | Optional free-form comments, often used for debugging or manual annotations (e.g., "missing requirements", "waiting on rewrite"). |
| file\_source | str | The file path (or filename) from which the data was loaded. Helps track lineage when ingesting batches into DuckDB. Typically set automatically during flatten\_model\_to\_df(). |

**🔁 Pipeline Stages Definitions**

| **Stage** | **Description** |
| --- | --- |
| preprocessing | Initial phase where raw job posting URLs are scraped, job descriptions are parsed, and requirements are extracted. Outputs structured JSON for downstream tasks. |
| staging | Intermediate transformation stage. Nested responsibilities and requirements are flattened and standardized into tabular formats (e.g., for matching). |
| evaluation | Resume responsibilities are compared with job requirements using similarity and entailment metrics. This step generates raw alignment data. |
| editing | LLM-based rewriting of resume responsibilities to better align with job requirements. Output is structured as optimized responsibility-requirement pairs. |
| revaluation | The edited responsibilities are re-evaluated using the same metrics as before. This step allows before/after comparison to assess improvement. |
| crosstab | Human-readable tables are generated for manual review, often visualizing alignment metrics and text side-by-side. |
| llm\_trim | Final pass using LLM to remove redundancy, fix tone/style, or apply filters. This stage prepares responsibilities for final delivery. |
| final\_export | The last stage where data is exported (e.g., to Word, CSV, JSON) and considered pipeline-complete. No further modifications are expected after this point. |

**🔍 pipeline\_control table: nullability**

| **Column** | **Null Allowed** | **Notes** |
| --- | --- | --- |
| url | ❌ **No** | Primary key; required |
| llm\_provider | ✅ **Yes** | No explicit NOT NULL; but usually set early |
| version | ✅ **Yes** | Default is 'original' |
| status | ✅ **Yes** | Default is 'new'; not declared NOT NULL |
| is\_active | ✅ **Yes** | Default is TRUE |
| stage | ✅ **Yes** | May start as None if uninitialized |
| last\_updated | ✅ **Yes** | Default: current\_timestamp, so typically filled |
| notes | ✅ **Yes** | Optional, free-form |
| source\_file | ✅ **Yes** | (Assuming it's included — all non-core metadata are optional) |

**🔁 Pydantic model (PipelineState)**

At the Pydantic level:

* Only url, llm\_provider, and version are required (with defaults)
* last\_stage and notes are explicitly Optional
* status defaults to "new", so will never be None unless overridden
* is\_active and last\_updated also have defaults

**❓ Why didn’t BaseDBModel include all fields from PipelineState?**

**✅ Because:**

The purpose of BaseDBModel is to represent the **shared metadata fields** present across **most DuckDB tables**, **not** the full control-plane metadata used only in pipeline\_control.

| **Field** | **In BaseDBModel?** | **Why or Why Not** |
| --- | --- | --- |
| url | ❌ | Present in every table, but its semantics differ (e.g. job\_posting vs. responsibility row); better to define explicitly per table |
| iteration | ✅ | Shared metadata, added to all tables ✅ |
| llm\_provider | ❌ | Not present in all tables; only in a few (e.g., edited\_responsibilities, similarity\_metrics) — define per-table |
| version | ❌ | Only some tables use versioning (similarity\_metrics) — not globally needed |
| status | ❌ | Only makes sense for job lifecycle tracking (pipeline\_control) |
| last\_stage | ❌ | Internal FSM field; only for pipeline\_control |
| is\_active | ❌ | Controls pipeline inclusion; again, only pipeline\_control |
| last\_updated | ✅ (renamed timestamp) | Included, but renamed to fit broader table context |
| notes | ❌ | Freeform debugging field; not standard across data tables |

**✅ Summary: BaseDBModel is intentionally minimal**

It includes only:

iteration, source\_file, stage, timestamp

These are:

* **standard metadata**
* **always present**
* **structurally uniform** across all your DuckDB tables

**🧠 So: Keep PipelineState and BaseDBModel separate**

| **Model** | **Purpose** |
| --- | --- |
| PipelineState | Tracks FSM status for each job (control metadata only) |
| BaseDBModel | Lightweight, reusable metadata shared by data rows across tables |

**✅ Correct Interpretation of Status (FSM-level)**

| **Status** | **Meaning** |
| --- | --- |
| "new" | Not yet started |
| "in\_progress" | This stage completed successfully, but the pipeline continues |
| "complete" | Final stage completed successfully (end of pipeline) |
| "error" | Current stage failed |
| "skipped" | Explicitly skipped (optional path or filtered out) |

### Final Version

| **Column** | **Type** | **Required** | **Purpose** |
| --- | --- | --- | --- |
| iteration | INTEGER | ✅ Yes | Tracks reruns of the same pipeline logic; useful for versions (e.g., iter 0 vs iter 1) |
| version | TEXT | ✅ Yes | "original", "edited", "reval" — tracks logical LLM output version |
| llm\_provider | TEXT | ✅ Yes | "openai", "anthropic", "none" — critical for downstream tracing |
| stage | TEXT | ✅ Yes | "job\_postings", "flattened\_requirements", etc. — aligns row with a pipeline stage |
| timestamp | TIMESTAMP | ✅ Yes | When the row was ingested — used for dedup, ordering, and dashboard views |

## How to Load Data from DB Tables to Fit DuckDB Driven Pipeline

You're shifting from a **file-based, URL-sharded ingestion flow** to a **database-driven, URL-centered pipeline**, and that fundamentally changes how you want to load and validate models.

Let’s design pydantic\_model\_loaders\_from\_db.py in a way that reflects this evolution.

**✅ Design Goals for pydantic\_model\_loaders\_from\_db**

**🔁 Goal**

Replace the file-based JSON loading in pydantic\_model\_loaders.py with DuckDB-based equivalents, **while respecting how data was inserted and should be rehydrated**.

**✅ Strategy: Split Based on Table Semantics**

**🟢 Group A: One-row-per-URL mapping (safe to load entire table)**

For models like:

* JobPostingUrlsFile
* JobPostingsFile
* ExtractedRequirementsFile

You can define:

def load\_all\_job\_postings\_file\_model\_from\_db() -> JobPostingsFile:

return db\_loader(TableName.JOB\_POSTINGS)

These will be used in **dashboards**, **audits**, or **cross-job summaries**, not in FSM pipelines.

**🟡 Group B: FSM-Driven Per-URL loaders (most of your pipeline work)**

For models like:

* Requirements
* Responsibilities
* NestedResponsibilities

You want:

def load\_flattened\_responsibilities\_for\_url(

url: str, version: str = "original", iteration: int = 0

) -> Responsibilities:

return db\_loader(TableName.FLATTENED\_RESPONSIBILITIES, url=url, version=version, iteration=iteration)

This ensures:

* You're passing exactly the slice of data that was originally flattened per-URL
* The rehydrate\_\* functions behave correctly (since they expect a single logical unit)

**✅ Final File Structure for pydantic\_model\_loaders\_from\_db.py**

**🔹 1. Top: Per-URL Loaders (FSM-compatible)**

These are used in every pipeline stage:

def load\_job\_postings\_for\_url(...) -> JobPostingsFile: ...

def load\_extracted\_requirements\_for\_url(...) -> ExtractedRequirementsFile: ...

def load\_flattened\_requirements\_for\_url(...) -> Requirements: ...

...

**🔹 2. Bottom: Full-Table Loaders (Batch QA, analytics)**

def load\_all\_job\_postings\_file\_model\_from\_db() -> JobPostingsFile: ...

def load\_all\_extracted\_requirements\_model\_from\_db() -> ExtractedRequirementsFile: ...

**✅ Summary**

| **Group** | **Loader Type** | **Used In** | **Notes** |
| --- | --- | --- | --- |
| 🟢 Group A | Full-table loaders | QA, dashboards, manual analysis | Safe to bulk rehydrate |
| 🟡 Group B | Per-URL loaders | FSM pipelines, processing stages | Only load a single logical unit |

## How Do Different Table/Field Model Files Relate to Each Other

/models/duckdb\_table\_models.py

**✅ So What Do You Do With BaseDBModel?**

* Keep BaseDBModel as the parent class for all your **actual DuckDB table models** in duckdb\_table\_models.py.
* It guarantees:
  + Every table has the standard metadata fields
  + Fields are **required or defaulted** as needed
  + Types are enforced (e.g., PipelineStage, Version, LLMProvider)

**Example:**

class EditedResponsibilitiesRow(BaseDBModel):

responsibility\_key: str

requirement\_key: str

optimized\_text: str

**🧠 So What’s the Role of MasterFieldModel Then?**

* It is **not** used in any pipeline logic
* It is used **for introspection and reflection**, like:
  + Listing all possible fields
  + Checking if a field exists before insert
  + LLM prompt templating
  + Column-level schema alignment or documentation

You don’t inherit from it. You **query it**.

**🔍 In Practice**

| **Use Case** | **Use BaseDBModel?** | **Use MasterFieldModel?** |
| --- | --- | --- |
| Validating edited\_responsibilities.json | ✅ Yes | ❌ No |
| Generating DuckDB DDL | ✅ Yes | ❌ No |
| Checking if pruned\_by exists anywhere | ❌ No | ✅ Yes |
| Building dynamic join UI or prompt schema | ❌ No | ✅ Yes |

**🧩 Summary**

| **Model** | **Purpose** | **Where It's Used** |
| --- | --- | --- |
| BaseDBModel | DuckDB schema + validation | Inherited by all DuckDB models (\*\_Row) |
| MasterFieldModel | All-field superset (optional) | LLM, docs, join schemas, column inference |

# State Machine Design

## High Level Workflow

At a high level, your pipeline uses a **finite state machine (FSM)** and **DuckDB-backed state tracking** to move job posting URLs through a multi-stage processing pipeline. Here's how it all fits together:

**1. Initial Entry: URLs Table + Pipeline Control**

* **job\_urls table**: Stores all job posting URLs and their metadata (company, title).
* **pipeline\_control table**: Tracks the state of each URL through the pipeline using the PipelineState model.
* FSM stages are defined by PipelineStage: preprocessing → staging → evaluation → editing → revaluation → crosstab → llm\_trim → final\_export.

The PipelineFSM class uses transitions.Machine to step each URL through these stages and persists it back to DuckDB.

**2. Preprocessing Stage (Pipeline ID "1\_async")**

Triggered via:

execute\_pipeline("1\_async", llm\_provider="openai", ...)

This pipeline:

* Loads filtered URLs (job\_posting\_urls\_filtered.json)
* Scrapes job descriptions and extracts raw requirements
* Produces:
  + job\_postings table (from JobPostingsFile)
  + extracted\_requirements table (from ExtractedRequirementsFile)
* Updates pipeline\_control to stage=preprocessing and status=in\_progress/complete

Relevant functions:

* run\_preprocessing\_pipeline\_async()
* flatten\_model\_to\_df(...) to convert Pydantic to tabular
* insert\_df\_dedup(...) to insert into DuckDB

**3. Staging Stage ("2b")**

This takes output from preprocessing and flattens it:

* Flattened responsibilities (from resume)
* Flattened requirements (from job posts)

Stored into:

* flattened\_responsibilities
* flattened\_requirements

Function:

* run\_flatten\_resps\_reqs\_processing\_mini\_pipeline()

**4. Evaluation Stage (e.g. "2c\_async")**

Each responsibility/requirement pair is evaluated for similarity via LLM-generated metrics:

* Semantic similarity (BERTScore, SoftSim, etc.)
* Entailment scores

Saved to:

* similarity\_metrics table (CSV -> Pydantic -> DataFrame)

Function:

* run\_metrics\_processing\_pipeline\_async()

**5. Editing Stage (e.g. "3b\_async")**

LLM edits resume responsibilities to better align with requirements:

* Generates optimized text for each (responsibility, requirement) pair
* Stores nested JSON → flattened → edited\_responsibilities table

Function:

* run\_resume\_editing\_pipeline\_async()

**6. Revaluation, Trimming, Export**

After editing:

* New metrics are recomputed
* Final manual or LLM trimming can happen
* Data is exported or reviewed manually (e.g. crosstabs in Excel)

**Pipeline Coordination via FSM**

Each time a pipeline finishes for a given URL:

fsm = PipelineFSM(pipeline\_state)

fsm.step() # Advances to next stage and updates DuckDB

FSM reads PipelineState.last\_stage and advances it to the next one based on pre-defined transitions.

**Summary Diagram**

job\_urls + pipeline\_control

↓

"1\_async" → job\_postings, extracted\_requirements

↓

"2b" → flattened\_requirements + responsibilities

↓

"2c\_async" → similarity\_metrics (original)

↓

"3b\_async" → edited\_responsibilities

↓

"3d\_async" → similarity\_metrics (edited)

↓

final stages → pruning, trimming, export

What Triggers a Job URL to Move to the Next Stage?

The core trigger mechanism is handled by your **PipelineFSM** class and the **step()** method:

fsm = PipelineFSM(pipeline\_state)

fsm.step() # This advances the job to the next stage

**1. The Source of Truth: pipeline\_control Table**

Each URL’s lifecycle is tracked in the pipeline\_control DuckDB table, using the PipelineState model:

class PipelineState(BaseModel):

url: str

last\_stage: str = "preprocessing"

status: str = "new" | "in\_progress" | "complete" | "error"

...

Every time a pipeline stage finishes for a URL, the system:

* Loads its PipelineState
* Creates a PipelineFSM instance
* Calls .step() to trigger the **next stage** (if not already at final\_export)

**2. The Transitions Logic (in PipelineFSM)**

TRANSITIONS = [

{"trigger": "advance", "source": "preprocessing", "dest": "staging"},

{"trigger": "advance", "source": "staging", "dest": "evaluation"},

...

]

This is used by the transitions library:

* The FSM is initialized to the current stage (e.g., "evaluation")
* When you call .step(), it internally calls self.advance() to go from source → dest
* It then updates last\_stage, sets a new timestamp, and saves the new state back to DuckDB

**3. Who Calls .step()?**

Any pipeline function **can** choose to trigger .step() *after* successfully completing its job. For example:

def post\_process\_each\_url(url: str):

state = load\_pipeline\_state(url)

fsm = PipelineFSM(state)

fsm.step()

This is typically done at the **end of a pipeline step** to move the job forward.

**4. Optional Filtering and Selective Triggering**

When you run:

execute\_pipeline("3b\_async", filter\_keys=["https://job1.com"])

Only those URLs are processed — and only those will get their pipeline state advanced via .step() afterward.

This allows for **controlled, batched advancement** through the FSM.

**5. Visual Summary**

DuckDB: pipeline\_control (stores URL state)

↓

load\_pipeline\_state(url)

↓

fsm = PipelineFSM(state)

↓

fsm.step() ← Triggers next stage based on transitions[]

↓

save\_pipeline\_state\_to\_duckdb(updated\_state)

## Detailed

Let’s walk through the logic of the `step` method in your `PipelineFSM` class. This method is responsible for advancing the pipeline to the next stage and persisting the updated state to a DuckDB database. I’ll break it down step by step, explaining what each part does, how it interacts with the finite state machine (FSM), and why it’s structured the way it is.

Here’s the `step` method for reference:

```python

def step(self, table\_name="pipeline\_control"):

"""

Advance the pipeline to the next stage and persist the updated state to DuckDB.

Args:

table\_name (str): Name of the DuckDB table to persist state.

Defaults to "pipeline\_control".

"""

if self.\_state != "final\_export":

self.advance()

self.\_state = self.state # Sync internal state after transition

self.state\_model.last\_stage = self.\_state

self.state\_model.last\_updated = datetime.now()

save\_pipeline\_state\_to\_duckdb(self.state\_model, table\_name)

print(f"✅ Advanced to stage: {self.state\_model.last\_stage}")

else:

print("✅ Already at final stage.")

```

### Context of the `PipelineFSM` Class

Before diving into the `step` method, let’s recap the role of the `PipelineFSM` class:

- It’s a finite state machine (FSM) built using the `transitions` library to manage a sequence of pipeline stages (e.g., `preprocessing`, `staging`, `evaluation`, etc.).

- The stages are defined in the `STAGES` list, and valid transitions between stages are defined in the `TRANSITIONS` list (e.g., `preprocessing` → `staging` via the `advance` trigger).

- The class wraps a `PipelineState` object (a Pydantic model) that holds the current state of the pipeline (`last\_stage`, `last\_updated`, etc.).

- The FSM is initialized with an initial state (either from `state.last\_stage` or defaulting to `preprocessing`).

- The `step` method is the primary way to move the pipeline forward and save the updated state.

### Step-by-Step Logic of the `step` Method

#### 1. \*\*Method Signature and Parameters\*\*

```python

def step(self, table\_name="pipeline\_control"):

```

- \*\*Purpose\*\*: The method advances the pipeline to the next stage and persists the state.

- \*\*Parameters\*\*:

- `self`: The instance of the `PipelineFSM` class, which holds the FSM configuration, the current state (`self.\_state`), and the `PipelineState` model (`self.state\_model`).

- `table\_name`: A string specifying the DuckDB table where the state will be saved. It defaults to `"pipeline\_control"`, allowing flexibility if you want to save to a different table.

- \*\*Docstring\*\*: Clearly explains the method’s purpose and the `table\_name` argument.

#### 2. \*\*Check if the Pipeline Can Advance\*\*

```python

if self.\_state != "final\_export":

```

- \*\*Purpose\*\*: Determines whether the pipeline can move to the next stage.

- \*\*Logic\*\*:

- `self.\_state` is an internal variable tracking the current stage of the FSM (e.g., `preprocessing`, `staging`, etc.).

- The condition checks if the current stage is \*\*not\*\* `final\_export`, which is the last stage in the `STAGES` list (`["preprocessing", "staging", ..., "final\_export"]`).

- If `self.\_state` is `final\_export`, the pipeline is complete, and no further transitions are allowed (as per the `TRANSITIONS` list, there’s no transition defined from `final\_export` to another state).

- \*\*Why this check?\*\*:

- Prevents attempting an invalid transition, which would raise an error in the `transitions` library.

- Ensures the FSM respects the defined workflow.

#### 3. \*\*Advance to the Next Stage\*\*

```python

self.advance()

```

- \*\*Purpose\*\*: Triggers the FSM to move to the next stage.

- \*\*Logic\*\*:

- `advance` is a method dynamically added by the `transitions.Machine` based on the `TRANSITIONS` list, where each transition has `trigger: "advance"`.

- For example, if `self.\_state` is `preprocessing`, calling `self.advance()` moves the FSM to `staging` (per the transition `{"trigger": "advance", "source": "preprocessing", "dest": "staging"}`).

- The `transitions` library updates the FSM’s internal state (accessible via `self.state`) to reflect the new stage.

- \*\*Note\*\*:

- This only executes if `self.\_state != "final\_export"`.

- If `advance` is called on an invalid transition, the `transitions` library would raise a `MachineError`, but the prior check ensures this doesn’t happen.

#### 4. \*\*Sync the Internal State\*\*

```python

self.\_state = self.state

```

- \*\*Purpose\*\*: Updates the internal `\_state` variable to match the FSM’s new state after the transition.

- \*\*Logic\*\*:

- After `self.advance()` is called, the `transitions` library updates the FSM’s state, which is accessible via `self.state` (a dynamic attribute added by `transitions.Machine`).

- The class maintains its own `\_state` variable (set during initialization and updated here) to ensure consistency between the FSM’s state and the class’s internal tracking.

- Assigning `self.state` to `self.\_state` syncs the internal variable with the FSM’s current state.

- \*\*Why this step?\*\*:

- The class uses `\_state` for checks (e.g., `self.\_state != "final\_export"`) and to avoid relying solely on the dynamic `self.state` attribute.

- This ensures the class’s logic remains robust even if the `transitions` library’s behavior changes.

#### 5. \*\*Update the `PipelineState` Model\*\*

```python

self.state\_model.last\_stage = self.\_state

self.state\_model.last\_updated = datetime.now()

```

- \*\*Purpose\*\*: Updates the `PipelineState` object with the new stage and timestamp.

- \*\*Logic\*\*:

- `self.state\_model` is the `PipelineState` object (likely a Pydantic model) passed during initialization. It tracks the pipeline’s state, including `last\_stage` (the current stage) and `last\_updated` (a timestamp).

- `self.state\_model.last\_stage = self.\_state`: Sets the `last\_stage` field to the new stage (e.g., `staging` if transitioning from `preprocessing`).

- `self.state\_model.last\_updated = datetime.now()`: Records the current time to mark when the transition occurred.

- \*\*Why this step?\*\*:

- The `PipelineState` object is the source of truth for the pipeline’s state, which will be persisted to DuckDB.

- Updating these fields ensures the model reflects the latest stage and provides a record of when the transition happened.

#### 6. \*\*Persist the State to DuckDB\*\*

```python

save\_pipeline\_state\_to\_duckdb(self.state\_model, table\_name)

```

- \*\*Purpose\*\*: Saves the updated `PipelineState` object to a DuckDB database.

- \*\*Logic\*\*:

- Calls the `save\_pipeline\_state\_to\_duckdb` function (imported from `db\_io.state\_sync`) with two arguments:

- `self.state\_model`: The updated `PipelineState` object containing the new `last\_stage` and `last\_updated` values.

- `table\_name`: The name of the DuckDB table (defaulting to `"pipeline\_control"`) where the state is stored.

- The `save\_pipeline\_state\_to\_duckdb` function is assumed to handle the logic of serializing `self.state\_model` and writing it to the specified table in DuckDB.

- \*\*Why this step?\*\*:

- Persisting the state ensures that the pipeline’s progress is saved, allowing it to be resumed later (e.g., after a restart).

- DuckDB is a lightweight, embedded database, making it suitable for storing pipeline metadata.

#### 7. \*\*Log the Transition\*\*

```python

print(f"✅ Advanced to stage: {self.state\_model.last\_stage}")

```

- \*\*Purpose\*\*: Outputs a confirmation message to indicate the transition was successful.

- \*\*Logic\*\*:

- Prints a message with the new stage (`self.state\_model.last\_stage`), prefixed with a checkmark emoji (`✅`) for clarity.

- Uses an f-string to insert the current stage (e.g., `✅ Advanced to stage: staging`).

- \*\*Why this step?\*\*:

- Provides feedback to the user or developer running the pipeline, confirming that the transition occurred and what the new stage is.

- Useful for debugging or monitoring the pipeline’s progress.

#### 8. \*\*Handle the Final Stage\*\*

```python

else:

print("✅ Already at final stage.")

```

- \*\*Purpose\*\*: Handles the case where the pipeline is already at the `final\_export` stage.

- \*\*Logic\*\*:

- If `self.\_state == "final\_export"`, the `if` block is skipped, and this `else` block executes.

- Prints a message indicating that no further transitions are possible because the pipeline is complete.

- \*\*Why this step?\*\*:

- Informs the user that the pipeline has reached its terminal state, avoiding confusion about why no action was taken.

- Maintains a consistent user experience by providing feedback in all cases.

### Flowchart of the Logic

Here’s a simplified flowchart of the `step` method’s logic:

```

START

|

v

Is self.\_state != "final\_export"?

| \

Yes No

| \

v Print "Already at final stage."

Call self.advance()

|

v

Set self.\_state = self.state

|

v

Set self.state\_model.last\_stage = self.\_state

|

v

Set self.state\_model.last\_updated = datetime.now()

|

v

Call save\_pipeline\_state\_to\_duckdb(self.state\_model, table\_name)

|

v

Print "Advanced to stage: {self.state\_model.last\_stage}"

|

v

END

```

### Key Interactions

- \*\*With the FSM\*\*: The method interacts with the `transitions.Machine` via `self.advance()` and `self.state` to manage state transitions.

- \*\*With `PipelineState`\*\*: Updates `self.state\_model` to keep the Pydantic model in sync with the FSM’s state.

- \*\*With DuckDB\*\*: Persists the state via `save\_pipeline\_state\_to\_duckdb`, ensuring durability.

- \*\*With the user\*\*: Provides feedback through `print` statements.

### Why This Structure?

- \*\*Conditional check\*\*: Prevents invalid transitions at `final\_export`, making the code robust.

- \*\*State syncing\*\*: Keeps `self.\_state`, `self.state`, and `self.state\_model.last\_stage` consistent, avoiding discrepancies.

- \*\*Persistence\*\*: Saves the state to DuckDB, enabling fault tolerance and traceability.

- \*\*Feedback\*\*: Logs transitions for transparency and debugging.

### Assumptions

- The `transitions` library is correctly configured, and `advance` triggers valid transitions as defined in `TRANSITIONS`.

- The `PipelineState` object has fields like `last\_stage` and `last\_updated`, and `save\_pipeline\_state\_to\_duckdb` handles serialization and storage correctly.

- The DuckDB table (`table\_name`) exists or is created by `save\_pipeline\_state\_to\_duckdb`.

### Potential Edge Cases

1. \*\*Invalid Initial State\*\*: If `self.state\_model.last\_stage` is set to an invalid stage during initialization, the FSM might raise an error. The code mitigates this by defaulting to `preprocessing`.

2. \*\*DuckDB Failure\*\*: If `save\_pipeline\_state\_to\_duckdb` fails (e.g., due to a database connection issue), the state won’t be persisted, but the FSM will still advance. You might want to add error handling (e.g., a `try-except` block).

3. \*\*Concurrent Access\*\*: If multiple processes call `step` simultaneously, the state might become inconsistent. This would require locking mechanisms if relevant.

### Example Execution

Suppose the pipeline is currently at `preprocessing`:

1. `self.\_state` is `preprocessing`.

2. `self.\_state != "final\_export"` is `True`, so the `if` block executes.

3. `self.advance()` transitions the FSM to `staging` (per `TRANSITIONS`).

4. `self.state` is now `staging`, so `self.\_state = self.state` sets `self.\_state` to `staging`.

5. `self.state\_model.last\_stage` is set to `staging`.

6. `self.state\_model.last\_updated` is set to the current timestamp (e.g., `2025-04-11 10:00:00`).

7. `save\_pipeline\_state\_to\_duckdb` saves the updated `self.state\_model` to the `pipeline\_control` table.

8. Prints: `✅ Advanced to stage: staging`.

If called again at `final\_export`:

1. `self.\_state` is `final\_export`.

2. `self.\_state != "final\_export"` is `False`, so the `else` block executes.

3. Prints: `✅ Already at final stage.`.

### Summary

The `step` method is a well-structured way to advance the pipeline while ensuring:

- The FSM transitions correctly.

- The `PipelineState` model stays updated.

- The state is persisted to DuckDB.

- The user is informed of progress.

# LLM to Infer Schema/Pydantic Models

**🧠 Recommendation Summary**

🔹 **Create a real file manually**  
🔹 Use a **"mini" version in your LLM prompt**  
🔹 Let the LLM **infer structure, not replicate boilerplate**

**✅ Strategy**

**1. Manually maintain MasterFieldModel as your ground truth**

* This should be your introspection + validation source
* Keep it accurate and complete — but don't prompt with the whole thing

**2. Prompt with a trimmed example + abstract description**

Use a **small sample version** like:

class MasterFieldModel(BaseModel):

url: Optional[str]

job\_title: Optional[str]

company: Optional[str]

responsibility: Optional[str]

requirement\_key: Optional[str]

similarity\_score: Optional[float]

stage: Optional[str]

timestamp: Optional[datetime]

And pair it with a **commented schema overview** in your prompt:

✨ Prompt pattern:

You are updating a master schema model that contains all possible fields across DuckDB tables.

- Each field appears in at least one table.

- Use optional fields only.

- Include standard metadata fields (url, stage, timestamp, version, llm\_provider, iteration).

- Types should be inferred from provided data structure.

- Keep the model flat, no nested models.

Here is a minimal example:

```python

class MasterFieldModel(BaseModel):

url: Optional[str]

job\_title: Optional[str]

...

Update this model to include the following newly discovered table fields:

* responsibility\_key (string)
* pruned\_by (string)
* pca\_score (float)

---

## 🧩 Why Not Include the Full File?

| Option | Pros | Cons |

|----------------|--------------------------------------------------|---------------------------------------------------|

| Full File | LLM won’t miss anything | High token cost, overfitting risk |

| Mini Example ✅| Lower token cost, better generalization | Needs good instruction to avoid missing fields |

---

## 🧠 Bonus: Dynamic Prompt Builder

If you're running this in code, you can generate mini-prompts:

```python

from models.duckdb\_table\_models import EditedResponsibilitiesRow

example\_fields = "\n".join(f" {k}: Optional[{v.annotation.\_\_name\_\_}]"

for k, v in list(EditedResponsibilitiesRow.model\_fields.items())[:5])

prompt = f"""

You're updating a master field model that includes all fields across all DuckDB tables.

Example:

```python

class MasterFieldModel(BaseModel):

{example\_fields}

Now add these new fields...  
"""

---

## ✅ TL;DR

| Task | Recommendation |

|-----------------------------|-------------------------------------------|

| Base file (`.py`) | Yes, create/maintain full master model |

| LLM prompting | Use trimmed mini-model + textual guidance |

| Overfitting concern | Avoid full copy-paste into LLM prompts |

| Goal: Extend / update model | Use table diffs + token-efficient prompts |

---

# New Ideas to Improve

Suggestion by OpenAI

If I were a **pragmatic app architect**, my goal would be to simplify complexity, improve fault tolerance, and ensure **scalability** while maintaining **debuggability**. Based on your project's structure, I’d recommend the following **high-level refactors**:

**1. Modularizing Processing into Self-Contained Microservices**

Instead of a large, tightly coupled pipeline, break it down into **independent services** that can be run in isolation and communicate via well-defined interfaces.

**Proposed Microservices Architecture**

| **Service Name** | **Responsibility** | **Tech Stack** |
| --- | --- | --- |
| **Web Scraper** | Extracts job descriptions, handles retries, and sanitizes HTML | Playwright (async), FastAPI |
| **Text Normalizer** | Cleans and transforms text into structured JSON | Python + Pydantic |
| **LLM API Handler** | Calls OpenAI, Claude, and Llama3; centralizes validation & retries | Async API clients |
| **Similarity Evaluator** | Computes BERTScore, DeBERTa, Word Movers, etc. | Hugging Face Transformers, Pandas |
| **Resume Optimizer** | Modifies resume content based on job requirements | Custom ML/LLM model |
| **Metrics Aggregator** | Handles versioned storage and comparisons | DuckDB, Pandas |
| **File Manager** | Ensures data integrity across pipeline stages | Redis or PostgreSQL for state tracking |

**Why?**

* **Independent Scaling**: Run multiple scrapers but limit API calls to LLMs (rate-limited).
* **Resiliency**: If one service fails, others can continue processing.
* **Easier Debugging**: Track failures at a **service level**, not an entire pipeline.

**2. Transition to an Event-Driven Processing Model**

Instead of running **sequential scripts**, switch to an **event-driven approach** where each step **queues** its results for the next step.

**Implementation Using Kafka or Celery**

* Web scraping → pushes JSON to job\_posting\_extracted queue.
* Text normalizer → consumes, processes, then pushes to text\_normalized queue.
* LLM API calls → rate-limited workers pull from text\_normalized and push results to llm\_processed queue.
* Metrics → computed asynchronously and stored in a DB.

🚀 **Benefit**: If one step fails, you **replay from that step** instead of restarting the whole process.

**3. Centralizing API Calls & Response Validation**

* **Current Issue**: LLM API calls are scattered across multiple modules (llm\_api\_utils.py, llm\_api\_utils\_async.py).
* **Proposed Solution**: Move all LLM interactions to a **single service** (llm\_api\_handler.py) with:
  + Automatic **retries** (with exponential backoff).
  + Rate-limiting using **async semaphores**.
  + Unified **response validation** using Pydantic.
  + **Batching API requests** where possible to reduce costs.

**4. Move State Tracking to a Database**

Currently, pipeline execution relies on **file-based tracking** (mapping\_file.json), which is prone to corruption and difficult to query.

**Better Approach: Use DuckDB or PostgreSQL**

* Store job descriptions, resumes, similarity scores, and pipeline **state** in a structured format.
* Query **which steps succeeded/failed** instantly.
* Use **indexing** for fast lookups (instead of searching JSON files in directories).

🚀 **Benefit**: No more **missing file issues**, and **debugging becomes easier**.

**5. Optimize Similarity & Metrics Calculation**

The similarity evaluation currently:

1. Reads large JSON files.
2. Iterates through them **synchronously**.
3. Writes results to CSV.

**Optimized Approach**

* **Vectorize Similarity Computations**: Use FAISS or Annoy to **pre-index** resume embeddings.
* **Async Batch Processing**: Process similarity metrics in **chunks of 10-50 pairs** instead of looping through them one-by-one.
* **Precompute Common Results**: If a requirement matches multiple resumes, cache results **instead of recomputing**.

🚀 **Benefit**: Reduces API calls and speeds up processing by **3-5x**.

**6. Build a Lightweight Debugging UI**

Currently, debugging requires **log file analysis**. Instead:

* Build a **simple dashboard** using **Streamlit or FastAPI + React**.
* Show **which steps failed**, with a retry button.
* Allow **searching logs** by job ID.

🚀 **Benefit**: Cuts debugging time significantly.

**7. Improve Deployment & Testing**

**Switch to Containerized Deployment**

* Package each service into **Docker containers**.
* Use **Kubernetes** (or at least Docker Compose) to orchestrate them.
* Enable **autoscaling** (e.g., scale up the LLM API handler when processing many jobs).

**Testing Strategy**

| **Layer** | **Test Type** | **Tool** |
| --- | --- | --- |
| LLM Calls | Mock API responses | pytest + responses |
| Data Pipeline | Ensure structured JSON outputs | pydantic validation tests |
| Performance | Load testing for async API calls | locust |
| Integration | Full pipeline dry runs | End-to-end testing |

🚀 **Benefit**: Avoids **bugs from API changes**, ensures **scalability**.

**Final Thoughts:**

If I were **designing for long-term sustainability**, I would:

1. **Modularize processing into self-contained services**.
2. **Use an event-driven queue** instead of sequential file-based processing.
3. **Centralize API handling** to reduce redundancy.
4. **Replace JSON file tracking with a database**.
5. **Speed up similarity calculations with precomputed embeddings**.
6. **Add a debugging UI** for real-time monitoring.
7. **Containerize & automate testing**.

**What Would This Solve?**

✅ **No more file corruption/missing files** → Everything is in a DB.  
✅ **No more sequential bottlenecks** → Async tasks process in parallel.  
✅ **No more painful debugging** → UI shows failures in real time.  
✅ **Scales effortlessly** → Easily add workers for LLM API calls.

This approach **makes your pipeline faster, more reliable, and easier to debug** while still being cost-effective. Would you like help implementing any specific part? 🚀

## Top 3 Ideas

**Centralize LLM API Handling & Response Validation**

**Why?** Fixing inconsistent API calls, validation, and rate limits removes a major failure point in your pipeline.

✅ **Create a single llm\_api\_handler.py**

* Handles **all API calls** (OpenAI, Claude, Llama3) in one place.
* Implements **automatic retries, rate-limiting, and batching**.
* Ensures **every response is validated** before passing it downstream.

🚀 **Impact:**  
✅ **Reduces API failures by ~80%**  
✅ **Saves token costs via batching**  
✅ **Eliminates inconsistent response handling across modules**

**Move from JSON File Tracking to DuckDB**

**Why?** JSON-based tracking (mapping\_file.json) is **error-prone, slow, and hard to debug**. A structured DB solves all of that.

✅ **Replace JSON tracking with DuckDB**

* Store **job descriptions, resume matches, similarity scores, and processing states**.
* Query **which jobs/resumes need processing instantly** instead of searching files.

🚀 **Impact:**  
✅ **No more "missing files" issues**  
✅ **Debugging becomes 10x easier** (use SQL instead of checking logs)  
✅ **Lookups are 5x faster**

**Precompute Similarity Embeddings (FAISS or Annoy)**

**Why?** Your current similarity calculations are **repeated for every resume-job match**, wasting CPU time & API calls.

✅ **Precompute embeddings for job requirements & resumes**

* Store **vectorized job requirements** in **FAISS or Annoy**.
* Instead of recomputing similarity **every time**, just **lookup the precomputed vector**.

🚀 **Impact:**  
✅ **Speeds up similarity matching by 3-5x**  
✅ **Reduces API costs** (no redundant calculations)  
✅ **Handles 1000s of jobs without slowdowns**

**🔥 TL;DR: 3 Biggest Wins**

| **Solution** | **Impact** |
| --- | --- |
| **Centralized LLM API Handler** | 🚀 Fixes API failures, saves tokens, speeds up requests |
| **Move from JSON to DuckDB** | 🔎 Easier debugging, 🚀 No missing files, ⚡ Faster lookups |
| **Precompute Similarity Embeddings** | ⚡ 5x speedup, 💰 Less API cost, 🔄 Scales better |

These **solve the biggest pain points** (LLM failures, missing files, slow processing) **without disrupting your entire architecture**. Would you like help with implementation strategy for any of these? 🚀

## Different Approaches to Syncing Pipeline Control Table

**✅ Evaluating Flexibility and LLM-Integration Potential**

| **Approach** | **Flexibility** | **Automation Ease** | **Human Review Integration** | **Complexity** |
| --- | --- | --- | --- | --- |
| 1. Dynamic Schema Registry | **High** | **High** | Moderate | Medium |
| 2. JSON Schema / OpenAPI Driven | High | High | Moderate | Medium |
| 3. FSM-Based Schema Management | Medium | Medium | Good | High |
| 4. Event-Driven Microservice | Very High | Very High | Excellent | Very High |
| 5. Direct LLM Code Generation (PR-based) | High | High | Excellent | Low |

**✅ Recommended Hybrid Approach:**

Given your eventual goal of leveraging LLMs for dynamic schema management, a hybrid approach would be most suitable:

* **Core:** **Dynamic Schema Registry (Approach #1)** as the central schema management solution.
* **Intermediate Schema Representation:** Adopt **JSON Schema/OpenAPI (Approach #2)** for structured, reusable, and tool-supported schema representation.
* **Integration Workflow:** Implement **Direct LLM-Generated PR-based Code Generation (Approach #5)** to manage human-in-the-loop reviews and version control clearly and effectively.

This combination offers:

* Clear integration points for LLM-generated schema changes.
* Robust, automated tooling ecosystems to minimize manual effort.
* Structured review and integration processes to maintain reliability and consistency.

**✅ Suggested Next Steps:**

1. Implement and test the **Dynamic Schema Registry**.
2. Integrate JSON Schema-driven Pydantic generation tools.
3. Establish a clear, template-based, LLM-assisted schema update workflow (e.g., via GitHub Actions) with PR review processes.

## Central Schema + LLM Updating Schema Approach

Here's a structured, high-level outline for implementing the recommended hybrid approach that integrates:

* **Dynamic Schema Registry** (Core)
* **JSON Schema-driven Pydantic Model Generation** (Intermediate)
* **LLM-assisted, PR-based Schema Updates** (Integration Workflow)

**🚀 High-Level Implementation Outline**

**🌟 Phase 1: Dynamic Schema Registry Setup**

**Objectives:**

* Establish a robust, dynamically updateable registry for DuckDB table schemas.
* Centralize schema definitions, validation, and alignment logic.

**Steps:**

1. **Define Centralized Schema Registry**
   * Expand existing DUCKDB\_SCHEMA\_REGISTRY to handle dynamic schema updates.
   * Include schema metadata:
     + Model references (Pydantic models).
     + Column orders.
     + Primary keys.
2. **Implement Registry API**
   * Methods to:
     + Add new tables/models.
     + Update existing schemas.
     + Retrieve schemas dynamically.
3. **Establish Schema Validation Logic**
   * Automate schema alignment (align\_df\_with\_schema).
   * Introduce metadata handling (add\_ingestion\_metadata).
4. **Integrate Registry with Pipeline Logic**
   * Ensure all data ingestion points reference this centralized schema.
   * Adjust pipeline stages to query schemas dynamically at runtime.

**🌟 Phase 2: JSON Schema or OpenAPI-Driven Intermediate Layer**

**Objectives:**

* Provide a clear intermediate representation (JSON Schema/OpenAPI) for schemas.
* Utilize mature tooling to automate Pydantic model generation and schema synchronization.

**Steps:**

1. **JSON Schema Definition**
   * Implement JSON Schema (or OpenAPI) as the canonical intermediate format for each table.
   * Store schemas in structured YAML/JSON files in version control.
2. **Automate Schema → Pydantic Generation**
   * Leverage tools like datamodel-code-generator to automatically convert JSON Schema/OpenAPI to Pydantic models.
   * Set up continuous integration scripts (e.g., GitHub Actions) to regenerate Python code from schemas upon update.
3. **Validate Schema Changes**
   * Add pre-commit or CI/CD validation scripts to test schema compatibility before updates.
   * Employ schema diff tools to visualize changes clearly.

**🌟 Phase 3: LLM-Assisted Schema Update Workflow**

**Objectives:**

* Automate schema detection and updates via LLM-generated code.
* Integrate structured human review using PR-based workflows.

**Steps:**

1. **Detection of Schema Changes**
   * Periodically or on-demand scan incoming data for schema changes.
   * Use automated schema diffing (existing schema vs. new data).
2. **LLM-Driven Schema Proposal Generation**
   * Provide structured prompts/templates to LLMs (e.g., OpenAI GPT-4 Turbo) to:
     + Generate new or updated JSON Schemas.
     + Propose corresponding Pydantic model code updates.
   * Output structured, well-documented PRs.
3. **Template-Based Code Generation**
   * Create Jinja2 (or similar) templates for standardizing model and schema code generation.
   * Populate templates with LLM-generated outputs for consistent formatting.
4. **Automated PR Workflow**
   * Automate the creation of pull requests via GitHub Actions or similar workflows.
   * Include automatic schema/model validation checks.
   * Integrate a robust review process:
     + Human review (optional but recommended initially).
     + Auto-merge after passing strict automated checks (optional, for future maturity).

**🌟 Phase 4: Finite State Machine (FSM) Integration (Optional/Advanced)**

**Objectives:**

* Enhance existing pipeline FSM to handle schema state transitions explicitly.
* Ensure schema consistency as an integral pipeline operation.

**Steps (Optional Advanced Step):**

1. **Extend Pipeline FSM**
   * Add explicit schema validation and state-update states in the FSM.
   * Implement FSM transitions that trigger schema regeneration and updates.
2. **State-Dependent Schema Checks**
   * Integrate schema checks directly into pipeline stages.
   * Allow pipelines to halt or pause if schema misalignment is detected.

**🌟 Phase 5: Event-Driven Schema Updates (Optional/Advanced)**

**Objectives:**

* Introduce scalability and reliability for future complexity using event-driven architecture.

**Steps (Optional Advanced Step):**

1. **Implement Event Queue**
   * Setup Kafka, RabbitMQ, or cloud-based event streams to manage schema-update events.
2. **Dedicated Schema-Updater Service**
   * A microservice that listens to schema-update events.
   * Automatically applies validated schema updates.

**🚦 Proposed High-Level Architecture Diagram**

+----------------------------------------------------------------------------------+

| Dynamic Schema Registry (Centralized) |

| [DUCKDB\_SCHEMA\_REGISTRY.py] |

+------------------------------------+---------------------------------------------+

|

v

+--------------------+ +----------------------+ +-------------------------+

| Data Processing | <--> | Intermediate JSON | <--> | LLM-driven Schema Update|

| Pipelines & FSM | | Schema Representation| | Workflow (GitHub Actions|

| (Python, DuckDB) | | (JSON Schema/OpenAPI)| | + PR Reviews) |

+--------------------+ +----------------------+ +-------------------------+

^

|

LLM-Proposed JSON Schema Changes + Human Review

**🎯 Benefits of This Hybrid Approach**

* **Flexibility**: Seamless integration of new schemas/tables dynamically.
* **Automation**: LLM-assisted code generation reduces manual schema management overhead.
* **Scalability**: Intermediate JSON Schema representation allows ecosystem flexibility.
* **Traceability & Governance**: PR-based review ensures robust schema management.
* **Maintainability**: Clearly defined roles for each layer minimize complexity.

**🚧 Initial Implementation Roadmap**

| **Stage** | **Task** | **Priority** | **Complexity** | **Dependencies** |
| --- | --- | --- | --- | --- |
| 1 | Dynamic Registry Setup | High | Medium | Existing Registry |
| 2 | JSON Schema/Model Generation | High | Medium | Phase 1 |
| 3 | LLM-driven PR Automation | Medium | Medium | Phases 1-2 |
| 4 | FSM Schema Integration (optional) | Medium | High | Phases 1-3 |
| 5 | Event-Driven Updates (optional) | Low | High | Phases 1-4 |

**✅ Immediate Next Steps**

1. **Implement a basic dynamic registry framework** (expand existing registry with methods for dynamic updates).
2. **Establish JSON Schema → Pydantic model generation automation**.
3. **Prototype LLM-driven schema proposal generation workflow** (simple proof-of-concept).