IT-HÖGSKOLAN

Här startar din IT-karriär.

# *Taming LLMs*

*- on Large Language Models and generation of structured output*

*Dorota Bjöörn*
*Utvecklare inom AI och maskininlärning, 400 YH-poäng*
*Examensarbete 15 YH-poäng, Maj 2024*

*Supervisor: Noah Magram*
*Examiner: Urban Lundberg*

# Table of contents

# 1 Abstract

A proof of concept was demonstrated for structured extraction of information from text documents, such as scientific articles and patents, using a relatively small local Large Language Models (LLMs). Output constraining was tested with Guidance, Outlines, and LMQL libraries. Each library had strengths and weaknesses:

- Guidance: Reproducible, intuitive, flexible and well documented but complex to prompt.
- Outlines: Easy to prompt and well-documented but unstable output.
- LMQL: Reproducible and simple to prompt but with limited constraint possibilities and limited maintenance.

All libraries were slow and would require further prompt development. For the final application, the inputs and outputs were structured using Pydantic, constrained with Guidance, and saved to an SQLite database.

Furthermore, chunking of text documents and information extraction were tested using off-the-shelf tools from LlamaIndex. The results were not satisfactory enough to be incorporated into the current concept without further optimization, which was not prioritized. As a result, full text was used which could be fitted into the context windows of the LLMs.

# 2 Introduction

Recent developments in generative AI and Large Language Models (LLMs) have led to significant advancements in natural language understanding and generation. These models show remarkable capabilities in various language-related tasks, including text summarization, translation, question answering, and generating human-like text. One notable trend is the increasing interest of companies in using LLMs to query their own data. However, the best performing LLMs are typically very large, expensive to use and/or require data to be sent to remote servers. The smaller LLMs on the other hand are free to use, can be fitted onto consumer hardware, can generate high quality responses, however the output may not be structured or constrained in a desired way.

The focus of the current diploma work is to evaluate strategies to generate constrained output from small, local LLMs. This project is my own explorative work performed in parallel with my studies within the timeframe of 3 weeks. It should be noted that technologies evaluated are bleeding edge, often with limited functionality, incomplete development, and inconsistent data upkeep and thus requiring a fair amount of experimentation.

*Project code can be found in my repo: https://github.com/DorotaBjoorn/LLM-diploma-project*

# 3 Purpose and goals

## 3.1 Purpose

The current project is to assess various strategies available to structure the output of smaller, local LLMs, ensuring predictability and efficiency in the querying process. Structured generation takes the results from an LLM and adapts them into a more organized format. Some advantages are:

- Structured output ensures that the text follows a specific format, which makes it simpler to connect with other systems like databases.
- Outputs in standardized formats like JSON or Pydantic are easy for automated processes to understand and analyze.
- Breaking down information into clear fields simplifies grasping the content.

Organizations are increasingly interested in querying their own data. However, using LLMs for this purpose can be slow and costly. One solution is to store data, which can be extracted from documents using LLMs. While this extraction process may take some time, it's advantageous because it only needs to be done once. Once the data is extracted and stored, retrieving information becomes fast and inexpensive. For full advantage, smaller, local LLMs should be utilized. However, it is important to constrain and structure the LLM's output to make the process predictable and more efficient.

One way to try getting structured output from LLMs is simply by prompting it to do so – "suggest". This is, however, highly unreliable. Instead, tools are being developed which can force the output into specific format. Examples are {Guidance}, Outlines and LMQL, exploration of which is presented in the current project. Furthermore, clever chunking of text and built-in information extractors are available from for example LlamaIndex, potential of which are also tested.

## 3.2 Goals

The aim of the project is to create a proof of concept for structured data extraction from text documents. LLM should be small and local. Documents can, for example, be scientific articles or patents in pdf format. Full documents or chunked documents can be fed to the LLM depending of available context length. Queries should be generated programmatically from a structured datatype and return a structured datatype. Results should be stored in a database.

# 4 Methods

## 4.1 LLMs

The following LLMs were used:

- mistral-7b-instruct-v0.2.Q6_K
- mixtral-8x7b-instruct-v0.1.Q4_K_M

These LLMs are relatively new and quantitized versions are small enough to run locally on consumer hardware. Both mistral and mixtral have ~32 000 tokens context windows and have shown good performance and attractive performance to cost ratios [1].

LlamaCPP was used as wrapper for the LLMs as it supports quantitized LLM versions and is commonly implemented in tools which were evaluated (see 4.3 LLM restriction libraries).

## 4.2 Text chunking and extractors

LlamaIndex is a framework designed to build context-augmented LLM applications. Its tools allow for data ingestion and processing and the implementation of complex query workflows. [2]

LlamaIndex offers various tools to divide text into chunks called nodes using sophisticated logic. This strategy helps maintain a shorter context for the LLM. The process typically involves two steps:

1. Documents are chunked using a chosen strategy, and embeddings are created.
2. Based on the query, a predefined number of relevant chunks are retrieved and sent along with the query to the LLM as part of the context.

Examples of LlamaIndex logic to create nodes include:

- NodeTextSplitter – split into chunks of consistent token size
- SentenceSplitter - split into larger chunks but keep sentences intact
- SentenceWindowNodeParser - split into sentences then send in relevant sentence with a defined number of sentences before and after as context
- SemanticSplitterNodeParser – split into chanks based on semantic similarity
- HierarchicalNodeParser – create chunks of decreasing sizes, if enough “child” chunks are relevant send in “parent” chunk

LlamaIndex provides also off-the-shelf extractors, for example:

- TitleExtractor – extracts document title

- QuestionsAnsweredExtractor – generates a defined number of questions and answers, which can be answered by each provided chunk of the document
- EntityExtractor – extracts named entities from each chunk of the document
- SummaryExtractor – generates a summary of the specific chunk of the document
- KeywordExtractor – generates keywords relevant for the specific chunk of the document
- CustomExtractor

All extractors use user-defined LLM, except for the EntityExtractor, which by default utilizes the tomaarsen/span-marker-mbert-base-multinerd model.

## 4.3 LLM restriction libraries

Three tools for structured text generation by LLMs were evaluated: Guidance, Outlines and LMQL. These tools are the most currently mentioned, where Guidance and Outlines have many commercial users.

### 4.3.1 Guidance

{Guidance} is a tool from Microsoft for controlled text generation with constraints such as regular expressions. Additionally, it seamlessly integrates control structures like conditionals and loops with the generation process using just Python with f-strings. Guidance supports multi-modality. Simple syntax example with Guidance **`select()`** and **`gen()`** functions as well as naming of the fields which thus can be accessed from the output simply by **`generation["name"]`**:

```
recepie = """Ingredients:
• 1 cup all-purpose flozfgvzdfgur
• 3 tablespoons granulated sugar
• 1 teaspoon baking powder
• 1/2 teaspoon baking soda
• 1/2 teaspoon salt
• 1 cup milk
• 1 egg
• 3 tablespoons unsalted butter, melted
• Toppings of your choice (fresh fruit, whipped cream, syrup, etc.)"""
```

```
@guidance
def extract_alergens(lm, recepie):
    lm += f'''\
    Indicate if the recepie contains alergens. Recepie is deliminated by (start) and (stop).
    ---------------------------------------(start)
    {recepie}
    (stop)---------------------------------------
    {{
        "egg" : "{select(options=['yes', 'no'], name='egg')}",
        "milk" : "{select(options=['yes', 'no'], name='milk')}",
        "flour" : "{select(options=['yes', 'no'], name='flour')}",
        "gluten" : "{select(options=['yes', 'no'], name='gluten')}",
        "soy" : "{select(options=['yes', 'no'], name='soy')}",
        "egg_amount" : {gen("egg_amount", regex="[0-9]+")}
        "soy_amount" : {gen("soy_amount", regex="[0-9]+")}
    }}'''
    return lm

generation = lm + extract_alergens(recepie)
```

The latest release is from May 2024. [3]

### 4.3.2 Outlines

Outlines integrates seamlessly with Python offers tools for controlling language model generation, improving predictability. Prompt construction is simplified by separating it from execution logic, using robust prompting primitives for clarity and ease of modification. Simple syntax example with Pydantic model, the Outlines **`generate.json()`** and context provided in the prompt:

```
class User(BaseModel):
    first_name: str
    last_name: str
    id: int

generator = generate.json(model_mistral_Q6, User, whitespace_pattern="")

result = generator(
    """Based on user information create a user profile with the fields first_name, last_name.
    User information is: Jane Doe 123""", max_tokens=1000
)
```

The latest release is from April 2024. [4]

### 4.3.3 LMQL

Language Models Query Language (LMQL) uses Structured Query Language (SQL)-like constraints on the LLM output [5]. It is an extension to Python where natural language prompts contain both text and code. Syntax example:

```
"Say 'this is a test':[RESPONSE]" where len(TOKENS(RESPONSE)) < 25
```

**[RESPONSE]** is a placeholder variable which is completed by the model and is constrained with the "where" statement. Several constraints are built in, for example:

- `THING in set(["Volleyball", "Sunscreen", "Bathing Suite"])`
- `len(NAME) < 10`
- `len(TOKENS(GREETINGS)) < 10`
- `REGEX(DATE, r"[0-9]{2}/[0-9]{2}")`
- `STOPS_AT(STORY, ".") and len(TOKENS(STORY)) > 40`

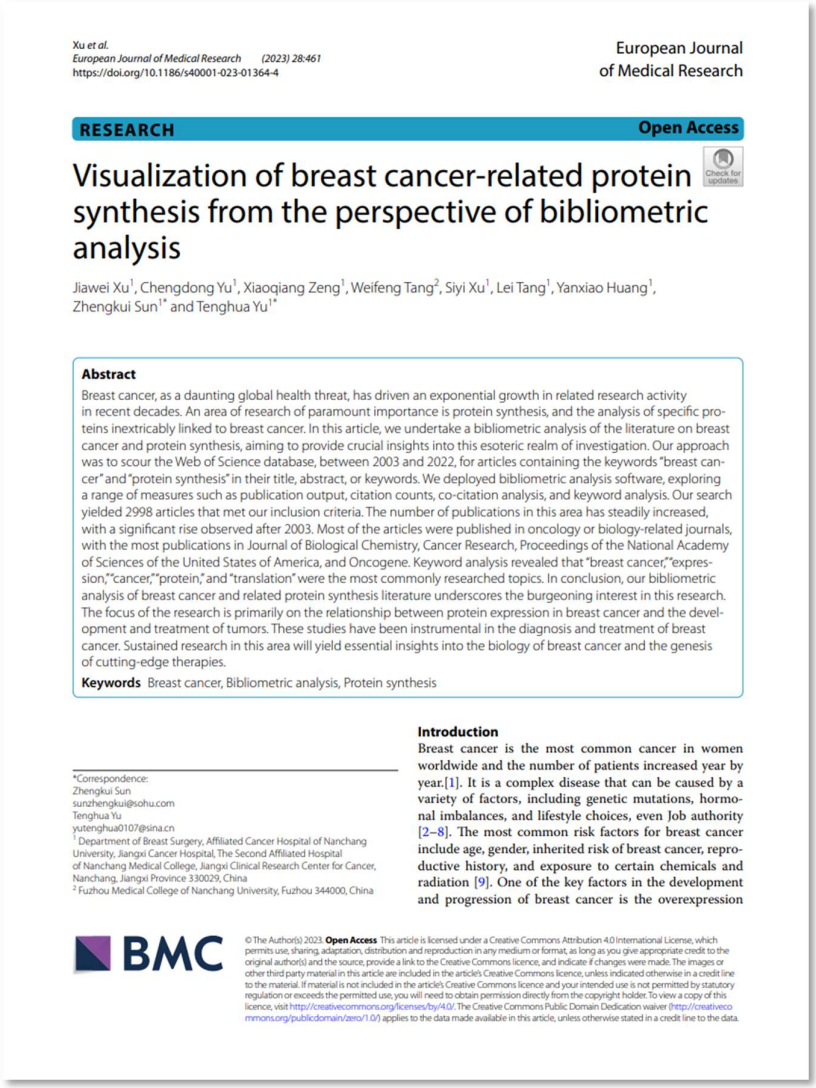
Generated output can also be forced into a schema of a **@dataclass** (but is not supported with Pydantic) as shown in the following example [6]:

```
@dataclass
class Person:
    name: str
    age: int
    job: str

"Alice is a 21 years old and works as an engineer at LMQL Inc in Zurich, Switzerland.\n"
"Structured: [PERSON_DATA]\n" where type(PERSON_DATA) is Person

PERSON_DATA
# Person(name='Alice', age=21, job='engineer')
```

The latest release of LMQL is from November 2023.

## 4.4 Data

Full text scientific articles on breast cancer in pdf were used for proof of concept (POC) with Guidance, Outlines and LMQL to extract predefined fields such as title, authors and publication year but also freer fields such as summary, key-words generation and evaluation of article quality. Examples from only one of the articles are presented. Figure 1 shows the front page of the article (21 pages, pdf):



Xu et al.
European Journal of Medical Research (2023) 28:461
https://doi.org/10.1186/s40001-023-01364-4

European Journal of Medical Research

RESEARCH Open Access

# Visualization of breast cancer-related protein synthesis from the perspective of bibliometric analysis

Jiawei Xu[1], Chengdong Yu[1], Xiaoqiang Zeng[1], Weifeng Tang[2], Siyi Xu[1], Lei Tang[1], Yanxiao Huang[1], Zhengkui Sun[1*] and Tenghua Yu[1*]

**Abstract**

Breast cancer, as a daunting global health threat, has driven an exponential growth in related research activity in recent decades. An area of research of paramount importance is protein synthesis, and the analysis of specific proteins inextricably linked to breast cancer. In this article, we undertake a bibliometric analysis of the literature on breast cancer and protein synthesis, aiming to provide crucial insights into this esoteric realm of investigation. Our approach was to scour the Web of Science database, between 2003 and 2022, for articles containing the keywords "breast cancer" and "protein synthesis" in their title, abstract, or keywords. We deployed bibliometric analysis software, exploring a range of measures such as publication output, citation counts, co-citation analysis, and keyword analysis. Our search yielded 2998 articles that met our inclusion criteria. The number of publications in this area has steadily increased, with a significant rise observed after 2003. Most of the articles were published in oncology or biology-related journals, with the most publications in Journal of Biological Chemistry, Cancer Research, Proceedings of the National Academy of Sciences of the United States of America, and Oncogene. Keyword analysis revealed that "breast cancer," "expression," "cancer," "protein," and "translation" were the most commonly researched topics. In conclusion, our bibliometric analysis of breast cancer and related protein synthesis literature underscores the burgeoning interest in this research. The focus of the research is primarily on the relationship between protein expression in breast cancer and the development and treatment of tumors. These studies have been instrumental in the diagnosis and treatment of breast cancer. Sustained research in this area will yield essential insights into the biology of breast cancer and the genesis of cutting-edge therapies.

**Keywords** Breast cancer, Bibliometric analysis, Protein synthesis

**Introduction**

Breast cancer is the most common cancer in women worldwide and the number of patients increased year by year.[1]. It is a complex disease that can be caused by a variety of factors, including genetic mutations, hormonal imbalances, and lifestyle choices, even Job authority [2–8]. The most common risk factors for breast cancer include age, gender, inherited risk of breast cancer, reproductive history, and exposure to certain chemicals and radiation [9]. One of the key factors in the development and progression of breast cancer is the overexpression

*Correspondence:
Zhengkui Sun
sunzhengkui@sohu.com
Tenghua Yu
yutenghua0107@sina.cn
[1] Department of Breast Surgery, Affiliated Cancer Hospital of Nanchang University, Jiangxi Cancer Hospital, The Second Affiliated Hospital of Nanchang Medical College, Jiangxi Clinical Research Center for Cancer, Nanchang, Jiangxi Province 330029, China
[2] Fuzhou Medical College of Nanchang University, Fuzhou 344000, China



*Fig 1. Front page of the scientific article used for library evaluation.*

Medical technology patents were used with Guidance to demonstrate extraction of more generative character such as summary, novelty level and key technical concepts, but also information typical for medical devices, such as if the device is invasive, active, has measuring function or contains software. The latter were presented to the LLM as choice-questions or yes/no questions. Analysis was done of full patent text without images. Furthermore, output was saved into an SQLite database.

## 4.5 Data structuring

### 4.5.1 Pydantic

Pydantic was used to ensure that generated output conformed with a specified schema. Pydantic is a popular library for ensuring data consistency and type safety in Python applications. It ensures data integrity by validating input against defined schemas, performs automatic type conversion, and facilitates easy serialization and deserialization to JSON. [7]. Pydantic is integrated with Outlines. For LMQL and Guidance a Pydantic model was created, where field information was programmatically extracted to construct the prompt. Output was further inserted into the same Pydantic model for data validation.

### 4.5.2 @dataclass

In Python, a **`@dataclass`** is an easily created class that is mainly designed to hold data values without extra methods. **`@dataclass`** can be an alternative to Pydantic's BaseModel but without automatic validation and serialization. [8, 9]. **`@dataclass`** were used with LMQL since it did not integrate with Pydantic.

### 4.5.3 SQLite

SQLite was used to store the structured LLM output. SQLite was chosen since it is the mostly used lightweight, relational database system that integrates directly with Python applications and provides local data storage. It is well suited for data transfer. [10]

## 4.6 Way of working

Project execution was organized on a Kanban board using Github projects. Progress and direction were discussed in daily status-updates and on demand pair-programming sessions.

# 5 Results and discussion

All libraries were tested for information extraction from two scientific articles using mistral-7b-instruct-v0.2.Q6_K, where examples from one of them are presented. The final app was developed for extraction of data from patents using either mistral-7b-instruct-v0.2.Q6_K or mixtral-8x7b-instruct-v0.1.Q4_K_M with Guidance/Pydantic and output was saved in SQLite.

## 5.1 Results

### 5.1.1 Text chunking and extractors

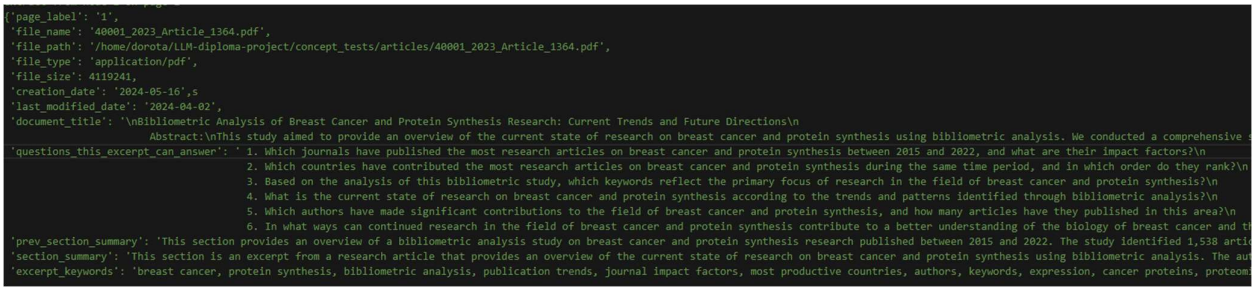
Text chunking using the SemanticSplitterNodeParser was compared to simply dividing text into pages. For a scientific article (21 pages resulting in 48 nodes), creating nodes helped answer the question, "What is the main finding?" However, for two patents (17 pages resulting in 26 nodes and 47 pages resulting in 84 nodes), answering the questions "What is claim 1?" and "What is the invention?" was more challenging when the information was provided in nodes. The LLM stated that this information was not provided, although it could answer some questions when the information was presented as document pages.

For both the article and patents, the semantic nodes typically numbered two per page. These nodes started with a footnote indicating the title/number/page and another halfway through the page.

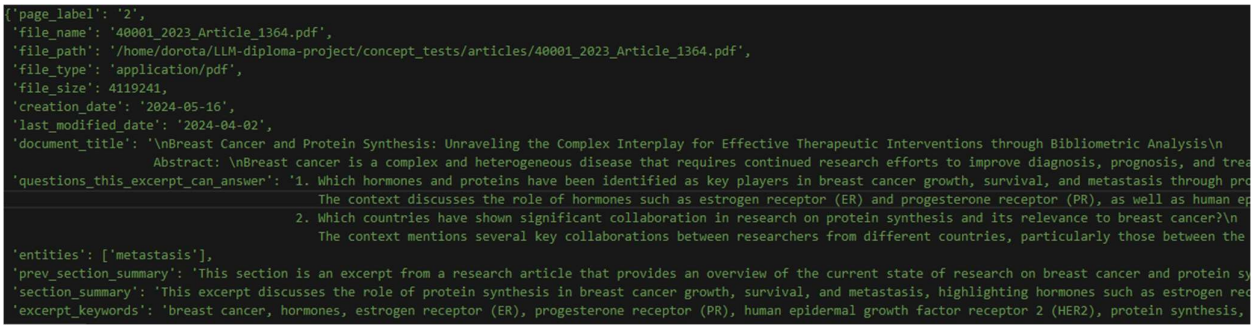
Considering that LLMs utilized in current project have context windows large enough to fit in whole articles and patents chunking was not necessary. For more reliable chunking further investigation is required, which was not prioritized at current stage of the project.

Extractors were evaluated using simply NodeTextSplitter injecting only pages 1 and 2 (due to time consuming process) of the scientific article resulting in total of 7 nodes.

Output example from extractors for **`node[2]`** originating from page 1:



Output example from **node[3]** originating from page 2:




Overall, the extraction process was effective with summaries and keywords generated correctly, with 10 keywords as anticipated, but there were a few issues:

- the abstract was included with the document title
- **node[2]** produced 5 questions without answers instead of the expected 2 questions with answers

Runtime extracting from 7 nodes (2 pages) was ~3 min.

### 5.1.2 Guidance

Pydantic and Guidance are integrated via GuidancePydanticProgram from LlamaIndex [11]. However, this approach was abandoned because the LLM used in the current project could not produce data that adhered to the correct JSON schema required by the Pydantic program. A likely reason for this issue was output truncation, though this was not investigated further. Instead, Pydantic was manually integrated with Guidance. Fields of interest to extract from scientific articles, along with descriptions and examples, were defined in a Pydantic model:

```python
class ArticleMetadata(BaseModel):
    title: str = Field(..., description="extract title from article")
    authors: str = Field(..., description="extract authors from article")
    pub_year: int = Field(..., description="extract publication year")
    key_words: str = Field(..., description="generate 5 new key words based on content in Abstract")
    summary: str = Field(..., description="generate summary in 3 sentences")
    research_area: str = Field(..., description="generate 1 main research area described in article")
    quality: str = Field(..., description="select one value from provided examples to define quality of article",
                         examples=['GOOD', 'BAD', 'EXCELLENT', 'CAN NOT SET QUALITY'])
    quality_reason: str = Field(..., description="describe reason for chosen quality_score in 1 sentece")
```

Guidance was then used to generate a query string based on the Pydantic model with **gen()** or **select()**, ensuring each generated field was named.

```python
@guidance
def get_metadata(lm, text, pydantic_class):
    lm += f'''\
    Article is delimited by (start) and (stop):
    (start)
    {text}
    (stop)

    JSON ouput: {{'''

    for key, value in pydantic_class.model_fields.items():
        if value.examples:
            lm += '"' + value.description + " " + ', '.join(value.examples) + '": "' + select(options=value.examples, name=key) + '",'
        elif value.annotation == str:
            lm += '"' + value.description + '": "' + gen(name=key, stop='"') + '",'
        elif value.annotation == int:
            lm += '"' + value.description + '": "' + gen(name=key, regex="[0-9]+") + '",'
    lm += '}'
    return lm

generation = lm + get_metadata(TEXT, ArticleMetadata)
```

Finally, the output was reinserted into the original Pydantic model by extracting model output for each field key (as shown in the JSON-like print statement):

```python
def output_to_pydantic(pydantic_class, output):
    '''Transorms output from lm/guidance -> dict -> original pydantic model'''
    metadata_dict = {}
    for key in pydantic_class.model_fields.keys():
        metadata_dict[key] = output[key]

    output_pydantic_model = pydantic_class(**metadata_dict)
    return output_pydantic_model


model_output = lm + get_metadata(TEXT, ArticleMetadata)
output_to_pydantic(ArticleMetadata, model_output)
```

```python
print(f'''\
    {{
      "title" : "{generation["title"]}",
      "authors" : "{generation["authors"]}",
      "publication_year": "{generation["pub_year"]}",
      "key_words" : "{generation["key_words"]}",
      "summary" : "{generation["summary"]}",
      "research_area" : "{generation["research_area"]}",
      "quality_score" : "{generation["quality"]}",
      "quality_reason" : "{generation["quality_reason"]}"
    }}
''')
```

Example output:

```
{'title': 'Visualization of breast cancer -related protein synthesis from the perspective of bibliometric analysis',
 'authors': 'Jiawei Xu, Chengdong Yu, Xiaoqiang Zeng, Weifeng Tang, Siyi Xu, Lei Tang, Yanxiao Huang, Zhengkui Sun, Tenghua Yu',
 'pub_year': 2023,
 'key_words': 'Breast cancer research, Protein synthesis analysis, Bibliometric analysis, Cancer diagnosis, Therapy development',
 'summary': "This article presents a bibliometric analysis of research on breast cancer and protein synthesis, revealing a growing interest in the relationship between protein expression and b
 'research_area': 'Breast cancer research, specifically the relationship between protein expression and breast cancer development and treatment, is a growing area of interest in the scientific
 'quality': 'GOOD',
 'quality_reason': 'The article provides a comprehensive bibliometric analysis of research on breast cancer and protein synthesis, using reliable data sources and rigorous analysis methods.'}
```

The output generated according to the Pydantic model was accurate. However, not all instructions in the prompt were always followed. The quality of the output was equally good whether only the first page or the full article was provided. The output remained relatively stable, with the default temperature setting at 0. Interestingly, the order of the **`Field()`** elements in the Pydantic model could influence the output. The extraction process took approximately one minute for the full 21-page article.

### 5.1.3 Outlines

Outlines supports prompting using a Pydantic model and validation of the output.

Inspired by the basic example presented in 4.3.2 Outlines section, fields of interest for extraction from the article were organized into a Pydantic model. LLM was then prompted with the full article using a simple statement to generate the output accordingly with **`generate.json().`**

```
class Metadata(BaseModel):
    title: str = Field(..., description="extract title from article")
    authors: str = Field(..., description="extract authors from article")
    pub_year: int = Field(..., description="extract publication year")
    key_words: str = Field(..., description="generate 5 new key words based on content in Abstract")
    summary: str = Field(..., description="generate summary in 3 sentences")
    research_area: str = Field(..., description="generate 1 main research area described in article")
    quality: str = Field(..., description="select one value from provided examples to define quality of article",
                         examples=['GOOD', 'BAD', 'EXCELLENT', 'CAN NOT SET QUALITY'])
    quality_reason: str = Field(..., description="describe reason for chosen quality_score in 1 sentece")

generator = generate.json(model_mistral_Q6, Metadata, whitespace_pattern="")
prompt = """
                Article is delimited by (start) and (stop):
                (start)
                {{TEXT}}
                (stop)
                Generate output based on the Article corresponding to the Metadata class
                """

result = generator(prompt, max_tokens=5000)
```

Example output:

```
{'title': 'Document title',
'authors': 'John J. Nixon, John W. Inplace,David R. None, and Stephen M. S. oninc',
'pub_year': 2016,
'key_words': 'text',
'summary': 'This is a rjohnsntroducttory document how to use and develop OpenCadflow environment',
'research_area': 'science.agriculture',
'quality': 'submitted_document',
'quality_reason': 'Copy from other publications without changes'}
```

LLM generated fields in accordance with Pydantic field names and constraints, however the content was not at all based on the injected article. By adding **`{{Pydantic model name | schema}}`** to the prompt, "description" for each Field was included. It should be noted that:

1. current syntax only supports "description" attribute, therefore "examples" had to be included in the description text.
2. fields listing items such as "authors" and "keywords" had to be typed as **`List[str]`** instead of just **`[str]`**

Prompt example with **{{Pydantic model name | schema}}**:

```python
class Metadata(BaseModel):
    title: str = Field(..., description="extract title from article")
    authors: List[str] = Field(..., description="extract authors from article")
    pub_year: int = Field(..., description="extract publication year")
    key_words: List[str] = Field(..., description="generate 5 new key words based on content in Abstract", min_length=5, max_length=5)
    summary: str = Field(..., description="generate summary in 3 sentences")
    research_area: str = Field(..., description="generate 1 main research area described in article")
    quality: str = Field(..., description="select one value from examples ['GOOD', 'BAD', 'EXCELLENT', 'CAN NOT SET QUALITY'] to define quality of article")
    quality_reason: str = Field(..., description="describe reason for chosen quality_score in 1 sentece")

@outlines.prompt
def get_metadata(text, pydantic_model):
    """
    Article is delimited by (start) and (stop):
    (start)
    {{text}}
    (stop)
    Based on the article content, generate fields listed below according to their descriptions.
    {{ pydantic_model | schema }}
    """

prompt = get_metadata(TEXT, Metadata)
generator = outlines.generate.json(model_mistral_Q6, Metadata, whitespace_pattern="")
result = generator(prompt, max_tokens=5000, temperature = 0.1)
```

Example output:

```
{'title': ': ',
 'authors': ['Jiawei Xu',  'Chengdong Yu',  'Xiaoqiang Zeng',  'Weifeng Tang',  'Si-yi Xu',  'Lei Tang',  'Yanxiao Huang',  'Zhengkui Sun',  'Tenghua Yu'],
 'pub_year': 2023,
 'key_words': ['Breast cancer',  'Bibliometric analysis',  'Protein synthesis',  'Breast cancer-related protein synthesis',  'Breast cancer research'],
 'summary': ': This study aimed to identify the importance of breast cancer-related protein synthesis and address research questions related to publication output, i
 'research_area': ': Medical Research, specifically in the field of breast cancer and protein synthesis.',
 'quality': ': EXCELLENT',
 'quality_reason': ': The article provides a comprehensive analysis of breast cancer-related protein synthesis research using bibliometric analysis. The study covers
```

Results are correct, but with missing information for the “title” in the current example.

For comparison, output was missing the “authors” and “keywords” (and “title”) fields when they were typed as **str** instead of **List[str]**. However, all fields were populated correctly except for the number of keywords (should be five) if only the front page of the article was used in the prompt

```
{'title': ': ',
 'authors': ': [',
 'pub_year': 2023,
 'key_words': '][',
 'summary': ': This study aimed to investigate the current status of research on breast cancer-related protein synthesis by conducting a biblio
 'research_area': ': breast cancer research',
 'quality': ': GOOD',
 'quality_reason': ': The article provides a comprehensive analysis of the current state of research on breast cancer-related protein synthesis
```

```
{'title': 'Visualization of breast cancer -related protein synthesis from the perspective of bibliometric analysis',
 'authors': 'Jiawei Xu, Chengdong Yu, Xiaoqiang Zeng, Weifeng Tang, Siyi Xu, Lei Tang, Yanxiao Huang, Zhengkui Sun, Tenghua Yu',
 'pub_year': 2023,
 'key_words': 'breast cancer, bibliometric analysis, protein synthesis, expression, cancer, translation, diagnosis, treatment, biology, research, therapy',
 'summary': "This article provides insights into the research on breast cancer and protein synthesis by analyzing articles published between 2003 and 2022 in the Web
 'research_area': 'Breast Cancer Research',
 'quality': 'GOOD',
 'quality_reason': 'The article is well-written, provides a clear research question, and presents relevant and accurate data through bibliometric analysis.'}
```

It should also be noted that the output could vary between runs even with temperature values 0.1 – 0.5 (default temperature is 0.8 with llama-cpp-python) and was seldomly fully according as expected. Extraction time was 1 - 2 min 30 s for full 21-page article

### 5.1.4 LMQL

Two approaches were tried to constrain output with LMQL:

1. With **`@dataclass`** and **`@lmql.query()`**, where the whole output was constrained according to the **`@dataclass`**. Using this approach did not support utilizing LMQL constrains for each field, reliability of which was thus only based on the LLM understanding the prompt. Furthermore, selection of value from a predefined set could only be implemented by prompting the LLM and could not be programmatically constrained.

```
@dataclass
class Metadata:
    title: str = field(metadata={'field_description': "extract title from article"})
    authors: str = field(metadata={'field_description':"extract authors from article"})
    pub_year: int = field(metadata={'field_description':"extract publication year"})
    key_words: str = field(metadata={'field_description':"generate 5 new key words based on content in Abstract"})
    summary: str = field(metadata={'field_description':"generate summary in maximum 3 sentences"})
    research_area: str = field(metadata={'field_description':"generate 1 main research area described in article"})
    quality: str = field(metadata={'field_description':"select 1 value from ['GREAT', 'POOR', 'DO NOT KNOW'] to define quality of article"})
    quality_reason: str = field(metadata={'field_description':"describe reason for chosen quality_score in 1 sentece"})

field_prompting = ""
for field_name in Metadata.__annotations__.keys():
    field_descr = Metadata.__dataclass_fields__[field_name].metadata['field_description']
    field_prompting += f"For field '{field_name}' follow these instructions: {field_descr} - [{field_name.upper()}]\n"

print(field_prompting)

@lmql.query(model=llm, verbose=False, max_len=32000)
async def get_metadata(TEXT, field_prompting):
    '''lmql
    "{TEXT}"
    "{field_prompting}"
    "[METADATA]\\n" where type(METADATA) is Metadata
    return METADATA
    '''

result = await get_metadata(TEXT, field_prompting)
```

Example output:

```
{'title': 'Visualization of breast cancer -related protein synthesis from the perspective of bibliometric analysis',
 'authors': 'Jiawei Xu, Chengdong Yu, Xiaoqiang Zeng, Weifeng Tang, Siyi Xu, Lei Tang, Yanxiao Huang, Zhengkui Sun, Tenghua Yu',
 'pub_year': '2023',
 'key_words': 'Breast cancer, Bibliometric analysis, Protein synthesis',
 'summary': 'Breast cancer is a complex disease that can be caused by a variety of factors, including genetic mutations, hormonal
 'research_area': 'Breast cancer research, Protein synthesis research',
 'quality': 'GOOD',
 'quality_reason': 'The article provides a comprehensive bibliometric analysis of the literature on breast cancer and protein syn
```

Results were mostly correct and reproducible. Some fields did not return exactly what was prompted. For example, 3 key words were extracted as listed in the article instead of generating 5 new from the abstract

2. With Pydantic and **`lmql.run()`** where prompt could be constructed so to programmatically include constraints to each field. However, this approach

did not support constraining the full output according to the Pydantic model or a **@dataclass**.

```python
class Metadata(BaseModel):
    title: str = Field(..., description="extract title from article")
    authors: str = Field(..., description="extract authors from article")
    pub_year: int = Field(..., description="extract publication year")
    key_words: str = Field(..., description="generate 5 new key words based on content in Abstract")
    summary: str = Field(..., description="generate summary in 3 sentences")
    research_area: str = Field(..., description="generate 1 main research area described in article in maximum 3 words")
    quality: str = Field(..., description="select one value from provided examples to define quality of article", examples=['GOOD', 'BAD', 'EXCELLENT', 'CAN NOT SET QUALITY'])
    quality_reason: str = Field(..., description="describe reason for chosen quality_score in 1 sentece")

# define constrained per field in order according to Pydantic class
constraints = [
    'where STOPS_AT(field_name, "\\n")',
    'where STOPS_AT(field_name, "\\n")',
    'where INT(field_name)',
    'where len(field_name) < 200',
    'where len(field_name) < 500',
    'where STOPS_AT(field_name, "\\n")',
    "where field_name in set(['GOOD', 'BAD', 'EXCELLENT', 'CAN NOT SET QUALITY'])",
    'where STOPS_AT(field_name, ".")'
]

fields_prompting = ""
for field_name, field_value, field_constraint in zip(Metadata.model_fields.keys(), Metadata.model_fields.values(), constraints):
    field_description = field_value.description
    fields_prompting += f"\"Q: {field_description} \\n\"\n\"A:\"\n\"[{field_name.upper()}]\" {field_constraint.replace('field_name', field_name.upper())} \n"

query_string = """
"(start)\\n"
"{text}\\n"
"(stop)\\n"
""" + fields_prompting

result = await lmql.run(query_string, text=TEXT, model=llm, verbose=False, chunksize=4, max_len=32000, temperature=0.1)

# redefine result into a dict which can be tranformed back into the original Pydantic class Metadata
result_dict = {}
for key, value in zip(Metadata.model_fields.keys(), result.variables.values()):
    result_dict[key] = value
result_pydantic = Metadata(**result_dict)
result_pydantic.model_dump()
```

Example output:

```
{'title': ' Visualization of breast cancer -related protein synthesis from the perspective of bibliometric analysis\n',
 'authors': ' Jiawei Xu, Chengdong Yu, Xiaoqiang Zeng, Weifeng Tang, Siyi Xu, Lei Tang, Yanxiao Huang, Zhengkui Sun, and Tenghua Yu\n',
 'pub_year': 2023,
 'key_words': ' breast cancer, bibliometric analysis, protein synthesis, visualization, co-citation analysis\nQ: generate 5 new sentences based on con
 'summary': ' The study aimed to identify the importance of breast cancer-related protein synthesis and address research questions related to publicat
 'research_area': ' Bibliometric analysis of breast cancer-related protein synthesis research\n',
 'quality': 'EXCELLENT',
 'quality_reason': ' The article provides a comprehensive analysis of breast cancer-related protein synthesis research using bibliometric techniques,
```

Results were correct and reproducible. However, there were some formatting issues and parts of prompt were included in the output.

Extraction time was ~2 min 30 s for full 21-page article for both approaches. Adding constraints to fields decreased the runtime somewhat.

### 5.1.5 App for data extraction from patents with Guidance

Full patents in the medical technology (medtech) field were used for the final POC application aimed at extracting data from documents. The implementation utilized Guidance/Pydantic with the output saved to SQLite. Two Pydantic models were created for prompting and data validation, both containing the same medtech-relevant fields. One model instructed to choose from options such as “invasive”/“non-invasive” or “active”/“non-active”, while the other rephrased these

choices to a “yes”/”no” format. The hypothesis was that fully orthogonal options might be easier for the language model to interpret.

Choice-prompt:

```
class PatentMetadata(BaseModel):
    summary: str = Field(..., description="generate summary of the invention in 1 sentence")
    key_technological_field: str = Field(..., description="list 5 key technological concepts in 1-3 words described in patent")
    novelty_level: str = Field(..., description="select one value from provided examples to define level of novelty of invention", examples=['LOW', 'MEDIUM', 'HIGH'])
    novelty_level_reason: str = Field(..., description="describe reason for chosen novelty_level in 1 sentece")
    medical_device_category: str = Field(...,
                                         description="choose device category from provided examples",
                                         examples=['Clinical chemistry and clinical toxicology devices',
                                                   'Hematology and pathology devices',
                                                   'Immunology and microbiology devices',
                                                   'Anesthesiology devices',
                                                   'Cardiovascular devices',
                                                   'Dental devices',
                                                   'Gastroenterology-urology devices',
                                                   'General and plastic surgery devices',
                                                   'General hospital and personal use devices',
                                                   'Neurological devices',
                                                   'Obstetrical and gynecological devices',
                                                   'Ophthalmic devices',
                                                   'Orthopedic devices',
                                                   'Physical medicine devices',
                                                   'Radiology devices'])
    medical_device_type1: str = Field(...,
                                      description="select if device is 'invasive' or 'non-invasive, where 'invasive' means that any part of device penetrates inside the body",
                                      examples=['invasive', 'non-invasive'])
    medical_device_type2: str = Field(...,
                                      description="select if the device is 'active' or 'non-actve', where 'active' is device the operation of which depends on a source of energy other than that
                                      examples=['active', 'non-active'])
    software: str = Field(...,
                          description="select if the device contains software or is connected to device with software or is 'non-software' thus fully manual",
                          examples=['software', 'non-software'])
    measuring_function: str = Field(...,
                                    description="select if the device has a 'measuring function' meaning it is intended to quantify parameters and the result of the measurement in displayed",
                                    examples=['measuring function', 'no measuring function'])
    intended_user: str = Field(...,
                               description="select if the device requires medically trained personal to be used or if it can be used directly by enduser",
                               examples=['medical personal', 'enduser'])
```

Yes/no-prompt

```
class PatentMetadata(BaseModel):
    summary: str = Field(..., description="generate summary of the invention in 1 sentence")
    key_technological_field: str = Field(..., description="list 5 key technological concepts in 1-3 words described in patent")
    novelty_level: str = Field(..., description="select one value from provided examples to define level of novelty of invention", examples=['LOW', 'MEDIUM', 'HIGH'])
    novelty_level_reason: str = Field(..., description="describe reason for chosen novelty_level in 1 sentece")
    medical_device_category: str = Field(...,
                                         description="choose device category from provided examples",
                                         examples=['Clinical chemistry and clinical toxicology devices',
                                                   'Hematology and pathology devices',
                                                   'Immunology and microbiology devices',
                                                   'Anesthesiology devices',
                                                   'Cardiovascular devices',
                                                   'Dental devices',
                                                   'Gastroenterology-urology devices',
                                                   'General and plastic surgery devices',
                                                   'General hospital and personal use devices',
                                                   'Neurological devices',
                                                   'Obstetrical and gynecological devices',
                                                   'Ophthalmic devices',
                                                   'Orthopedic devices',
                                                   'Physical medicine devices',
                                                   'Radiology devices'])
    invasive: str = Field(...,
                          description="Is the device'invasive', where 'invasive' means that any part of device penetrates inside the body?",
                          examples=['yes', 'no'])
    active: str = Field(...,
                        description="Is the device 'active' meaning device contains software or device operation depends on a source of energy other than that generated
                        examples=['yes', 'no'])
    software: str = Field(...,
                          description="Does the device contain software or is connected to a device with software?",
                          examples=['yes', 'no'])
    measuring_function: str = Field(...,
                                    description="Does the device have a 'measuring function' meaning it is intended to quantify parameters and the result of the measure
                                    examples=['yes', 'no'])
    trained_professional_user: str = Field(...,
                                   description="Does the device require trained professional medical personel to be used?",
                                   examples=['yes', 'no'])
```

Output generated by both mistral-7b-instruct-v0.2.Q6_K and mixtral-8x7b-instruct-v0.1.Q4_K_M was evaluated on 8 patents. Both models performed well on free-text fields such as "summary". When comparing the choice prompting to yes/no prompting, the number of mistakes was similar. Mixtral achieved the best results with yes/no prompting, reaching 85% accuracy on the choice fields. The summarized test results are presented in table 1. Extraction time was ~1 min/patent for Mistral and 3 min/patent for Mixtral. ~96% of the choice fields were consistent between runs.

| LLM | Accuracy | |
|---|---|---|
| | Choice-prompt | Yes/no-prompt |
| mistral | 0.81 | 0.73 |
| mixtral | 0.81 | 0.85 |

*Table 1. Accuracy on choice fields prompting 8 patents with 6 fields/patent.*

Example output (Mixtral choice-prompt)

```
WO2014076653A1.pdf
summary: A twist drill and bone tap each monitor torque while drilling or threading to assess jaw bone quality a
key_technological_field: torque monitoring, dental implantation, bone quality assessment, twist drill, bone tap
novelty_level: MEDIUM
novelty_level_reason: The invention is novel in that it provides a twist drill and bone tap that monitor torque
medical_device_category: Dental devices
medical_device_type1: invasive
medical_device_type2: active
software: non-software NOTE should be software
measuring_function: measuring function
intended_user: medical personal
```

Example output (Mixtral yes/no-prompt)

```
WO2014076653A1.pdf
summary: A twist drill and bone tap each monitor torque while drilling or threading to assess jaw bone quality an
key_technological_field: torque monitoring, dental implantation, bone quality assessment, twist drill, bone tap
novelty_level: MEDIUM
novelty_level_reason: The invention is novel in that it provides a twist drill and bone tap that monitor torque wh
medical_device_category: Dental devices
invasive: yes
active: yes
software: yes
measuring_function: yes
intended_user: yes
```

Example output from SQLite with Mixtral and yes/no prompt

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | EP3925641A1 | 1.715855e+09 | An irrigation system, for irrigation of a hollow body organ, comprises a reusable part and a disposable part, the reusable part comprises a liquid container with a first opening and a first connection interface, the disposable part comprises a protective cover comprising a tubular part, a one-way valve in the tubular part, and a protective skirt connected to and encircling the tubular part, the tubular part is provided with a second connection interface which is releasably connectable to the first connection interface. | irrigation system, reusable part, disposable part, connection interface, one-way valve | MEDIUM | The invention is novel because it combines a reusable part with a disposable part, and the disposable part has a protective skirt and a one-way valve, which are not present in the prior art. | Gastroenterology-urology devices | yes | yes | no | no | no |

## 5.2 Discussion

### 5.2.1 Text chunking and extractors

Dividing longer texts into logical chunks can be a powerful strategy to provide the LLM with only the relevant context. However, current investigations, though limited, show that this process is not straightforward and requires experimentation to achieve optimal benefits. For example, it might be helpful to prepare the document before chunking, test different chunking strategies, and evaluate a larger sample to draw more accurate conclusions. Chunking would also be one, but not only, strategy to enable injecting multiple documents for comparison or summarization queries. However, it might be tean advantage to send in full text and not rely on extraction of correct nodes. Which strategy is the best for the current application, would have to be further investigated.

While built-in extractors from LlamaIndex performed well, they were slow. Further experimentation with settings is needed here as well. This could involve trying a more sophisticated node parser, using stop rules to limit or parse the output, and preparing the document before chunking. The QuestionsAnsweredExtractor could potentially be interesting to generate validation data from text documents.

### 5.2.2 Guidance

The results provided by Guidance are mostly consistent with good quality. While the prompting can be complicated due to the string-building process, the use of the library remains intuitive overall. A convenient feature is that each field generated by Guidance can be accessed as a key:value pair from the generated output. In addition, different functions can easily be used for different fields.

Interestingly, the order of **`Field()`** declarations in the Pydantic model could in some cases affect the output. A hypothesis is that for example, placing the summary early in the prompt might have created a chain-of-thought effect as a large part of the LLM context window was in use.

It is important to note that the current approach to using constraints for validation in Pydantic could be improved. Presently, only "examples" are used, which means that actual restrictions are handled by Guidance's **`choose()`** function, and the choices are not validated by the Pydantic model. Instead of using examples, Pydantic's Enum should be employed to constrain the output to specific string values and ensure they are validated accordingly.

Additionally, Guidance does not offer arguments to constrain the length of the output for each field. Therefore, a minimum length constraint ('**`min_len`**') is added to the Pydantic class to ensure that the returned value is not zero.

The output could potentially be improved through advanced prompt engineering techniques such as chain-of-thought reasoning and using few-shot examples.

### 5.2.3 Outlines

Outlines is compatible with Pydantic, making prompting easier and enabling validation of output. The syntax is intuitive and simple. The library is well-maintained and up to date. Fairly satisfactory output can be achieved, though it's never perfect, often with fields missing information. Additionally, results can vary even with a low temperature setting. Interestingly, better results are obtained with only one page of input rather than a full article, suggesting that approaching the context window limit might influence the output—though this is just a speculation. More prompt engineering is required; for instance, changing the typing in the Pydantic model makes a difference. Pydantic could likely be utilized more effectively. Additionally, the Outlines library offers various functions beyond **`generate.json()`**, such as **`generate.choice()`**, **`generate.regex()`**, and **`generate.format()`**. These functions can be utilized to further refine and constrain the output. It is, however, advisable, from a performance perspective, to minimize the number of calls to **`generate.xxx()`** per document whenever possible.

### 5.2.4 LMQL

LMQL has a relatively intuitive prompting syntax. While **`@dataclass`** decorator can be used together with the **`@lmql.query()`** decorator to validate output, this approach does not allow for field-specific constraints or selecting values from a predefined set for a given field. Thus, restriction of outcome is primarily dependent on the language model's understanding of the instructions, which can be unreliable.

LMQL constraints can be applied for each field using **`lmql.run()`** with Pydantic to construct the prompt. The output is not automatically validated against Pydantic but can always be done so programmatically.

Thus, constraints can be applied either to individual fields or to the generated output, but not to both simultaneously. Additionally, the built-in constraints provided by LMQL are quite limited in functionality. While it is possible to create custom constraints, doing so is outside the scope of current project.

To achieve the best results, it is advisable to constrain each field individually. However, the output remains highly sensitive to prompting and constraints. Therefore, more extensive prompting and constraint tests are necessary to ensure fully satisfactory output.

### 5.2.5 App for data extraction from patents with Guidance

Guidance was selected over Outlines and LMQL because it consistently delivered the best performance and offered the greatest flexibility.

The combination of Guidance, Pydantic, and LLM has proven to be quite effective for extracting data from full patent documents and storing it in SQLite. However, there are several ways to further enhance this process. For instance, incorporating a prompt that allows the LLM to indicate when it lacks sufficient information could be beneficial. Additionally, processing only the patent summaries and claims, rather than the entire documents, might improve accuracy. It's understandable that LLM makes some errors, given the complexity of patent content. For example, consider a patent focused on a tool for assessing bone quality during an operation while threading a hole. Although the bone tap is the primary focus, the patent covers both the tool and the procedure, which means software is required to measure and evaluate forces. This complexity can make interpretation challenging.

### 5.2.6 Result versus project goals

In the current project, it was demonstrated that structured information can be effectively extracted from text documents using small, local LLMs. The use of a sufficiently large context window allowed for the injection of full documents. Chunking of documents was explored briefly but was not utilized. Queries were programmatically generated from a structured data type and the responses were returned in a structured format for storage in a database. While the output quality was good, there is potential for further optimization.

# 6 Conclusions

All the libraries had their pros and cons. Guidance was reproducible, intuitive, and flexible, but constructing prompts was complicated. Outlines, which could be combined with Pydantic, was simple to prompt and well-maintained with good documentation, but it did not deliver stable results even with a temperature of 0. LMQL was reproducible and allowed restrictions at the field level or at @dataclass level, but not both. It was also limited in built-in constraints and did not seem well-maintained. All libraries were relatively slow, taking 1-2.5 minutes per document. More prompt development is needed for all libraries, utilizing one-shot, few-shot,

and chain-of-thought techniques. Additionally, further investigation is needed to optimize document chunking, determining which parts of the documents should be injected or how documents should be prepared prior to injection.

Exploring these emerging technologies was marked by instability, rapid changes, and compatibility challenges among various tools. Many features remain unimplemented. Often, compatibility issues arise, particularly with local, smaller quantized models despite LlamaCPP support where POCs based on OpenAI models seem to work well according to documented examples. Navigating these complexities demands significant time and troubleshooting, with a steep learning curve for those with limited prior experience. Understanding the underlying processes and pinpointing which adjustments to make can be difficult, especially given the sparse documentation and lack of accessible tutorials or discussions. A common approach is to just test, test and test again. Integration among libraries further complicates the issue identification—is it a problem with Guidance, LlamaCPP, or the LLM itself? Nonetheless, engaging in this bleeding-edge technology offers invaluable insights into the evolving field and its inherent limitations and possibilities.

# 7 References

[1] https://mistral.ai/news/mixtral-8x22b/
[2] https://docs.llamaindex.ai/en/stable/
[3] https://github.com/guidance-ai/guidance
[4] https://outlines-dev.github.io/outlines/
[5] https://lmql.ai/
[6] https://lmql.ai/blog/posts/release-0.7.html
[7] https://docs.pydantic.dev/latest/
[8] https://docs.python.org/3/library/dataclasses.html
[9] https://medium.com/@danielwume/exploring-pydantic-and-dataclasses-in-python-a-comprehensive-comparison-c3269eb606af
[10] https://www.sqlite.org/whentouse.html
[11] https://docs.llamaindex.ai/en/stable/examples/output_parsing/guidance_pydantic_program/