ChatGPT

Intelligent GenAI Options for Loan Document Comparison

Introduction

Comparing loan advisor documents can be tedious and error-prone for humans. AI-powered text analysis can dramatically speed up this process and catch subtle differences in wording or terms that manual review might miss [1]. For example, AI systems can detect not just explicit changes in interest rate numbers, but also nuanced shifts in tone or context between two contract clauses [1]. By introducing **Generative AI (GenAI)** into a web app, users can upload two loan documents and have a GPT-based chatbot intelligently highlight differences in interest rates, repayment terms, penalties, and more.

However, deploying GenAI for this task requires careful choice of platforms and techniques. Large Language Models (LLMs) sometimes generate *hallucinations* (fabricated outputs), which is especially risky in legal/financial contexts – studies show hallucinations in about **1 of 6 answers** on legal questions [2]. Therefore, we must weigh each option's accuracy, limitations, and implementation effort.

This report provides a comprehensive comparison of leading GenAI platforms and techniques for document comparison. We cover:

- **GenAI Platforms**: OpenAI's GPT-4, Anthropic's Claude, Google's Gemini (PaLM family), Meta's LLaMA models, Mistral AI, and other open-source LLMs.
- **Techniques**: Prompt engineering, retrieval-augmented generation (RAG), agent-based reasoning, embeddings for semantic search, and fine-tuning models on custom data.
- For each option, we examine the **developer effort** to integrate it, **infrastructure/resources** needed, key **limitations** (e.g. latency, context length, cost, hallucination risk), and **suitability** for structured document comparison (like loans).

Finally, we include a summary table comparing all methods across these dimensions, and we provide both a detailed PDF-style report and a PowerPoint-style outline for developers.

GenAI Platform Options for Document Comparison

OpenAI GPT-4 (Closed API Platform)

OpenAI's **GPT-4** is a state-of-the-art LLM known for its strong reasoning and language understanding capabilities. It would likely excel at comparing complex loan documents due to its high accuracy and ability to follow instructions. **Integration** is straightforward via OpenAI's cloud API – developers send the document text in prompts and receive the comparison analysis in return. There is no need to host the model or manage ML infrastructure, which keeps developer effort low. Key characteristics of GPT-4 for this use case include:

- **Quality and Reliability:** GPT-4 generally produces very coherent and detailed outputs, with fewer mistakes than earlier models. It is adept at understanding financial terms and performing comparisons if properly prompted. That said, it can still misinterpret or hallucinate details occasionally, so prompts must explicitly direct it to use only given document content.
- **Context Length:** Standard GPT-4 models support up to ~8,000 tokens context, and a 32,000-token variant is available for longer inputs [3]. This context length might be a limiting factor if both loan documents are very long. However, OpenAI has introduced new versions (GPT-4 Turbo) with context windows up to **128,000 tokens** [4], which can easily accommodate most document pairs. Larger context models come with higher costs but allow you to feed entire documents in one go.
- **Cost:** GPT-4 is a paid service and one of the more expensive models to use. Pricing (as of 2024) is roughly $0.03 per 1K input tokens and $0.06 per 1K output tokens for the 8k model, and about $0.12 per 1K for the 32k model [3]. The new 128k context GPT-4 Turbo is cheaper per-token (about $0.01 per 1K input tokens) [4], but of course if you send huge documents the total tokens (and cost) add up. Developers must budget for these API costs, especially if many comparisons will be done.
- **Latency:** GPT-4's advanced reasoning comes at the cost of speed. It is generally slower to respond than smaller models – a complex prompt might take several seconds to tens of seconds to generate a response. This could impact user experience if the comparison is requested interactively. In practice, GPT-4 is often the **slowest** of the major models for a given prompt length, so optimizing prompt size (or using the faster "GPT-4 mini/nano" models if available) might be necessary for real-time apps.
- **Usage Constraints:** As a cloud API, using GPT-4 means document data will be sent to OpenAI's servers. OpenAI does offer data privacy assurances (data not used for training by default), but organizations with strict data security requirements might be cautious about sending sensitive financial documents to an external service. There may also be rate limits on API calls that require scaling considerations if many users use the feature simultaneously.
- **Suitability:** GPT-4 is highly suitable for structured document comparison **if paired with proper prompt design or retrieval**. Out-of-the-box, GPT-4 can compare two texts and articulate differences well, thanks to its strong comprehension. For example, with a prompt like *"Here are two loan documents (Document A and Document B). Analyze and list the differences in interest rate, repayment term, fees, and penalties."*, GPT-4 can produce a detailed comparison. It will excel at understanding rephrased terms or implicit differences. The main concerns are ensuring it doesn't fill in any details that aren't actually in the documents (hence one might ask it to quote or reference the source text for verification). Using GPT-4 in combination with retrieval (providing the relevant sections of the documents in the prompt) can further ground its answers. Despite the higher cost and slight latency, GPT-4's accuracy and reasoning make it a top choice for this domain, especially for a first implementation where quality is paramount.

Anthropic Claude 2 (Closed API Platform)

Claude 2 by Anthropic is another powerful LLM accessible via API, distinguished by its extremely large context window. Claude has been designed with an emphasis on helpfulness and safety, making it less likely to produce harmful content. For our document comparison use case, Claude's ability to ingest very long inputs is a major advantage:

- **Context Length:** Claude 2 supports up to **100,000 tokens** in its context [5], equating to around 75,000 words of text. This means you can feed entire lengthy documents (hundreds of pages) from both advisors directly into one prompt. For instance, Anthropic demonstrated loading an entire novel (~72K tokens) into Claude and it could answer questions requiring synthesis across the book [6]. In our scenario, you could concatenate Document A and Document B (perhaps prefixed with identifiers) into a single prompt and ask Claude to compare them – no need for complex chunking or retrieval logic if they fit in 100K tokens. Claude can *"digest and analyze"* that much text *"in less than a minute"* [6], which is impractical for smaller-context models. This massive context capability can simplify development: you don't have to pre-select relevant sections; Claude can consider all the content at once.
- **Accuracy and Behavior:** Claude is quite capable at summarization and QA tasks. It tends to be conversational and oriented to follow the *"Constitutional AI"* guidelines Anthropic set (aiming to be truthful and harmless). In practice, Claude's quality on many tasks is comparable to OpenAI's GPT-3.5 and approaching GPT-4. It may sometimes be slightly less precise or detailed than GPT-4 on tricky problems, but it handles straightforward comparison questions well. Importantly, having the full documents in context means Claude is *less likely to hallucinate facts*, since it can directly quote or refer to the given text. Anthropic even notes that for complex cross-referenced questions, using a large context (like Claude's) can work *"substantially better than vector search based approaches"* because the model sees all relevant info together [7]. This suggests that Claude can directly synthesize information across different sections of the documents without needing an external retrieval step, as long as everything is in the prompt.
- **Integration and Effort:** Using Claude is similar to GPT-4 – it's an API call (Anthropic provides a cloud endpoint for Claude). Developer effort is minimal in terms of ML ops; you send prompts and receive responses. To leverage the 100K context fully, you may need to format your prompt carefully (perhaps as *"Document A text... Document B text... Now compare XYZ..."*). Also, very large inputs will require streaming or careful handling in code due to their size (and the API may have some throughput limits on large payloads). But overall integration is easy, and you avoid building a separate retrieval pipeline if you rely on Claude's context window.
- **Cost:** Claude's pricing is competitive given its context size. For example, input tokens for Claude Instant (a faster, slightly less accurate version) have been on the order of ~$1.60 per million tokens for 100K context, and ~$5.51 per million for outputs, according to some sources [8] [9]. (These costs might have evolved; Anthropic's latest pricing can be obtained from their documentation.) To illustrate, one experiment showed that querying Claude with a ~170K-token input (like two large documents) cost only around **$1.5–$2** and took about 150 seconds to process [10]. So while Claude allows huge inputs, the cost per query will scale with token count – using the full 100K tokens might cost a few dollars each time. This is something to consider if the app will compare very large documents frequently.
- **Latency:** Claude can handle huge inputs in under a minute [6], which is impressive, but still slower than small context queries. In the Great Gatsby example, Claude responded in **22 seconds** for a 72K-token input plus analysis [6]. In our use case, if we feed two documents totaling, say, 20K tokens, the response might come back in just several seconds. Overall, Claude's speed is reasonable; it may even outperform GPT-4 on long inputs since it's optimized for that scenario. For interactive use, you might implement an async or streaming response to keep the UI responsive during analysis of very long texts.

- **Limitations:** One limitation is that **Anthropic's model might sometimes be overly verbose or overly confident**. Like any LLM, it can still misstate something if instructions aren't clear. Also, handling 100K tokens requires careful prompt construction to keep the model focused (it could get distracted by irrelevant parts of such a large input). Another practical limitation is access: Claude 2 (especially the 100K version) may not be openly available to all developers without a waitlist or through certain platforms (it's offered on services like Slack's GPT and AWS Bedrock). So obtaining API access and quotas for large context might require coordination with Anthropic.
- **Suitability:** Claude 2 is **highly suitable** for structured document comparisons, especially when documents are lengthy. You could literally drop both full documents into the prompt and ask: *"Compare Document A and Document B in terms of interest rates, repayment period, prepayment penalties, and any other financial terms. Provide any differences in these sections."* Claude can then read both entirely and produce a comprehensive answer. This one-step approach minimizes the risk of missing context that might occur with a narrower context model + retrieval. It's an elegant solution if you can afford the token cost and slightly longer wait for answers. For developers, using Claude might eliminate the need for building a separate retrieval mechanism (since the model itself can ingest everything). Thus, Claude offers a **trade-off**: higher input costs in exchange for simplicity and comprehensiveness of analysis. In summary, for comparing *full* documents and capturing subtle differences, Claude with its 100K context is one of the best options.

Google Gemini (PaLM Family on Vertex AI)

Google's **Gemini** (part of the PaLM 2 family and its successors) is an advanced GenAI platform that by 2025 has become a strong competitor to OpenAI and Anthropic. Google's models can be accessed via **Google Cloud's Vertex AI** or the AI Studio, and are integrated into Google's ecosystem. For loan document comparison, Gemini/PaLM offers some unique strengths:

- **Ultra-Long Context:** Google's latest models boast the **longest context windows** in the industry. Gemini 1.5 introduced a standard **128,000-token** context window for its Pro model [11], and even demonstrated processing **1 million tokens** in a private preview [11]. This is an enormous capacity – 128k tokens is roughly 100k words (like an entire book or several documents), and 1M tokens approaches an entire library of documents. In practical terms, this means a Gemini-powered API could accept not just two documents, but potentially dozens of documents at once. For our use, you can easily fit two loan docs (even very large ones) into the prompt with room to spare. This similar advantage to Claude's 100k context means less need for external retrieval; the model itself can be given all relevant text explicitly.
- **Performance:** Google's PaLM 2 (and likely Gemini models) are on par with other top-tier LLMs. PaLM 2 was known for strong multilingual and reasoning capabilities, and Gemini has been described by Google as *"our most intelligent model yet"* with improvements in reasoning and coding ability [12]. By early 2024, Google reported Gemini matching or exceeding GPT-4 levels on certain benchmarks as it evolved (e.g., Gemini 1.5 achieved quality comparable to their previous top model, and Gemini 2.x showed further gains). While specific benchmark data isn't public in this document, we can assume Gemini can handle complex comparison tasks effectively. Its answers should be accurate and detailed as long as the prompt is well-structured.
- **Integration:** Using Gemini typically involves the **Vertex AI** platform on Google Cloud. Developers need to have a Google Cloud project and enable the AI services. Integration steps include selecting the model version (e.g., "gemini-2.5-pro" etc.), and then making API calls similar to other providers (Google provides SDKs and REST endpoints). The initial setup is slightly heavier than OpenAI/ Anthropic because of the GCP configuration (you might need to deploy a model endpoint or use the

"chat" API within Vertex). Once set up, the usage from an app is straightforward. Google's platform also allows fine-tuning and model customization through their **Model Garden**, so developers could fine-tune a PaLM model on domain-specific data if needed (subject to availability of that feature for specific model sizes).

- **Infrastructure and Resources:** By leveraging Google's cloud, you avoid hosting any model yourself. However, costs can accrue for using these models. Google's pricing for large context models is tiered; for instance, **Gemini 1.5 Pro** with 128k context might have a certain price per thousand tokens, and if one opts into the experimental 1M token context, there could be premium pricing [13]. Google indicated plans for pricing tiers where 128k is base and higher contexts cost more [13]. Also, such large contexts can result in high memory usage and slower evaluation – but Google's infrastructure presumably handles it behind the scenes. For developers, one consideration is that using Google's LLMs often ties you into Google's cloud ecosystem (which might be fine if you are already on it, but could be an overhead if not).
- **Capabilities for Document AI:** Google has done extensive work in **document AI** (e.g., Google's DocAI and AI for Google Workspace). While those are separate from Gemini, it means Google's ecosystem might offer complementary tools: for example, one could use Google's Document AI OCR to parse PDFs, then feed text into Gemini for comparison. In fact, Google's models are increasingly multi-modal; Gemini is expected to handle not just text but also incorporate vision (the mention of "Gemini Live" with camera input [14] suggests it's moving toward image understanding). This could be relevant if, say, the loan docs are scanned PDFs – a future Gemini might take PDFs directly. For now, assuming text input, Gemini's strength is simply dealing with *a lot of text at once*.
- **Limitations:** One limitation is **availability** – in 2024–2025, Google's most advanced models like Gemini 2.5 were in **Preview** for certain customers [15]. This means not everyone can immediately use the highest-end model. You might get access to PaLM 2 or an earlier Gemini version with slightly smaller context if you're not in the preview program. Another challenge is **cost and speed**: pushing 100k+ tokens through the model will be slow (possibly tens of seconds or more) and expensive. If the use case only occasionally needs such capacity, that's fine, but for frequent interactive queries you'd likely still use RAG to reduce input size. Also, debugging and iterating on prompts can be cumbersome when each prompt is huge – it's often easier to test on smaller content or use RAG to limit what the model sees.
- **Suitability:** Google's platform is **very well-suited** for structured document comparison, especially if you are already leveraging Google Cloud in your app. The ability to stuff both documents in the prompt and just ask for differences is similar to Claude's approach. For example, you could send: *"Document A: [full text]; Document B: [full text]; Identify differences in interest rate, term, fees, and penalties."* Gemini can then utilize its massive context to directly compare side by side. The output can be expected to be thorough. Additionally, Google's model could potentially be fine-tuned or instructed to output in a structured format (like a bullet list or a table of differences) easily. The tight integration with other Google AI tools (like their embedding services or vector search in Vertex AI) means you could also combine approaches: use embeddings to find relevant parts and then feed those to Gemini for an even faster result. In summary, if accessible, Google's Gemini provides a cutting-edge option that, like Claude, minimizes the need for complex architecture at the expense of relying on a single very powerful (and possibly expensive) model call.

Meta LLaMA (Open-Source LLMs by Meta)

LLaMA refers to Meta's family of open-source large language models, which have become prominent alternatives to proprietary models. Starting with LLaMA 1 (released in 2023 to researchers) and then **LLaMA 2** (released openly in mid-2023), Meta has continued to improve these models (as of 2025, **LLaMA 3** has been announced with significant leaps in capability [16]). Using LLaMA models for our use case means opting for an **open-source, self-hosted solution** or using third-party services that host them. Key points for LLaMA-based options:

- **Performance:** LLaMA 2 was notable for achieving high performance with relatively fewer parameters. The largest version, LLaMA-2 **70B**, is roughly on par with OpenAI's GPT-3.5 in many benchmarks, and not too far from GPT-4 on some tasks. In fact, LLaMA-2 70B Chat (the instruction-tuned variant) slightly **outperformed ChatGPT (GPT-3.5)** in some head-to-head evaluations [17] . By the time of LLaMA 3, Meta claims the new 70B model sets a *"new state-of-the-art"* for open models, with major improvements in reasoning, coding, and following instructions [16] . This means an open model can now approach the quality of top proprietary models, especially after fine-tuning. However, out-of-the-box LLaMA might be less specialized in financial/legal language, so it could need prompt tuning or fine-tuning to really shine in document comparison.
- **Models & Sizes:** The LLaMA family comes in different sizes (e.g., 7B, 13B, 70B parameters for LLaMA 2). Smaller models (7B/13B) are much faster and require less hardware, but they have more limited understanding. A 7B model might miss subtleties in a complex contract or even misread some context if not explicitly guided. The 70B model has far better comprehension and is likely the choice if using LLaMA for serious analysis – Meta's research indicated LLaMA-2 70B could *"match the performance of smaller models with less training compute"* and had strong results on reasoning tasks [18] [16] . There is also a new 34B LLaMA 3 model mentioned (noting LLaMA 1 had a 65B, 33B variant; LLaMA 2 had no 33B but 70B; presumably LLaMA 3 might have 70B as well). For our needs, we'd likely consider the 70B class to ensure the model can handle the complexity of loan documents.
- **Context Length:** One limitation historically with LLaMA models was a smaller context window (around 4K tokens for LLaMA-2). Some community efforts have extended this via fine-tuning or employing techniques like **RoPE scaling**, but reliably, LLaMA won't match Claude or Gemini's context. This means if the documents are large, a naive approach of putting both into one prompt won't work – you'd need to employ **retrieval or chunking** with LLaMA. (However, as open models evolve, we see techniques like **Grouped-Query Attention** which Mistral used to extend context efficiently [19] . It's possible newer LLaMA-based models have >4K context by 2025 through such improvements, but likely not anywhere near 100K without sacrificing quality.) For now, plan to use RAG or split the task for long documents when using LLaMA.
- **Integration & Infrastructure:** Using LLaMA requires **hosting the model** yourself or via a service. This is the biggest difference from the closed APIs. Hosting a 70B model for inference typically needs one or two high-end GPUs. For example, running LLaMA-2 70B in 16-bit precision requires ~80GB of GPU memory, but with quantization (like 4-bit GPTQ) it can run in about 20GB with some speed trade-off. Many developers run 70B on a single 48GB GPU with 4-bit quantization, or on two 24GB GPUs sharded. The infrastructure could be on-premises (if you have a machine with those GPUs) or on cloud VMs (e.g., an AWS EC2 with an NVIDIA A100 or H100). This is a non-trivial effort: you must set up the model weights, the inference framework (PyTorch or specialized inference libraries like **vLLM or FasterTransformer** for efficiency), and ensure the app can communicate with this service (often via a REST API you host, or using something like HuggingFace's text-generation-inference server). There are also managed solutions: some providers or startups host LLaMA and let you call it via API (for a fee), combining the best of both worlds (control + no infra).
- **Cost:** While the model weights are free (LLaMA 2 is under a permissive license for commercial use, and LLaMA 3 presumably as well), the cost comes from infrastructure. Running a 70B model 24/7 on cloud GPUs can cost hundreds to thousands of dollars per month depending on usage. However, if your usage is lower volume, you could run it on-demand or scale the server up/down. Compared to

paying per token for an API, an owned model could be cheaper at scale. For example, one could fine-tune and run LLaMA-2 7B or 13B on a standard cloud instance without GPU (using CPU with quantization, albeit slowly), making it virtually cost-free to use aside from hardware rental. The trade-off is latency and quality. **In summary, open-source models shift cost from per-call pricing to fixed infrastructure costs and developer time.** This can be worthwhile if you need to avoid external API costs or privacy concerns.

- **Fine-Tuning Capability:** A big advantage of open models is you can **fine-tune** them on your data. With Meta's models, techniques like Low-Rank Adaptation (LoRA) have made fine-tuning feasible even on a single GPU for larger models. If you have a dataset of loan document comparisons or want the model to adopt a certain style (e.g., always output a structured difference report), you can train LLaMA to do that. Meta's LLaMA-2 chat models were themselves fine-tuned on instruction data, and building on that with domain-specific instructions can yield excellent results. Fine-tuning could make an open model *as good as or better than GPT-4 on your specific task*, at least one Meta exec claims LLaMA 3 models are *"a major leap"* in capabilities [16] partly due to advanced fine-tuning methods. We will discuss fine-tuning more in the Techniques section, but it's worth noting here as a platform strength: you have **full control** over the model's training when using LLaMA.
- **Suitability:** LLaMA-based solutions are suitable if you desire **maximum control and privacy**. For a loan comparison app inside a bank's secure environment, using an open model means no data leaves your servers. LLaMA 2 or 3 at 70B can certainly parse legal text and find differences, but you will almost certainly need to implement **Retrieval Augmented Generation (RAG)** to feed it relevant snippets due to context limits. A likely architecture is: embed the documents, use similarity search to find matching sections (interest rate section from Doc1 and Doc2, etc.), then prompt LLaMA with those sections to comment on differences. The quality of the output might require some prompt engineering, and possibly additional fine-tuning if the raw model's legal understanding is lacking. LLaMA might not know certain financial domain nuances out-of-the-box (its training data is broad but not specifically focused on finance contracts). This can be mitigated by providing sufficient context or fine-tuning on financial text. In terms of **developer effort**, going with LLaMA is the more **labor-intensive route**: you'll be managing model endpoints, vector databases, etc., rather than calling a single API. But for many developers (especially those with ML background), the flexibility and cost benefits at scale are worth it. In summary, Meta's LLMs give you a path to a **self-hosted AI agent** that can be optimized for your loan comparison task, with the caveat that you must build and maintain the supporting infrastructure.

Mistral AI (Open-Source 7B Model and Beyond)

Mistral 7B is an example of a new wave of efficient open-source models. Released by Mistral AI in late 2023, this model is only 7.3 billion parameters yet it achieved performance levels previously seen in much larger models. The team introduced training improvements that made Mistral 7B **outperform LLaMA-2 13B** on all evaluated benchmarks and even match or surpass some 30B+ models [20]. For our purposes, Mistral represents the possibility of using a *much smaller, faster model* without giving up too much understanding:

- **Performance vs Size:** Mistral 7B's creators report it *"significantly outperforms Llama 2 13B on all metrics, and is on par with Llama 34B"* in their internal evaluations [21]. That's remarkable – it means a 7B model performing like a model 4-5 times larger. They also provided a chat-tuned version that *"outperforms Llama 2 13B chat"* [22]. In practice, Mistral 7B can handle a variety of tasks including reasoning to an extent, although it will still be weaker than a 70B model on very complex instructions. For document comparison, a well-tuned 7B might identify straightforward differences (like numeric differences in interest rates, or added/removed clauses) quite well. It might struggle more with subtle or implicit differences unless guided carefully.

- **Efficiency and Speed:** The advantage of a 7B model is that it's **lightweight**. Mistral uses **Grouped Query Attention and Sliding Window** strategies to allow somewhat longer input sequences and faster inference [19]. It can run on a single consumer-grade GPU (or even on CPU with optimized libraries) and still return answers quickly. The small size yields **low latency** – this can enable real-time interactive comparisons without heavy hardware. For example, running Mistral 7B on a laptop with 16GB GPU RAM is feasible, and responses might come in under a second for moderate-length prompts. This makes it attractive for embedding directly in a web app environment or edge device.
- **Integration:** Like LLaMA, Mistral is open and requires self-hosting. But due to its small size, integration is easier. You might use libraries like Hugging Face Transformers or the **Ollama** server which packaged Mistral for easy local inference. Mistral's license (Apache 2.0) is very permissive [23], so it's straightforward to use commercially. One can fine-tune Mistral on custom data with modest compute as well – fine-tuning 7B might even be done on a powerful single GPU or smaller cloud instances. The barrier to entry for experimenting with Mistral is low; a developer could get it running locally and iterate on prompts quickly.
- **Limitations:** The main limitation is **raw capability**. Despite its efficiency, a 7B model is still fundamentally less knowledgeable and less fluent than a GPT-4 or even LLaMA-70B. If the loan documents are complex or the differences are subtle (e.g., a slight change in phrasing that alters legal meaning), a small model might miss it or misinterpret it. There's also the issue of context – Mistral 7B by default might support around 4k tokens context, similar to LLaMA. So RAG is needed for longer docs. Additionally, small models are more prone to hallucination when unsure. They might "guess" an answer if they don't fully understand the context. One mitigation is to constrain their output or use a tool-assisted approach (for example, have the model output which sections it thinks differ, then verify those with another pass). But that adds complexity.
- **Suitability:** Mistral or similar small open models could be suitable if the goal is to **deploy an AI assistant in resource-constrained environments** or if cost must be kept minimal. For instance, if this comparison feature is offered for free to many users, running a cluster of GPT-4 or Claude calls might be cost-prohibitive, whereas a few servers with Mistral 7B could handle it at fixed cost. Also, if the documents being compared are relatively formulaic (many loan documents follow standard templates), a smaller model fine-tuned on that domain might perform surprisingly well because the patterns are easier to learn. Mistral could be fine-tuned on a dataset of contract pairs and differences so that it inherently knows where to look for differences. This fine-tuned mini-model could then instantly output a structured difference summary without needing massive processing. In short, Mistral exemplifies the **"small but mighty"** approach – lower accuracy ceiling than giant models, but if you engineer the system (with retrieval, fine-tuning, etc.) it can potentially meet the business requirements at a fraction of the running cost. It's best suited when you need **on-premises deployment, low latency, and have the expertise to compensate for its limitations via prompt and system design**.

Other Notable Open-Source Models

Aside from LLaMA and Mistral, the open-source community has produced many LLMs that could be considered, each with their own strengths:

- **Falcon** (TII/UAE's model): Falcon 40B was top in open-source benchmarks around 2023, particularly good at generic tasks. It has a permissive license for commercial use. A 40B model like Falcon could be a middle ground – more powerful than a 7B, but lighter than a 70B. It might require slightly less infrastructure than LLaMA-70B and still give good results after fine-tuning on domain-specific data.

- **MPT (MosaicML)**: Mosaic's MPT models (7B, 30B) are open and optimized for training efficiency. MPT-30B was quite capable, and Mosaic had a vision of modular training; their 30B could be adapted for long contexts (they had an "MPT-30B-8K" and even 16K context versions). MPT could be considered if long context open-source is needed – e.g., MPT-30B with 16k context might handle moderately large documents without chunking.
- **GPT-J / NeoX / Pythia**: Older open models like EleutherAI's GPT-J-6B or GPT-NeoX-20B or the Pythia series might be considered if using older infrastructure, but frankly newer models like LLaMA have made them mostly obsolete in performance.
- **Domain-Specific Models**: There are emerging LLMs fine-tuned for legal or financial text. For example, *LegalGPT* or *Lawyer LLMs* fine-tuned on case law, or *FinGPT/BloombergGPT* trained on financial data. BloombergGPT (from 2023) was a 50B model trained on finance documents, which might make it naturally good at understanding loans. If available, such a model could be very apt for comparing financial documents since it "speaks the language" of the domain. However, many of these are proprietary or not generally available. The user could, in theory, fine-tune an open model on a corpus of financial documents to get similar specialization.
- **Emerging smaller models**: By 2025, we also see efforts like **Alpaca** (Stanford's fine-tuned LLaMA), **Vicuna** (fine-tuned on conversation data), **WizardLM** (enhanced instruction tuning) and others. These are variants of base models tailored for better instruction following. For instance, Vicuna-13B was quite good at chat-style responses (somewhere between GPT-3.5 and GPT-4 level at times). Such models could be considered if one needs an open model that's already aligned to follow user instructions closely. They would still need help (via retrieval or fine-tuning) to handle document content accurately.

In summary, the open-source landscape is rich and evolving. The general pattern is that open models require more effort to adapt and integrate, but they offer **flexibility** (you can fine-tune, self-host, avoid per-query costs) and **community support** (many plugins, extensions, and forums for troubleshooting). In contrast, proprietary models offer **convenience and cutting-edge performance at a cost**. The next section will delve into techniques that can amplify the capabilities of any of these platforms for our document comparison scenario.

Key Techniques for Document Comparison with GenAI

Beyond choosing a model, **how** you use the model is crucial. Several techniques can be employed to improve accuracy, handle large documents, and structure the comparison task. Here we describe the main techniques and how they apply to comparing two structured documents like loan agreements:

Prompt Engineering

Prompt engineering is the art of crafting the input to the LLM to elicit the best possible response. Even with a powerful model like GPT-4 or Claude, a poorly phrased prompt can lead to irrelevant or incorrect answers, whereas a well-designed prompt can significantly improve accuracy and consistency.

- **Structured Prompts:** For document comparison, a good prompt might explicitly set the context and ask for specific output. For example: *"You are an AI assistant that compares loan documents. Below are two documents: Document A and Document B. Provide a detailed comparison, listing differences in interest* *rate, loan term, repayment schedule, fees, and penalties. Quote the relevant sections from each document in your answer. Do not assume any facts not stated in the documents."* This prompt does several things: it defines the role, it enumerates the exact aspects to compare, and it instructs the model not to hallucinate (by quoting the source text). Providing such **step-by-step instructions and examples** can improve results [24]. For instance, one could include a small example in the prompt: "*(Example) Document A: interest rate 5%; Document B: interest rate 5% –> No difference in interest rate.*" as a guide. Few-shot examples like this often help the model understand the task format.

- **Avoiding Hallucinations:** Legal/financial accuracy is critical, so prompts should emphasize not making up information. Techniques include instructing *"If information is not found in the text, say you cannot find a difference"* or *"only reference details present in the documents."* Experiments in the legal AI domain found that simpler, straightforward prompts sometimes yield more accurate outputs than overly complex ones [24]. So it's about finding a balance: give the necessary guidance but do not confuse the model with overly convoluted instructions.
- **Role and Tone:** Prompt engineering can also ensure the response is in the desired tone or format. You might want the output as bullet points under headings "Interest Rate: ...; Term: ..." etc. You can prompt for that explicitly. If the target audience is developers or loan officers, you may keep the language concise and factual. The prompt can say "Respond in a formal, factual tone suitable for a financial report."
- **Iterative Refinement:** Often you will test prompts and refine them. The nice thing about prompt engineering is that it's **fast to iterate** – no training needed, just trial and error. This is usually the first step to try when implementing the feature: get a decent result just by improving the prompt wording.
- **Limitations:** One limitation is that prompt engineering alone might not overcome fundamental model limitations. If the document is long and the model's context is short, no clever prompt will allow it to consider text it doesn't see – you then need other techniques (like RAG). Similarly, if the model doesn't actually know certain domain facts, you can't prompt it into magically knowing them (aside from providing that info in the prompt). Prompt engineering also can yield brittle solutions – a prompt that works well on one pair of documents might not generalize to another with a different structure. Therefore, prompt strategies often need to be robust (e.g., rely on the model's ability to follow instructions rather than specific keywords that might not appear in all documents).
- **Tooling:** There are emerging prompt design tools and libraries (like Promptify or LangChain prompt templates) that help manage complex prompts or do prompt versioning. For our scenario, maintaining a library of **predefined prompt templates** (one for comparing interest rates, one for summarizing differences in penalties, etc.) could be useful. The app might even let the user pick a specific comparison focus, which triggers a specific prompt variant. This is a way to implement "intelligent predefined prompts" – essentially have a set of well-engineered prompts that the user's query selects among or that the system uses depending on context.

In summary, prompt engineering is a **low-effort, high-impact technique**. It should be the first line of approach: carefully instruct the model on what you want from the document comparison. Often, the difference between a confusing answer and a perfect answer is just tweaking how the question is asked.

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is a powerful technique to deal with large documents and to ground the model's answers in factual data. The idea is simple: rather than asking the LLM to answer questions purely from its internal knowledge (which can be outdated or hallucinated), you provide it with **retrieved excerpts** from the documents that are relevant to the question, and then ask it to base its answer on those excerpts.

For comparing two documents, a RAG approach would work like this: when a user (or system prompt) asks to compare specific aspects of Document A and B, the system first **searches each document for the sections about that aspect**, and pulls those sections into the prompt for the LLM to analyze. This way the model only sees a focused subset of the text – the parts that matter – and thus can give a targeted, accurate comparison.

- **Embedding and Search:** Typically, RAG uses **embeddings** to achieve the search. Each document is pre-processed: it's split into chunks (like paragraphs or sections), and an embedding model converts those chunks into high-dimensional vectors. These vectors capture the semantic meaning of the text. The chunks and their vectors are stored in a **vector database**. When a query comes in (e.g., "compare the interest rates"), the query is also embedded into a vector, and the vector DB is queried for the most similar chunks from Document A and Document B. The top relevant chunks (say the paragraphs that mention interest rates in each document) are returned. This semantic search finds information even if different wording is used (for instance, one doc might say "Annual Percentage Rate (APR)" and the other says "Interest Rate"; an embedding search can recognize those as related even if keywords differ). The concept is illustrated by Microsoft in describing RAG: *"a RAG system first uses semantic search to find articles that might be helpful… then sends the matching articles with the prompt to the LLM to compose an answer."* [25] . Essentially, we're **augmenting** the LLM's input with just the information it needs to answer correctly.
- **Prompt with Retrieved Context:** After retrieval, the prompt to the LLM can be something like: *"Document A relevant section: … [text snippet] …; Document B relevant section: … [text snippet] …; Based on these, compare Document A and B on interest rates."* By providing the actual text, the LLM can quote or refer specifically to them. This greatly **reduces hallucination**, because the model has concrete data to work with. In fact, RAG is known to combat hallucination by *"anchoring responses in actual data"*, thereby **reducing the risk of fabricated outputs** [26] . The model is effectively *open book* now – it doesn't have to invent anything, just read and summarize differences from the given text.
- **Handling Long Documents:** RAG shines when documents are too long to feed entirely into the model's context (which is likely for many real loan documents if using an 8k/16k context model). Instead of truncating or skipping content, you let the model see relevant parts of potentially **unbounded-length documents**. This makes the system scalable: no matter if the docs are 10 pages or 100 pages, you can retrieve specific parts as needed. The model's input length stays small and manageable.
- **Implementation Effort:** To implement RAG, developers need to set up a few components:
- An **embedding model** to vectorize text. This could be OpenAI's `text-embedding-ada-002` model (which is high-quality and easy to use via API) or an open-source embedding model (like SentenceTransformers or Instructor XL). There's some cost if using an API (OpenAI's embedding is ~$0.0004 per 1K tokens, quite cheap [8]), or computational load if doing it in-house. But this is usually done once per document (e.g., when the user uploads the documents, you immediately generate embeddings for all sections).
- A **vector database** to store embeddings and perform similarity search. There are many options (Pinecone, Weaviate, Faiss, Milvus, etc.). For a simple app, even an in-memory search using Faiss or a library is fine if document count is small. If the app will have many documents or concurrent users, a proper vector DB service is recommended.
- The logic to **retrieve** for each query. In our case, it might be slightly complex because it's a comparison: you want to retrieve the related chunk from Document A *and* from Document B. One way is to store embeddings with metadata indicating which document they belong to. Then at query time, you run two searches: one restricted to Doc A's chunks, one to Doc B's chunks. Or run one

search on combined chunks but ensure you get at least one from each doc. This ensures the model sees context from both sides.

- Finally, the **prompt assembly**: inserting those retrieved texts into a prompt template that asks for differences. We need to be careful to label them clearly (e.g., **Doc A says:** "...", **Doc B says:** "...") so the model knows which is which.

Libraries like **LangChain** can simplify some of this, as they have ready-made components for retrieval QA. LangChain, for example, can do a "Similarity search then combine docs then ask LLM" in a few lines of config. Another library, **LlamaIndex (GPT Index)**, is designed to connect documents with LLMs seamlessly and could be used to set up this pipeline with minimal code. So while RAG is extra work beyond a single API call, it's a well-trodden path with a lot of community support and tooling.

- **Pros and Cons:** The advantages of RAG are clear – **grounding and scale**. It often dramatically improves factual accuracy because the model doesn't need to rely on memory; it has the text in front of it. It also makes the solution more **auditable**: you can show the user the excerpts the AI used to generate the answer (in a UI, you might highlight "here's the clause from Doc A and clause from Doc B that were compared"). This builds trust and allows verification. Another advantage is **freshness**: if loan documents contain latest terms or regulations not seen in the model's training data, RAG surfaces that text directly, so the model isn't limited by outdated knowledge. As IBM writers succinctly put it, *"RAG plugs an LLM into stores of current, private data... returning more accurate answers with added context of internal data"* [27] – exactly our case, the internal data being the loan documents provided by users.

The trade-offs include: (1) **Complexity** – more moving parts (the system can fail if the retrieval fails or if embeddings aren't good). (2) **Possible retrieval misses** – if the wording in documents is very different, the similarity search might not retrieve the perfectly corresponding pieces. Proper tuning of embedding model and maybe using keywords as backup (like also do a keyword search) can mitigate this. (3) **Prompt length** – if many sections are relevant, feeding all of them could approach the context limit of the LLM. One has to balance how many snippets to include. Usually top 2-3 from each doc suffice. If too much text is included, the model might get overwhelmed or incur higher cost. But since typically we ask about specific fields (interest rate, etc.), this isn't too bad – you retrieve maybe one paragraph from each doc for each field of interest. (4) **Answer coherence** – because the model only sees chunks, it might not have the broader context of the doc's structure or earlier definitions. So sometimes the answer might lack context. However, since we ask it to compare, as long as each chunk is self-contained for that field (which they usually are: interest rate clause is usually in one place), it's fine. If not, one might retrieve a bit more surrounding text or allow the model to ask follow-up (see Agent approach next).

- **Example:** Suppose the user clicks "Compare interest rates". The system knows the query is about interest rates, so it uses an embedding-based search to grab the interest rate clause from Doc A and Doc B. It then prompts: *"Document A - Interest Rate Section: [excerpt] Document B - Interest Rate Section: [excerpt] Question: What are the differences in the interest rates between Document A and Document B?"*. The LLM's answer will be directly grounded on those excerpts, e.g., *"Document A's interest rate is 5.0% fixed, whereas Document B's interest rate is 4.5% variable (indexed to prime). Thus, Doc B offers a lower initial rate but it can fluctuate, unlike Doc A's fixed rate* [26] *."* – ideally the model even quotes as shown (the citation here is just illustrative that it's drawing from real data as instructed). This demonstrates high accuracy because the info came straight from the documents.

Overall, RAG is considered a **best practice** for many enterprise LLM applications because of its improvements to accuracy and ability to work with proprietary data. In a loan document comparison app, using RAG (especially in combination with a strong model) is a reliable approach to ensure the AI's outputs are fact-based and relevant.

Agent-Based Systems

Agent-based systems involve using an LLM in a loop where it can **plan actions, use tools, and break down tasks** to achieve a goal. Rather than a single prompt-response, an agent interacts with the environment or subtasks. For document comparison, an agent approach might allow more complex or interactive analysis – for example, reading the documents in segments, performing calculations, or iteratively deepening the comparison.

- **What is an Agent?** In this context, an "agent" is typically implemented with frameworks like **LangChain**, **Chainlit** or custom logic, where the LLM is given the ability to call functions or tools. It might have a prompt like: *"You are a document analysis agent. You can ask for specific sections of Document A or B, you can call a calculator, and you can output the final report."* The agent then can operate step-by-step, deciding which tool to use next. For instance, it may first retrieve the interest rate from each doc, then compute the difference, then format an answer. This is powered by the LLM but guided by a pre-defined set of possible actions.

- **Use Cases for Agents in Doc Comparison:** One reason to use an agent is if the comparison needs to be very detailed and systematic. For example, maybe the requirement is: *"List all differences between the two documents, section by section."* This could be a lengthy process if done in one prompt (it might exceed context length or result in a very long answer that's hard to validate). An agent could tackle it by iterating over sections: it could have the ability to *"read next section from Doc A and Doc B"* and then compare those small pieces in a loop. It essentially decomposes a big task into many small LLM calls. This is more controllable – you can ensure each section is handled, and perhaps parallelize or cache results. Another reason is when some aspects of comparison require external logic: e.g., verifying if numerical sums match, checking dates (maybe one doc's schedule has dates that should correspond to the other's). The LLM agent can be given a **calculator tool** or a **date parser** to offload exact arithmetic or date computations. This avoids errors (LLMs can sometimes do arithmetic incorrectly or misunderstand date differences). By using tools, the agent can achieve a higher level of accuracy in those sub-tasks.

- **Integration Complexity:** Implementing an agent is more complex than a single prompt or simple RAG. With LangChain, you would define tools like:

- A tool to get a specific document section by heading or number.
- A tool to run a search over the document (similar to RAG, but agent-initiated).
- Possibly a tool to fetch the entire document (if needed).
- Utility tools like a calculator (LangChain has a simple math evaluator).
- Then you set up an **Agent executor** with these tools and the LLM as the reasoning engine.

The prompt (often called the "agent's prompt") will have instructions and examples on how to use the tools. The agent LLM will output something like "Action: Search Document A for 'interest rate'" then the system executes that, returns result, then the agent sees it and continues reasoning: "Observation: Found interest rate = 5%. Action: Search Document B for ‘interest rate’” and so on, until it finally says “Action: Finish” and outputs the final answer. All these intermediate steps are not shown to the user; they are internal. The end result is the answer which is hopefully more precise.

This approach leverages the LLM’s ability to perform **chain-of-thought** reasoning explicitly and to interact with external functions. It’s like how a human might systematically compare two contracts: read a section from A, then B, note differences, move to next section, etc., possibly taking notes or doing calculations along the way. The agent emulates that process.

- **Advantages:** The main advantage is **fine-grained control and potentially improved accuracy** on complex tasks. Agents can handle very large documents by reading them piecewise (since each read action can fetch a small part, you’re never stuffing the whole doc in context at once). They can also dynamically decide which part to focus on – for instance, if an initial question is broad (“find all differences”), a smart agent might identify key sections to compare rather than linearly going through everything (though a simple approach might just go linearly). Agents with tool use can avoid some LLM limitations: if asked “what’s the difference in total fees over the loan term?”, the agent could actually sum up fees from tables in each doc using a calculator tool then compare, rather than trusting the LLM’s internal math. This yields **higher confidence in numerical accuracy**.

Agents also allow **interactive conversation** patterns. For example, the user might ask a follow-up like “What about differences in collateral requirements?” – an agent can handle this by using the same tools to retrieve collateral clauses and analyzing them, within the context of the ongoing session. It can maintain memory of what was compared already or not if designed to do so. This is essentially building a mini-AI analyst that can navigate the documents.

- **Limitations:** The complexity of agent systems is their biggest downside:
- There’s more that can go wrong: the agent might take too many steps, or get stuck in a loop, or use a wrong tool if not carefully constrained. Prompting agents is an art of its own – you have to define the tasks well so it doesn’t hallucinate an action that doesn’t exist or try something weird.
- Each step is an LLM call, which means **potentially higher latency and cost**. If comparing two 30-page documents section by section, and there are 20 sections, the agent might make dozens of LLM calls (one or two per section plus overhead). This could be significantly slower and more expensive in token usage than a single prompt that tries to do all at once. Mitigations include using a cheaper/faster model for the agent’s reasoning (maybe GPT-3.5 or an open model) if it can still do the job.
- Debugging: it can be hard to test all the paths an agent might take. You’ll need to monitor and perhaps set up timeouts or step limits to avoid runaway cases.

- Simpler might be sufficient: In many cases, a well-implemented RAG with one-shot prompts is enough. Using an agent might be over-engineering if the task can be handled in one or two prompt calls. Always consider if the added complexity is justified by significantly better outcomes.

- **When to Use Agents:** If the documents are highly structured and you want a thorough, possibly automated, multi-step analysis, agents shine. For instance, suppose after comparing, you also want the agent to *update a spreadsheet* or *create a summary table of terms* – an agent can call an API or function to do that. Or if the user might ask extremely broad or unbounded questions like “Do these documents have any significant differences I should worry about?”, an agent could break that down into checking each possible category of difference (interest, term, etc.) sequentially, which a single prompt might not reliably do (it could forget to mention something). Agents are also useful if integrating with other enterprise systems: e.g., after comparison, agent could auto-email a report, or fetch additional data (like current interest rates from an API to contextualize differences). Essentially, it's about **automation** and **integration** beyond just Q&A.

In summary, agent-based systems offer a **dynamic and robust** approach to document comparison, at the expense of increased complexity. They make the AI more **like a human analyst** that can take actions and intermediate steps. For mostly developer audiences, implementing an agent will involve working with frameworks like LangChain (which many developers are exploring nowadays), and ensuring the LLM has reliable tools (document readers, searchers, calculators). If done well, it can yield an AI that not only compares documents but can handle a variety of related tasks in a flexible workflow – effectively a custom *AI agent for documents*. This could be a differentiator if your application needs to handle more than just static Q&A.

Embeddings and Semantic Search

We already touched on embeddings in the RAG section, but embeddings themselves deserve emphasis as a technique. **Embeddings** are vector representations of text that capture semantic meaning. In the context of document comparison, embeddings can be used in creative ways beyond just retrieving for a given query.

- **Document Similarity & Section Alignment:** One challenge in comparing two documents is aligning their sections or finding which parts correspond to each other. Embeddings can help automate this. For example, you can take every section (or paragraph) of Document A and compute its embedding, and do the same for Document B. By comparing these vectors, you can find which section in B is most similar to each section in A. If the documents are two versions of a contract, ideally each clause in A has a matching clause in B, and the closest embeddings should pair them up. This way you can programmatically detect if a clause was removed or significantly changed – if a section in A has *no* close match in B above a certain similarity threshold, that might mean B is missing that clause or it's very different. Likewise, if one section's best match is actually a different labeled section, it could indicate reordering or merging of sections. Using cosine similarity between embedding vectors gives a quantitative measure of how related two pieces of text are [28]. A high cosine similarity (close to 1.0) indicates they are covering the *same content or meaning* [28]. This could be used to create a mapping between documents.
- A practical approach: cluster Document A and B's sections together by similarity; each cluster would contain sections from A and B that talk about the same topic. Then differences can be extracted per cluster, possibly using an LLM to summarize "what's changed in this cluster between A and B".

- If doing a side-by-side comparison feature, embeddings could even drive a UI that highlights which sections to compare. This is more *algorithmic* and could complement the LLM output by focusing it. For example, you might run this alignment first, and then for each aligned pair of sections, ask the LLM: "These two sections seem to match, summarize the differences." And for any unaligned sections, ask "Document A has this clause that Document B doesn't – summarize it as an extra clause." This ensures thorough coverage.

- **The Right Embedding Model:** There are specialized embedding models for legal text or long documents. However, often general-purpose models like OpenAI's Ada or SentenceTransformer's all-mpnet work well for semantic similarity. Since loan documents likely involve technical terms, you may consider an embedding model that has been exposed to legal/finance text for best results. If not, general embeddings should still capture most meaning (they handle words and context well). Embeddings are typically low-dimensional (like 1536 dims for Ada) and you can store a lot of them efficiently. If each doc has 50 sections, that's just 100 vectors to compare – trivial computationally.

- **Limitations:** Embeddings capture semantic similarity, but they might not capture *nuance*. For example, if Document A says "prepayment allowed after 2 years with 1% fee" and Document B says "prepayment allowed after 2 years with 2% fee", those two might still appear very similar in embedding space, because the sentences are identical except one number. The embedding might not be sensitive enough to treat them as very different (numbers often don't heavily influence the embedding, which is more tuned to overall meaning). So purely using embeddings to detect differences could miss numeric or small phrasing differences that actually matter. This is why combining embeddings with LLM is powerful: embeddings can point out *"these two pieces talk about the same thing"*, then the LLM can be brought in to actually parse the text and notice *"ah, the fee percentage differs"*.

- There are some ways to mitigate this: one could augment the text with certain tags or normalized values to help embeddings, or ensure that when something like interest rate differs, you rely on the LLM or explicit code to catch it rather than the embedding.
- Also, embeddings might mistakenly align sections that are topically similar but not actually corresponding. E.g., maybe both documents have a generic "Miscellaneous" section – they could be similar but contain different specific clauses. The LLM in the loop (or some metadata like section titles) would be needed to verify actual correspondence.

- **Embeddings for Query Understanding:** In a chatbot scenario, if the user asks a question in a different way (like "How do these documents differ in terms of early repayment?"), you can use embeddings to map that to the sections or to predefined prompt templates. For example, you might embed the user's question and have stored embeddings for various known comparison aspects ("interest rate differences", "repayment term differences", etc.) to see what they're asking about. This is a form of semantic parsing using embeddings. It could route the query to the appropriate logic (maybe you have a function specifically to handle interest comparisons). While this goes beyond basic doc comparison, it's a technique to add robustness to user interactions in the app.

In essence, embeddings are the **connective tissue** in many LLM applications. They enable *semantic search*, *clustering*, and *alignment*. In our use case, they ensure the right information from the documents is identified and fed into the model or the logic. The result is a system that doesn't rely solely on the model's internal knowledge, but actively uses the *content of the user-provided documents*. This makes the comparisons much more trustworthy.

Fine-Tuning Models

Fine-tuning involves training an LLM further on custom data so that it performs better on a specific task or domain. Instead of trying to engineer prompts or provide context every time, you *teach* the model what to do by example. Fine-tuning can be applied to open-source models (where you have weight access) and also to some API models (OpenAI allows fine-tuning on GPT-3.5 Turbo, for example, though not yet on GPT-4 at the time of writing).

For the loan document comparison app, there are a couple of ways fine-tuning could be beneficial:

- **Task-Specific Fine-Tuning:** You could assemble a dataset of document pairs and their difference analyses. For instance, maybe take a hundred pairs of loan contracts (or similar documents) and manually or programmatically create the "expected output" – which could be a structured list of differences. Then fine-tune an LLM to take as input something like "Document A: ... Document B: ..." and output a comparison. This would effectively train the model to do the entire task in one go. A fine-tuned model on this could outperform a base model with just a prompt, especially if the fine-tuned model is smaller. The model learns the patterns of how differences are typically described. It might also pick up domain-specific terminology and the importance of certain clauses. According to IBM, *"a fine-tuned LLM typically outperforms its base model when applying its training with domain-specific data, allowing it to generate more accurate responses"* [29] . This suggests that if we fine-tune on the domain of loan documents, the model will understand that space much better – e.g., it will know that "APR" and "interest rate" are related, that a "prepayment penalty" is a negative for borrowers, etc. It essentially internalizes some of the comparison logic.
- **Domain Adaptation:** Another approach is fine-tuning not on the specific *task*, but on *domain text*. For example, take a large corpus of loan agreements or financial contracts and fine-tune a model on predicting the next token (language modeling). This isn't about Q&A or comparison directly, but about making the model's base vocabulary and style more in line with our documents. A model like GPT-3.5 or LLaMA fine-tuned on financial texts might more accurately parse and recall details from similar documents. However, this requires a lot of data and careful training to not ruin the model's general abilities (we wouldn't want it to become too narrow). More commonly, fine-tuning is done with *instruction data* for the domain: e.g., feed it question-answers that are relevant. If we had a list of 1000 Q&A pairs like "Q: What is the interest rate in Doc A vs Doc B? A: Doc A has X%, Doc B has Y%." etc., we could fine-tune on that to make it really good at direct questions about the documents.
- **OpenAI Fine-Tuning vs Open-Source:** If using OpenAI's GPT-3.5, one could fine-tune it to have a certain style or focus. For example, fine-tune it to always produce output in a structured JSON with differences. OpenAI's fine-tuning might help if the out-of-the-box model doesn't follow instructions exactly as you like. That said, OpenAI's models are already quite instruction-tuned, so the gain might be modest unless your format is very specialized. Also, OpenAI does not (as of 2025) allow fine-tuning on GPT-4; only on earlier models like GPT-3.5 Turbo. So you'd be trading some model power for customization. With open-source (like LLaMA), you can fine-tune any size model. Fine-tuning a 70B model is resource-intensive (needs multi-GPU training, perhaps distributed), but methods like LoRA allow adding a low-rank adaptation matrix such that even 70B can be fine-tuned on a single high-end GPU in some cases. Fine-tuning a 7B or 13B is much easier – these could be done on a single GPU in an hour or two for small datasets.
- **Cost and Effort:** Fine-tuning is typically a **heavy upfront effort** compared to prompt engineering or retrieval. You need training data – which might be scarce (do you have a bunch of example comparisons?). If not, you'd have to generate it, maybe by using GPT-4 to produce some synthetic comparisons or by using known differences from versioned documents. Then you need to actually run the training. This requires ML expertise: setting hyperparameters, making sure you don't overfit or introduce bias, etc. And then deployment of a fine-tuned model means you have a custom model to host (if open-source) or a custom endpoint to call (if OpenAI). If the documents vary a lot in structure or content, the fine-tuned model might not generalize well beyond the patterns it saw. For example, if all training pairs had an interest rate difference of a few percent, and then a new pair has

one document lacking an interest clause entirely, the fine-tuned model might not handle that scenario unless it saw something similar in training. RAG, by contrast, always uses the actual content, so it's more flexible to novel inputs.

- **Maintenance:** One drawback IBM highlights is that fine-tuning uses a **static snapshot** of data – the model can become outdated and requires retraining to incorporate new information [30]. In our context, that might be less of an issue (loan terms don't change as rapidly as, say, world events), but if new regulations come and change how loans are structured, a model fine-tuned before that may not know the new structure. RAG would handle it easily by just retrieving whatever is in the docs, whereas a fine-tuned model might miss nuance if not retrained.

- **When Fine-Tuning Makes Sense:** If you plan to use an open-source model and you need to maximize its performance on the task, fine-tuning is very powerful. For instance, an open 13B model fine-tuned on your domain might match a 70B base model performance on that domain – meaning you save runtime costs. Also, if you require **custom output formats** (like a JSON difference report that can be automatically parsed), fine-tuning the model to reliably output exactly that format can be worthwhile since getting that consistency via prompting alone can be tricky. Fine-tuning might also improve a smaller model's tendency to hallucinate by explicitly teaching it to say "I don't know" when something isn't present, if your fine-tuning data includes such cases. According to Red Hat's guidance, consider your **team's skill**: fine-tuning *"requires experience with NLP, deep learning, data preprocessing, etc., and can be more time-consuming"* whereas RAG is often more accessible to implement for a dev team [31]. So you'd embark on fine-tuning if you have or can acquire that ML expertise and if the expected ROI (in performance gain or cost savings) is high.

In summary, fine-tuning is like **investing upfront to tailor the model** to your exact needs. It can lead to a model that performs the comparison task faster (no need for as much prompt or retrieval) and perhaps more accurately on seen patterns. But it's a significant project on its own – gathering data, training, and validating. For many developer teams, starting with prompt/RAG approaches (which require no model training) and only fine-tuning when necessary is a prudent strategy. If fine-tuning is done, it might be on an open-source model for full control, unless using OpenAI's fine-tune on GPT-3.5 for convenience despite the model being weaker than GPT-4.

Implementation Effort and Infrastructure Considerations

Implementing an intelligent document comparison involves assembling the right combination of the above platforms and techniques. Here we summarize the **developer effort**, **integration steps**, and **infrastructure** needs for each major approach:

- **Using a Hosted API Model (GPT-4, Claude, Gemini):** This is the **least effort path** in terms of ML ops. You do not have to manage any model servers – you simply call the provider's API. For GPT-4 or Claude, integration means obtaining API keys, perhaps installing the provider's SDK (or just using HTTP calls), and then sending prompts with your document data. The main work for the developer is in prompt engineering and handling the API responses. Infrastructure needed on your side is minimal: just your web app server to make API calls. The cloud provider handles all the heavy lifting of GPUs and model inference. However, you must ensure your system can handle the asynchronous nature (these calls might take a few seconds, so maybe use a background job or async call). Also, consider data governance: if deploying in production, some companies set up proxies or use something like Azure OpenAI (which offers GPT models in a more enterprise-managed way) to satisfy internal policies. In terms of scaling, if usage grows, you might run into API rate limits, so you might need to request higher quotas or distribute load across multiple keys/accounts. But overall, the scaling is handled by the provider – you don't worry about adding more GPUs yourself. **Developer effort focus:** prompt design, handling errors (like API timeouts or model refusals), and cost optimization (ensuring you don't send unnecessarily large prompts).

- **Using Open-Source LLM (Self-Hosted):** This approach requires setting up **ML infrastructure**. A developer (or ML engineer) will need to:

- Choose a model (e.g., LLaMA 2 70B or Mistral 7B) and obtain its weights (downloading several GBs).
- Set up a server or cloud instance with sufficient GPU resources. This could be a single high-memory GPU for smaller models or multiple for larger. Tools like Docker images (e.g., **text-generation-inference** container from HuggingFace) can simplify deployment – it exposes a REST API for the model. Alternatively, one might integrate the model within the application code using frameworks like PyTorch Lightning or JAX, but typically using a dedicated serving solution is better for performance (they handle batching, quantization, etc.).
- Optimize the model for runtime: apply 4-bit quantization if needed to fit GPU memory, use optimized transformers libraries or runtimes (like HuggingFace's Accelerate or NVIDIA's TensorRT if applicable).
- Ensure the serving endpoint is stable and can handle concurrent requests. This may involve using model parallelism or multiple replicas if heavy throughput is needed.
- Set up monitoring for GPU utilization and memory – so you know if you need to scale up for more load.

The **integration** into the app from there is calling your own API or function instead of OpenAI's. Latency might be a bit higher for big models (as they run on your hardware possibly at lower throughput than OpenAI's superclusters). But you gain independence from third-party services.

Developer effort focus: DevOps around ML. This includes possibly writing some boilerplate to load the model and keep it in memory, designing a fallback or load-balancing strategy if the model is busy (maybe queue requests). Also, if updates are needed (like switching to a new model checkpoint), you manage that deployment. Testing is required to ensure the model outputs what you expect (some open models might need slight prompt format adjustments, e.g., they might expect a system prompt like "<s>[INST] ..." tokens for proper formatting if they were trained that way). So, more trial and error at the start to integrate the model properly.

- **Implementing Retrieval (RAG) Pipeline:** If you choose RAG (which is highly recommended for long docs), plan to integrate a **vector database or search component**. Developer steps:
- Pick a vector DB: could be a managed service like Pinecone (easy: just use API), or an open-source one like Weaviate or ElasticSearch's vector support (which you have to deploy or have an instance of).
- When a document is uploaded, chunk it (maybe use headings or just split every N sentences), embed each chunk with an embedding model. If using OpenAI's embedding API, it's a quick call (just ensure you handle rate limits if embedding many docs at once). If using an open embedding model (like sentence-transformers) you might run it locally (fast on CPU for moderate amounts of text).
- Store the resulting vectors and metadata (doc id, section text) in the DB.

- At query time, implement logic to query the DB with the question or the known aspect (if it's a predefined prompt scenario like clicking a button "compare interest rate", you might bypass semantic search and directly retrieve sections that have titles or keywords like "Interest" – a simpler approach if docs are well-structured).
- Take the top relevant pieces and construct the LLM prompt.

This requires writing code for the above or using a library. LangChain can do a lot of it with their `RetrievalQA` or `ConversationalRetrievalChain` classes – you'd still need to set up the DB and embeddings though. The effort is moderate: perhaps a few days of work to get all the plumbing right if not familiar with these tools, but it's a one-time setup and largely configuration after that.

Infrastructure: If using a managed vector store, just factor in that cost (often charged by vector count or queries, but for small scale it's minimal). If self-hosting a vector DB (like running Weaviate or a simple Faiss in memory), ensure the server has enough memory to hold vectors and that there is a way to persist them (so you don't re-index every time it restarts). Usually text data for a couple documents is tiny in vector form (a few hundred kilobytes maybe), so scaling is an issue only if you have many documents or very large corpora.

Developer effort focus: Data processing (text splitting, ensuring no important context is lost when splitting), and the interface between search results and the final prompt (like making it seamless, e.g., if multiple chunks are needed to capture a clause that spans paragraphs, you might need to retrieve both and include both). Testing the RAG pipeline is also important: does it retrieve the right things for various queries? You might need to adjust chunk sizes or add some keywords to the query to guide it. For example, sometimes combining keyword search with vector search improves reliability (some frameworks do a hybrid search).

- **Fine-Tuning a Model:** If you decide to fine-tune, the effort ramps up on the **ML engineering** side:
- **Data Prep:** You need to compile a dataset for training. For supervised fine-tuning, this means input-output pairs (or a set of prompts with ideal completions). You might use actual examples of differences or generate synthetic ones. Ensure diversity so model learns broadly.
- **Training Environment:** Set up a training environment, which could be on a cloud GPU instance or a local rig. Use libraries like Hugging Face Transformers which make fine-tuning fairly approachable (they provide scripts for fine-tuning that you largely need to configure).
- **Parameter Efficient Tuning:** Decide if you do full fine-tune (updating all weights) or parameter-efficient (like LoRA, adapters). LoRA is popular as it allows merging weights later or toggling between base and fine-tuned easily, and it doesn't require as much memory. There are many resources and open scripts for LoRA training on LLaMA for example.
- **Compute and Time:** Fine-tuning can take from hours to days depending on model and data. A smaller model (7B) might train in an hour on a single GPU for a small dataset; a 70B might take multiple GPUs and careful distributed setup. This is a non-trivial effort – possibly needing someone with ML background on the team or consulting with a service that can do fine-tunes.
- **Validation:** You'll need to evaluate the fine-tuned model on some test cases to ensure it actually improved and didn't overfit or start spouting wrong info. Possibly compare its outputs to the base model on a set of tasks.
- **Deployment:** Once fine-tuned, deployment is similar to the open-source self-host scenario – but now you have custom weights to deploy. If using OpenAI's fine-tuning, they handle deployment (you get a model name you can call via API).

Fine-tuning can be an *iteration* as well – you might do a round of tuning, see where it fails, add more data, tune again. It's more akin to classical software training cycles rather than just coding.

Developer effort focus: ML tasks (data curation, running training jobs, hyperparameter tuning). If the team is mostly application developers without ML specialization, this can be challenging. It may be easier to use a fine-tuning service or a platform like Hugging Face Hub (which offers AutoTrain or similar) to reduce the burden.

- **Implementing an Agent:** Setting up an agent system will involve writing prompts and possibly using an agent framework:
- If using **LangChain**, you would choose an Agent class (like `AgentType.ZERO_SHOT_REACT_DESCRIPTION` or their Tool-using agent). You then define your Tools (each tool has a name, description, and a function that it calls in Python). In our case, tools might be: `get_section(doc, section_name)`, `search_document(doc, query)`, `calculate(expression)`, etc.
- Write the prompt prefix for the agent that describes when to use which tool. LangChain automatically appends the tool list and format, but you may need to experiment with phrasing so the agent uses them correctly.
- Test the agent with various tasks to see if it behaves. You might find it needs more guidance, so you add example usage (few-shot examples of tool use).
- Ensure you put limits (LangChain allows max iterations, etc.) to prevent infinite loops or very long chains.
- Infrastructure-wise, an agent just uses the same LLM backend as you have (OpenAI or your hosted model) but will make multiple calls. So be aware of compounding cost. You might want to use a cheaper model for the agent if you can trade off some quality for cost. Or limit the agent to certain complex queries and use direct prompting for simpler ones.

Building an agent is somewhat like orchestrating a mini workflow engine where the LLM is the decision-maker. The developer's role is to set the boundaries and provide the tools. This is a bit of a **software engineering** task combined with prompt engineering. Compared to plain RAG, it's more code to write and maintain.

Developer effort focus: Designing the agent's logic and ensuring reliability. It might require debugging the model's thought process by reading its step-by-step logs and adjusting prompts or adding constraints if it goes off course. Over time, you might refine the agent as you see how it performs on real user queries.

- **Data Privacy and Security:** A quick note – whichever approach, developers must consider where document data is flowing. If using external APIs (OpenAI, Anthropic, etc.), ensure it's allowed to send that content out. If not, lean towards open-source on-prem solutions. If storing data in vector DBs, encrypt or secure access as needed (since financial documents are sensitive).

- **Testing and Validation:** With any approach, plan to test the system on a set of known document pairs with expected differences. For prompt-based approaches, you check if the output covers all key differences and has no hallucinations. For RAG, check that it's actually pulling the right references and not missing something because maybe a term was phrased oddly. For fine-tuned models or agents, ensure consistency and reliability (maybe an agent initially needs close monitoring to be sure it's not mis-using tools or skipping content).

In summary, **developer effort ranges from very low (API calls and prompt tweaking) to very high (custom model training and multi-step agents)**. Many teams opt for a middle ground: use an API model with RAG – this leverages powerful models and ensures factual grounding, without having to train models from scratch. That approach requires moderate integration work (embedding + vector DB), but it's manageable with modern libraries and gives a good balance of quality and development speed.

Limitations and Constraints to Consider

When designing the GenAI-powered comparison feature, it's critical to be aware of the limitations and constraints of each approach. Here we highlight key factors and how they impact the different options:

- **Latency:** Response time matters in a web app. Larger models like GPT-4 and open 70B models have higher latency. GPT-4's responses can take several seconds up to tens of seconds for lengthy outputs, which might feel sluggish to users used to instant results. Claude with a 100K context might take ~20+ seconds for huge inputs (as seen in the Great Gatsby example) [6], though for moderate tasks it can be quicker. Open-source models running locally will depend on your hardware – a 7B model might generate text very fast (perhaps ~50 tokens/sec), whereas a 70B might be much slower (~10 tokens/sec on one GPU). If your comparison output is, say, 500 tokens long, and generation is 10 tokens/sec, that's 50 seconds – unacceptable for live interaction. Strategies to mitigate latency:
- Use **smaller or distilled models** if possible for parts of the task. For example, maybe use GPT-3.5 (which is very fast and cheaper) to do an initial summary or retrieval of differences, then have GPT-4 verify or expand on it. Or use an open 13B instead of 70B if fine-tuned well – 13B can be roughly 2-3x faster than 70B.
- If using an agent with multiple steps, consider that each step adds overhead. Combining steps into one prompt (if feasible) is faster but might reduce reliability.
- Utilize asynchronous processing: you might accept the document upload, then in the background generate the comparison and notify the user or display when ready, rather than blocking the UI the whole time.

- Some providers (like OpenAI via Azure) offer **streaming** responses, where the model's output token stream is sent in real-time. This can improve perceived latency – the user starts seeing the answer as it's generated (maybe key difference bullet points appear one by one).

- **Context Length:** This is a constraint on how much information can be considered at once. Each model has a maximum context (prompt + output) length. If you exceed it, the prompt will be truncated or the model simply can't handle it. GPT-4 (8k/32k), GPT-4 Turbo (128k) [4], Claude (100k) [5], Gemini (128k+ likely) [11], LLaMA (4k), etc., all differ. If using a smaller context model (like GPT-3.5 at 4k or LLaMA without extended context), you **must rely on RAG** or summarization to feed in only relevant info. If you attempted to stuff two full documents into a 4k context model, you'll almost certainly run into truncation or the model ignoring a lot. Conversely, if you have access to something like Claude 100k or GPT-4 128k, you have the luxury of feeding everything, but remember the **cost scales** with context. Also, just because you *can* give a model 100k tokens doesn't always mean you *should* – the model might lose focus, and it could be expensive. Evaluate the typical size of documents: if each loan doc is, say, 10 pages (~5k tokens each), then even GPT-4 8k cannot take both fully (10k total) – you'd need at least the 32k version or a retrieval strategy. If docs are small (a few pages each), then maybe you can fit them in mid-sized context and direct prompting is fine. Always design with worst-case lengths in mind (what if someone uploads a 50-page loan contract?). Possibly enforce limits on uploads or use RAG as a scalable solution.

- **Hallucination and Accuracy:** As noted, hallucination is the tendency of LLMs to output fabricated or incorrect info confidently. This is dangerous in our use case – you don't want the model saying "Document B has a 7% fee" when it's actually nowhere in the text. While techniques like RAG greatly reduce this by grounding answers [26], no solution is 100% immune. The model might misattribute which doc a clause came from, or it might fill in a guess if something was ambiguous. Testing on domain-specific edge cases is important. For instance, if a clause in one doc is implied but not explicit, will the model incorrectly assume the other doc has it? We might need to add rules like if one doc lacks a section entirely, explicitly mention that rather than assume symmetry.

- Fine-tuned models might hallucinate less in their domain if trained to not do so, but it's not guaranteed.
- Agents could potentially verify facts (e.g., the agent might extract values and then compare them, leaving less room for free-form invention).
- Another method is to **post-process the output**: after the model writes differences, you could automatically cross-check any values it stated against the documents. For example, if it says "5%" was the rate in Doc A, you could search Doc A for "5%" to confirm, and if not found, flag a potential error. This kind of verification system adds complexity but can catch hallucinations, essentially grounding via brute-force. Some research and tools (like "Scarecrow" or other hallucination detectors) can assist, but in practice, domain-specific verification code might be simplest (e.g., a regex for interest rates in text and compare with model output).

- The bottom line is you should **never fully trust** the raw output for critical decisions without some validation. At minimum, keep the human in the loop (the user should see enough context or quotes to trust the output).

- **Cost Constraints:** Cost comes in multiple forms:

- **API usage cost:** If using OpenAI, Anthropic, etc., each comparison call might cost a certain amount of money in tokens. Multiply that by number of users and you have monthly expenses. For instance, a single GPT-4 (8k) run comparing moderately sized docs might consume 4k prompt + 1k output = 5k tokens, which is ~$0.18 per run. If you do 1000 such comparisons, that's $180. With larger contexts, costs go up (the 32k model is double price per token, etc.). Claude might be cheaper for large input as noted (~$1.50 for 170k tokens processed in one test) [10]. But still, if your app grows, you could be looking at thousands of dollars per month in API fees.
- **Infrastructure cost:** If self-hosting, you have fixed costs like cloud GPU rentals or on-prem hardware depreciation. A single high-end GPU can be $2-$3k upfront or ~$1-3/hour on cloud. Running 24/7 is ~$700-$2000/month per GPU. If you need multiple for performance or redundancy, multiply that. Open-source models avoid per-query fees but have these fixed costs. If usage is low, you might pay a lot for idle capacity; if usage is high, self-host might turn out cheaper than API.
- **Developer cost:** Effort is money too – spending a month building a fine-tune or agent system has an engineering cost. If a quick API integration can deliver an MVP, that might be more cost-effective initially, even if the per-call cost is higher. Later, one might invest in optimization if volume grows.

You should analyze expected usage patterns. If it's an internal tool used a few times a day, API costs are negligible and the simplicity of API is worth it. If it's a public feature potentially used thousands of times, optimizing for cost (via open models or at least using cheaper models like GPT-3.5 for some tasks) becomes important.

One tactic is a **hybrid approach**: use a cheaper or smaller model with RAG for initial answers, and only escalate to GPT-4 or similar if the smaller model is unsure or if the user requests a more detailed analysis. Another tactic is caching – if the same documents are compared often or the same questions asked, cache the results to avoid repeating cost.

- **Model Limitations (Knowledge Cutoff and Domain):** Remember that models like GPT-4 have a knowledge cutoff (e.g., GPT-4's training data goes up to 2021 or early 2022). They won't know specifics of say a new law in 2023 unless you tell them. For our use, that means if the loan docs reference a very recent regulation or a novel clause, the model might not inherently understand it. This is where providing context or definitions in the prompt can help. But more straightforwardly, the model might not need external knowledge – it just compares what's there. The issue would be if it needs to understand a clause's significance (which is not really the task, just identification of differences).
- Some open models (LLaMA) are trained on lots of internet data but not specifically on legal contracts. They might be unfamiliar with certain legalese or abbreviations, leading to misunderstandings. Fine-tuning on domain text or providing a glossary in the prompt can mitigate this.

- Also, no model can do math or logic perfectly all the time. If the comparison requires some arithmetic (e.g., summing fees or understanding schedules), a pure LLM solution might falter. Either incorporate a calculation step (agent tool) or explicitly ask the model in a careful way (like "calculate the difference in interest rate numerically" – GPT-4 can do basic arithmetic usually, but not always reliably for complex math).

- **Ethical and Legal Concerns:** Since the target is loan documents, one must be cautious about the outputs:

- **Confidentiality:** Ensure that if using third-party APIs, you are not violating any privacy agreements. Perhaps mask certain personal identifiers in documents if they exist (like names, addresses) before sending to the model, if that's a concern.
- **Accuracy in Finance:** Even minor mistakes could be problematic. If the AI says "Document B has no prepayment penalty" and in fact it did but worded differently, a user might make a financial decision based on that error. So it's wise to either disclaim that the AI result is for assistance and not final, or ensure a human double-checks critical comparisons. Over time the model can be trusted more as it proves accuracy, but initial deployment might be labeled as "beta" or "assistive" tool.

- **Bias:** While not obvious in this context, if the documents involve subjective terms (like interpreting if one is "more favorable" than the other), the model might inject an opinion. We should constrain it to factual differences, unless analysis is desired. For instance, if asked "Which loan is better?", that's subjective and goes beyond just differences – the AI might express a biased view or could mislead. It's safer to stick to objective comparisons (numbers, presence/absence of clauses) rather than recommendations, unless you deliberately want that feature and handle it carefully.

- **Limits of Structured Data Handling:** Loan documents may contain tables (amortization schedules, fee breakdowns), or bullet lists. LLMs are primarily text oriented and might not perfectly handle a dense table. They might read it row by row in a weird way or skip some. If the critical info is in a table or form, consider extracting that via a different method (like an OCR or PDF parser that can give structured data). Then feed that data in a digestible format to the model. Tools exist (e.g., if using an agent, an **OCR tool** or a PDF parser tool can feed data). Or pre-process the PDF to text carefully such that tables become lists of values with labels.

- Also consider units and formatting: one doc might state an interest as “5 percent”, another as “5%” or “0.05”. A model will likely see those as the same, but just be cautious about any inconsistencies (embedding search might fail if one doc uses the word “percent” and query uses “%” – not likely but possible; ensure text normalization in such cases).

In summary, understanding these constraints helps in designing a robust solution. Often a combination of techniques is used to mitigate them: e.g., RAG to handle context length and hallucination, prompt engineering to reduce irrelevant info and focus the model, fine-tuning to embed domain knowledge, agents to break down complex tasks, etc. The aim is to deliver a system that provides **fast, accurate, and trustworthy comparisons** to the end-user under real-world conditions.

Suitability of Each Approach for Structured Loan Document Comparison

Finally, let’s evaluate how well each platform and technique serves the specific use case of comparing structured loan documents, considering all the factors above. Different combinations may be optimal depending on project priorities (accuracy vs. speed vs. cost vs. control). Here’s a breakdown:

- **OpenAI GPT-4 (with Prompt Engineering):** Very **suitable** in terms of accuracy and capability. GPT-4 can understand complex language and will likely catch nuances in loan documents. With proper prompts, it can list differences clearly. It’s a great choice if you want the highest quality analysis out-of-the-box. However, context limits mean you’d use it either with some summarization or with the 32k/128k context versions (if you have access). If using 8k context, you’ll definitely need RAG for longer docs. Hallucination risk is moderate – GPT-4 is less prone than smaller models, but you should still ground it with actual text via RAG or ask it to quote sources. Cost is on the high side, so if each comparison is lengthy, monitor usage. For a developer prototype or low-volume internal tool, GPT-4 is ideal since it minimizes development friction (just call the API with a good prompt). The output is likely to be well-structured if prompted to be. In summary, GPT-4 + good prompt = **high success rate** for getting correct differences, just keep an eye on context and cost.

- **Anthropic Claude (100k context):** Extremely suitable when documents are long. Claude could take both full documents in one shot (if combined length <100k tokens, which is roughly <75k words). This means you can literally do: “Here’s doc A and doc B, find differences.” That simplicity is a big win – you leverage the model’s ability to synthesize across a lot of text. The quality of Claude’s output is quite high as well, though perhaps a tad more verbose. It might enumerate differences nicely, possibly even ones a retrieval approach might miss if they were spread out (because Claude sees whole picture). Given Anthropic’s note that Claude can *“follow instructions and return what you’re looking for, as a human assistant would”* across many parts of text [7], it’s arguably one of the best fits for a holistic comparison. Suitability concerns: cost if using big contexts often, and ensuring that it accurately matches sections (with so much text, we must format prompt clearly so it knows which part is which document). Claude is likely to be very literal – if a difference is subtle, it might mention it as well (which is good). There's less dev effort in terms of building retrieval pipelines, etc., so if you have Claude access, it's a straightforward and robust solution. Overall, **Claude is well-suited for structured comparisons** especially if minimal development overhead is desired.

- **Google Gemini/PaLM (via Vertex AI):** Also highly suitable, particularly because of the huge context window. If you have enterprise data on Google Cloud, this could integrate nicely (e.g., you store docs in a Cloud Storage bucket, use Vertex AI to process). The model's multi-modal potential could eventually allow feeding PDFs directly, but assuming text input, it's similar to Claude's proposition: feed everything, get an answer. Google's models have strong multilingual capabilities too, so if your documents had any non-English portions or unusual formatting, it might handle them well. Suitability from a cost perspective will depend on Google's pricing; it might be competitive. Also, with Vertex, you could pipeline other Document AI features (like their knowledge in document parsing). For instance, Google has pretrained models for contract understanding (like identifying parties, dates, etc.). In the future, coupling those with Gemini could yield even more structured comparison (like aligning clauses by their semantic role). But that's an advanced integration. Out-of-the-box, Gemini with 128k context should easily handle two loan docs and produce a reliable comparison. The main barrier could be access (if not generally available yet) and the need to set up on GCP, which is slightly more involved than using OpenAI/Anthropic's public APIs.

- **Meta LLaMA (with RAG or Fine-Tuning):** This approach can absolutely work, but it is **most suitable for scenarios where data privacy or cost control is crucial**. With RAG, a LLaMA-70B (or even 33B) can be fed small relevant chunks and give decent answers. LLaMA's chat-tuned models are pretty good at following instructions, so if you say "compare these provided sections", it will do it. However, compared to GPT-4, it might miss some nuance or not phrase it as clearly. Fine-tuning LLaMA on domain data can increase its suitability: e.g., a fine-tuned LLaMA-70B on legal QA might approach GPT-4 performance on those tasks [16]. This could make it very suitable if you invest that effort. LLaMA-2 (and likely LLaMA-3) being open means you can embed this capability into your app without external calls, which is great for an internal enterprise app (no worries about leaking documents). The trade-off is the heavy lifting on development/infrastructure we discussed. In terms of structured doc comparison, an open model might not inherently know how to structure the output as nicely without prompt help, so you might have to enforce formatting through either prompt or additional code (maybe post-process to ensure the output covers each key field). But if done right, LLaMA is **capable** of the task. So we'd rate its suitability as moderate-to-high for those willing to handle the complexity – technically it can do it, but whether it's the easiest path is another matter.

- **Mistral 7B (with RAG and possibly fine-tune):** Mistral (and similar smaller models) would be the **"lightweight" solution**. Suitability here depends on the requirement of accuracy. If the differences to catch are straightforward (numbers, presence/absence of clauses), a 7B with RAG might suffice. It's actually an intriguing proposition: because RAG will supply the exact sentences to compare, even a smaller model can often correctly point out the difference between those sentences. For example, if Mistral is given: "DocA: Interest rate 5%, DocB: Interest rate 4.5%", it should easily say one is 0.5% higher. Where it might struggle is if the difference is implicit or requires a bit of reasoning ("Doc A's language implies the borrower can extend the term, Doc B's doesn't"). A larger model picks that up better. So, if using Mistral, you might restrict the scope to more direct comparisons or supplement with some rules (like maybe a library for comparing numbers or dates so you don't rely on the model for those). Fine-tuning Mistral on a bunch of Q&A about contract differences could significantly help – essentially teaching it the pattern. Considering Mistral's very low resource needs, it's suitable for deployment in edge or high-throughput environments. For example, if you wanted this feature on a client's device (not sending data to server at all), a 7B model might even run on a powerful smartphone or laptop locally, enabling offline comparisons. That's a niche advantage. Overall, Mistral's suitability is **good for cost-sensitive, privacy-sensitive contexts**, but you sacrifice some accuracy/robustness. It would be a good candidate for a prototype if you want to avoid any external service and have a quick local solution, then maybe upgrade to larger model if needed.

- **Prompt Engineering (alone):** Relying on prompt engineering alone (with a good model) is suitable to a point. If documents are short and you can include them entirely in prompt, then just a well-crafted prompt to GPT-4 or similar might do the job directly. This is the simplest approach (just one API call with a prompt containing both docs and an instruction). Suitability drops if documents are long (past context) – you then need to combine with RAG. Prompt tweaks can also enforce structure, which is nice (you can make the model output a table of differences, etc.). However, prompt engineering can't fully solve issues like hallucination if the model lacks info, nor can it let a model process more text than its limit. So prompt engineering is **highly suitable as a first attempt or for smaller-scale comparisons**, but likely needs to be combined with other techniques for reliability on bigger tasks. It's essentially a necessary component in all approaches (you always have to design your prompt), so in that sense, prompt engineering is universally suitable and required.

- **RAG (Retrieval Augmentation):** We find RAG to be one of the **most universally useful techniques** here. It pairs well with almost any model choice. For closed models, it reduces token usage (you feed only relevant bits instead of whole docs) and increases accuracy by grounding answers [26]. For open models, it is almost mandatory to handle long docs. RAG is very suitable because loan documents have a clear structure – you can often keyword search or semantically search for sections like "Interest", "Repayment", etc. Those relevant parts are easy to retrieve. RAG ensures the model does *not forget to mention an important difference* – because if you specifically retrieve the clause about prepayment from both, the model will definitely address it, whereas if you just gave the whole docs, the model might skim and possibly overlook something. So RAG plus an LLM is a robust combo to systematically cover each aspect. The only scenario where RAG might be less needed is if you have a huge context model (Claude/Gemini) that can ingest everything – even then, one could argue that RAG would save cost by not sending everything every time. So yes, RAG is extremely suitable for structured comparison tasks as it aligns well with the need to compare corresponding sections side by side.

- **Agent-Based Approach:** Agents are **suitable when the comparison needs to be highly granular or interactive**. If the goal is to mimic how an analyst would go through the documents step by step, an agent can do that. For example, an agent could produce a difference report section by section, and if the user says "tell me more about the difference in penalties," the agent can dig back into those sections and elaborate. Agents also excel if the structure of the documents varies a lot – the agent can dynamically find what to compare instead of relying on static prompt instructions. However, if the task is well-bounded (we know we just need interest, term, fee differences), a simpler prompt or retrieval approach might suffice. Agents shine for **complex workflows or when using multiple tools** (like if we incorporate external knowledge or have to fetch updated interest rate benchmarks, etc.). For base document diff, it might be overkill. So I'd say agent approach is moderately suitable – it's powerful but only necessary if you need that level of detail or automation. For a mostly developer audience, implementing an agent might be a great learning experiment, but one must weigh the maintenance overhead.

- **Embeddings for Section Alignment:** Using embeddings to align and compare sections is very suitable in a structured doc scenario. Loan documents likely have overlapping sections (both have “Interest Rate” clauses, etc.), though wording differs. This technique can ensure nothing is missed and can even drive a UI (like highlight each clause and mark changed ones). It’s not a complete solution on its own (since after aligning, you still need to describe differences, which an LLM or simple diff algorithm can do), but it’s a **valuable auxiliary method**. It’s highly recommended to at least use embeddings to help identify which parts of Document A correspond to which parts of Document B. This can catch, for example, if Document B has an extra clause that Document A doesn’t (it will have no match, flagging it). Without embeddings, an LLM might not mention an extra clause if not asked explicitly. So embeddings make the comparison more **comprehensive**.

- **Fine-Tuning:** Fine-tuning, if done successfully, can make a model extremely well-suited to this task – it could reliably output difference summaries quickly and with less prompting needed. But considering the effort, it’s only worth it if you foresee repeated use at scale or want to deploy the model in a constrained environment (e.g., on a device, you could fine-tune a small model to do this one task very well). For most developer teams, fine-tuning is a later-stage optimization. Initially, using existing models with retrieval will likely meet requirements. Fine-tuning becomes suitable if, say, you find GPT-4 does a good job but you can’t use GPT-4 in production due to cost, so you decide to fine-tune LLaMA on a bunch of examples to approach GPT-4 quality. That could be a viable path. Or if company policy forbids external APIs and the only way to get quality from an internal model is to fine-tune it, then it’s necessary. Fine-tuning also allows controlling model behavior more tightly (maybe you want to ensure it always outputs in a specific format, which it learns via training). Summing up, fine-tuning is **suitable for maximizing performance on structured comparisons** but only if you have the data and resources to do it. Otherwise, techniques like RAG give a lot of benefit without needing to alter the model.

To conclude the discussion, let’s consolidate everything into a **comparison table** that summarizes how each platform or method stacks up on key dimensions relevant to developers (effort, resources, limitations, and suitability):

Option / Method	Dev Effort & Integration	Infrastructure & Resources	Key Limitations	Suitability for Loan Doc Comparison
OpenAI GPT-4 API (prompting)	Low effort – simple API calls; focus on prompt design.	OpenAI hosts model; no infra needed beyond API usage.	Context ~8k/32k tokens (larger 128k model in beta [4]); can be slow; $$ per token; data sent off-site.	**Highly suitable** – excellent reasoning and accuracy; best used with careful prompts or RAG for long docs.

Option / Method	Dev Effort & Integration	Infrastructure & Resources	Key Limitations	Suitability for Loan Doc Comparison
Anthropic Claude 2 (100K)	Low effort – API call; minimal prompt engineering needed for differences.	Anthropic hosts model; handle large prompt sizes.	Somewhat slower on huge inputs (e.g. 20+ sec for 70K tokens [6]); still costs tokens (though cheaper per token for big ctx); 100K access may require approval.	**Highly suitable** – can ingest both docs fully (75K words) [5], yielding comprehensive comparisons; minimal system complexity.
Google Gemini (Vertex AI)	Moderate – set up GCP project, use Vertex AI SDK; integrate with GCP services.	Google-managed; need Vertex AI account; supports huge context (128K+ tokens) [11].	Early access/ preview for newest versions; cost structure tiered by context size [13]; requires commitment to Google Cloud.	**Highly suitable** – massive context window allows one-shot doc comparisons; strong quality, but access and cost must be managed.
Meta LLaMA (open-source)	High – self-host model, manage GPU servers; possibly integrate RAG pipeline.	Requires powerful GPUs (e.g., 70B model on 48GB+ GPU); use optimized runtimes; no per-query cost, but hardware cost.	Context ~4K (needs RAG for long docs); initial model not domain-tuned (fine-tune recommended); significant devops overhead.	**Suitable** for privacy and control – can be fine-tuned to domain for accuracy [29]; with RAG, handles long texts; more effort to deploy and maintain.
Mistral 7B (open-source)	Moderate – easier to host (runs on single GPU/CPU); integrate RAG due to limited context.	Light infra – 7B runs on modest hardware (8–16GB GPU or even CPU); very fast responses.	Lower raw accuracy; may miss subtle differences without fine-tune; context ~4K; requires RAG for anything lengthy.	**Moderately suitable** – good for basic differences (with retrieved context) at low cost; fine-tuning can improve domain skill; ideal when resources or budget are limited.

Option / Method	Dev Effort & Integration	Infrastructure & Resources	Key Limitations	Suitability for Loan Doc Comparison
Prompt Engineering	Low/Moderate – iterative crafting of prompts; no additional systems needed.	No extra infra (just uses the chosen model's endpoint).	Cannot overcome model limits (knowledge or context) by itself; prompts can be brittle across varied inputs.	**Essential** – Effective prompts significantly improve output clarity and correctness; forms the foundation for all approaches (to guide model to compare systematically).
Retrieval-Augmented Gen (RAG)	Moderate – implement embedding, vector DB, query logic (many libraries to assist).	Needs vector storage for embeddings; can be lightweight (in-memory or managed service).	Added system complexity; must ensure good chunking/search so relevant info isn't missed; slight overhead in query latency.	**Very suitable** – greatly reduces hallucination [26] and handles long docs by feeding only relevant text; ensures factual, section-by-section comparison.
Agent-Based System (LLM + Tools)	High – set up multi-step reasoning, define tools (doc reader, calculator, etc.), and handle agent outputs.	Runs on top of model (multiple calls); may require additional APIs (for tools like OCR, calc); monitor agent loop.	More points of failure (tool misuse, long execution); higher latency from multiple steps; complex to test/debug.	**Situationally suitable** – excels for complex, structured workflows or interactive analysis; ensures thoroughness (the agent can check each section, use calculator for numbers, etc.), but likely overkill for straightforward comparisons.
Embeddings & Semantic Search	Moderate – use pre-trained embedding model; index documents; straightforward with existing libraries.	Minimal – can even run embeddings on CPU for small data; vector DB optional for small scale (simple cosine sim computation).	Not a standalone solution – needs to feed into an LLM or algorithm for the actual comparison output; might not detect very fine-grained differences on its own (e.g., numeric changes).	**Highly suitable** (as auxiliary) – aligns corresponding sections between docs, guiding the LLM to compare "like with like"; improves coverage and structure of comparisons.

Option / Method	Dev Effort & Integration	Infrastructure & Resources	Key Limitations	Suitability for Loan Doc Comparison
Fine-Tuning an LLM	High – requires training data prep, running fine-tune jobs, and evaluation; ML expertise needed.	Significant compute for training (especially large models); once trained, inference infra similar to hosting open model.	Data dependent – if training set isn't comprehensive, model may not generalize; retraining needed for new document types; longer development cycle.	**Potentially very suitable** – yields a model specialized in loan text, giving fast and consistent results (e.g., a fine-tuned model can directly output a formatted comparison) [29]. Best applied when volume is high enough to justify investment, or when using open models that need domain adaptation.

Table: Comparison of GenAI Platforms and Techniques – Effort, infrastructure, limitations, and fit for the loan document comparison use case.

In conclusion, developing an intelligent loan document comparison feature involves a mix of these GenAI options. For **fast initial results** with minimal engineering, using a powerful API model (like GPT-4 or Claude) with well-crafted prompts (and perhaps some retrieval to handle length) is a good strategy. As requirements on privacy, scale, or cost evolve, one can incorporate **retrieval augmentation** (to keep outputs factual and handle larger docs) and consider **open-source models** (fine-tuned or with agents) to gain more control. Ultimately, the best solution may combine multiple techniques: e.g., using embeddings to align document sections, RAG to feed an LLM those sections, and prompt engineering to format the output, with an agent or fine-tuned model as a future enhancement for added reliability. By understanding the strengths and trade-offs of each approach – as detailed above – developers can select the option (or hybrid approach) that best meets their project's needs for accuracy, efficiency, and maintainability. [6] [26]

---

[1] How to Build a Document Comparison AI Agent
https://www.stack-ai.com/blog/how-to-build-a-document-comparison-ai-agent

[2] [24] How to use ChatGPT to compare two documents (with prompts)
https://www.draftable.com/draftable-legal-blog/how-to-use-chatgpt-compare-two-documents-with-prompts

[3] [4] How much does GPT-4 cost? | OpenAI Help Center
https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost

[5] [6] [7] Introducing 100K Context Windows \ Anthropic
https://www.anthropic.com/news/100k-context-windows

[8] Pricing - Anthropic API
https://docs.anthropic.com/en/docs/about-claude/pricing

[9] AWS Marketplace: Claude (100K) (Amazon Bedrock Edition)
https://aws.amazon.com/marketplace/pp/prodview-bkpmgfnbvoexm

[10] Testing Anthropic Claude's 100k-token window on SEC 10-K Filings
https://www.llamaindex.ai/blog/testing-anthropic-claudes-100k-token-window-on-sec-10-k-filings-473310c20dba

[11] [13] Introducing Gemini 1.5, Google's next-generation AI model
https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/

[12] Gemini (language model) - Wikipedia
https://en.wikipedia.org/wiki/Gemini_(language_model)

[14] Gemini gets more personal, proactive and powerful - Google Blog
https://blog.google/products/gemini/gemini-app-updates-io-2025/

[15] Release notes | Gemini API | Google AI for Developers
https://ai.google.dev/gemini-api/docs/changelog

[16] [18] Papers Explained 187a: Llama 3. Llama 3 is a new set of foundation... | by Ritvik Rastogi | Medium
https://ritvik19.medium.com/papers-explained-187a-llama-3-51e2b90f63bb

[17] Llama 2 vs. GPT-4: Nearly As Accurate and 30X Cheaper - Anyscale
https://www.anyscale.com/blog/llama-2-is-about-as-factually-accurate-as-gpt-4-for-summaries-and-is-30x-cheaper

[19] [20] [21] [22] [23] Mistral 7B | Mistral AI
https://mistral.ai/news/announcing-mistral-7b

[25] [28] Augment LLMs with RAGs or Fine-Tuning | Microsoft Learn
https://learn.microsoft.com/en-us/azure/developer/ai/augment-llm-rag-fine-tuning

[26] RAG Hallucination: What is It and How to Avoid It
https://www.k2view.com/blog/rag-hallucination/

[27] [29] RAG vs. Fine-tuning | IBM
https://www.ibm.com/think/topics/rag-vs-fine-tuning

[30] [31] RAG vs. fine-tuning
https://www.redhat.com/en/topics/ai/rag-vs-fine-tuning