MetaMesh: Multi-Agent Collaboration for Meta-Prompting

Tawab Safi
Columbia University
New York, NY
as7092@columbia.edu

***Abstract*—Large Language Models (LLMs) offer significant promise for automating contract analysis, yet they often fail to capture critical nuances in lengthy, domain-specific documents. MetaMesh addresses this gap by integrating multi-agent meta-prompting with structured intermediate representations. The proposed framework dynamically generates specialized agents—each guided by a Planner Agent and executed by a Plan Executor—to parse legal contracts into JSON-like schemas. Evaluation on a subset of the Contract Understanding Atticus Dataset (CUAD) reveals that MetaMesh greatly reduces token usage and inference overhead while enabling offline preprocessing. Despite lower accuracy and F1 scores relative to directly feeding raw contracts, our approach demonstrates the potential of collaborative meta-prompting to enhance the interpretability and scalability of LLM-driven document understanding.**

1. Introduction

Large Language Models (LLMs) have shown impressive natural language understanding and generation capabilities, but they struggle with nuanced tasks that require strict adherence to detailed documents, particularly in domains such as legal contracts. Traditional Retrieval-Augmented Generation (RAG) approaches often fail to capture the subtle interpretative requirements of legal language or other complex documents, resulting in inaccurate or incomplete analysis. Additionally, conventional single-agent or single-prompt approaches often struggle to unpack or synthesize complex instruction sets in an iterative manner, limiting overall accuracy and compliance. As LLMs gain traction for enterprise use cases, ranging from utility management in healthcare to legal compliance, the need to ensure meticulous adherence to complex instructions becomes essential.

This paper presents MetaMesh, a multi-agent meta-prompting framework designed to improve contract comprehension and question answering. By dynamically creating specialized LLM-based agents that collaborate on an execution plan, MetaMesh enhances both the depth and accuracy of the final output. Our system leverages structured intermediate representations (e.g., JSON) and agent personas (e.g., IPAgent, NonCompeteAgent) to parse, interpret, and unify contract sections. Validated on yes and no questions sourced from the Contract Understanding Atticus Dataset (CUAD) dataset, our approach demonstrates the potential and usefulness of meta-prompting using multi-agent collaboration in legal context and other domains that require adherence to long-form complex policy, procedure, or legal documents for a downstream task.

2. Related Work

Recent advances in prompt engineering have sought to address the inherent limitations of Large Language Models (LLMs) when processing long-form or domain-specific documents. Early efforts typically rely on *Retrieval-Augmented Generation (RAG)*, in which external documents are appended directly to prompts. While RAG boosts information recall, it often struggles in intricate domains such as legal contracts, where subtle language can critically alter the interpretation of clauses. This drawback has motivated alternative approaches to structure, refine, and validate prompts before final inference.

Meta-prompting has emerged as one strategy to improve LLM performance in these scenarios. In [1], the authors introduced task-agnostic scaffolding to guide LLMs in processing complex instructions, and [2] further explored theoretical underpinnings, showing how meta-prompts can influence an LLM's reasoning patterns. While these works demonstrate enhanced comprehension, they largely operate under single-agent settings, lacking iterative refinement and overlooking the potential of collaboration among multiple specialized agents. Even with advancements like [3], which incorporates meta-prompts to optimize retrieval, directly embedding entire documents into prompts often fails to capture the granular interpretations required for domain-critical tasks.

Simultaneously, there has been growing interest in leveraging *multi-agent systems* to tackle more multi-faceted tasks. AutoGen[4] introduced a conversational framework enabling multiple LLM-based agents to exchange intermediate outputs, thereby improving task completion. Extending this idea, AutoAgents[5] proposed dynamically creating specialized agents with unique roles. However, these methods are mostly focused on code generation tasks and doesn't address strict adherence to dense instruction documents or integrate meta-prompting for iterative prompt refinement.

Building on these foundations, **MetaMesh** merges meta-prompting with multi-agent collaboration and iterative refinement, specifically targeting legal-domain adherence. By introducing structured representations (e.g., JSON), our framework ensures that nuanced language from contracts is neither lost nor misinterpreted. This integrated approach addresses the limitations of both single-agent meta-prompting and general-purpose multi-agent frameworks, offering a more robust solution for high-stakes, detail-oriented tasks.

3. Problem Statement

A persistent challenge in using Large Language Models (LLMs) for document analysis lies in ensuring that the model interprets domain-specific nuances accurately. In legal contracts, for example, subtle variations in clause wording can profoundly impact how questions are answered. Conventional approaches often feed the entire contract directly into an LLM-based analyzer (Route 1 in Figure 1), but this can lead to incomplete or incorrect interpretations due to the sheer length and complexity of the text.

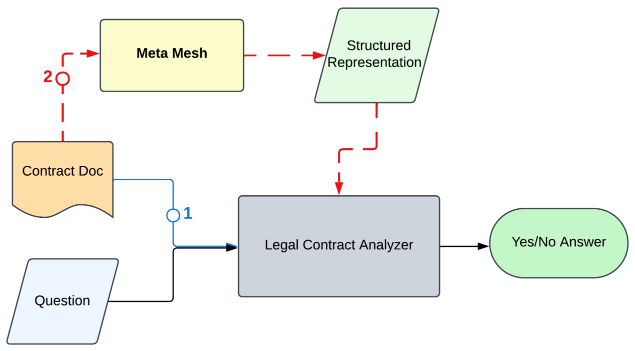


Figure 1: Two routes for question answering over a legal contract. Route 1 (blue) directly feeds the contract to the Legal Contract Analyzer. Route 2 (red) uses the MetaMesh framework to generate a structured intermediate representation before passing the content to the analyzer.

To address this, my work explores whether a structured intermediate representation can improve question answering performance (Route 2 in Figure 1). Specifically, I investigate two key questions:

1) *Can an LLM-based agent better answer legal questions if the contract is provided in a structured format rather than as raw text?*
2) *Can multiple agents collaborate to create a more comprehensive representation of the contract, thereby capturing essential details often lost in a single-pass approach?*

By introducing a multi-agent system (MetaMesh) to transform raw contracts into organized JSON-like structures, I hypothesize that the Legal Contract Analyzer can parse relevant clauses more effectively and generate more accurate yes/no answers. The framework leverages specialized agent personas, each focusing on distinct aspects of the contract, to collaboratively build a richly annotated representation. My overarching goal is to determine whether breaking down and refactoring the contract into a structured format mitigates misunderstanding and allows for more reliable interpretation of contracts and other complex documents for various downstream tasks.

4. Methodology

This section outlines my overall approach for generating intermediate structured representations of legal contracts and subsequently using them for question answering. I divide the methodology into two main parts. In Section 4.1, I describe the *Legal Contract Analyzer Agent*, which can take either contract document or intermediate structured representation of it as input to answer yes/no questions. Section 4.2 then introduces the *MetaMesh Framework*, a multi-agent system that builds these structured representations. Subsections 4.2.1 and 4.2.2 elaborate on the core components of MetaMesh: the *Planner Agent* and the *Plan Executor*.

4.1. Legal Contract Analyzer Agent

The **Legal Contract Analyzer Agent** is responsible for answering yes/no questions about a contract. It supports two input forms: a raw *contract document* or a *structured intermediate representation* (Figure 1). Once it receives one of these formats along with a specific question, it outputs a concise string containing the yes/no answer and a brief rationale.

The inputs to the agent are:

- **Contract or Structured Representation:** The raw contract text or the JSON-like data produced by MetaMesh.
- **Question:** A yes/no query regarding contractual clauses or stipulations.

The agent outputs:

- **Answer:** A textual response ("yes" or "no") followed by a short explanatory statement.

The agent employs a variety of LLM backends, including `gpt-4o`, `gpt-4o-mini`, or local language models through Ollama. The selection depends on availability and cost considerations. The analyzer logs the **processing time** and **token usage** for each contract-question pair, enabling performance tracking across different input formats and model variants.

4.2. MetaMesh Framework

The **MetaMesh Framework** is a multi-agent system designed to transform raw contracts into structured representations for improved question answering. The framework has two key components:

- **Planner Agent:** Creates a structured template and an agent execution plan after reviewing a contract.
- **Plan Executor:** Dynamically spawns specialized agents using the Planner output, ultimately producing the final structured JSON representation.

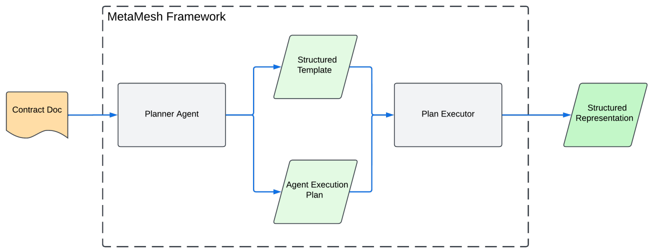


Figure 2: High-level overview of the MetaMesh Framework. The Planner Agent processes the raw contract to generate a structured template and agent execution plan. The Plan Executor uses these artifacts to produce a complete JSON representation.

The overall flow is illustrated in Figure 2. A contract document is fed to the Planner Agent, which generates (i) a hierarchical template describing sections and data points, and (ii) a plan that defines how specialized agents will be invoked. The Plan Executor then iterates through these sections to build the final structured output.

4.2.1. Planner Agent. This agent serves as a meta-level orchestrator, tasked with creating a blueprint for capturing important information from a contract. It parses the document to produce:

1) **Structured Template:** A hierarchical breakdown that best captures all the contract information in named sections and data points.
2) **Agent Execution Plan:** A set of specialized agents, each with a role and instructions to populate a particular section of the template.

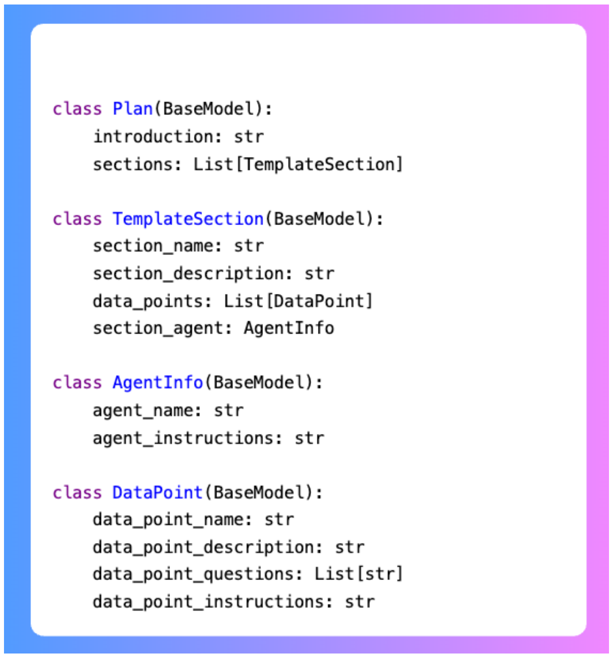
The Planner's prompt instructs the model (either `o1-preview` or `gpt-4o`) to identify critical components of the contract. It then designs a custom template schema composed of *sections* (e.g., IP information, liability clauses) and *data points* (e.g., renewal dates, confidentiality statements). Each section is associated with one specialized agent persona.

The planner returns a structured JSON-like response, conforming to our `Plan` and `TemplateSection` *pydantic* classes (Figure 3). Each `TemplateSection` has:

- `section_name` and `section_description` describing the contract segment.
- A list of `DataPoint` objects, each containing:
  - `data_point_name`, `data_point_description`
  - `data_point_questions`: a list of guiding questions
  - `data_point_instructions`: instructions on how to parse the data
- `section_agent` specifying the `agent_name` and `agent_instructions`.

The Agent Execution Plan is the part of the plan that provides agent-specific instructions along data-specific instructions and questions that can be used to guide the execution of the agent.

Just like the Legal Contract Analyzer Agent, the Planner logs the token count and processing time for generating each plan. This allows us to compare overhead costs for different contract sizes and model variants.



```
class Plan(BaseModel):
    introduction: str
    sections: List[TemplateSection]

class TemplateSection(BaseModel):
    section_name: str
    section_description: str
    data_points: List[DataPoint]
    section_agent: AgentInfo

class AgentInfo(BaseModel):
    agent_name: str
    agent_instructions: str

class DataPoint(BaseModel):
    data_point_name: str
    data_point_description: str
    data_point_questions: List[str]
    data_point_instructions: str
```

Figure 3: Pydantic models for the Planner Agent's response. Each plan is a hierarchical schema of sections, data points, and specialized agents.

4.2.2. Plan Executor. This takes the `Plan` generated by the Planner Agent and iterates over each `TemplateSection`. For every section, it builds and runs a specialized LLM-based agent. This agent's prompt contains:

- The contract text
- The agent-specific instructions from `section_agent`.
- The data points and guiding questions and instructions defined in the Planner's output.

Figure 4 illustrates the Plan Executor workflow. For each section:

1) **Build Agent:** A role-specific agent prompt is created, embedding the data points and instructions.
2) **Run Agent:** The agent extracts or summarizes the requested information from the contract.

3) **Store Section Result:** The results (including `data_point_answers` and `data_point_verbatim`) are saved locally.
4) **Loop Continuation:** If sections remain in the plan, the executor proceeds to the next section.

Once all sections are processed, the executor compiles a **Structured Representation** (Figure 5) containing each `Section` and populated `DataPoint` fields (e.g., `data_point_overview`, `data_point_answers`, `data_point_verbatim`). This final JSON-like artifact can be used by smaller or specialized models for accurate question answering.

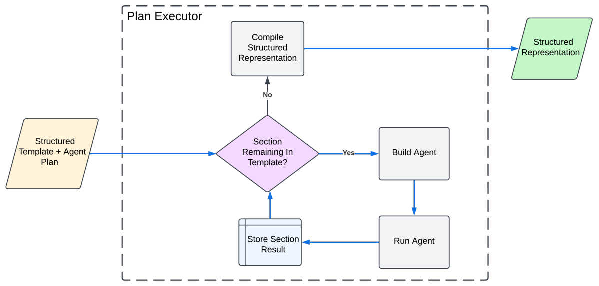
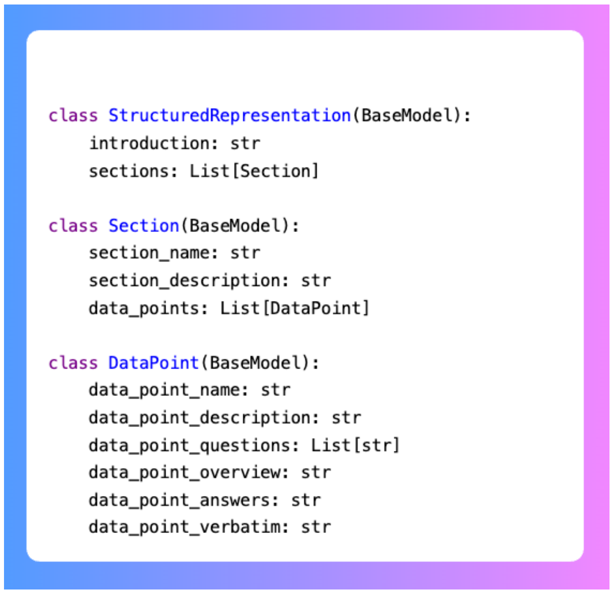

Plan Executor
Compile Structured Representation
Structured Representation
No
Structured Template + Agent Plan
Section Remaining In Template?
Yes
Build Agent
Store Section Result
Run Agent

Figure 4: Plan Executor workflow. Each section in the template triggers agent creation and data extraction until the entire structured contract representation is built.

For each section-agent run, the Plan Executor tracks **token usage** and **execution time**. This allows for detailed analysis of performance trade-offs when scaling the number of specialized agents or varying the LLM backend.



```
class StructuredRepresentation(BaseModel):
    introduction: str
    sections: List[Section]

class Section(BaseModel):
    section_name: str
    section_description: str
    data_points: List[DataPoint]

class DataPoint(BaseModel):
    data_point_name: str
    data_point_description: str
    data_point_questions: List[str]
    data_point_overview: str
    data_point_answers: str
    data_point_verbatim: str
```

Figure 5: Pydantic models representing the final StructuredRepresentation. Each DataPoint now includes `data_point_overview`, `data_point_answers`, and `data_point_verbatim` for richer semantic and verbatim coverage.

The structured output serves as a contract "blueprint" that smaller or domain-focused agents can query to answer yes/no questions more effectively than raw text-based prompts. This multi-agent synergy is a central feature of MetaMesh, aiming to improve interpretability and reduce inference overhead.

5. Evaluation

In this section, I describe the dataset and metrics used to assess the performance of the MetaMesh framework, followed by a detailed analysis of the experimental results. I compare *baseline* contract question answering (using only contract document) against three representation variants created by MetaMesh, each differing in the LLMs used by the Planner Agent and Plan Executor.

5.1. Evaluation Dataset

The evaluation dataset is derived from the *Contract Understanding Atticus Dataset (CUAD)*, which contains over 500 real-world legal contracts and 31 yes/no questions per contract. Due to limited OpenAI API budget and compute resources, I sampled a subset of 20 contracts using the following procedure:

1) **Initial Scoring:** I ran a Legal Contract Analyzer agent (LLaMA 2:13B) on the full dataset to collect yes/no predictions for each contract-question pair.
2) **Performance Ranking:** Based on the predicted answers, I computed F1, precision, recall, and accuracy metrics per contract and derived a *weighted performance score*.
3) **Distribution Constraints:** I only considered contracts where the distribution of yes/no answers was balanced enough ($\geq 35\%$ yes, $\leq 65\%$ no). This guards against highly skewed question-answer distributions.
4) **Contract Selection:**
    a) *Five worst-performing contracts* (lowest weighted performance score meeting the distribution criteria).
    b) *Fifteen average-performing contracts* (scores near the median, also meeting the distribution criteria).

This procedure yielded a final set of 20 contract files, each paired with 31 yes/no questions. While these contracts may not comprehensively reflect the entire CUAD dataset, they give a diverse sample of challenging yet not overly skewed examples to evaluate our MetaMesh approach in resource-constrained settings.

5.2. Evaluation Results

I evaluated using the Legal Contract Analyzer Agent with `gpt-4o-mini` across four input types:

- **Baseline:** Directly feeding the raw contract to the analyzer.
- **Version A:** Intermediate representation generated by *Planner* (`o1-preview`) + *Plan Executor* (`gpt-4o`).
- **Version B:** Intermediate representation generated by *Planner* (`o1-preview`) + *Plan Executor* (`gpt-4o-mini`).
- **Version C:** Intermediate representation generated by *Planner* (`gpt-4o`) + *Plan Executor* (`gpt-4o`).

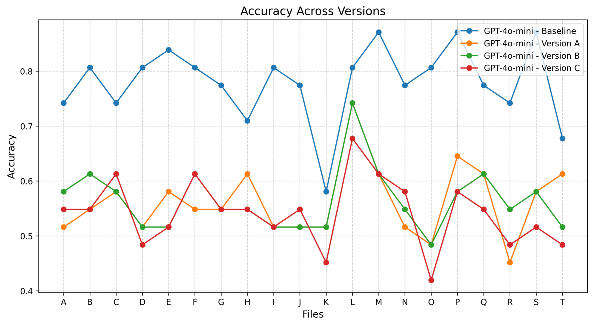


Figure 6: Accuracy across 20 contract files. Baseline (blue) often outperforms MetaMesh versions (A, B, C).

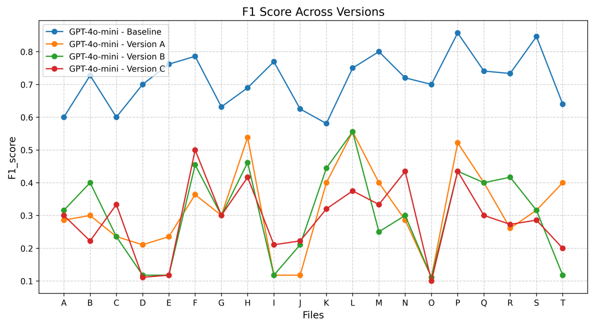


Figure 7: F1 Score across 20 contract files. Baseline again shows higher overall scores than the MetaMesh variants.

5.2.1. Accuracy and F1 Score. Figures 6 and 7 compare *accuracy* and *F1 score* for the four input modes across the 20 selected contracts. The baseline (direct raw text) generally achieves higher accuracy and F1, and the MetaMesh versions underperform on this subset. I hypothesize that the **structured intermediate representation** may omit or oversimplify crucial contract nuances. This gap highlights potential improvements in the MetaMesh Framework that are discussed in the Future Work section.

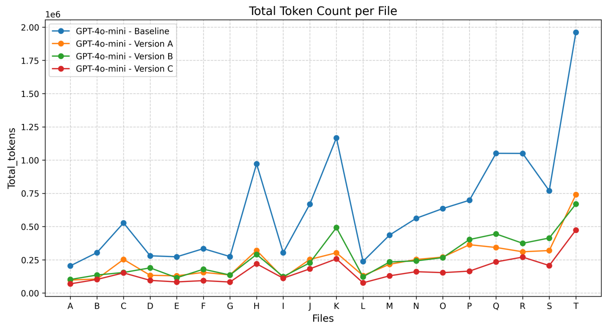


Figure 8: Total token usage across 31 questions per file. MetaMesh approaches (A, B, C) include tokens from both question answering *and* building the structured representation.

5.2.2. Token Usage. Interestingly, despite the additional overhead of generating an intermediate representation, all MetaMesh versions show a **significant reduction in total token usage** relative to the baseline (Figure 8). Note that I combined the tokens consumed by the Planner Agent, Plan Executor, and the question-answer stage for each of the 20 contracts. Even with these combined costs, the structured approach often uses fewer tokens overall than feeding raw documents directly. This finding suggests that well-designed intermediate representations can reduce inference overhead by narrowing the "effective context" an LLM must process.

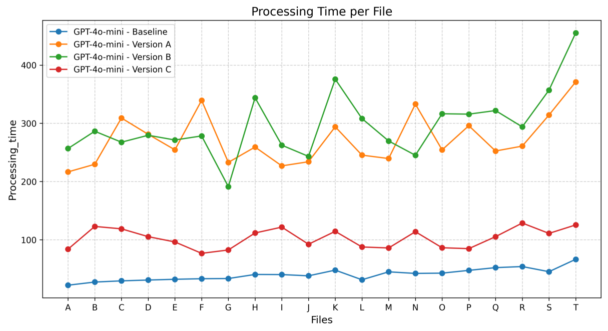


Figure 9: Total processing time per file for baseline and MetaMesh (A, B, C). MetaMesh includes time for plan creation and agent execution.

5.2.3. Processing Time and Latency. Figure 9 presents the total processing time (in seconds) per contract when combining *both* the question answering phase and the overhead for generating the intermediate representation. As expected, MetaMesh's multi-agent workflow (versions A, B, C) introduces additional latency, surpassing the baseline.

However, many real-world use cases (e.g., policy compliance checks, contract review) allow for **offline preprocessing**, where the structured representation is generated once and then **reused** for multiple queries. If the Plan Executor's overhead is amortized, question answering on structured data alone can be faster. Figure 10 indicate that once the representation is stored, the inference time per question is significantly lower compared to the baseline.

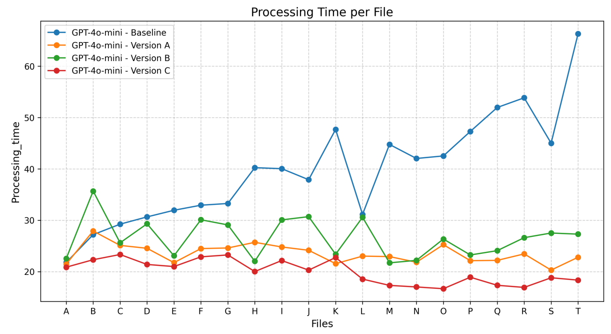


Figure 10: Total processing time per file for baseline and MetaMesh (A, B, C). MetaMesh only includes inference time.

6. Future Work

While MetaMesh shows promise for reducing token usage, improve latency during inference and potentially improving comprehension in certain scenarios, several enhancements are needed to bridge the accuracy gap observed in our evaluation. First, **self-evaluation** and **collaborative evaluation** mechanisms could be integrated into both the Planner Agent and Plan Executor. Allowing each specialized agent to iteratively reflect on its extracted data—akin to "LLMs as their own evaluators"—may improve data fidelity in the final structured representation. Empowering the Planner with meaningful feedback loops could also lead to more representative templates and agent execution plans.

Future extensions include evolving the **multi-agent environment** from a linear pipeline to one where agents dynamically collaborate or refine each other's outputs across multiple sections. Such an approach could close coverage gaps and capture interdependencies between contract clauses. Additionally, supporting alternative representation formats (e.g., knowledge graphs, dataframes) in the Planner would accommodate diverse document types and domain needs. Finally, I plan to conduct detailed analyses using **local models** to assess how well smaller or open-source LLMs benefit from structured representations, potentially broadening the real-world applicability of MetaMesh beyond the constraints of commercial APIs.

7. Conclusion

In this work, I introduced **MetaMesh**, a multi-agent meta-prompting framework designed to improve document comprehension by generating structured intermediate representations. By orchestrating specialized agents, each focused on distinct sections of a contract, MetaMesh reduces inference overhead and streamlines question answering. While our evaluation showed that the baseline typically outperforms MetaMesh in accuracy and F1 score, the significantly lower token consumption and potential for offline preprocessing underscore the system's efficiency advantages. My future work will explore improved template designs, self-evaluation strategies, and richer agent collaboration to bridge the observed performance gap, ultimately broadening MetaMesh's utility across a range of domain-specific tasks.

8. References

[1] M. Suzgun and A. T. Kalai, "Meta-prompting: Enhancing language models with task-agnostic scaffolding," 2024. [Online]. Available: https://www.semanticscholar.org/reader/59eeb8a259ef2a9c981868470480f53b67854060.

[2] A. Wynter, X. Wang, Q. Gu, and S.-Q. Chen, "On meta prompting," *arXiv preprint arXiv:2312.06562*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.06562.

[3] J. Rodrigues and A. T. Branco, "Meta-prompting optimized retrieval-augmented generation," 2024. [Online]. Available: https://www.semanticscholar.org/reader/20587151ec740dcc0b0910783b868938a60cd7d0.

[4] Q. Wu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023. [Online]. Available: https://arxiv.org/abs/2308.08155.

[5] G. Chen, S. Dong, Y. Shu, *et al.*, "Autoagents: A framework for automatic agent generation," *arXiv preprint arXiv:2309.17288*, 2024. [Online]. Available: https://arxiv.org/abs/2309.17288.