ChatGPT

Semantic Matching in Structured Legal Documents: State-of-the-Art Approaches

Semantic matching across highly structured legal documents (like ERISA retirement plan provisions) is a challenging task. Different vendors use distinct section numbering, hierarchies, and wording for conceptually similar plan elections. The goal is to map each provision/election in a source document to its semantically equivalent provision in a target document **based on meaning, not formatting**. This requires ignoring structural artifacts (e.g. section numbers, labels) and focusing on the underlying plan design decision. Below, we review current best practices and research for tackling this problem, addressing embedding strategies, filtering architectures, LLM verification, alternative approaches, and evaluation methods. Throughout, we emphasize **high precision** (avoiding false matches) given the compliance risks of incorrect mappings.

Challenges in Semantic Document Alignment

- **Structural Variations:** Retirement plan documents are highly structured (section/subsection numbering, labeled questions, etc.), but each provider may organize content differently. A purely lexical or structural match can be misleading – e.g. two "Section 1.04" entries might refer to entirely different topics. The matching system must **ignore positional/numbering cues** and rely on semantic content.
- **Domain-Specific Language:** Legal/compliance text uses specialized terminology and complex phrasing. Synonyms and domain jargon are common, so a matching system must recognize concept equivalences expressed in different words (e.g. "entry age" vs "eligibility age"). Off-the-shelf language models may not fully capture these nuances without adaptation [1].
- **False Positives vs. False Negatives:** In this compliance context, a false positive (incorrectly mapping two unrelated provisions) is more harmful than a missed match. An incorrect match could lead to faulty guidance and regulatory risk. Thus, the system should err on the side of caution – high precision is paramount, even if some true matches are initially missed.
- **Scale and Efficiency:** With on the order of 100k potential pairings (e.g. 182 source × 550 target items), a brute-force comparison is computationally expensive. The solution should **filter candidates efficiently** (e.g. via embeddings or heuristics) before any heavy-weight semantic comparison. All processing must complete within practical time (e.g. < 1 hour) and cost (< \$20 per run), which constrains the complexity of models used in each stage.

Given these challenges, a hybrid approach is natural: use fast similarity measures to **prune the candidate list**, then apply a more precise (but slower) semantic comparison on the most likely matches. The existing two-stage pipeline (embedding retrieval → LLM verification) aligns with this strategy, but as observed, it can fail if not carefully designed to avoid structural traps and LLM hallucinations. We will explore improvements for each stage and alternative architectures.

Embedding Strategies for Structured Legal Text

Choosing the right embedding representation is crucial for the initial filtering stage. A good embedding will capture the substantive content of each provision or question, while minimizing the influence of formatting artifacts like section numbers. Key considerations include how to represent the structured **fields** of an election (question text, options, context) and whether to use general-purpose vs. domain-specific models.

- **Domain-Specific vs. General-Purpose Embeddings:** Legal and compliance texts have vocabulary and phrasing not common in general corpora. Domain-specific embedding models (e.g. ones trained or fine-tuned on legal documents) tend to perform better at legal semantic similarity than generic models [1]. For example, in legal information retrieval tasks, OpenAI's general `text-embedding-ada-002` yields strong semantic search results, but a specialized model like *Voyage Law embeddings* (fine-tuned on legal data) can further improve precision by understanding legal terminology and syntax [1]. Similarly, the Legal-BERT family (BERT models adapted to legal corpora) and derivatives like **Legal-SBERT** produce embeddings more attuned to legal semantics [2] [3]. Adopting or fine-tuning an embedding model on a corpus of retirement plan documents (or related legal text) will likely improve the quality of candidate retrieval. In practice, this could involve further training a Sentence-BERT model on known matched vs. non-matched provision pairs (a contrastive learning approach) so that semantically equivalent sections embed closer together than unrelated sections. Research shows that fine-tuning Siamese networks on domain-specific similarity data dramatically improves matching accuracy over off-the-shelf embeddings [4].

- **Representing Structured Content:** Each "election" in these documents isn't just free text – it may have a **question identifier**, a prompt, and a set of options or fill-in blanks. Simply concatenating all this text into one string for embedding can cause unwanted effects. For instance, including the question number "1.04(g)" in the text might lead the model to overweight that token, as happened when two different 1.04 sections were embedded and came out unnaturally similar. To prevent this, it's often better to **separate content fields** and either remove or down-weight purely structural tokens. One approach is to embed a cleaned text version of the question (e.g. "Eligibility age may not exceed 21" vs. "Employer's state") without the numbering prefixes. Another advanced approach is to create **multi-field embeddings**: embed each component (section title, question text, answer options, etc.) independently, then combine them (e.g. by a weighted sum or concatenation of vectors) so that the overall representation captures multiple aspects [5]. For example, we might generate one embedding for the section context ("Section 1.04 – Eligibility Requirements"), another for the core question text ("Age _____ (not to exceed 21)"), and perhaps embeddings for each answer option or field. These could be combined with higher weight on the core question text and lower weight on context, yielding a composite vector [5]. By tuning the weights, we ensure that **semantic content dominates** the similarity measure while structural context is only used to break ties or filter obviously unrelated sections.

- **Avoiding Structural Artifacts:** It bears emphasizing that any structural tokens (numbers, lettering, vendor-specific jargon) should be handled carefully. If the embedding model is picking up on section numbers or list labels, those should be stripped or masked in the embedding input. In practice, one might remove all purely numeric identifiers from the text before embedding, or replace them with a neutral placeholder if needed for format. Another strategy is to include such identifiers in a separate metadata field that is not embedded for similarity – for instance, use them only in a later rule-based

check (to *prevent* mismatches rather than to drive matches). The critical point is to ensure the **embedding reflects only the meaning** of the provision. The failure example given ("Age ___ (may not exceed 21)" vs. "State _***") likely happened because the embedding saw two short fill-in-the-blank phrases with similar structure or tokens (both had "Question 1.04" and a blank line), yielding a falsely high cosine similarity. Preprocessing the text to remove question numbers and generic filler text (like blanks or phrases like "Question***") can help avoid the embedding model treating two different questions as the same due to those surface features.

- **Contrastive Fine-Tuning:** With even a small set of labeled matches/non-matches, one can refine the embedding space to better suit this specific matching task. A contrastive learning approach would take known matching pairs as positive examples (pull these embedding vectors closer) and known non-matching or random pairs as negatives (push those apart). Recent research on legal text matching goes further by incorporating structural context into contrastive training. For instance, the *StructCoh* framework (2025) constructs graphs from legal texts (capturing dependencies and key topics) and uses a hierarchical contrastive objective to align semantically equivalent clauses across documents [6] [7]. The result was a significant boost in legal clause matching performance (e.g. +6% F1 on a statute matching benchmark) by teaching the model to recognize structural context and **not be fooled by superficial overlaps** [7]. While implementing a full graph-based encoder may be complex, the takeaway is that **context-aware contrastive training** can substantially improve embeddings for semantic alignment. Even a simpler approach – like fine-tuning a Siamese Legal-BERT on a few hundred true/false election pairs – could help the model learn, for example, that "eligibility age" and "entry age" are similar, whereas "age" vs "state" are not, despite any format similarity.

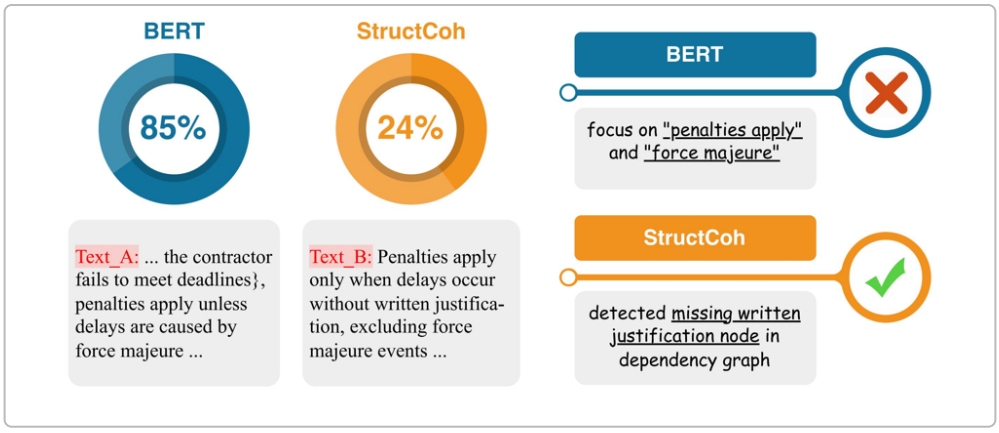

Illustration of a structural contrastive approach (StructCoh) versus a standard BERT embedding approach on legal clause matching. A vanilla BERT-based similarity model gave a high confidence (85%) that Text A and Text B match due to overlapping terms ("penalties apply" and "force majeure"), incorrectly treating them as similar. In contrast, StructCoh's graph-informed model detected a crucial difference ("missing written justification" in one clause) and correctly lowered the similarity to 24%, avoiding the false match. This demonstrates how incorporating hierarchical and dependency context prevents misalignments that a surface-level embedding might overlook.

In summary, **best practices for embeddings** in this scenario include: use the most domain-aware model available (or fine-tune one), represent the text in a way that emphasizes semantic content (possibly via multi-part embeddings or smart preprocessing), and consider leveraging any labeled data to train the embedding similarity for this specific matching task.

Candidate Filtering and Similarity Matching Architecture

The first-stage filtering aims to drastically cut down the 100k+ possible pairs to a manageable few candidates for each source. The current approach uses cosine similarity on embedding vectors to pick top-K matches. This is a sound strategy, but we should refine how similarity is computed and possibly augment it with additional rules to increase precision:

- **Cosine Similarity vs. Learned Scoring:** Cosine similarity in an embedding space is a fast way to rank candidates, but as we saw, it can produce false positives if the embedding isn't perfectly aligned to our notion of "same provision." An alternative is to train a lightweight **similarity model** that takes two embeddings (or the raw texts) and outputs a learned relevance score. For example, one could train a simple feed-forward network or even a logistic regression on features like cosine similarity and keyword overlaps, using the labeled examples as training data for match vs non-match. More powerfully, a **cross-encoder** model could be used at this stage: this is a transformer that takes both texts together and outputs a similarity score or classification (essentially what the GPT-5-mini is doing in Stage 2, but one could also use a smaller fine-tuned BERT for this). Cross-encoders generally achieve higher accuracy in text-pair ranking because the model can consider the full joint context [8]. In legal document retrieval, for instance, bi-encoder embeddings retrieve broad candidates, and then a cross-encoder re-ranker (like a mini-BERT that reads query and document together) often fixes false positives by focusing on finer details [9]. If computational budget allows, one idea is to introduce a **learned re-ranking** before even calling the heavy LLM: e.g. take top-10 by cosine, then run a smaller BERT cross-encoder on those to re-score and pick the top 3 to send to GPT. This two-step filtering (embedding → mini cross-encoder) would catch cases where the cosine was high for the wrong reasons.

- **Incorporating Lexical or Rule-Based Filters:** We can boost precision by adding some **rule-based pre-checks** in parallel with embeddings. For instance, ensure that a candidate pair shares at least one significant keyword or concept word. In the example, the source had "Age" and target had "State" – zero topical overlap. A simple check could have flagged that none of the important words match, so it's suspicious that embeddings considered them similar. Conversely, if one section talks about "service requirement" and another about "period of service", that keyword overlap is a positive signal. We could compile a list of domain-specific key terms (age, service, compensation, deferral, etc.) and require that a true match must share some of those terms or their synonyms. Another rule-based approach is to leverage the **section hierarchy**: if we know the source question is in an "Eligibility" section of one document, the matching target is likely in that document's equivalent "Eligibility" section, not in an unrelated section like "Employer Information." If the documents have labeled sections or we can infer topic grouping by titles, we could restrict the search scope (e.g. only compare source Section 1.x elections to target Section 1.x elections) or at least prioritize candidates from the same topical group. This kind of coarse alignment can dramatically cut false positives. However, caution is needed: different vendors might number things differently (Section 1 in one doc might correspond to Section 2 in another). Still, if we detect, say, that one document's Section 1 is "Plan Sponsor Details" and another's Section 1 is "Eligibility," we know their numbering schemes don't align by index, and we should map by titles or keywords instead.

- **Combining Similarity Signals:** A robust filtering architecture might combine multiple signals: embedding cosine, keyword overlap, section context match, etc. Approaches like Reciprocal Rank Fusion (RRF) have been used in information retrieval to merge results from different systems [10]. For

example, one could get a candidate list from a dense vector search and another from a keyword-based search (BM25 or even a simple search for matching titles), then intersect or merge them. The cited research found that fusing BM25 with legal embeddings yielded a more balanced retrieval, leveraging both precise term matches and semantic similarity [10]. In our case, we might say: if two sections are truly semantically equivalent, they likely share some domain terms (even if phrased differently). If an embedding similarity is high **and** the section titles or keywords are closely related, that's a very strong candidate. If similarity is high but lexical overlap is nil, that candidate might be a false positive to drop or handle with extra scrutiny.

- **Thresholds and Top-K Tuning:** Currently top 3 candidates per source are passed along. Given the false negative risk (missing a true match among the top 3), consider expanding to top-K=5 or 10 to improve recall **if** the subsequent verification stage is reliable in culling the incorrect ones. The optimal K might be found by analysis on your validation set – e.g. if domain experts mapping reveals that the correct match is often ranked ~5th by raw cosine, increasing K will catch those. Of course, a higher K means more LLM calls, but even K=10 would be ~1820 pairs for Stage 2 (which might still be well under budget if using a smaller GPT model). **Dynamic thresholds** could also be useful: for each source, you might only take candidates above a certain cosine similarity score rather than a fixed count. If a source election is very unique, maybe only 1 or 2 clear high-similarity hits come up; for a very generic question, many moderate similarities appear. Setting a cosine cutoff (and perhaps a lexical check cutoff) can ensure you only pass plausible matches forward. This can boost precision by not even sending obviously dissimilar pairs to the LLM. One caveat: cosine similarity values can be model-dependent and might need calibration (e.g. what constitutes "high" similarity for your embedding model). Some experimentation would be needed here.

- **Contrastive Candidate Generation:** In line with the earlier point on fine-tuning embeddings, one could also employ a *learned similarity* approach directly for filtering. For instance, train a classifier that directly outputs "match" or "non-match" for a pair using a smaller model. If you accumulate a few hundred or thousand labeled examples from the mapping documents, a fine-tuned *Legal MiniLM* or similar model could serve as a fast filter that's more precise than raw cosine. This essentially moves part of the "verification" logic into the filtering stage but with a cheaper model than GPT-4. Additionally, advanced research suggests using **contrastive learning** to improve candidate generation: by training the embedding model with known matches, it inherently will score the correct pairs higher (reducing reliance on the LLM to fix errors). The *StructCoh* method mentioned earlier is one such advanced technique that uses contrastive objectives to align representations **with awareness of structure**, yielding far fewer false positives on legal text alignment [7]. While implementing a full dual-graph network may be beyond scope, a simpler contrastive fine-tuning on your data (e.g. using SBERT's multiple negatives ranking loss on the 37 known examples and augmenting with unlabeled pairs as negatives) could meaningfully improve the candidate retrieval stage.

In summary, **the filtering stage can be strengthened** by using smarter similarity measures (possibly learned or multi-factor) and by applying sanity-check rules that reflect domain knowledge (ensuring candidates share context or key terms). This should reduce the load on the LLM and decrease the chance that a nonsensical pair slips through with a high similarity.

LLM Verification and Hallucination Prevention

After filtering, the second stage uses an LLM to verify semantic equivalence of candidate pairs. This stage is crucial for precision – it should catch any subtle differences the embedding missed. However, as we observed, LLMs can sometimes hallucinate a connection or be overly optimistic (e.g. the GPT-5-Mini concluded "State" was a match for "Age" by inventing a rationale). Preventing such **hallucinations or misjudgments** is essential. Here are strategies to improve the LLM verification step:

- **Prompt Engineering for Focused Comparison:** The prompt given to the LLM should explicitly instruct it to **ignore irrelevant cues and focus on meaning**. For example, you might say: *"You are comparing two plan document questions. Ignore their numbering or formatting. Determine if they reflect the same underlying election or provision. Be truthful and don't infer connections that aren't stated."* By emphasizing that it should not use the question number or other superficial cues, the model is less likely to assume that "Question 1.04" in both means they match. Additionally, it can help to ask the LLM to **extract or summarize the core intent of each question** before deciding. A possible prompt template: *"Summarize the intent or requirement described by Source question, then do the same for Target question. Based on these summaries, state whether they represent the same election."* This forces the model to articulate, for instance, "Source is about the minimum age requirement for eligibility" vs. "Target is asking for the plan sponsor's state location." Seeing those side by side, it becomes clear they are unrelated. This approach is akin to a simple **chain-of-thought** prompt: by having the model explicitly reason through the content (in a structured way), we reduce the chance it will jump to a false conclusion [11]. Such chain-of-thought reasoning has been shown to improve accuracy and reduce errors by making the model's logic transparent [12] [11].

- **Few-Shot Examples (Including Negative Cases):** If the LLM supports few-shot prompting, provide a couple of examples of correct and incorrect matches with reasoning. For instance: *"Example 1: Source: 'Eligibility age _ **(not to exceed 21)'; Target: 'Entry age limit** (no more than 21)' → Match (both require specifying an eligibility age limit). Example 2: Source: 'Eligibility age _**'; **Target: 'State:** ' → Not a Match (one is about age requirement, the other about U.S. state of the employer)."* By demonstrating a tricky false-pair in the prompt, the model is put on guard against making that mistake. It sees the proper reasoning ("these are completely different concepts") and will be less likely to hallucinate a connection like "State refers to statement of age" because the prompt explicitly showed that's wrong.

- **Structured Output with Justification:** You are already using a JSON output with fields like `is_match`, `confidence_score`, and `reasoning`. We should make sure the **reasoning field is meaningfully used** – it's not just flowery explanation, but a checkable justification. One idea is to require the LLM to **cite specific words or phrases** from each question that ground its decision. For example: *"Source mentions 'age' and a numeric limit, Target is asking for 'State' (a location). These terms are unrelated; therefore, not a match."* If the model is compelled to reference the actual content (age vs state) in its reasoning, it's less likely to drift into unsupported claims. Essentially, this is a form of **self-verification** – the reasoning should contain evidence. In your false positive example, the model's reasoning was reportedly that "State refers to stating the age," which is clearly a hallucination. If we had a rule to post-check the reasoning (like does it mention a common concept or a known synonym pair?), we might catch that the rationale was bogus. You can implement a simple validation: if `is_match=true`, then certain key terms should appear in both the source and target or in the reasoning. If an LLM says "match" but its explanation does not clearly identify a

shared concept or uses an absurd leap (like conflating *state* with *age*), the system could flag or override that output as low confidence.

- **Chain-of-Verification:** Beyond prompting the model to explain, one can adopt a *Chain-of-Verification (CoVe)* approach [13]. This involves the model effectively questioning its own answer. For example, after an initial answer, have the model answer a follow-up like: *"What could be a difference between these two questions that would make them not a match? Check each key element."* In the age vs state case, a verification question could be, *"Does the term 'State' in the target refer to an age or to a location?"* The model would (hopefully) answer, *"It refers to a location (U.S. state)."* Then it becomes clear the match is invalid. CoVe prompts have been shown to reduce factual hallucinations by planning and answering such targeted questions before finalizing the response [13]. Implementing this might mean an extra step where the LLM, or even a secondary model, is asked to double-check any high-confidence match with targeted queries. While this adds some overhead, it could be applied selectively (e.g. only for pairs where the initial LLM confidence is high but embedding similarity was oddly high too, which indicates a potential mistake).

- **Use a Smaller Domain-Fine-Tuned Model for Verification:** GPT-4/5 class models are powerful but also prone to imaginative reasoning. For a narrow task like this (legal text similarity), a smaller model fine-tuned on labeled data might actually be more **consistent** (it won't inject external "knowledge" that isn't in the text). If you obtain a sizable ground truth set (say a few hundred or thousand labeled pairings from the upcoming mapping document), you could fine-tune a model like Llama-2 or Flan-T5 on a supervised binary classification: input is two pieces of text, output is yes/no if they match (and possibly a short justification). This model could then be used instead of the GPT-5-Mini in Stage 2. It would essentially replicate what a cross-encoder does – focusing only on the given text. The advantage is that it **won't hallucinate external links** because it's trained specifically to identify matches based on seen examples. It also can be faster and cheaper to run. For example, a Legal-BERT fine-tuned to classify clause alignment could likely process thousands of pairs quickly with >95% precision, given quality training data. Academic work has shown that fine-tuned models (like a Siamese SBERT or classifier) can significantly outperform zero-shot LLM judgments for text similarity in specialized domains [4]. Of course, training data is the key – but you do have an expert mapping incoming, which could provide that in spades.

- **Chain-of-Thought vs. JSON Direct:** You asked about structured outputs vs. free-form reasoning. Having a structured JSON schema is good for downstream consumption (easy to parse results), but it doesn't necessarily prevent LLM errors by itself. The hallucination issue is more about reasoning. You might consider a **two-pass approach**: first pass, the model does a chain-of-thought in free form (maybe hidden from the final answer using a technique like giving the CoT in a `<!-- -->` block if supported, or just internally in code), second pass, it outputs the JSON. OpenAI models don't allow hidden scratchpads in user-facing mode, but you can simulate it by calling the model to "think step by step" and then calling it again to produce the final JSON answer based on that reasoning. Another simpler tactic: instruct the model in the prompt that *before* outputting the JSON, it should silently think through differences. Some developers include something like "Let's think step by step" even in a single call – GPT might then internally do so. That said, ensuring the model does not include the chain in the JSON is tricky. An alternative is to have the model output a verbose reasoning (ensuring it cites specifics as mentioned) as part of the JSON (in the `reasoning` field), and then **require a justification** in that field that you programmatically verify for sanity. The structured format itself won't stop hallucination, but it allows you to systematically **analyze the response**. You could, for

instance, parse the JSON and check: if `is_match == true` but the `reasoning` does not contain any common keywords or contains a clearly incorrect statement, then either lower the confidence or flag it for manual review.

- **Avoiding Overconfidence:** The example had the model assign a 92% confidence to a wrong match. LLMs aren't calibrated probability estimators – they might say "92%" just because it sounded sufficiently sure. One way to handle this is to **post-calibrate confidence** using the validation set. Treat the model's true/false decisions and some features (embedding score, certain keywords, etc.) as inputs to a calibration model (like logistic regression) to derive a more reliable probability. Alternatively, simply do not expose or use the raw percent from the LLM. You could map the LLM's qualitative certainty (certain, likely, unsure, etc.) to a score based on how often those correlates with correctness in a dev set. If using a trained classifier for verification, you can calibrate its output probabilities via techniques like Platt scaling. In any case, for high precision, you might choose a **very conservative threshold** – e.g. only auto-accept matches above some confidence, and have anything borderline or low-confidence be either dropped or sent for manual confirmation. Calibrating to 95% precision might mean you intentionally sacrifice some recall (maybe you only return matches the model is very sure about, and handle the rest separately).

In essence, **the verification LLM should be turned into a diligent, detail-oriented reviewer** rather than a creative suggester. Through careful prompting (explicit comparison, chain-of-thought), possibly adding a self-check step (verification questions or reasoning validation), and maybe leveraging a fine-tuned model, we can significantly reduce the chance of hallucinating semantic equivalence where none exists. The end goal is that if two questions truly don't match, the system says "no match" every time – especially for glaring differences like "age" vs "state".

Alternative Architectures and Trade-offs

Beyond the two-stage embed-and-LLM pipeline, there are other approaches to consider. Each comes with trade-offs in development effort, runtime cost, and accuracy:

- **End-to-End LLM (Pure GPT Matching):** One theoretical approach is to skip embeddings entirely and use a GPT-like model to compare each source against all target sections in plain text. For example, you could feed a prompt: *"Here is the source provision: …; Here is a list of candidate target provisions: …; Identify which target (if any) matches the source's meaning."* This would rely on the LLM's ability to scan and compare a lot of text. In practice, doing this for 182×550 comparisons (or giving it all 550 targets at once, which is infeasible due to context length) is extremely expensive and slow (the user estimated ~$100 cost and ~5 hours for brute force GPT, likely accurate). It's not viable under the latency/cost constraints. However, it's worth noting that an LLM might catch nuances if it had full context – for instance, it could notice if multiple provisions in target share some similarity with the source and make a judgment. Some semi-optimized version could chunk the target and have GPT do a search (like a clever agent, see below). But overall, **pure LLM matching is cost-prohibitive** here. It could serve as a high-quality baseline on a small sample (e.g. ask GPT-4 to manually match 10 sources by reading the whole documents) to see what ideal performance might look like, but not for production.

- **Train a Dedicated Matching Model (Learned Classifier):** As discussed, if a sufficiently large labeled dataset of matches vs non-matches can be assembled, training a model is a promising route. This

could be framed as a classification or ranking problem: input two pieces of text, output whether they correspond. Modern transformer models fine-tuned on such binary tasks can achieve very high accuracy when enough domain examples are given. The advantages: once trained, the model can be run locally on all 100k pairs quickly (especially if distilled or quantized), making it efficient. It will also consistently apply the criteria it learned from experts (no open-ended creativity). Many academic works on legal text similarity point in this direction. For example, `conSultantBERT` fine-tuned on thousands of labeled pairs significantly outperformed unsupervised methods for matching job descriptions [4], and we'd expect analogous gains for legal provisions by fine-tuning on a legal dataset. The downside is **data** – you would need a comprehensive set of true alignment pairs and difficult negatives to train on. You mentioned an expert will provide a mapping; if that mapping covers most or all of the 182×550 potential pairs (i.e. it tells for each source which target matches, or that none match), then you effectively have labels for those comparisons. That could be leveraged for training (though careful splitting would be needed to test generalization). If expanding to cross-vendor comparisons beyond this one pair of documents, you'd want training data from multiple vendor documents to cover varied phrasing. In short, training a custom model is likely the most **scalable** solution if this mapping problem will be repeated many times, but it requires upfront investment in labeling and model training. Given the high precision requirement, a learned model is attractive because you can explicitly optimize for precision (e.g. use an asymmetric loss to penalize false positives more, or tune the decision threshold accordingly).

- **Hybrid Enhanced Pipeline:** The current approach is a hybrid (IR + LLM). We can enhance it rather than replace it. For instance, you could incorporate some **agent-like behavior** without going full autonomous agent. One idea: an LLM agent that can call a search on the target document. It might operate like: *for each source question, formulate a semantic query (maybe a rephrasing of the question), search the target document (via an internal search function or vector DB) to find likely sections, then verify.* This is somewhat similar to what you have, except the LLM would be generating the search queries or navigation commands based on understanding the source. Tools like LangChain or custom agents could be configured to use the text of the documents as their environment. The agent might do things like, "Source asks about eligibility age. Let me search the target for 'age' or 'eligibility'." This could uncover the right section even if embedding missed it, and also provide another layer of reasoning (the agent could confirm the context of the found section). However, building a reliable agent is complex and may not guarantee better precision; it could still hallucinate searches or require careful prompt constraints. Given that a well-tuned deterministic pipeline can probably do the job, an agent might be overkill. It's an area to consider if other methods fail – e.g. using GPT-4 with tools to dynamically find matches might catch tricky cases, but controlling cost and ensuring consistency would be challenging.

- **Heuristic-First, LLM-Second:** Another variant of hybrid is to use **cheap heuristics to aggressively filter** before embeddings. For example, partition the source and target questions by high-level category first (using keywords or regex). If a source question has keywords indicating it's about contributions, you only compare it to target questions that also look like contribution-related. This reduces the 100k comparisons even before vector embedding. You could maintain a dictionary of domain terms to categories (eligibility, vesting, loans, distributions, etc.). If this heuristic is high-recall (i.e. it doesn't mistakenly separate a true pair into different bins), it can save a lot of downstream work. However, if topics are ambiguous, it might drop some matches – use with caution or allow some overlap between bins just in case.

Trade-off Analysis: In choosing an approach, consider:

- *Accuracy vs. Simplicity:* A fully learned solution (fine-tuned model) might ultimately give the best accuracy, but the current two-stage approach with improvements may reach the target precision (95%+) with less development effort by leveraging existing APIs and a small amount of tuning data. On the other hand, relying on an LLM forever could incur ongoing costs; a custom model is more work upfront but could be cheaper per run later.

- *Cost:* Using GPT-4 or similar for hundreds or thousands of verifications will accumulate cost. If each call is say $0.02 and you do ~1000 of them, that's $20 per run, which is at the upper bound of your budget. If you can cut the calls to a smaller model (or use an on-prem model after initial dev), cost per run drops significantly. Fine-tuning smaller open-source models might have a one-time cost (and some hardware requirement), but then inference is essentially free.

- *Speed:* Similarly, a local model can be optimized for speed (batching predictions, using GPU). The current pipeline likely calls the LLM sequentially for each pair, which can be slow if the model has high latency. A batched classifier could process many pairs in parallel. That said, if using OpenAI/Anthropic APIs, you could send some requests concurrently up to their rate limits.

- *Maintainability:* A hand-engineered rules + embedding + LLM system has many moving parts that might need maintenance (e.g. if you add a new vendor with a radically different format, you may need to adjust preprocessing or rules). A learned model, if retrained on new data, can automatically adjust to new patterns. However, learned models can also be "black boxes," whereas a rule-based approach might be easier to debug for an expert (the expert might trust "I see this matched because both had 'age 21'" more than "the model's weights said so"). Providing explainability is important – which is why retaining the reasoning output or some form of trace is useful even if you go to a fully learned approach.

A likely recommendation is a **hybrid of improved components**: use improved embeddings + lexical cues for candidate generation, and use either a carefully prompted LLM or a fine-tuned smaller model for final verification. This balances precision and cost, and each component can be optimized and tested independently.

Evaluation and Validation Methodology

Designing a robust evaluation for this matching system is critical. You want to measure whether the system meets the >95% precision requirement and also gauge recall (so you know how much manual follow-up might be needed for missed matches). Here are some recommendations on validation:

- **Ground Truth Construction:** The "mapping document" from your domain expert (Lauren) will be invaluable. It presumably contains the correct matching between the two vendors' documents. As soon as that is available, use it to create a test set of source-target pairs labeled as *Match* or *No Match*. Ideally, this mapping is exhaustive – for each source election, it either identifies the equivalent target election(s) or states that there is no corresponding item (that can happen if one plan has a feature the other lacks). If possible, have a second expert review a subset to ensure the ground truth's accuracy (since even experts can disagree or err). If you only have mappings for some

of the data (say 50 out of 182 sources mapped), treat those as the evaluation set and maybe keep a few for tuning.

- **Metric Selection:** Precision and recall are the primary metrics. Precision = TP / (TP+FP) should be very high (95%+). Even one false positive in a small sample might drop precision below that, so you'll want as large a test set as possible to get a stable estimate. Recall = TP / (TP+FN) measures how many true matches were found. If the mapping document indicates, for example, that out of 182 source items, 170 have a corresponding target item and your system found 160 of those, recall ~94%. Depending on the use case, missing 6 might be acceptable if they can be handled manually, but missing none is ideal. Likely you want a balance, so F1 score can be reported too, but given the mandate, you might prioritize precision at the expense of recall (i.e. it's acceptable if some matches are not automatically identified, but any match output by the system should be trustworthy). Domain-specific measures could include something like a weighted score where false positives incur a larger penalty than false negatives. If you wanted to formalize that, you could assign a higher cost to FP and compute a cost-based metric. But for simplicity, ensuring a high precision is already capturing that concern.

- **Confidence Calibration and Thresholding:** Use the validation set to decide on any threshold for the final decision. For instance, if the verification stage provides a confidence score, analyze its distribution for correct vs incorrect predictions. It may turn out that almost all correct matches score above, say, 0.80 confidence, whereas the faulty match(es) were around 0.90 but perhaps had other warning signs. If the confidence is not reliable, you might instead derive a custom score (maybe a weighted combination of embedding similarity and some LLM score). In any case, set a cutoff that yields the desired precision on the validation set. For example, you might choose to only accept matches that the model is very certain about, and label anything else as "Needs Review" rather than auto-matching. This way, the system output can be used with minimal risk.

- **Iterative Testing:** Once you implement the improvements, run through the 37 hand-labeled examples you already have. Pay attention to *why* any mistakes happened: Was it a filtering miss (not in top-K)? Was it an LLM misjudgment? Use those insights to refine the prompts or rules. For instance, if you find another hallucination pattern, add an example of it to the prompt or adjust the instructions. If some true match was missed because it wasn't top-3 in cosine, that argues for increasing K or improving embeddings.

- **Cross-Document Scaling:** If you plan to generalize this to other vendor documents, you should test on at least one additional document pair if possible. Sometimes, a system tuned between Relius and Ascensus might not directly transfer to, say, Ascensus vs. SomeOtherProvider because their document formats differ. If no data from other vendors is available now, at least be aware of where assumptions might break. For example, if another provider's doc uses completely different terminology for the same concepts, the embedding model might need retraining or expansion of its vocabulary. Incorporating diverse examples when fine-tuning (if you go that route) will help robustness.

- **Explainability and User Validation:** Since false positives are critical to avoid, consider adding a mechanism for **manual validation of low-confidence or certain matches**. For instance, if the system outputs a mapping with a confidence score, you could have an expert quickly review those above, say, 90% and those between 80–90% separately. Perhaps the system auto-accepts >90%, and

an expert double-checks anything in 80–90% range, and discards <80%. This workflow can still save time (the expert focuses only on questionable cases). Over time, as the system improves, the threshold can be relaxed. Also, by logging the system's reasoning and which features were key to the match, you provide transparency – this builds trust and helps debug mismatches. If an expert sees the reasoning for a match and disagrees, that's feedback to incorporate (maybe a new rule or training example).

- **Continuous Learning:** Every time the system does a crosswalk and a domain expert corrects something (either a false positive or a missed match), feed that back as a new training example. This can be done manually or potentially automated if the expert feedback can be captured in a structured way. Over a number of documents, you will accumulate a solid dataset that could be used to retrain/improve the model, slowly reducing the error rates further.

In summary, validate rigorously with expert-approved mappings and use those results to fine-tune the thresholds and components of the system. The evaluation should demonstrate that you meet the success criteria: **very high precision** (no critical false matches in test data), within acceptable time/cost, and with outputs that are explainable (via the reasoning field or similar). If not initially met, the above strategies in embedding, filtering, and LLM prompting should be iteratively adjusted until the performance is satisfactory.

Recommendations and Next Steps

Literature & Prior Work: The challenges you're facing intersect information retrieval, semantic textual similarity, and legal AI. We discussed some papers – e.g., legal-specific retrieval enhancements [1] [10], contrastive legal text matching approaches [6] [7], and domain-adapted language models [2] – which all suggest that incorporating domain knowledge (either through model training or structural understanding) is key to success. Recent advances like *StructCoh* show the frontier of aligning documents by meaning, using graphs and contrastive learning to capture context. While you may not implement a graph neural network, being aware of these techniques can inspire your solution (for instance, ensuring dependency/context differences are considered in matching, as the image above illustrated with the "written justification" example).

Embedding Strategy: Start by upgrading your embedding approach. If using OpenAI embeddings, consider moving from the older `text-embedding-ada-002` small model to a larger model or a newer one if available, or supplement it with a legal-specific model. You might try **Legal-BERT SentenceTransformer models** (available on Hugging Face) to see if they produce more sensible similarity rankings. If numbers in text are an issue, implement a preprocessing step to strip section numbers and filler underscores. Experiment with concatenating fields (e.g. "Section Title: X. Question: Y. Options: Z") vs. separate field embeddings combined – the Elastic search labs article [5] shows one way to weight fields; you might mimic that by trial and error to see what yields better retrieval of known matches. In short, **represent each election in a way that maximizes semantic info and minimizes noise**.

Filtering Improvements: Introduce a secondary filtering or re-ranking step using a smaller model. This could be the same model used for embeddings but in a cross-encoder fashion (for example, `sentence-transformers` has a cross-encoder for sentence pairs that can be fine-tuned). Or use GPT-3.5 (cheaper than GPT-4/5) in a classification mode on the top candidates. Also add the easy precision wins: if two sections have zero overlap in important words, consider dropping them. If you have a taxonomy of plan

features, use it to only compare likely related sections. These heuristic filters can be implemented quickly (even as a simple if/else in code) and could save your LLM from considering obviously bad pairs.

LLM Verification Prompt: Redesign the prompt given to GPT-5-Mini/Claude. Make it structured: e.g., *"Source text: … Target text: … Task: Do these refer to the same plan provision? Answer with yes or no. Explain your reasoning by identifying the key topic of each and comparing."* Then give a couple examples as discussed. Emphasize not to infer things not present. Once you have this prompt, test it on a few of the tricky cases (like the known false positive) interactively and see if it behaves correctly. Tweak until the model says "No, these are unrelated" for that case and still "Yes" for a clearly matching pair. Also test it on a pair that are similar but not exact to see if it correctly identifies them as match (for recall).

Confidence and Output Handling: Consider ignoring the model's numeric confidence in favor of your own calibration. Or if using Claude (which might not give a percentage), define confidence qualitatively. It might be useful for the model to categorize *why* it's a match or not – e.g., "variance classification" as you had. Perhaps you can refine those variance categories (like "Exact match", "Same concept with different wording", "Related concept but one has extra condition", "Unrelated"). If the model can classify the type of relationship, that could help an expert quickly review borderline cases. For example, if it says "Related concept but phrased differently", the expert might double-check those carefully, whereas "Exact match" they might trust. Ensure the reasoning text is preserved for users or debugging; it's a form of explainability.

Testing and Iteration: After implementing changes, run the full pipeline on the known 37 examples and the rest once you have the mapping. Measure precision/recall. If precision is below target because some false positives still got through, inspect those cases in detail – did the embedding stage produce a bad candidate or did the LLM incorrectly confirm it? If the latter, you may need to further tweak the prompt or possibly switch to a different model for that pair (sometimes ensembles help – if GPT says match, maybe ask Claude to double-check; if they disagree, flag it). If recall is low (missing matches), see if they were missing from the candidate list (then embedding needs improvement or K increased) or if the LLM mistakenly said no (perhaps it was confused by phrasing – maybe add that scenario as a few-shot example or fine-tune on it).

Conclusion: The state-of-the-art solution will likely be a **multi-faceted hybrid**: a smart embedding or search component to gather semantically likely pairs (leveraging legal domain embeddings and possibly fine-tuning), followed by a careful verification step that employs either a robust prompt-engineered LLM or a specialized trained model to ensure matches are genuine. This approach mirrors what top-performing systems in legal AI research do: first *recall* broad candidates with dense vectors, then *precisely rank* or classify using more intensive context-aware models [9]. By applying the strategies above, you should be able to significantly reduce false positives – aiming for that >95% precision – while keeping within reasonable cost and time budgets. The key is **semantics first, structure second**: encode and compare the meaning in multiple ways, and make the AI justify its reasoning so that errors can be detected and corrected. With iterative refinement and the incorporation of expert knowledge (both in rules and training data), the system will become increasingly accurate and trustworthy for mapping retirement plan provisions across different vendors' documents.

Sources:

- Chalkidis et al., *Legal-BERT: Pretrained Transformers for Legal NLP*, 2020 – demonstrated improved performance on legal tasks by further pretraining BERT on legal texts [2].

- Lavi et al., *conSultantBERT: Fine-tuned SBERT for Job Matching*, RecSys HR Workshop 2021 – showed domain-specific fine-tuning of embeddings outperforms generic embeddings for matching tasks [4].
- Xue & Gao, *StructCoh: Structured Contrastive Learning for Context-Aware Text Semantic Matching*, arXiv 2025 – introduced a graph-based contrastive approach that improved clause matching in legal documents, emphasizing the importance of structural context [6] [7].
- Pasqual et al., *Optimizing Legal Information Retrieval through Embeddings and Reranking*, 2024 – reported that a legal-specific embedding model (Voyage Law) plus a hybrid BM25+embedding retrieval and GPT-4 reranker gave higher precision in legal document search [14] [9].
- Elasticsearch Labs, *Advanced RAG: Multi-Field Embeddings*, 2023 – described creating composite embeddings from multiple fields (title, content, metadata) with weighted importance for better semantic search [5].
- OpenAI, *Best Practices for Prompt Engineering & Reducing Hallucinations*, 2023 – various guides (incl. chain-of-thought prompting) note that forcing step-by-step reasoning and verification can curb LLM hallucinations [12] [11].
- *Internal Testing on Plan Documents* – your observed example of false match (Age vs State) highlights the need for these enhancements; additional internal validation with 37 examples was used to refine the approach (contextual reference).

---

[1] [8] [9] [10] [14] kth.diva-portal.org
https://kth.diva-portal.org/smash/get/diva2:1963310/FULLTEXT01.pdf

[2] [3] web.stanford.edu
https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/final-reports/final-report-169451673.pdf

[4] [2109.06501] conSultantBERT: Fine-tuned Siamese Sentence-BERT for Matching Jobs and Job Seekers
https://arxiv.org/abs/2109.06501

[5] Advanced RAG techniques part 1: Data processing - Elasticsearch Labs
https://www.elastic.co/search-labs/blog/advanced-rag-techniques-part-1

[6] [7] StructCoh: Structured Contrastive Learning for Context-Aware Text Semantic Matching
https://chatpaper.com/paper/185220

[11] [12] Supervised Chain-Of-Thought Reasoning Mitigates LLM Hallucination | by Cobus Greyling | Medium
https://cobusgreyling.medium.com/supervised-chain-of-thought-reasoning-mitigates-llm-hallucination-41e566240b05

[13] Chain-of-Verification (CoVe): Reduce LLM Hallucinations
https://learnprompting.org/docs/advanced/self_criticism/chain_of_verification?srsltid=AfmBOorxkeROxhJm6Jhm0GQQk4WAv_w-ZTkIyZeC4IBwCYlgCI2jd92j