LEUPHANA
UNIVERSITÄT LÜNEBURG

Data Wizard: An Embeddable and Reusable Approach for Structured Data Extraction Using Large Language Models

Lukas Mateffy

Bachelor Thesis

University:	*Leuphana University Lüneburg*
Field of Study:	*Business Information Systems*
Matriculation Number:	*3038817*
Primary Examiner:	*Prof. Dr. Rer. Nat. Guido Barbian*
Secondary Examiner:	*Prof. Dr. Jan Wilk*

Abstract

In an era defined by extensive digitalization, the challenge of efficiently extracting structured information from unstructured digital documents remains a significant bottleneck for both people and organizations. Despite advancements in digital document formats, the embedded data often remains inaccessible to automated systems, necessitating time-consuming and error-prone manual data entry. This thesis introduces Data Wizard, a novel approach to address this challenge: a reusable and embeddable tool for structured data extraction leveraging the power of Large Language Models (LLMs). It aims to answer the question whether a solution can be built that is flexible enough to be useful in a variety of contexts while remaining simple and easy to work with. As such, Data Wizard is designed to be seamlessly integrated into existing software ecosystems, being adaptable to diverse document formats and extraction tasks. This work details the architecture, implementation, and evaluation of Data Wizard, showcasing its ability to create data extraction processes through configurable strategies, support for multiple LLM providers, and an intuitive user interface. The evaluation across various real-world scenarios demonstrates Data Wizard's effectiveness in extracting structured information with minimal configuration, highlighting its potential to provide access to LLM-powered data extraction and accelerate digital transformation initiatives across industries.

Table of Contents

1 **Introduction:** The Data Input Bottleneck

The past decades we have witnessed a significant push for digitalization and digital transformation across nearly every sector, with companies investing heavily in transitioning from traditional paper-based processes to digital alternatives [1]. This initial wave of digitalization often focused on converting physical documents into digital formats, resulting in a widespread adoption of the *Portable Document Format* (PDF) , digital spreadsheets, and other electronic document types.

While this shift represents a step forward in reducing paper waste, enabling internet-based collaboration and promoting digital backups, it frequently falls short of achieving true digital transformation. The core issue is that simply having documents in a digital format does not automatically translate to machine-readability and efficient data utilization, which are necessary for unlocking the automation capabilities of modern technology. For computers, the information embedded within PDFs, Word documents, and even Excel spreadsheets often remains largely opaque.

The data, while digitally stored, is not inherently structured or easily accessible for automated processing, analysis, or integration into other software systems. This creates a significant bottleneck: extracting meaningful, structured data from these digital documents to use with advanced software solutions remains a predominantly manual task, requiring considerable time and human effort. Too often, data entry tasks are given to interns or part-time workers, who are tasked with manually transcribing data from documents and spreadsheets into a newly adopted software system. This process is both time-consuming and error-prone, as the task itself is repetitive and tedious. Additionally, if the work is not done by people with domain knowledge, the data quality can suffer as some information may be misinterpreted or left out entirely.

1.1 **A Persistent Pain Point:** Data Silos and the Burden of Legacy Systems

This reliance on manual data extraction presents a major slowdown on the road to achieve the full potential of digitalization. Companies find themselves struggling to leverage modern, data-driven technologies, such as machine learning and advanced analytics, because the foundational data required to power these systems is locked away in unstructured or semi-structured digital documents, inaccessible without significant manual work. Building a robust and readily usable data baseline requires substantial investment in manual data collection and entry, a costly and often discouraging undertaking. This problem is furthered by the prevalence of legacy software systems, which often act as data silos, hindering data accessibility and interoperability.

Consider, for instance, the real estate industry, where older software like iX-Haus [2] is used. This Windows-centric software, while still functional for its core purpose, lacks modern *Application Programming Interfaces* (APIs) and cloud integrations, effectively isolating the valuable data it contains from other, more modern systems. Extracting data from iX-Haus is a cumbersome process, as the included export functionality results in poorly formatted Excel and PDF files, far from the structured, readily processable data standards used in modern software development, like JSON (*JavaScript Object Notation*) and XML (*Extensible Markup Language*) or at the very least flat-file CSVs (*Comma-separated values*).

Similarly, many organizations, particularly in German-speaking regions, rely heavily on the DATEV software for accounting and tax preparation [3]. While DATEV does offer an API, accessing it is far from straightforward. Gaining API access requires explicit permission from DATEV, incurring a minimum upfront cost that can exceed 2.000 Euros, and navigating a complex, bureaucratic process involving multiple meetings [4]. Even with access granted, their Cloud API mainly focuses on data entry, having few endpoints for getting data out of the system, even in an aggregated form. For this, their Desktop API is still the go-to solution, which requires a Windows machine and a DATEV installation to use and deploy [5].

Furthermore, a significant number of businesses, especially smaller enterprises or departments within larger corporations, operate primarily within office suite ecosystems like Microsoft 365 or Google Workspace [6] [7] [8]. In these environments, critical business data can be scattered across a multitude of individual Word documents, Excel spreadsheets, and similar office files. While these files are digital, the data within them is not effectively machine-readable in a practical sense. Finding specific information becomes a time-consuming and often frustrating exercise, as data is spread across multiple, often deeply nested folders and files. While navigating this maze of files has been improved by the introduction of modern search functionality, the underlying data remains locked away in unstructured formats. Extracting and aggregating data from these files for analysis or integration into other systems typically requires manual opening, copying, and pasting – a far cry from automated, efficient data processing. Software solutions like Microsoft's Power Automate [9] and Power BI [10] offer some relief by enabling basic automation of repetitive tasks, but they still need to be configured specifically for each use case and rely on the fact that employees keep their data formatted in a specific way.

These examples illustrate a common theme: companies often find themselves locked into legacy systems or inefficient digital workflows that hinder true data utilization. The reluctance to migrate to modern, data-centric solutions is rooted in the perceived high costs and disruptive nature of data migration from these silos, coupled with a potential underestimation of the hidden costs associated with maintaining outdated systems and an overestimation of the expenses involved in switching to more modern alternatives. These hidden costs include inefficient processes, reduced employee productivity and

morale, difficulty attracting tech-savvy talent, and expensive training on outdated systems, which can all add up to a significant financial burden over time [11] [12].

1.2 From OCR to LLMs: Evolving Approaches to Extraction

Automated data extraction is not a novel challenge, and various technologies have been developed over the years to address this need. *Optical Character Recognition* (OCR) has long been used to convert images of text into digital text, enabling the processing of scanned documents and images. The Tesseract OCR software, for example, got started in the 1980s, where it was developed by HP Inc. and later open-sourced [13]. While the old versions use traditional OCR pattern matching, the 4.0 release of the engine implemented OCR using a deep learning approach [14]. Normally, OCR output is unstructured text and can struggle with complex document layouts and handwriting. Its primary goal is to create digital text from images, not necessarily to understand its content and derive new data structures from it.

An improved approach was the development of rule-based systems, relying on regular expressions and query scripting, like IBM's SystemT [15]. It is particularly effective for highly structured or standardized documents with predictable layout and format. However, this kind of approach is brittle, as it requires significant manual configuration and maintenance, and is easily disrupted by even minor variations in document structure.

There are some machine learning techniques that are useful for working with unstructured data and are even used by modern OCR software to improve text recognition, like Tesseract [16] or Amazon Textract [17]. These methods, as do most machine learning techniques, typically require substantial amounts of labelled training data for each document type and extraction task, limiting their adaptability to new document formats and variations and often necessitating complex feature engineering. This is not a scalable approach as even today, both machine learning engineers and GPUs (*Graphical Processing Units*) are in high demand and the process of labelling data is both time-consuming and expensive [18].

Large Language Models (LLMs) represent a significant breakthrough in document understanding, offering a level of flexibility and generalization previously unattainable with earlier technologies. LLMs possess a remarkable ability to handle unstructured and semi-structured data, enabling them to process diverse document formats, including data from PDFs, Word documents, Excel spreadsheets, scanned images, and more, overcoming the fixed nature inherent in template-based and rule-based systems. Their contextual understanding and semantic interpretation capabilities allow them to understand and extract data based on meaning and context, rather than relying solely on keywords or predefined patterns, leading to improved accuracy in interpreting complex information [19] [20].

Furthermore, LLMs reduce the need for extensive manual configuration and task-specific training data. Pre-trained on vast datasets, they exhibit *"few-shot"* or even *"zero-shot"* learning capabilities, allowing them to adapt to new extraction tasks with minimal examples or even none at all [21]. Crucially, a single LLM-based tool can be adapted to extract data from various document types and for different extraction tasks simply by modifying its configuration, for example, through JSON Schema definitions, significantly enhancing reusability and reducing development effort for diverse data extraction needs.

1.3 Streamlining LLM Data Extraction: The Reusability Advantage

While the capabilities of LLMs provide an obvious solution to the data extraction bottleneck, building reusable tools that effectively harness this potential is not a trivial undertaking. Due to computational constraints, LLMs are mainly accessed via well-defined API endpoints. Still, developing LLM-based data extraction solutions requires an API client implementation, capabilities to work with all kinds of document types and robust error handling mechanisms. Although possible, it would be inefficient and impractical for every software application that requires data extraction functionality to independently implement its own bespoke LLM integration and extraction logic. This provides an opportunity for the development of reusable, embeddable data extraction tools to streamline the integration of LLM-powered data extraction into a wide range of applications.

The rapidly evolving landscape of LLMs is characterized by a diverse ecosystem of providers, each offering distinct APIs with unique model capabilities. A reusable tool should be able to use as many of these as possible, and thus abstract away these API-level differences, providing a unified interface for multiple LLM providers. This allows users to choose the most suitable model based on factors such as cost, performance, data privacy requirements, and company policies. Furthermore, the field of LLM technology is constantly advancing, with new models, APIs, and extraction techniques emerging continuously. Just recently, Deepseek introducing its R1 model caused a stir in the industry and on the stock market, as it was on-par with OpenAI's o1 model in terms of performance, while costing a fraction of the price to train and run [22] [23]. If anything, these drastic advancements prove that there is still a lot to come in the field of LLMs and that the technology is far from stagnant. A reusable tool needs to be adaptable to this environment and in some ways future-proof, capable of incorporating advancements in LLM technology without requiring fundamental rewrites in every application that utilizes it.

Language and tooling ecosystems also play a crucial role. Not every programming language or development environment has equally mature and readily available libraries for interacting with LLM APIs for multiple providers or even just one. While languages like Python or JavaScript have universal libraries like the Vercel AI SDK [24] (*Software Development Kit*) or LangChain [25], other languages like R only have provider-specific libraries like the OpenAI R wrapper [26]. A pre-built, reusable and language-agnostic tool effectively bridges this gap, and by providing readily accessible LLM-powered data extraction capabilities across diverse technology stacks by being integrable using standardized web technologies like iFrames and HTTP APIs (*Hypertext Transfer Protocol*), could be used in a large variety of settings.

By using such a reusable component, developers can focus their efforts on building core application logic and features, rather than diverting valuable development time and resources to building and maintaining complex LLM integration and data extraction infrastructure. iFrame-based embedding allows for seamless integration of data extraction functionality into existing web applications, regardless of their underlying technology stack, while programmatic APIs enable more advanced and customized integrations for applications requiring deeper control over the extraction process. This thesis presents *Data Wizard*, a concrete response to this need. By offering a practical implementation of a reusable and embeddable LLM-based data extraction tool, it aims to address the challenges outlined above and to facilitate broader adoption of LLM-powered data extraction across diverse application types.

1.4 **Establishing Focus:** The Research Question

Given the expansive nature of data extraction as a field, the research question at the heart of this thesis is threefold.

Primarily, it investigates the technical feasibility of developing a reusable and embeddable tool for structured data extraction using Large Language Models. Integral to this exploration is the question of flexibility: can such a tool be sufficiently adaptable to be viable across a wide range of applications? That is, can it remain effective for various domain-specific data extraction challenges without being hardcoded on specific issues? Finally, this research aims to evaluate the crucial factor of usability: can such an application remain sufficiently simple, both in terms of deployment and integration into existing software systems, requiring minimal configuration to function?

To address these questions, this thesis details the planning and implementation of the Data Wizard software solution. To validate the assumptions made and to evaluate the practical utility of the resulting software, the thesis will assess its performance across various data extraction tasks, taken from multiple business sectors, each presenting unique technical challenges. To provide a structured framework for this investigation, the following section will outline specific technical requirements that are essential for achieving the stated objectives.

2 Requirements for a Reusable LLM-Based Data Extraction Tool

To address the challenges of data input in digitalization and to effectively leverage the potential of LLMs for data extraction, a reusable software component must meet a set of key requirements. This section outlines the essential functionalities, integration capabilities, deployment options, and architectural considerations for an ideal LLM-based data extraction tool. These requirements ensure the tool is not only powerful and accurate in extracting data but also user-friendly, easily integrated, adaptable, and maintainable in the long term.

2.1 Core Functionality: Data Extraction and Flexibility

The core functionality of the software must be the ability to reliably and efficiently extract structured data from unstructured and semi-structured digital documents. This extracted data must then be readily available in a structured format, suitable for seamless integration into other software systems, databases, and downstream workflows.

Firstly, the tool must offer **broad input format support**. Real-world data resides in a multitude of document formats, often within the same organization. An effective data extraction tool must handle a wide variety of these formats out of the box, including, but not limited to: PDFs, Word documents (`.docx`), Excel spreadsheets (`.xlsx`, `.xls`), and scanned images (`.jpeg`, `.png`, ...). The user experience should be intuitive and seamless, ideally allowing users to simply *"drag and drop"* files of various formats into the application, with the tool automatically handling format detection and processing. While not every possible format needs to be supported from the start, the tool should be designed with extensibility in mind, allowing for the addition of new format handlers as needed.

Secondly, the software should incorporate **flexible extraction strategies**. Different document types and data extraction tasks may benefit from varying approaches to leveraging and instructing LLMs. The tool should therefore support different extraction strategies, allowing users to select the most appropriate method for their specific needs. Examples of such strategies include providing entire small documents at once to maximize contextual knowledge but allowing large documents to be processed in chunks. Furthermore, some data sources describing a complex single data entity require some base data to be sent along with each chunk to provide context, whereas documents containing multiple small data records can be processed independently of each other.

Thirdly, **multiple LLM provider integration** is essential for a truly reusable and adaptable tool. The LLM landscape is dynamic, and different providers offer models with varying strengths, weaknesses, cost structures, and data privacy policies. The software should be designed to be provider-agnostic, supporting integration with multiple leading LLM providers such as OpenAI, Anthropic, and Google, among others. This abstraction should shield users from the complexities of provider-specific APIs, allowing them to easily switch between LLMs based on their own needs and preferences.

Furthermore, the tool must incorporate a robust **error correction workflow**. Large Language Models, while powerful, are not infallible and can sometimes generate inaccurate or nonsensical outputs, a phenomenon often referred to as "*hallucination*" [27] [28]. To ensure data quality, the software must include a built-in error correction process. This should involve mechanisms for automated validation

of extracted data against predefined schemas and user review of extraction results through intuitive interfaces for manual correction of any errors or inaccuracies.

To facilitate both automated validation and user-driven correction, the tool should leverage **data validation based on JSON Schema**. JSON Schema provides a standardized and widely adopted way to define the structure and data types of JSON documents, including validation rules such as required fields, data type constraints, and allowed values [29]. This should be utilized to both guide the LLM in structuring its output and to automatically validate the extracted data against the defined schema through automated checking and UI generation.

Finally, to provide a unique edge in data extraction, the software should support getting **embedded images** out of the documents and allow referencing them in the extracted data. Many document types include photos or other images that are directly relevant to the extracted data. For example, an accident report of a car crash may include images of the crash scene, which are valuable for insurance claims and legal proceedings. Even though the embedded images are simply a part of the document contents, most PDF readers do not support a "*right-click to save image*" functionality like it is available in web browsers. The software should enable referencing these files in the extracted data, which gives it an edge even over manual data entry, as most people would not be able to easily use these images from the document without resorting to lossy methods like screenshots.

2.2 Seamless Integration into Existing Software

For such a data extraction tool to be truly reusable, it must be designed for **seamless integration into existing software applications**. Integration should be straightforward and require minimal development effort, allowing organizations to easily incorporate LLM-powered data extraction into their current workflows and systems.

Firstly, **JSON Schema-driven configuration** is crucial for ease of use and rapid integration. Since configuring data extraction workflows should be intuitive and require minimal coding to get started, JSON Schema is a great fit for defining the target data structure and guiding the LLM in its extraction process. Ideally, the minimal configuration required to get started should be simply providing a JSON schema that describes the target data structure. Additionally, most LLM APIs have standardized on JSON Schema as their input format for features like tool calling, which allows the input schema to be reused without a lot of modification [30] [31]. By leveraging the power of JSON Schema, the tool can also generate user interfaces for data validation and correction automatically, streamlining the error correction process and enhancing user experience. This allows even some non-technical users to get started with the tool by only having to learn the relatively simple JSON Schema syntax [32].

Secondly, as mentioned above, the tool should provide an **embeddable user interface**. For many integration scenarios, a simple user interface that can be embedded into existing applications is essential for non-tech-savvy users to interact with the data extraction tool. Providing such an interface out of the box allows developers to quickly integrate the tool into their applications without having to build a custom user interface from scratch. This interface could then be embedded using iFrames, enabling seamless integration into web applications, desktop applications, and other software systems. This embedded UI should be customizable (e.g. with custom colours and fonts), allowing it to seamlessly blend into the host application's user interface.

The tool should also allow for writing **custom extraction code in the form of strategies** for advanced users or specific use cases. While the default extraction strategies should cover a wide range of common scenarios, advanced users may require finer-grained control over the extraction process. This enables

users to tailor the extraction to highly specific document types or complex data structures. However, this customization should be optional and not required for basic usage.

Lastly, for more complex integration scenarios, the tool must offer **programmatic API access**, ideally through an HTTP API. Such an API allows other software applications to programmatically initiate extraction jobs, monitor their progress, and retrieve the extracted data without user interaction. This programmatic access allows for integrating the tool into data pipelines that do not require user interfaces or for building entirely custom user interfaces on top of the tool's functionality.

2.3 Flexible Deployment Options

To be usable both as a standalone application and as an embeddable component, it should be **easily deployable in various infrastructure setups**. This is easily possible by distributing the application as a Docker container including everything the app needs to run. Docker packages the application and all its dependencies into a single container image, which can then be deployed on any system that has the Docker runtime installed [33]. In a way, this concept is similar to Java and the JVM, where the JVM abstracts away the underlying operating system and CPU hardware, allowing Java applications to run on any system with a JVM installed. With Docker containers, this is taken a step further, as the container can include databases and storage configurations, allowing users to get the tool up and running by executing a single command.

2.4 Architectural Considerations

While the usefulness of the data extraction tool is a core pillar of this thesis, the unified LLM interface and the ability to support multiple LLM providers are equally useful on their own. This usefulness warrants the abstraction to be separated from the main application and made available as a **separate, open-source library**.

To provide a unified LLM interface, support for multiple LLM inference API formats has to be implemented. Since OpenAI was among the first to introduce such an LLM API, many other providers have followed suit and adopted their API schema, requiring only the base URL to be changed [34] [35] [36]. However, not all APIs support this format. Such an abstraction is useful in a multitude of other applications as well, as it allows for easy switching between LLM providers without having to rewrite the entire application. Decoupling it from the Data Wizard application itself allows for reusability in other projects, keeps the logic more maintainable by separating it from any application-specific code, and promotes community contributions and collaboration through open-sourcing a more widely useful part of the application.

Data Wizard has been developed with these considerations in mind. To gain a better understanding of how these requirements manifest themselves in the final application, the following section will provide an in-depth look at how Data Wizard is be used and how an end-user interacts with the application.

3 **Data Wizard:** How It Works

The following graph gives an overview of how an external software application can embed Data Wizard and use it to add data extraction features to their software workflows, be it for data entry or other processing purposes. In this section, we'll go through this process step by step, using an example extraction task to illustrate the functionality of Data Wizard.

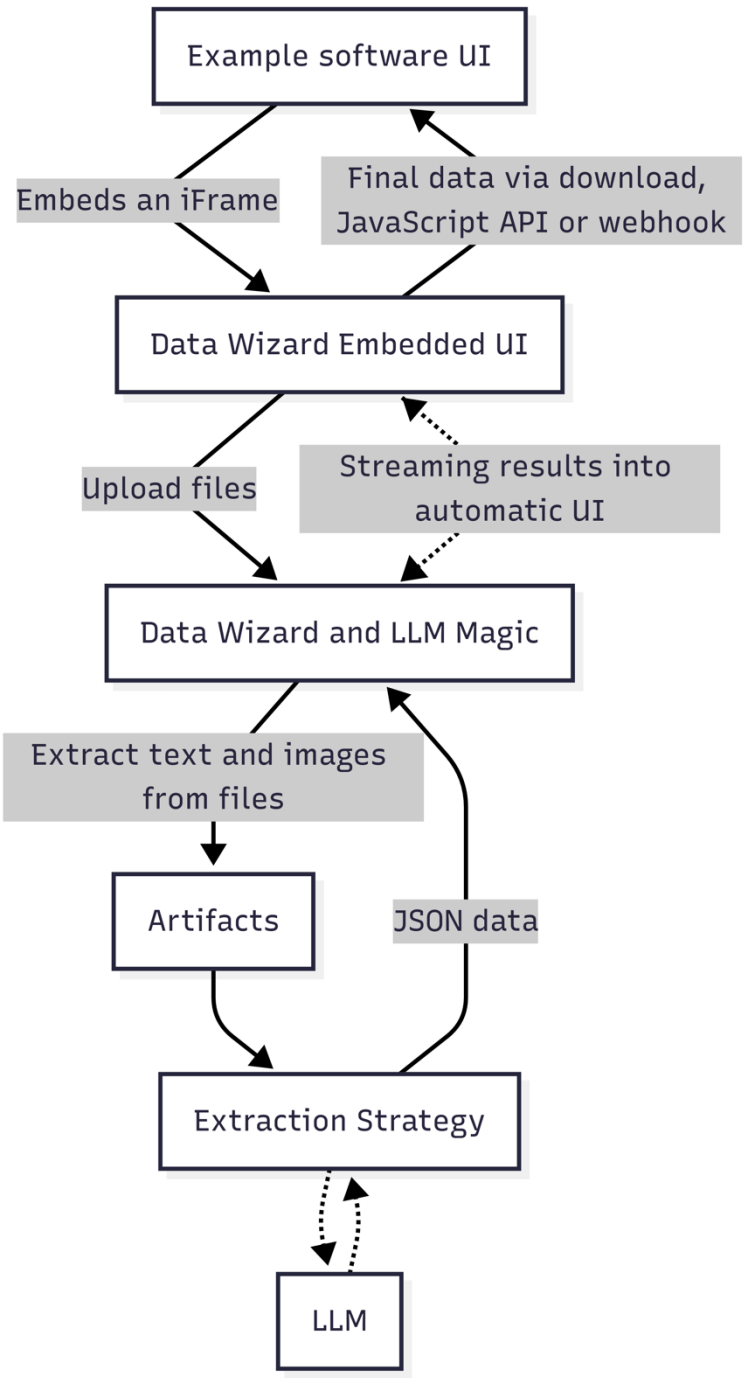


Figure 1: How Information flows through Data Wizard

3.1 **The Example Dataset:** Products from LIDL Brochures

As the example extraction task, we will choose the extraction of product and pricing information from discount flyers of German supermarkets like LIDL or ALDI. These flyers are multiple pages long and contain a lot of products and prices. "*Plus-Membership*" prices that are only available with membership cards and not valid for regular customers are often included in these flyers too, and the goal is to extract only the products and prices available to the regular customer.

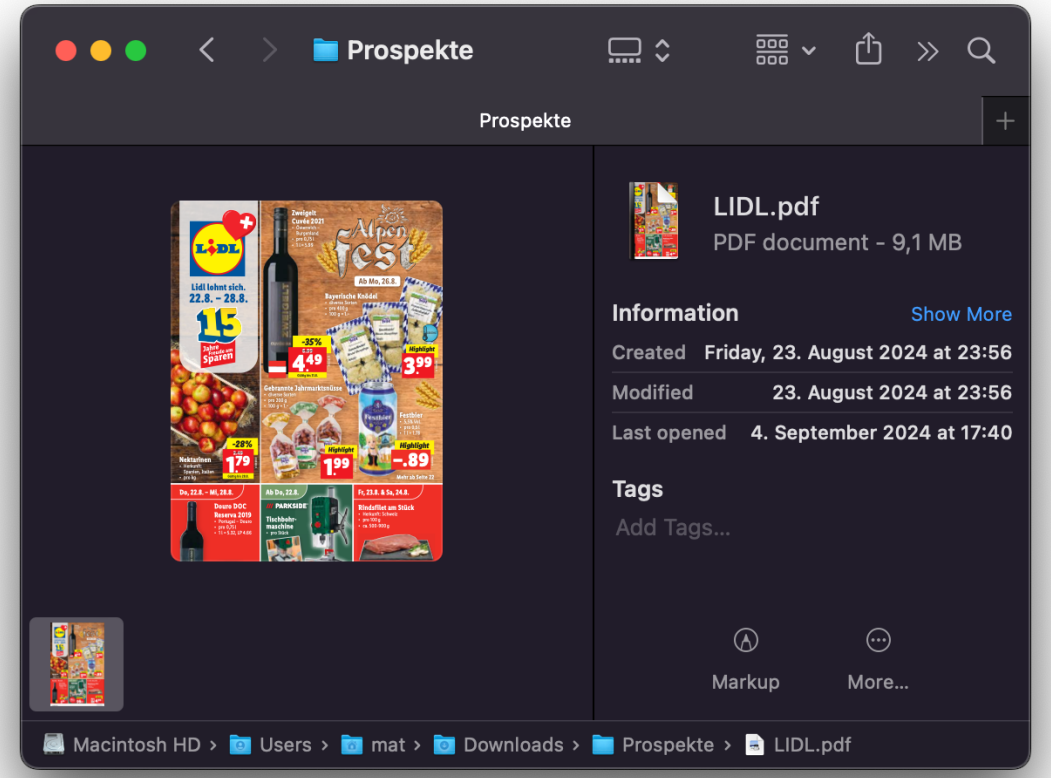


Figure 2: The LIDL brochure PDF used as an example.

The following is an example of the kind of JSON data that we would like to extract from the flyer. It contains a list of products, each with a name, an original price, and a discounted price, if available. Entering this kind of data manually would be a very tedious and time-consuming task, especially for larger flyers with hundreds of products. This makes it a prime candidate for automation.

```
[
  {
    "name":"Bottle of red wine",
    "original price": 14.99,
    "discounted price" :9.99
  },
  {
    "name":"Bottle of white wine",
    "original price": 12.99,
    "discounted price": 6.99
  }
]
```

Code Snippet 1: Example JSON object representing a list of supermarket products

3.2 Data Wizard Through the Eyes of the End-User

First, the user interacts with the external software application's user interface, which embeds Data Wizard's user interface using an iFrame. The fact that the UI is embedded may be visible to the user, but colour, font and other CSS customizations can be used to make the UI blend in with the rest of the application.

In this embedded UI, the user can upload the source files containing the data to be extracted. If the source data is already present in the example software, it can be uploaded by the application itself through the HTTP API instead, while the upload step can be skipped, and the extraction started immediately.

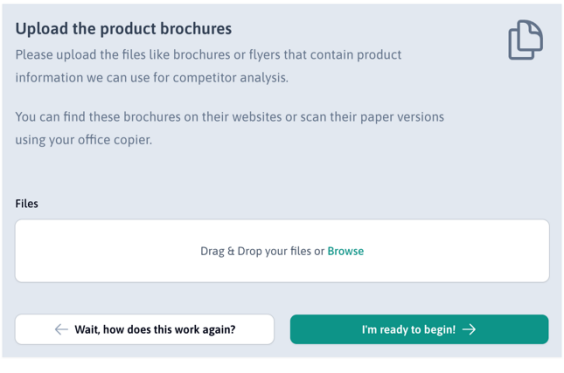


Figure 3: The empty uploading UI

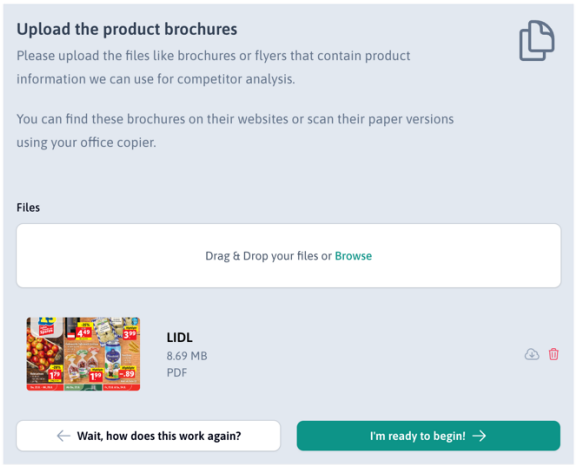


Figure 4: The UI after a file has been uploaded

After the user has uploaded all the files that should be processed, they can start the extraction process. The extraction process is a long-running task that is handled in the background by Data Wizard's

extraction strategies. However, the user can see the real-time progress of the extraction, as the user interface is updated with the extracted data as it comes in. Depending on the extraction strategy, the data visible may not be the complete data and may only be a small chunk of it. This is because some strategies only return partial data, which is then combined into the final result at the very end. Data Wizard shows the chunks as they come in these scenarios.

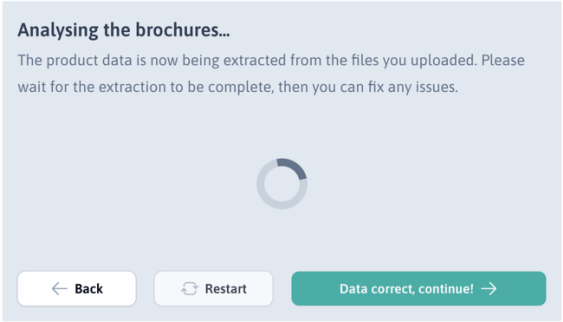


Figure 5: Initial loading indicator after starting the extraction

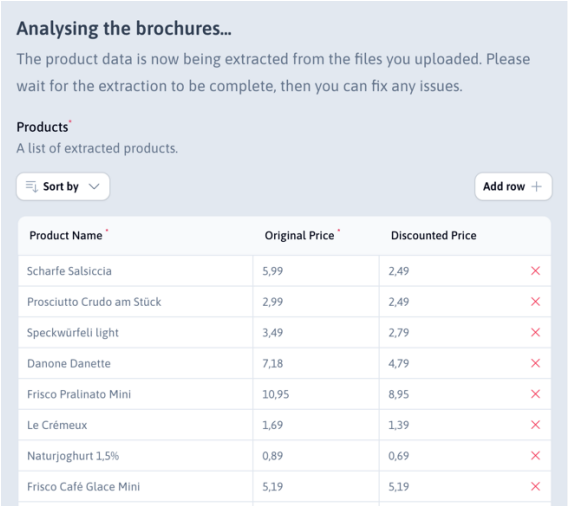


Figure 6: Once data comes in, it is shown in a generated form

Instead of seeing the raw JSON data being returned, the user is seeing a user interface that has been generated from the input schema. For less technical users, this is a much more intuitive way to interact with the data, as they do not have to understand anything about JSON or data structures. Since the data labels in the UI are taken from the property names of the JSON schema, they may contain underscores or other characters that are not user-friendly. These labels and other UI aspects can be customized using some custom, non-standard properties directly inside the schema, but this is entirely optional.

In the example above, the data is shown inside a table, which is useful when dealing with large quantities of small data objects like our example products. This is configured in the JSON schema by using the `magic_ui` property, which is not part of the JSON Schema standard but is used by Data Wizard to generate the UI. Without this customization a card layout containing input fields for each property would be shown.

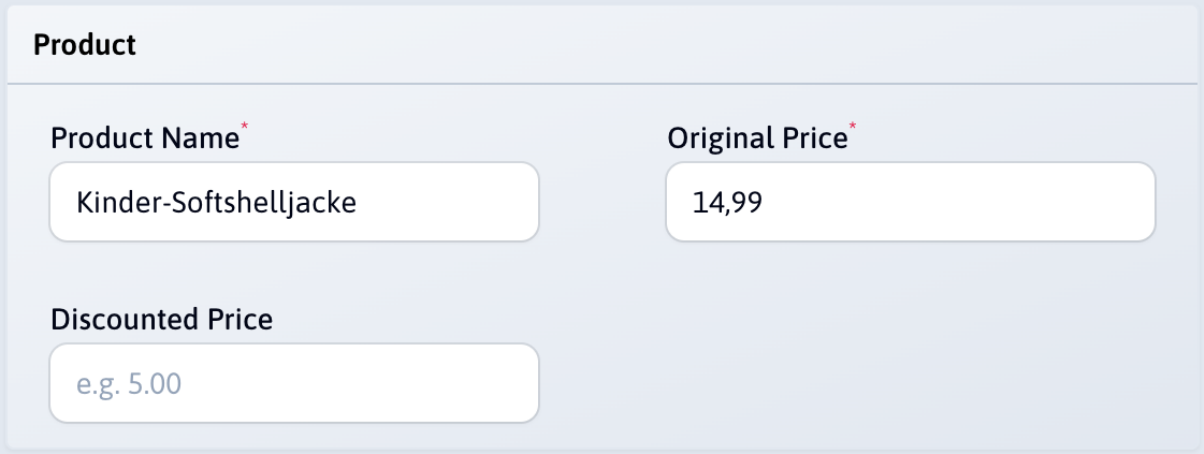


Figure 7: The generated form if `table` is not specified using the `magic_ui` directive

The form elements in the user interface are disabled while the extraction is still in progress. This way, the user can tell that the data is not yet final and that they still have to wait for the extraction to finish. Once the extraction is done, however, the interface becomes interactive, and the user can start editing the data and fixing any mistakes that the LLM may have made. This is a crucial part of the data extraction process, as LLMs are not perfect and will occasionally make mistakes, especially with complex data structures or when the input data is not clear. If any data is missing, new rows can be added to the dataset, while unneeded data can be removed.

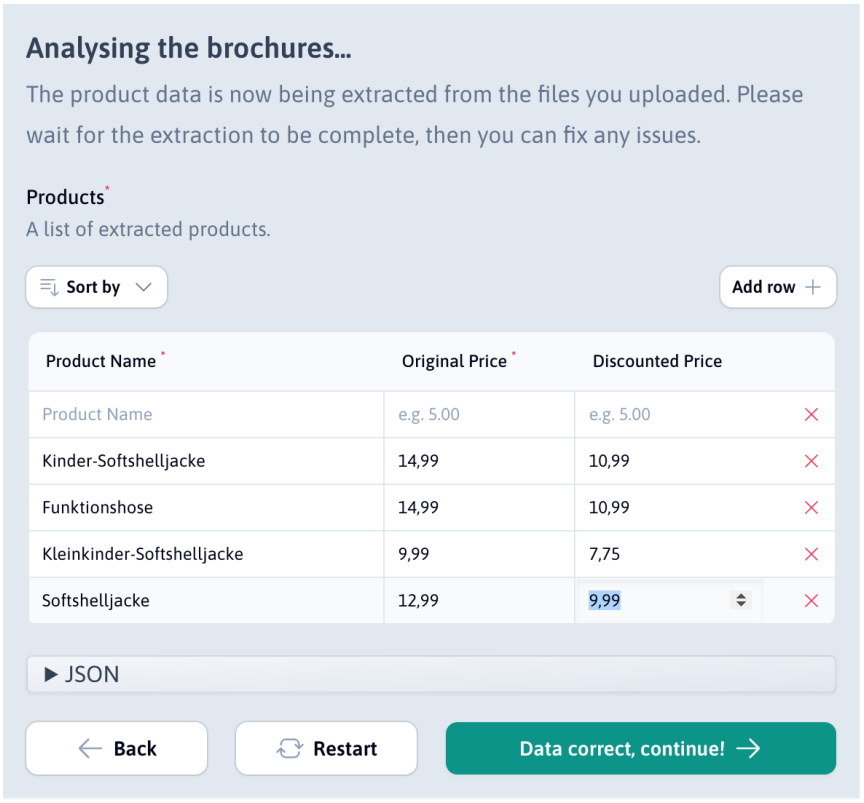


Figure 8: Editing the extracted data

While making changes to the data, it is sent to the server in the background, where it is validated against the JSON schema. If the data contains any errors, the user cannot continue until these errors are fixed. This applies both to missing data, for example, if a required field is not filled out, and to data that does not match other requirements, like a number being too large or too small. Simple rules like `minimum` and `maximum` values are directly validated client-side, using the respective attributes on the HTML5 input elements. But since a full JSON schema validator is running on the server, more complex rules like `multipleOf` or `pattern` can also be checked. If a value does not conform to the schema, the user is shown an error message, and the field is highlighted in red.

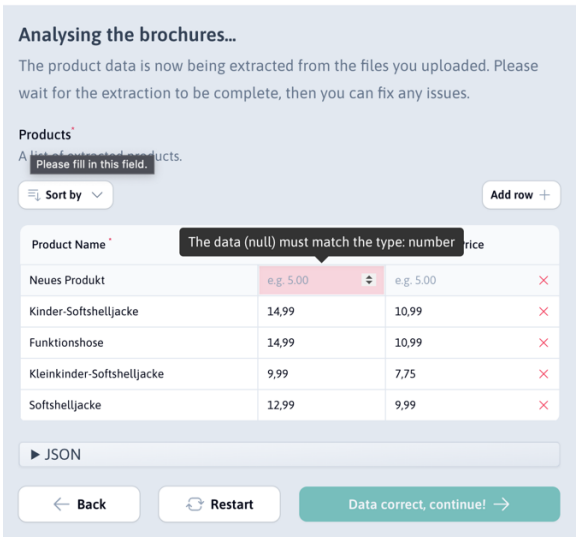


Figure 9: How errors are displayed in tables

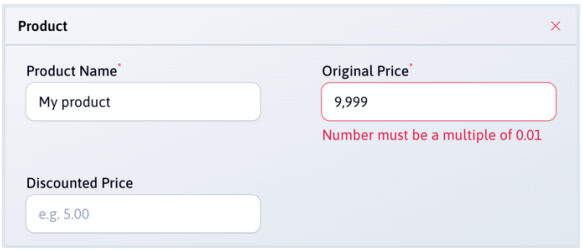


Figure 10: How errors are displayed in normal forms

Once the data is validated and the user is satisfied with the result, they can press continue to go to the next step. After pressing continue, this step will also result in a JavaScript event and a webhook being triggered, that the external software application can listen to and react to.

The final step enables the user to download the extracted data in JSON, XML and CSV formats. Since this is not always necessary to do depending on the implementation, it might make sense for an implementing application to hide the iFrame after receiving this JavaScript event and then show the data in the application itself. After completing the extraction it can receive the data through said JavaScript event, through the webhook it set up or by requesting the Data Wizard API for the result.

At this point, the job of Data Wizard is complete, and the normal application flows can continue with the extracted data.

3.3 Configuring the Extraction Process

The following section describes how the above process can be configured and customized by the user.

In Data Wizard, the configuration for the extraction process is stored in something referred to as an "*extractor*". This data model contains the JSON schema to be extracted, the strategy to be used for the extraction, the actual LLM to be used, and any additional instructions for the LLM. Settings for how to integrate the extracted data back into the application are also stored in there, like the webhook URL or texts and labels for the embedded UI. This makes it one of the three central data models in the application, the other two being a "*bucket*" containing the uploaded files and "*extraction runs*" which store the extraction results of each attempt.

All the required settings and configurations are done in the Data Wizard backend interface, which is accessible through a web browser. The user can create new extractors, edit existing ones, and view the results of past extraction runs.

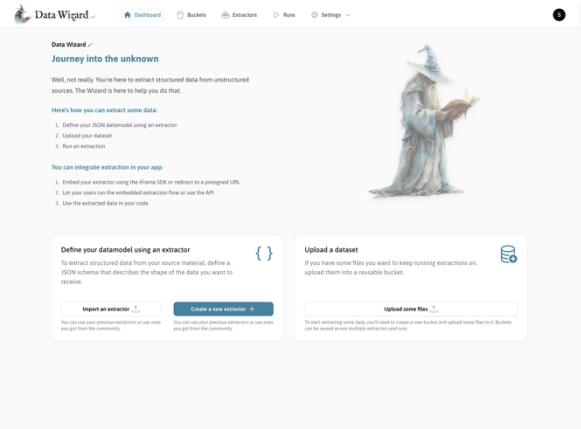


Figure 11: The dashboard of the Data Wizard backend

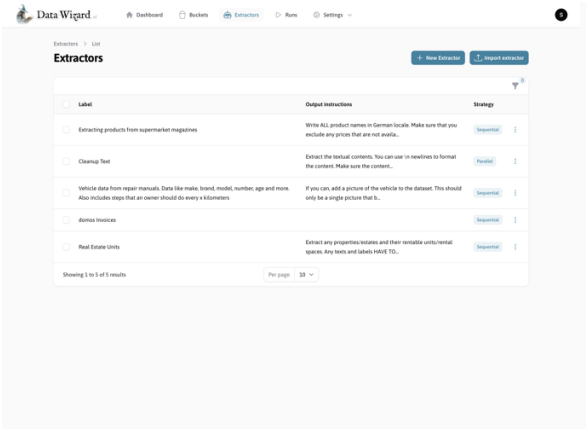


Figure 12: A list of all the configured extractors

The creation of a new extractor is assisted by a wizard-like interface that guides the user through the process of setting up the extraction. Within this wizard, the users state in plaintext what kind of data they want to receive and how it should be structured. An LLM call is then made to generate a JSON schema based on this description, which is inserted into the extractor as a first draft.

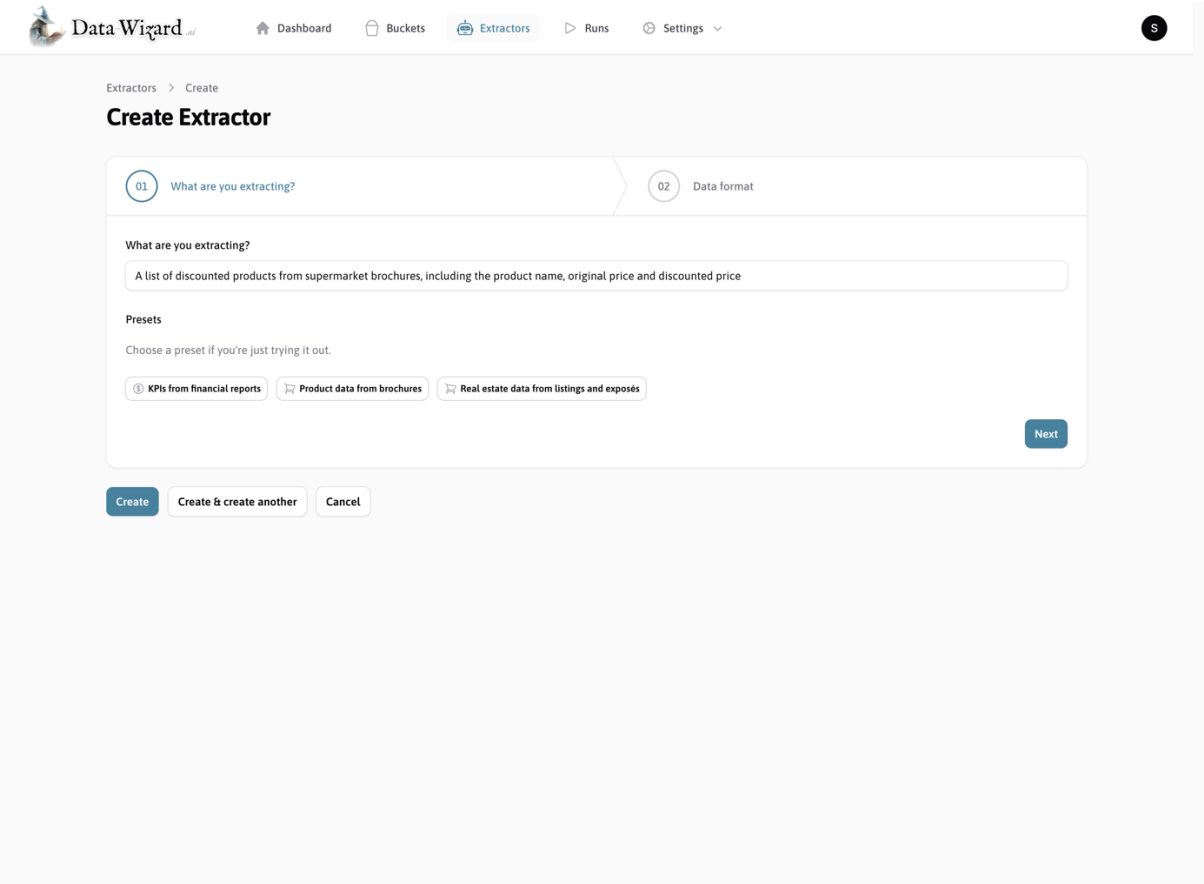


Figure 13: The creation wizard with AI support

For the example of the LIDL brochure, entering "*A list of discounted products from supermarket brochures, including the product name, original price and discounted price*" generates the following JSON schema:

```
{
  "type": "object",
  "required": ["products"],
  "properties": {
    "products": {
      "type": "object",
      "required": ["name", "original_price"],
      "properties": {
        "name": { "type": "string" },
        "original_price": { "type": "number" },
        "discounted_price": { "type": "number" }
      }
    }
  }
}
```

Code Snippet 2: Automatically generated JSON schema for supermarket products

This is a great starting point, but the user can further customize the schema to fit their needs. In our case, it would make sense to add some validation rules to only allow positive numbers with two decimal places for the prices. We can do so by adding the `minimum` and `multipleOf` properties to the schema.

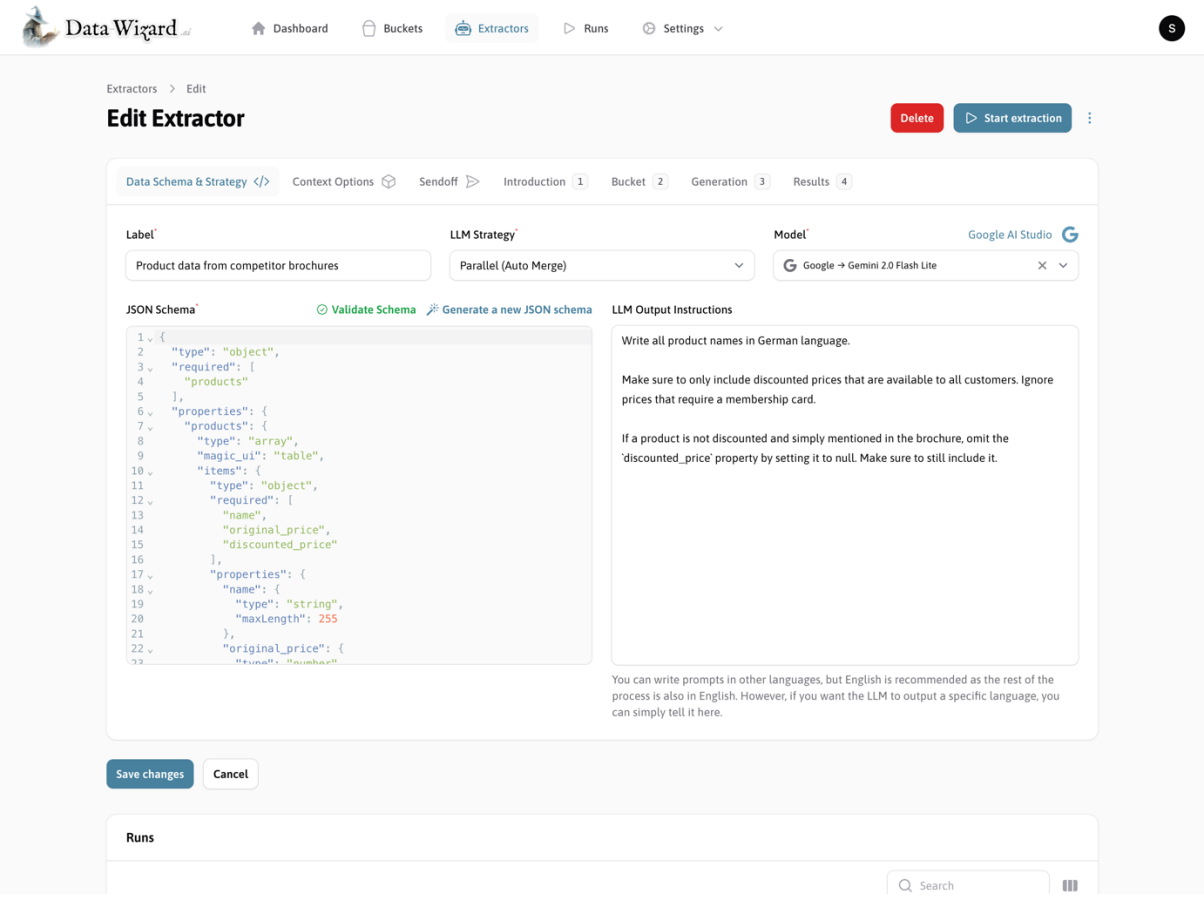


Figure 14: The extractor configuration UI

Another constraint we're facing is the length of the name column. The external software application uses a `VARCHAR(255)` column in its database to store the product names, so we want to make sure that the names are not longer than 255 characters. This can be done by adding the `maxLength` property to the name property in the schema.

What has become noticeable throughout the development of Data Wizard, is that LLMs deal with optional properties in different ways. While some models will use `null` for empty properties, others will simply omit them from the output. To normalize this behaviour, it might make sense to include all the properties in the `required` array, which will force them to be present, but to allow them to be `null` by using an array for the type of the property.

Additionally, it can be helpful to add some `description` properties to the schema, which will give the LLM some more context about how to fill the data. In case of the LIDL brochure, sometimes a discounted price is only available for customers with a membership card. We're only interested in prices that are generally discounted, so adding a `description` like *"The discounted price for all customers. Prices only applying to customers with a membership card should not be included here."* to the schema of the `discounted_price` property can help the LLM understand what we're looking for.

After making these changes, the updated schema looks like this:

```
{
  "type": "object",
  "required": ["products"],
  "properties": {
    "products": {
      "type": "object",
      "required": ["name", "original_price", "discounted_price"],
      "properties": {
        "name": {
          "type": "string",
          "maxLength": 255
        },
        "original_price": {
          "type": "number",
          "minimum": 0,
          "multipleOf": 0.01
        },
        "discounted_price": {
          "type": ["number", "null"],
          "description": "The discounted price for all customers…",
          "minimum": 0,
          "multipleOf": 0.01
        }
      }
    }
  }
}
```

Code Snippet 3: An improved version of the product schema

After the extractor has been created, even more options become available to be configured. Most importantly, the user can now choose the extraction strategy as well as the LLM to be used. Next to the data schema itself, these are the two options with the most impact on the extraction results. The extraction strategy determines how the LLM receives the chunked input data and how the results are combined into the final output. The LLM choice, on the other hand, determines the quality and speed of the extraction, as well as the cost of the extraction run. This is where the choice to use a unified LLM interface pays off, as the user gets to choose from a large list of LLMs, each with different strengths and weaknesses. What models to use is a decision that should be made based on the requirements of the extraction task, as well as the budget and time constraints of the user.

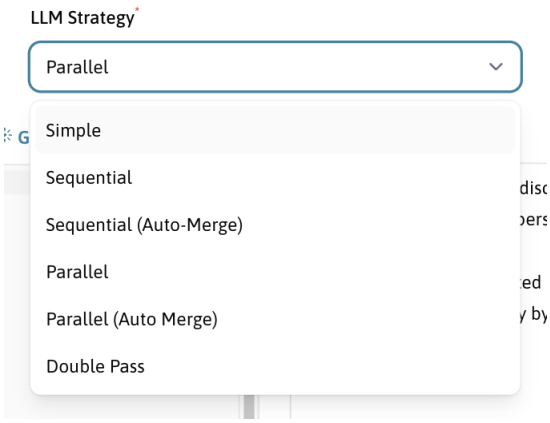


Figure 15: Selecting from the available extraction strategies, including any custom ones

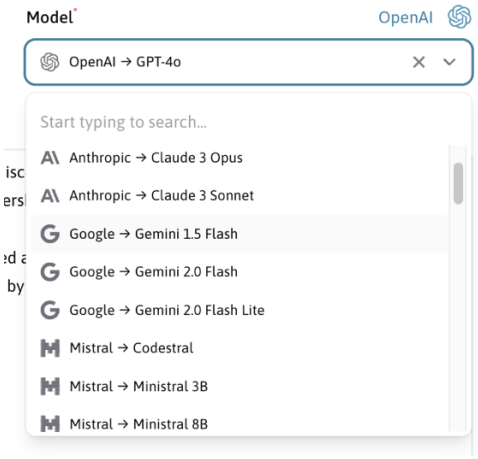


Figure 16: A large number of LLMs are available thanks to the API abstraction layer

In addition to the LLM choice, the user can also add some additional instructions for the LLM. This is directly placed in the system prompt of the LLM, which will make anything written in here have a direct impact on the extraction results. Additional rules and constraints that the LLM should follow can be placed here, as well as hints on how to deal with certain edge cases or what to do in case of missing data. It can also help to describe the reasons for the extraction in more detail, as this might be helpful for the LLM to interpret the data correctly.

Since these output instructions are placed in the system prompt, best-practices for prompt engineering can be followed to achieve better results. The major LLM providers have released guidelines on how to structure prompts for best results, often referred to as "*cookbooks*" [37] [38]. Depending on the model choice, including instructions in a certain format can lead to better results. Anthropic models, for example, work really well when XML-tags are used to structure the prompt [39] [40].

At the top of the extractor configuration is a list of tabs that reveal further customization options. Of these, the context options tab is the only one that contains settings with direct impact on the extraction results. In here, the user can configure what part of the document contents should be included in the LLM context when running inference.

For one, the size of the document chunks in tokens can be set to a specific value. What works here depends on the model used as well as the input data. Using a small value of around ~2.000 tokens will result in lots of small LLM calls being made to go through a document. While this may allow the LLM to place more focus on the included document sections, it will also result in a higher cost and slower extraction. Using more LLM calls also increases the chance that an LLM may drop some important contextual information in between calls. Using a larger value starting at 20.000 all the way up to 500.000 tokens will result in fewer LLM calls, since each has a lot more context to work with.

Other options in this tab include settings for what images should be sent along with the textual contents of the document. Both the embedded images, as well as "*screenshots*" of the full document pages can be enabled or disabled here. The user can also choose to mark the images with identifiers, which the LLM can use to reference the images in the output. For the page screenshots, enabling this option results in a red border being drawn around the image in the page, in addition to the identifier. The rest of the tabs include options for setting the webhook URL and HMAC SHA-256 secret (*Hash-Based Message Authentication Code using the Secure-Hash-Algorithm with a digest size of 256 bits*) to be used, configuring a

redirect after the extraction is done, and customizing the texts and labels for the embedded UI. Placing these options on the extractor-level allows a single Data Wizard instance to serve multiple extraction use-cases and applications.

If the user is done configuring the extractor, they can get started integrating it into their software. Alternatively, they can test the extractor right in the backend, which provides a useful user interface for both executing extraction runs as well as to debug them.

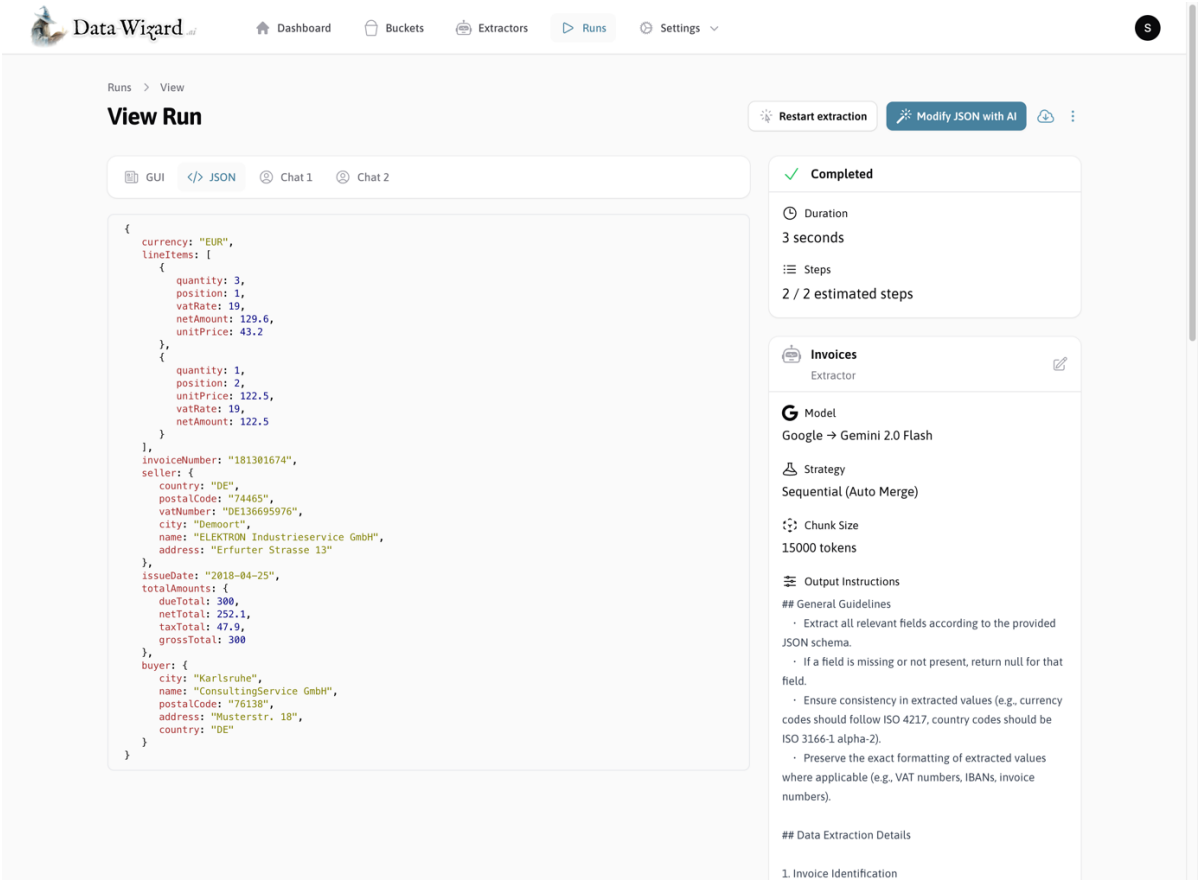


Figure 17: The backend interface to perform, observe and debug extraction runs

3.4 Integrating Data Wizard into External Applications Using iFrames

As mentioned earlier, Data Wizard can be integrated into an external application in multiple ways. The most straightforward way is to embed the Data Wizard UI using an iFrame, as shown in the example above. This has the advantage of being very easy to set up and requires minimal development effort to get started. To embed the iFrame, all the user needs is the URL of the Data Wizard instance, the ID of the extractor they want to use as well as some security parameters to ensure that only authorized users can access the data.

Security is implemented by requiring pre-signed URLs which require a valid HMAC SHA-256 signature to be present in the URL. This can be thought of like an API token that is not stored in the database but instead generated and verified on the fly. A hash of the URL is created and attached, so that any changes made to the URL or its query parameters will invalidate the signature. This technique is commonly used in file storage systems like Amazon *Simple Storage Service* (S3) and Google Cloud Storage to allow temporary access to files without having to expose the actual API key [41]. Web frameworks like Laravel have since included this functionality as a built-in feature, making it easy to implement in the backend [42]. Another common implementation of this technique can be found with *JSON Web Tokens* (JWTs), which allows distributed systems to authenticate and even authorize requests without needing access to a token database [43].

A pre-signed URL for embedding the iFrame can be obtained in multiple ways. The extractor backend interface provides a button to generate a pre-signed URL for the current extractor, which can then be copied and pasted into the external application. The expiration of this URL can be modified or even set to never expire, which is useful to get started quickly but should probably not be used in production. Alternatively, a pre-signed URL can either be generated programmatically by implementing HMAC SHA-256 signing and attaching the appropriate parameters to the URL, or by simply creating a bucket ahead of time through the HTTP API, which will also return a pre-signed URL that can be used.

The iFrame itself is a simple HTML element that can be placed anywhere in the external application's user interface:

```
<iframe src="https://data-wizard.ai/embed/123456...?signature=..."></iframe>
```

Code Snippet 4: An example of an iFrame with a Data Wizard Embed URL

When using the iFrame, a JavaScript API is available to communicate with Data Wizard and the extraction process. This allows the embedding application to receive real-time updates and to react to events like the extraction being complete.

The following events are emitted by the iFrame:

- `screen.changed` - The user has navigated to a different screen of the embedded extractor
- `extraction.started` - Extraction process has begun
- `extraction.progress` - Real-time updates during extraction
- `extraction.completed` - Extraction finished successfully
- `extraction.error` - An error occurred during extraction
- `extraction.submitted` - Final data has been submitted

These events can be received using standard JavaScript event listeners, by listening for the `message` event:

```
const wizard = document.getElementById('data-wizard-frame').contentWindow;
window.addEventListener('message', message => {
    if (message.source !== wizard) {
        return; // Skip messages that are not from our iFrame
    }

    if (message.data.event === 'extraction.completed') {
        // Extraction is done, hide the iframe and show the data in the application
        wizard.remove();
        showDataInMyUI(message.data.data);
    }
});
```

Code Snippet 5: Example JavaScript code to receive iFrame events

Additionally, the iFrame can be styled using *Cascading Style Sheets* (CSS) to match the look and feel of the embedding application. To avoid potential security risks, custom CSS can only be included by setting the `DATA_WIZARD_CSS_URL` environment variable to a custom CSS file that will then be loaded into the iFrame. Since some applications implement a dark mode, the iFrame can be set to match the parent application's appearance by setting a `theme` query parameter to either `light` or `dark`. If the theme changes while the iFrame is open, a `set_theme` message can be sent to the iFrame to update the appearance on the fly.

The result can be observed in Figure 18, where an extractor has been integrated into real estate management software. In this example, the user can upload real estate exposés and other similar documents to skip having to manually create a property and fill it with information. Instead, Data Wizard returns both structured data and any included media files, which are immediately usable in real estate listings. The fact that this part of the software is comping from another application is completely invisible to the end user, which shows the power such a simple iFrame based solution can provide.

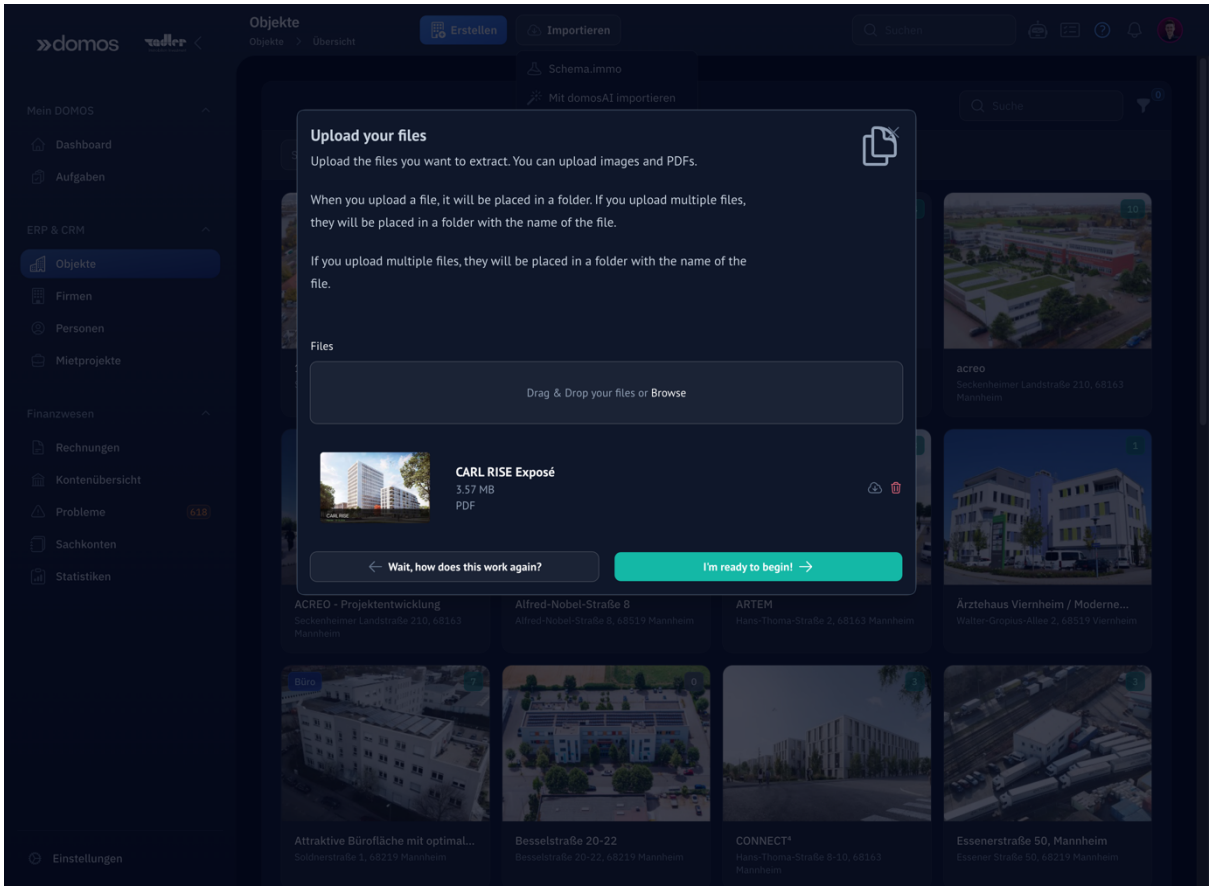


Figure 18: An example of Data Wizard being embedded directly into the domos real estate software

4 Implementation

This thesis focuses on the implementation of a specific use case, the Data Wizard application, rather than exploring general web application development methodologies. While there are numerous technology stacks that could be used, the primary selection criteria are development time efficiency, long-term maintainability, and rapid time-to-solution. Traditional web application architectures, including *Single-Page-Applications* (SPAs) built with JavaScript frameworks, offer powerful user interfaces but often necessitate significant initial setup and a decoupled backend.

This is why the PHP (*PHP: Hypertext Preprocessor, ex: Personal Home Page*) ecosystem, particularly in combination with the Laravel framework, presents a compelling alternative. While PHP used to have a reputation for being a somewhat slow and insecure language, modern PHP versions (anything after the 7.0 release) have significantly improved the performance and developer experience [44]. Frameworks like Laravel builds upon this, providing much needed structure and nicer APIs to work with. While I personally have a lot of experience working with Laravel already, the framework's opinionated structure and extensive ecosystem provide a solid foundation to build and rely upon. Laravel has built-in support for multiple databases and comes with file storage capabilities out of the box [45]. Since 2024 it is also backed by substantial venture capital investment, securing its future as a stable and well-supported platform [46]. Laravel distinguishes itself by a backend-centric design, emphasizing data modelling and business logic, while offering multiple ways to build frontend interfaces. These range from traditional server-rendered and static HTML templates all the way to completely custom Vue or React SPAs using Inertia.js [47].

In addition, Laravel Livewire [48] is a relatively new technology that allows for building interactive frontend interfaces using server-side components, enabling a more integrated approach to frontend and backend development. Livewire uses components in a similar way to modern frontend frameworks, but is driven by server-side PHP code, reducing the need for complex state management or API calls. State updates automatically trigger UI updates and JavaScript can simply call backend functions as if they were local, which enables a whole range of new possibilities for application development. Filament is another library built on top of Livewire that provides a structured way to build backend administration interfaces through configuration and reusability, reducing the need to manually rebuild the same basic user interfaces, which is especially useful for CRUD-heavy apps (*Create, Read, Update, Delete*).

It works by defining form fields, validation rules and table schemas right at the data model level, which can then be used by reusable `ListRecords`, `ViewRecord` and `EditRecord` Livewire components. Leveraging Laravel's opinionated APIs and the Eloquent *object–relational mapping* (ORM) models, Filament can also infer model relationships, which can then be connected right inside the forms or as part of relation tables. This "*define-once, use-everywhere*" approach drastically reduces redundancy and boilerplate, especially with just a single developer working on the project. This makes it a great fit to build the backend of the Data Wizard application, while only the very custom aspects of the app like the embedded UI and the LLM integration need to be built from scratch.

The following is an example Filament resource with the minimum configuration required, to give an impression of how easy it is to define forms and tables with it. This resource directly accesses the Eloquent model functions like `Model::create()` or `Model::update()` in a safe, validated way. The table can also use all the of the ORM's query features like sorting, filtering and eager loading, all while supporting pagination as well.

```php
class SavedExtractorResource extends Resource
{
    public static string $model = SavedExtractor::class;

    public static function form(Form $form): Form
    {
        return $form
            ->schema([
                TextInput::make('name')
                    ->label('Name')
                    ->maxLength(255)
                    ->required(),
                ModelSelect::make('model')
                    ->label('LLM'),
            ]);
    }

    public static function table(Table $table): Table
    {
        return $table
            ->columns([
                TextColumn::make('name')
                    ->label('Name'),
                TextColumn::make('model')
                    ->sortable()
                    ->label('LLM'),
            ]);
    }
}
```

Code Snippet 6: The code of a basic Filament resource

The rest of the architecture consists of a relatively standard Laravel setup, with SQLite and PostgreSQL being usable as the database [49] [50]. PostgreSQL is a reliable and widely-used relational database that has everything required for this application, and could even be used as a vector store for embeddings in the future [51]. SQLite on the other hand works in an embeddable way and can be bundled together with an application, which is especially useful in containerized deployment environments. The framework also has built-in support for background tasks, which are particularly useful for the long-running LLM inference operations we're dealing with. The Laravel ecosystem itself is also very mature, with lots of widely used libraries being available for all kinds of purposes. There are libraries dealing with JSON Schema validation [52], spreadsheet reading [53] and even more broadly useful features like automatic API generation [54] and handling file uploads [55]. This leads to rapid development progress, with development efforts focused on the core aspects of this application, which are by themselves problems complex enough to explode the scope of this project. Therefore, this thesis will not go further into a general comparison of web application development methodologies but will instead concentrate on the specific implementation details of the LLM integration and the embeddable part of the application.

4.1 **The Data Model:** Extractors, Buckets and Runs

Laravel uses the Eloquent ORM to interact with the database, which is a powerful and expressive way to define database models and relationships. The full Data Wizard application consists of 8 different models, of which 6 are at the core of what's required for the extraction process, the other two being the `User` model and another one to store LLM API keys. A class diagram of the extraction data model can be viewed in *Appendix 1*.

At the core of it all is the `SavedExtractor` model, which stores all the configuration parameters for extraction tasks. Having a separate model for these settings makes them reusable without having to re-input them every time a new extraction is started. It stores the data schema, output instructions and strategy, in addition to the properties for the external integrations and embeddable UI. Required information like `json_schema`, `model` and `strategy` is stored here and properties like `chunk_size` and `include_embedded_images` are used by the LLM abstraction to build the context window. Others like `webhook_url` or `introduction_view_heading` are used to configure the integration with the external application and the embedded UI.

The `ExtractionBucket` model acts as a grouping mechanism for multiple `File` models, which are the uploaded files that are to be processed. The `File` model extends the `Media` model provided by the *laravel-medialibrary* [55] package, which provides easy support for handling file uploads and even generating thumbnails. This dependency is not necessarily required, as we could use the images generated in the pre-processing steps as thumbnails. But not having to worry about the upload and storage of files is a big time-saver and makes the application more robust, as it is a well-tested and widely used package. Filament itself even provides a `FileUpload` form component, which directly integrates with the library.

The `ExtractionRun` model is the central model for the extraction process itself. It stores the current status of the extraction, including any partially generated data, the final results, and any errors that may have occurred. It also stores some of the same settings as the `SavedExtractor` model. This duplication has a reason, as it allows for slight variations in the extraction process without having to create an entirely new extractor every time. This is particularly useful for evaluating the performance of different strategies or LLMs.

The last two models are the `Actor` and `ActorMessage` models, which are used to store the LLM interactions for inspection in the backend. Since the LLM strategies may make multiple API calls, each `Actor` represents a single LLM instance, while the `ActorMessage` models store the individual messages that were sent and received. This makes it possible to debug the extraction process and to see what the LLM was outputting at each step. An `ActorMessage` stores the raw message data as JSON, including any images that were sent along as Base64-encoded strings. This results in the table growing quite large, which is why it is recommended to frequently truncate this table. Data Wizard provides a command-line helper to do this, which can be run as a scheduled task to keep the database size in check. This is not enabled by default, as keeping the data may be useful for debugging and compliance reasons.

4.2 **LLM Magic:** A Unified Abstraction Layer for Invoking Large Language Models

The primary objective of designing a unified API for interacting with various Large Language Models is to abstract away the unique complexities of each API, providing a consistent and developer-friendly interface. This unified API should enable seamless integration with multiple LLM providers, allowing developers to focus on building their applications rather than dealing with the unique quirks of each API itself. Integrating the APIs of multiple LLM providers into a single interface is not a trivial task, as the LLM landscape is characterized by rapid evolution. A prime example of this dynamic environment is the shift towards message-based Chat APIs, which seem to have become the dominant interface for interacting with contemporary LLMs. This paradigm, now considerable as the de facto standard, was not prevalent in earlier model iterations such as GPT-3, which used APIs centred on simple text completion tasks.

An LLM abstraction layer designed during that era would likely prove inadequate for accommodating the functionalities of newer, chat-centric models without a lot of modifications. However, the chat-based API interface has demonstrated relative stability over the past two years and a lot of other tooling exists around this concept [25]. Furthermore, this paradigm aligns well with the requirements of structured data extraction. Many chat-oriented APIs incorporate tool calling capabilities, which allow the LLM to invoke predefined functions with structured input parameters using JSON and JSON Schema. This is perfect for data extraction tasks, and the tool calling mechanism also opens up more possibilities for new strategies in the future, such as directly accessing external data sources through tools that can dynamically fetch data.

Additionally, an LLM abstraction has advantages for other applications that require a similar level of interaction with the LLM. Being able to use a variety of LLM APIs and to swap them out as needed is not just beneficial for this project, as I have personally encountered the need for such a library in some of my other PHP projects as well. That makes this part of the codebase a prime candidate for open sourcing as a standalone package, which could then be reused in other projects. While this potentially makes the scope itself more complex by not directly integrating with the rest of the codebase, it does make the project more modular and forces the separation of application and LLM interaction logic, simplifying maintenance in the long run.

Building such a unified LLM abstraction requires careful thought and planning to ensure that it can accommodate the LLM APIs available today, while offering a great developer experience too. As such, it needs a well-defined API that can be easily extended to support new LLMs as they emerge.

4.2.1 Designing the API and the Unified Messaging Format

The LLM Magic SDK is designed to provide a seamless and intuitive interface for developers. Making a request to an LLM is inherently something with a lot of configuration options. For this reason, the builder pattern is a good fit for an SDK like this. This allows lots of optional configuration options to be set in a fluent way, while still providing a clear API for the developer to use. Also, the required options can be kept to a minimum, enabling the use of powerful behaviour with just a few lines of code.

```php
$answer = Magic::chat()
    ->prompt('What is the capital of France?')
    ->stream()
    ->text();
// -> "The capital of France is Paris."
```

Code Snippet 7: The simplest possible usage of LLM Magic

Calling `Magic::chat()` returns the builder object that we can then configure to our liking. Using the `->prompt($text)` method, we can provide a basic text message to send to the LLM. The `->stream()` call then executes our LLM call, returning a list of messages wrapped in a `MessageCollection` class. This class extends the Laravel `Collection` class and provides some helper methods to deal with the returned data. In this case, the model would likely only return a single text message, but tool calls and other responses may result in multiple being returned. To provide a sane API surface, the function always returns a list of them. However, the `MessageCollection` helper methods like `->text()` can then be used to work with the array of messages. The `text()` method concatenates the textual contents of all the received messages together. There are also methods like `->firstText()` or `->lastText()` which are useful in other contexts. In addition to text data, the messages themselves or any data contained within them can also be returned using helpers like `->firstTextMessage()`, `->lastToolResultMessage()` or `->lastData()`. The latter works for both tool calls and tool responses and will return the respective input/output data as a PHP array. The builder object supports a couple more methods to further customize the LLM call:

```php
Magic::chat()
    ->model(Gemini::flash_2_lite()) // Set the LLM model
    ->system('You are an XYZ expert…') // Set the system prompt
    ->temperature(0.5) // Set the temperature for randomness
    ->topP(123) // Set the top-p sampling parameter
    ->maxTokens(1000) // Set the maximum number of tokens
```

Code Snippet 8: Additional LLM Magic configuration options

One of the more important ones is the `->model()` method. It accepts both `string` values as well as classes implementing an `LLM` interface. These classes contain the logic required to interact with the respective LLM APIs. So, the `OpenAI` class can be passed to call GPT-4o and similar models, while using `Anthropic` will result in an inference call being made to the Anthropic API. While these classes can be instantiated using normal PHP constructors like `new OpenAI('gpt-4o')`, they provide static factory helpers for type-safe instantiation. For example, `OpenAI::gpt_4o()` could be used instead. The `OpenAI` class is extended by models like `Gemini` or `Mistral`, as they use OpenAI-compatible APIs and can simply reuse the same execution logic. If a raw string is passed to the `->model()` method instead, the library tries to resolve a model class itself by looking through all the supported models in the background. This model string needs to be in the `<provider>/<model>` format for this to work. For example, the Gemini 2.0 Flash model can be created using the `google/gemini-2.0-flash` string value. However, it is recommended to use the model classes directly, as this enables static analysis of the codebase to trigger warnings with invalid models. It is still a useful feature, and Data Wizard uses it to make the extractor models configurable in the backend.

All of these models require API keys. To make configuring models as flexible as possible, a `TokenResolver` interface exists, which is called by the underlying model logic with a provider ID. By default, a `ConfigTokenResolver` is used, which fetches the API keys from the Laravel config files, which allows configuration through environment variables like `OPENAI_API_TOKEN`. A token resolver is created through the Laravel service container and dependency injection logic, which allows a custom resolver

to be configured globally within the Laravel `AppServiceProvider`. Data Wizard itself implements the `DatabaseTokenResolver`, which fetches tokens from the database instead, allowing them to be configured in the backend UI.

The other methods available include the `->system()` method, which sets the system prompt and is a very important part of any LLM interaction. Modern LLMs are fine-tuned to more closely follow the system prompt than the user prompt, so utilizing this correctly is crucial for good model output. The SDK also provides a set of methods to configure the temperature, top-p hyperparameter, and a maximum output token limit, which are common parameters for most LLMs.

In addition to passing a simple string to the `->prompt('…')` method, it also accepts a class implementing the `Prompt` interface. This enables prompt reusability by being able to package an LLM prompt into a single class that can then be created as often as necessary and even customized on a case-by-case basis by using input parameters in the class constructor. The interface itself requires the `system(): ?string`, `messages(): array` and `tools(): array` methods to be implemented.

The `messages()` function can be used to configure multiple LLM messages as part of the prompt. This enables writing more complex prompts, for example by including images or by prefixing the user request with a couple of user-assistant message pairs to facilitate few-shot prompting.

To abstract the different LLMs effectively, we cannot let the end-user of the library work with the original JSON-based LLM messages directly. Thus, a messaging format that can be used across all kinds of LLMs is needed. This format does not only need to be capable of representing all the different message types that an LLM can accept but also be flexible enough to adapt to new message types in the future. For example, while most models currently only support text and image input, support for audio blobs may be needed in the future.

Two kinds of message structure have emerged in the LLM space:

1. A flat message array, where each message is a simple object with a role and a content field.
2. A nested and step-based message array, where each item is a step in the conversation and contains a role and content field, but the latter is itself an array of message parts which can be text, images or tool calls.

Both have their advantages, with the flat message array being less complex and easier to type, while the step-based message array is more structured and allows for more complex conversations. The library must pick one of these formats to work with internally, as this ensures that writing scripts with the library is consistent and the same regardless of the LLM used. One downside of the flat message array is that it technically allows for invalid message structures to be created. If two user messages are sent in a row, the LLM API may reject the API call, as it expects a model response in between. I've only collected anecdotal evidence in personal testing for this happening while using some smaller models with the OpenRouter API, but since the step-based message structure could always be turned back into a flat message array if required, it is most likely the safer choice. This ensures consistency within the different model API implementations, as they can expect to only have to deal with the nested array structures. We can still help the developer save some keystrokes by including helper methods on the `Step` class.

The final message API looks like this:

```
Magic::chat()
    ->messages([
        Step::user('Hello, how are you?'),
        Step::assistant('Great, how can I help you today?'),
        Step::user([
            Text::make('Please find save the following invoices to the database:'),
            Image::base64('...'),
            Image::url('https://example.com/image.jpg'),
            Image::disk('s3', 'path/to/image.jpg'),
        ]),
        Step::assistant([
            Text::make('I\'ll analyze the images using some AI magic.'),
            ToolUse::make('createInvoices', [
                'invoices' => [
                    ['name' => 'Invoice 1', 'amount' => 100.00],
                    ['name' => 'Invoice 2', 'amount' => 200.00],
                ],
            ]),
        ])
    ]);
```

Code Snippet 9: An example showing the message API to pre-fill LLM responses

In addition to the helper methods on the Step class, similar methods are available for the Image content types, which allows for some more direct integrations with the Laravel ecosystem by providing an `Image::disk()` helper function that can read images directly from the Laravel storage systems. The images are converted to Base64 strings before being sent to the LLM, which appears to be the most common denominator for image messages across different LLM APIs. While some providers support using image URLs directly, we can make the library more useful by handling the conversion to Base64 ourselves.

4.2.2 Calling the LLM and Parsing the Responses

We want to make the LLM interaction as simple as possible for the developer. As shown above, the LLM used can be swapped by changing a single line of code. To enable this functionality, we need a way to transform our unified messaging format into provider specific messages and back. This is a non-trivial part, as LLM providers have built their APIs on different assumptions and with different use-cases in mind. For example, some LLMs only allow text messages, while others have Vision support and allow for image messages as well [56]. Also, the meta-options available to the developer can slightly differ between providers. While almost all have `maxToken` and `temperature` options, some LLMs including the Gemini family even support safety options for filtering out harmful content [57].

Thankfully, due to OpenAI's dominance in the LLM space, a lot of providers have adopted the OpenAI API to offer a drop-in replacement for OpenAI models or to at least provide some kind of compatibility with existing codebases. This allows the use of Mistral, Grok, Gemini, Deepseek and other models with the same API as OpenAI models [35][58]. Open-source models can also be used with this API through hosted services like TogetherAI and Hugging Face or by self-hosting them through software like Ollama [36] [59]. There are even proxy services like OpenRouter, which try to make every model available through a single API and even offer load balancing and other advanced features on top [60]. But not all providers have adopted this API, such as Anthropic, which has its own API that is not directly compatible with OpenAI's. If one wants to use the Azure OpenAI or Amazon Bedrock offering for compliance reasons, an implementation using the Azure and Amazon Web Services (AWS) cloud SDKs is also required. Also, some advanced features like Gemini safety settings or prompt caching are also

only available through provider specific APIs and there is no guarantee that model providers will provide OpenAI compatibility layers forever.

For the moment though, the situation allows LLM Magic the luxury of only having to implement two different API interfaces to gain access to the most powerful LLMs on the market: the OpenAI API and the Anthropic API. While both the AWS, Azure and Gemini-specific APIs would be nice to have, the models they provide are available through other means too, which allows their implementation to be skipped for now.

Conveniently, PHP already has a widely used library for the OpenAI API, which manages all of the needed HTTP requests and even does some basic error handling and data parsing [61]. For the Anthropic API, however, no widely adopted PHP library exists yet, so we must implement the API interface ourselves. Late into the development of Data Wizard and LLM Magic, another library following a similar unified approach has emerged, which has since grown in popularity [62]. While this library technically also supports both OpenAI and Anthropic APIs, it was released too late into the development work on LLM Magic and after all the model implementations had already been written. Besides that, it does not have event hooks deep enough to allow for the same kind of progress callbacks that Data Wizard requires and also does not support streaming for tool call parameters either. While replacing the HTTP layer with this library is a decision to be revisited in the future to save on maintenance efforts, the limitations present might not make it a good fit anyway.

On the topic of streaming, this is something that LLM Magic prioritises greatly. Few things are worse than having to wait for a long LLM response, just to discover that the model misinterpreted something. While supporting streaming for normal text output is relatively simple, it gets more complicated when trying to support streaming for tool call parameters. Both the OpenAI and Anthropic API use Server-Sent-Events (SSE) to implement streaming. Since the library makes its own HTTP requests to the Anthropic API, it has full control over this stream of data. Unfortunately, this is slightly different with the OpenAI PHP library. While it does receive these packets as they are being generated, the API client buffers them until the full message is complete. This means that for the moment, the library must wait for the full tool call message to be received before it can be shown to the user, and streaming tool parameters is not possible with it. While this does reduce the user experience somewhat, it is not a problem major enough to justify writing a custom OpenAI HTTP client for now, especially not as part of this thesis. The streaming is not exposed to the user during data extraction in Data Wizard and is only relevant for more generic use of the library. In the future, a fix for this could be contributed to the OpenAI PHP library itself, which would then fix the issues observed without requiring any other modifications to LLM Magic.

The SSE packets themselves are somewhat tricky to parse correctly. While each data packet is a JSON object, we need to keep track of the current message and append the new data to our buffer accordingly. The aim of the library is to not expose implementation details to the developer, including the use of SSE. Instead, it provides universal progress indicators to be received using the `onMessage` and `onMessageProgress` callbacks.

Parsing the packets of tool calls is even more difficult, as the tool parameters are streamed in as raw text that can only really be converted to JSON once the full message has been received. This would mean that for large JSON responses, using streaming becomes relatively useless, as we must wait for the full message before we can parse it. To work around this, the library uses a slightly modified version of a partial JSON parser created by Greg Hunt [63] to try and parse the JSON as it comes in. This allows the application to react to the data immediately. Since we get an unfinished but valid JSON object, an application could even show this data in its UI to make it feel even more seamless. To accomplish this,

a `PartialMessage` interface is used to enable appending streamed data to the current message, by implementing the `append(string)` method. Both text and tool call messages support this interface.

4.2.3 Tool Calling using Reflection

While tool calling itself is a powerful feature of LLMs, it is surprisingly easy to implement in a unified manner. All models that support tool calling use JSON Schema to define the input parameters, which is very convenient since Data Wizard already uses these schemas for defining the output data that it wants. This allows the tool calling implementation of the LLM library to focus more on the developer experience.

An annoying thing about working with normal LLM APIs is the fact that the called function as well as its parameters need to be read and used on a case-by-case basis, all one receives is the JSON data that the LLM returns, which contains information about what function the model wants to execute. If one wants the LLM to correct its own mistakes or do follow up tasks, custom retry logic needs to be implemented every time, which also non-trivial to do correctly without some kind of abstraction.

All of this is not ideal and a potential source of errors, and the tool call itself is also not guaranteed to actually be valid JSON according to the schema provided. LLM providers simply do not validate the model's output before returning it to the user, which is not even possible to do when using streaming. OpenAI has an entire Structured Outputs feature to work around this, which unfortunately is only supported by OpenAI itself [64]. If not for this fact, it would have been a great way to implement data extraction. But as it stands today, we need to use tool calling with manual schema validation to achieve the same functionality.

One feature that some programming languages support is source code reflection, which allows the program to inspect its own structure and definitions. While less useful in older versions, PHP 7.0 introduced type hinting of native scalar types to the language, allowing the function parameter syntax to contain `int`, `float`, `string`, `bool`, and `array` types. Using function reflection, LLM Magic can inspect a function's signature and its parameters, to automatically generate the JSON schema needed for the LLM to understand how to call it [65]. This is quite powerful, as it allows the developer to write their tools as native PHP functions, with the library taking care of validating the input beforehand and even supporting automatic correction of the input data by sending it back to the LLM if it is invalid, along with an appropriate error message.

This leads to an API surface of the SDK that allows very powerful behaviour with few lines of code. Below is a simple implementation of a calculator agent that can perform basic arithmetic operations:

```php
$solution = Magic::chat()
    ->prompt('What is the result of ((1000 - 900) + 26) / 3?')
    ->tools([
        'add' => fn (float $a, float $b) => $a + $b,
        'subtract' => fn (float $a, float $b) => $a - $b,
        'multiply' => fn (float $a, float $b) => $a * $b,
        'divide' => fn (float $a, float $b) => $a / $b,
    ])
    ->toolChoice(ToolChoice::Required)
    ->send()
    ->lastText();
// -> "The result is 42"
```

Code Snippet 10: A simple calculator agent using LLM Magic tool calling behaviour

In this example, the LLM is given four tools for arithmetic operations, which it can use to calculate the result step by step. The library then analyses the function signatures using PHP's reflection functionality and generates the JSON schema for the LLM to know what the function requires to be called. After execution, the tool output is then passed back to the LLM, which can then perform the next step in the calculation. When the model considers its task complete, it returns the final result to the user in a text message. This is a simple example, but it shows the power of the tool calling feature, especially compared to the complexity of having to implement the same feature by manually parsing the responses. The `toolChoice` option is used to specify how the LLM should choose the tools. By default, tool calls are optional. But setting it to required or by passing the tool name directly, an LLM that supports this feature can be forced to use any or a given tool.

If the developer is interested in directly working with the result of the calculation, a simple text response is inadequate. For this use case, the library supports the `Magic::end($data)` helper function, which, when returned from a tool, will end the conversation immediately. Data Wizard's extraction strategies use this internally to access the generated data.

```php
$solution = Magic::chat()
    ->prompt('What is the result of ((1000 - 900) + 26) / 3? Use the finish tool for final answer.')
    ->tools([
        // add, subtract, multiply, divide functions
        'finish' => fn (float $a) => Magic::end($a),
    ])
    ->send()
    ->lastData();
// -> (int) 42
```

Code Snippet 11: An example of a tool using the `Magic::end` helper

In some cases, a developer may want to add additional information to the tool call definitions, such as a description or more finely tuned JSON schema. For these cases, the library supports the use of DocBlocks [66], which allow some metadata to be added to functions and classes. DocBlocks are normally used for documentation purposes, like providing descriptions or specifying the types of function parameters. We can use this to our advantage by using both the text contents of the comments and parameter definitions to generate the JSON schema for the tool call. Alternatively, the library also provides the custom `@type` tag, which can be used to provide a custom JSON schema directly in the DocBlock. The parser is relatively basic for now, so `@type` definitions are limited to a single line, which does reduce readability somewhat.

```php
Magic::chat()
    ->tools([
        /**
         * This text will be added as the `description` field in the JSON schema for this tool.
         * @param array<array{title: string, description: string}> $articles
         */
        'save' => fn (array $articles) => ...,

        /**
         * This tool uses the schema tag to provide a JSON schema directly.
         * @type $articles {"type": "array", "items": {"type": "object", "properties": {"title":
{"type": "string"}, "description": {"type": "string"}}}}
         */
        'save' => fn (array $articles) => ...,
    ]);
```

Code Snippet 12: Adding additional metadata to tools using DocBlocks

To improve the readability of complex tools as well as to promote their reusability, class-based tools are also supported and are similar to reusable prompts.

```php
class SaveArticles implements InvokableTool
{
    public function name(): string
    {
        return 'save_articles';
    }

    public function description(): ?string
    {
        return 'This tool saves the provided articles to the database.';
    }

    public function schema()
    {
        return [
            'articles' => [
                'type' => 'array',
                'items' => [
                    'type' => 'object',
                    'properties' => [
                        'url' => ['type' => 'string', 'format' => 'uri'],
                        'title' => ['type' => 'string'],
                        'description' => ['type' => 'string'],
                    ],
                    'required' => ['url', 'title'],
                ],
            ]
        ];
    }

    public function execute(ToolCall $call): mixed
    {
        foreach ($call->arguments['articles'] as $article) {
            Article::create($article);
        }
    }
}
```

Code Snippet 13: An example of a class-based tool

```php
Magic::chat()
    ->tools([
        new SaveArticles,
        // Tools can be included multiple times with different parameters, if needed
        'save_with_options' => new SaveArticles(option: 'value'),
    ]);
```

Code Snippet 14: Examples of how class-based tools can be used

4.3 Accessing Text and Images by Pre-Processing the Files

Before any data can be extracted from the files, they need to be processed into a format that the LLM can understand. How well this is done can greatly influence the quality of the extraction results, as the LLM can only work with the data it is given. For the best extraction results, we want to give the LLM as much of the original data as possible. This mainly includes the textual contents of the files, but also any images that might be embedded inside of them. Images can provide a lot of context to the LLM, especially when they are referred to as figures or diagrams in the text. However, the raw form of most document types is not directly suitable for LLMs, as they are either in a format that they do not understand directly, contain a lot of bloated data that is not relevant to the extraction task, or are too large to be processed in one go. This is why Data Wizard needs to pre-process and split the files before being able to send them to the LLM.

For models that have vision capabilities by supporting image input, a seemingly obvious shortcut would be to just send screenshots of each page to the LLM instead of trying to parse and read the text from the files. While vision LLMs can certainly work with these images and even extract text from them, this approach has some drawbacks. Firstly, the text contents inside the image might not be readable correctly, depending on the size and quality of the image. Providers like OpenAI even do some resizing of the images before giving them to the model, which can lead to an unpredictable loss of quality as text may become unreadable during this rather opaque process [56]. This would fail silently, as the LLM would still generate some output, but it would be incorrect or incomplete. Secondly, images consume a lot more tokens than raw text, which can lead to higher costs for the extraction process and even limit the output quality as less context can be given to each LLM call. This is especially wasteful when just uploading text documents as images, as they would contain a lot of empty space that the LLM would still have to process. Thirdly, not all LLMs support image input, which would limit the choice of models that can be used for the extraction process. This could make extracting from text-only documents more expensive than necessary, as there is no gain from using a vision model if smaller, text-only models suffice.

4.3.1 **Artifacts:** A Unified File-Content Representation

To make working with the files easier, LLM Magic uses a unified file representation interface called `Artifact`. These help to abstract away the differences between different file types by focusing on the content of the files, rather than the file format itself. It works by splitting document contents into separate `Slice` objects, which can be thought of as small parts of the document, like a paragraph or an embedded image. Each slice also has a page number associated with it, which can be useful for keeping related slices together when passing them to the LLM. For example, a financial report will likely contain explanations of graphs and figures on the same page as the graph itself. By passing this text and the image to the LLM together, the model can better understand the context of the image and provide more accurate extraction results.

Since an `Artifact` is an interface rather than a class itself, the application code becomes isolated from the storage implementation details. In fact, LLM Magic already has two kinds of `Artifact` implementations: `DiskArtifact` and `VirtualArtifact`.

A `DiskArtifact` is tightly connected to a directory on a disk, where extracted contents of a file are stored. This is useful, as it means other programs or scripts can be used to create the artifact contents, which LLM Magic can then simply read from. This allows using the best tool for the job, as

the extraction process can be written in a language with good support for the file format, namely Python in the case of PDF files.

The `DiskArtifact` requires a specific directory structure. The source file is stored in the root of the directory with the filename source and the file extension of the original file. The main contents of the document are being stored in a `contents.json` file. This content file then contains all of the slices of a document, with each slice being a separate object with a page and type field. The basic slices include `text` and `image`, and `image-marked`. These are the extracted contents of the documents. The `marked` version of the image has its ID baked-in on-top. In addition to these, the `contents.json` file also includes slices for `page-image` and `page-image-marked`. The latter two are screenshots of each page of the document, with the `page-image-marked` items having the embedded images highlighted with a red border and numbered with their corresponding ID. While the text for each page is contained within the `contents.json` file itself, the images and page screenshots are stored in separate directories with their relative paths being stored instead.

These page screenshots are useful if we want to send the document to the LLM as an image instead of text, should an extraction strategy wish to do so. This provides the LLM with more layout context, which can be useful if text is drawn on top of images or if the document contains complex formatting that is not easily represented as text. Using the marked page screenshots gives the LLM the ability to then use the image IDs in the extracted data, allowing for these images to be referenced in the final output. For example, the LLM could assign pictures of a rental space to the corresponding data like rent, area or location when extracing from a real estate exposé.

If the file that has been provided is a simple image, a `DiskArtifact` can still be used. In this case, the `contents.json` file will only contain a single slice of type `image`, which contains the image's path and dimensions. The path of this slice will point to the `source.{png, jpg, ...}` file in the root of the directory. This way, we do not need to treat images differently from other files, and we can rely on the slicing mechanism to handle them correctly. The same applies to Word documents and spreadsheets, which are also processed into a `contents.json` representation. There is no custom parser for Word documents, and the file is simply converted to a PDF file using LibreOffice before being processed further. There are no good libraries for easily accessing the images in a Word file, so while having this conversion step seems less optimal, it actually increases the usefulness of the document. Spreadsheets are processed natively in PHP using the PhpSpreadsheet library [53], with their contents being split into chunks based on a maximum row count.

```
- <artifact_id>/
    - metadata.json
    - contents.json
    - source.{pdf, docx, png, ...}
    - [marked.pdf]
    - images/
        - image1.jpg
        - ...
    - images_marked/
        - image1.jpg
        - ...
    - pages/
        - page1.jpg
        - ...
    - pages_marked/
        - page1.jpg
        - ...
```

Code Snippet 15: The directory structure of a `DiskArtifact`

An example `contents.json` file would look like this:

```
[
    {
        "page": 1,
        "type": "text",
        "text": "This is the text on the first page of the document. Lorem ipsum dolor sit amet..."
    },
    {
        "page": 1,
        "type": "page-image-marked",
        "mimetype": "image/jpeg",
        "path": "pages_marked/page1.jpg"
    },
    {
        "page": 1,
        "type": "page-image",
        "mimetype": "image/jpeg",
        "path": "pages/page1.jpg"
    },
    {
        "page": 1,
        "type": "image",
        "mimetype": "image/jpeg",
        "path": "images/image1.jpg",
        "x": 455.0,
        "y": 28.55999755859375,
        "width": 88.32000732421875,
        "height": 688.3200073242188
    },
    ...
]
```

Code Snippet 16: Example contents of a `contents.json` file

A `VirtualArtifact` is an implementation that does not have a directory on disk but is instead created in memory. This is useful for testing purposes, but it also allows for the creation of artifacts from other sources, such as a database or a remote storage solution. Instead of reading the contents from the disk, raw data can be passed to the `VirtualArtifact` constructor, which then creates the slices and pages in memory.

The common `Artifact` interface then defines methods for accessing this data, and for getting the contents of embedded slices no matter how they are stored. Another yet unmentioned implementation of an `Artifact` is the `SplitArtifact`, which only contains a subset of the original `Artifact`'s slices. They work with any kind of `Artifact`, as they keep a reference to the original `Artifact` internally, which is used to resolve image paths and contents when needed.

To split an Artifact, both the `ArtifactSplitter` and `ArtifactBatcher` are utilized. The first class is responsible for walking through the `contents.json` slices until either a token or image count limit is reached, after which a new split is created. The batcher on the other hand will combine `Artifacts` together with similar limits. While this seems contradictory at first, the combination of the two allows large files to be split so they can be included at all, while still filling the context window as much as possible. Not having this filling step would result in a lot of LLM calls being made, if a bunch of tiny documents were to be analysed, as each LLM call would be made with only a single split included.

4.3.2 Extracting the Text and Images from Multiple File Types

While Laravel has great support for managing, uploading and working with files, one thing PHP does not have great support or libraries for is working with PDFs. This is unfortunate, as PDF is one of the most common file formats for documents, especially in business settings. A language that does is Python, which has libraries like PyMuPDF [67] and pdf2image [68] that can directly read and work with PDF file contents.

For this reason, LLM Magic uses a custom Python script to extract the text and images from PDF files using the PyMuPDF library. PyMuPDF provides a Python binding for the MuPDF library, which is a lightweight PDF, XPS (*XML Paper Specification*), and E-book viewer written in the C language [69]. It not only allows for reading the text and image contents of a PDF file, but also for rendering the pages as images and even drawing on top of them. The latter is used to create the marked page screenshots.

This Python script is called from within a Laravel background job, which then stores the extracted text and images on the disk. We do not need any complex PHP to Python communication, as the Python script can simply write the extracted data to a JSON file, which is then read by the Laravel application. In case of an error during the execution, the Python script outputs a JSON object with an error message, which is then captured by the Laravel application and thrown as an exception. To make this easier, the `PythonRunner` class is a simple wrapper around PHP's `shell_exec` function to allow easy execution of Python scripts with parameter and error handling.

One difficulty with running Python scripts from PHP is that the Python environment needs to be set up correctly. This is especially true when considering that both LLM Magic and Data Wizard can be run in a variety of environments, such as local development on macOS or in a Docker container on a Linux server. Python is notorious for being difficult to set up correctly across different environments [70]. This is because of the many ways Python can be installed and configured, leading to the potential coexistence of different versions on the same system. Furthermore, managing Python packages in an environment-independent manner adds another layer of difficulty, as packages can be installed using various package managers such as `pip`, `conda`, or even non-Python-specific tools like `brew` on macOS.

Recently, there has been some developments to solve this problem, particularly with the introduction of the `uv` package manager [71]. `uv` is a universal package and project manager for Python that can be used to manage Python installations, install packages, and run scripts in a consistent manner across different environments. Additionally, running a script with `uv` does not require the dependencies to be installed beforehand, as `uv` will automatically install them when the script is run the first time [72]. This is useful for LLM Magic, as no additional setup is required to run the Python script, apart from having `uv` itself installed on the system.

The Python script itself is relatively straightforward, as it only needs to serve this one purpose. First, it creates the `DiskArtifact` directory structure at a given path, reads the file using PyMuPDF, and then extracts the text and images from the PDF. In a second step, it draws the markings around embedded images directly into the PDF and writes their image ID on top of them. Finally, it saves the modified PDF file as `marked.pdf` and then extracts the images and page screenshots again, this time to the `marked` directories.

Should the extraction fail, we can use the retry functionality of Laravel's background jobs to try again for a set number of times until finally giving up. Since the whole process is asynchronous, we keep track of the preparation status of a file using a `status` column in the database of the `File` model. This allows the user to observe if the preparation was successful, is still being processed, or if the processing has failed.

The same pre-processing logic applies to simple image and text files as well, with the exception that the Python script is not needed in these cases. Instead, we simply use the Laravel filesystem to write the file contents to the source file and then manually create the `contents.json` file containing a single slice of type text or image.

As mentioned before, Word support is implemented by converting the file to PDF and then re-using the logic above. To do this the Doxswap package is used, which is a PHP wrapper around the LibreOffice `soffice` binary [73].

4.4 Running the Extraction Process

Once the files have been pre-processed into `Artifacts`, the extraction process can begin. However, this is not as simple as just sending the files to the LLM, telling it to extract the data, and then receiving the results. While that is certainly the basic idea, there are lots of tiny details that need to be considered to make the extraction process as efficient and reliable as possible.

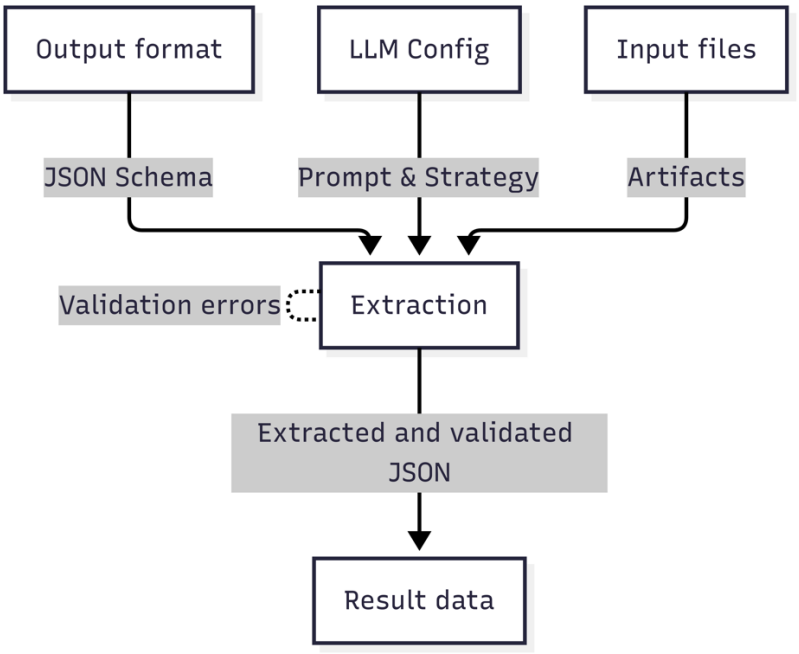


Figure 19: An overview of the data flow of the extraction process

For one, many large documents cannot be processed in one go by the LLM, as they exceed the maximum token limit. This fundamentally changes the way the extraction process needs to be designed, as the operation has to be split into multiple LLM calls. This splitting will then also add complexity to working with the output data, as it will also be split across multiple responses. For example a 100-page document might need to be split into 10 parts, each containing 10 pages. If each page contains information on a single vehicle, each LLM response would include a list of vehicles. In such a case, the data can be merged together relatively easily, as the returned lists can simply be concatenated.

A more difficult example would be a large document that contains information about a single vehicle. In this case, the return data would not be a list but a single object, which would need to be merged from multiple responses. Some of the input chunks would have more of the relevant data than others, while some pages might just contain prose that is useless for what we're trying to extract. Merging this data together correctly is not easy to do in a generic or pre-defined way, as we cannot know which fields from which response should take precedence over others. How these fields are merged highly depends on the kind of data that we're working with, and the optimal solution will differ from case to case.

Additionally, if the LLM processes certain chunks entirely independent of the others, some calls may get a totally different idea of what the data is about. This can lead to big problems when merging the data back together, as the LLM might have made different assumptions about the data in each chunk, which may be incompatible with each other. This is a problem that Data Wizard needs to be able to solve, and it does so by allowing the user to choose between different extraction strategies.

4.4.1 Choosing the Right LLM

Just as critical as choosing an extraction strategy is choosing the right LLM to accomplish the task. The marketplace for LLMs has grown massively since the release of ChatGPT and the GPT-3 API, with other large software players like Google and Meta also releasing their own models. Just on Hugging Face alone, almost 200.000 text generation models are available, which, while mostly being fine-tuned versions of the same base models, still shows the sheer amount of choice that is available [74].

Performing data extraction on a variety of tasks in a reusable way is helped by the fact that the big LLMs handle in-context learning really well. This allows a user to simply specify what they want as part of the prompt, and maybe add a few examples of what good output looks like, and the LLM will adapt to this context [21]. This is especially useful for software like Data Wizard, where an *"out of the box"* experience can be provided that should work for most use-cases. While the larger model families focus on a broad audience, trying to encompass as many use-cases as possible, the smaller models are often more specialized and fine-tuned for specific tasks. Using fine-tuned models can have both performance and cost benefits compared to using generalized LLMs [75], which is especially true when the savings are cumulative if the model is being used lots of times. When fine-tuning, smaller-sized base models can be used, while still resulting in on-par or even better result quality [76]. This can significantly reduce the cost of running the model, also making it much more feasible to run them on local hardware. Solutions like Ollama provide OpenAI-compatible APIs for locally running LLMs, which LLM Magic already has support for [36]. This lets a user get started with bigger models, and then switch to smaller, more specialized models later. Data Wizard could even be used in helping create a dataset for fine-tuning a model, by using the bigger models to generate the initial few examples, which can then be used to train or fine-tune a smaller base model on. But for now, a focus on the larger LLMs provides a plentiful amount of choice to work with.

Data Wizard can take advantage of the already large variety in LLM quality and cost of the big players, by allowing the user to choose which model to use for the extraction process. There are a few factors to consider when deciding on what model to use, which are worth discussing in more detail. Mainly, the choice of model can be broken down into three main factors: model performance, context length, and cost.

4.4.1.1 Model Performance

Different LLMs have different performance characteristics for different tasks based on their training methods and datasets. Depending on how, how long and what kind of data the models were trained with, the data extraction performance can vary greatly. If a model has not seen a lot of JSON data during training, it will likely perform worse on JSON extraction tasks than a model that has seen a lot of JSON data in its training dataset. However, this is not a big problem for our case, since we'll be focusing on the current generation of LLMs by the big LLM laboratories, namely OpenAI, Anthropic, Google and Mistral. Their publicly available frontier models are state-of-the-art and have been trained on a wide variety of data, making them suitable for a wide range of tasks.

All these providers offer multiple models, with different parameter sizes and performance characteristics. Either the model is released with different sizes (e.g. Llama 3.2 with 1B, 3B, 11B, 90B and 405B parameters [77]) or grouped into different families and quality presets (e.g. `gpt-4o`/`gpt-4o-mini` or `claude-3-opus`/`claude-3-sonnet`/`claude-3-haiku`). The choice of model will greatly influence the extraction performance. On the DocBench benchmark GPT-4 scores a `69.8` compared to GPT-3.5's `49.6` result. Claude 3 Opus scores `67.6` on the same benchmark [78]. The more recent Livebench benchmark [79] has more up-to-date data, including more recently released models like Deepseek R1 and OpenAI o1 [80].

The model performance across different tasks is what LLM makers generally compete on the most, as the goal of some of them appears to be to create "*Artificial General Intelligence*" (AGI) that can replace a human at most if not all digital tasks [81]. New models are released regularly, with the performance of the models improving with each new release. For this reason, looking at benchmarks like Livebench can be helpful to determine which model has the most fitting performance characteristics for a given task.

The following table shows the performance data of some popular LLMs according to recent evaluation on the Livebench leaderboard [80]:

Model	Global Average	Coding Average	Data Analysis Average
o3-mini-2025-01-31-high	76.69	82.74	70.64
claude-3-7-sonnet-thinking	74.30	74.54	74.05
deepseek-r1	68.26	66.74	69.78
o1-2024-12-17-high	67.58	69.69	65.47
o3-mini-2025-01-31-medium	65.97	65.38	66.56
claude-3-7-sonnet	65.43	67.49	63.37
chatgpt-4o-latest-2025-01-29	63.28	60.56	66.00
gemini-2.0-flash	60.73	53.92	67.55
o3-mini-2025-01-31-low	61.75	61.46	62.04
claude-3-5-sonnet-20241022	61.08	67.13	55.03
gemini-2.0-flash-exp	58.02	54.36	61.67
gpt-4o-2024-08-06	56.17	51.44	60.91
gemini-2.0-flash-lite	56.26	47.08	65.45
gpt-4o-2024-11-20	51.12	46.08	56.15
gemini-1.5-pro-002	51.88	48.80	54.97
gpt-4-turbo-2024-04-09	51.68	49.00	54.36
chatgpt-4o-latest-0903	52.68	47.44	57.93
o1-mini-2024-09-12	52.99	48.05	57.92
gpt-4o-mini-2024-07-18	46.56	43.15	49.96
claude-3-5-haiku-20241022	49.91	51.36	48.45
claude-3-opus-20240229	48.24	38.59	57.89
mistral-large-2411	48.62	47.08	50.15

Table 1: Benchmark results of modern LLMs according to the Livebench leaderboard

For Data Wizard, models that perform well on Data Analysis and Code Generation tasks are likely to perform well on data extraction tasks as well, since dealing with and generating JSON data can be associated with both areas. However, Livebench does not offer a specific data extraction benchmark for up-to-date comparisons.

4.4.1.2 Context Length

Another key factor to consider when choosing the LLM is the context length. This is a critical parameter to the extraction process, as it determines the maximum amount of data that can be processed in one go, thus putting a strict limit on the chunk size.

LLMs have a maximum token limit for both input and output. This is due to the underlying architecture that LLMs use, which is the Transformer and its self-attention mechanism [82]. The architecture results in memory usage that is scaling quadratically with the length of the input data. This means that the longer the input, the more computationally expensive it is to process. While some model architectures provide tricks to work around these issues, it remains the fundamental problem.

Context sizes of early LLMs like GPT-3 were relatively restricted, only allowing for a context window of about 2048 tokens. Since then, lots of advancements have been made in making larger context windows possible, with models like GPT-4 having a context window of up to 32k tokens [83], Claude-3-family models supporting up to 200k tokens [84] and newer Gemini models going all the way up to 2 million tokens [85].

The following table provides an overview of the context lengths of some state-of-the-art LLMs [85] [86] [87] [88] [89] [90] [91] [92]:

Model	Max Input Tokens	Max Output tokens
gemini-1.5-pro	2M	8k
gemini-2.0-flash, gemini-2.0-flash-lite	1M	8k
Anthropic claude-3.5-sonnet, claude-3.5-haiku)	200k	8k
OpenAI o1	200k	100k
OpenAI o1-mini	128k	65k
OpenAI gpt-4o, gpt-4o-mini, gpt-4-turbo	128k	16k
OpenAI gpt-4-turbo	128k	4k
Meta llama-3.2-90b	128k	2k
Mistral Large 2	128k	8k
gpt-4-32k	32k	8k
gpt-4	8k	8k

Table 2: Input and output context sizes for some popular LLMs

By having the largest context windows, Gemini models are very suitable for data extraction tasks, as big parts of the document, if not the entire thing, can be included in a single LLM call. This has significant advantages over models with smaller context windows, as the chunking process could potentially be avoided entirely, leading more contextual understanding of the data. However, with only 8.000 output tokens being available, the model might not be able to fit the entire extraction results into its output, especially when strategies with LLM-based merging are used. A trade-off might have to be made, where a model like GPT-4o or a different strategy is used, so that the output can still be generated. As evident in the table, output tokens are generally much more restricted than input tokens, as the latter seems to have been prioritized by LLM makers. However, the trend shows this number also increasing with each new model generation, so this might be a problem that solves itself, given enough time.

For the moment, an application like Data Wizard will have to work around this limitation, as some documents can exceed the context length of even the largest models. But for the moment, the context length is a limiting factor that must be considered when choosing the right LLM.

4.4.1.3 Cost per Token

Lastly, the actual cost per input and output token is something that also has a big impact on running extraction processes too. While not directly connected to output quality, using a model that is super expensive to run can quickly add up to a large bill, reducing the cost-effectiveness and applicability of the application. This is especially true for large documents, which result in large amounts of input tokens and possible multiple LLM calls.

LLMs are expensive to run and this cost is measured in cost per token. Often, units like *"price per million tokens"* are used to make the cost easier to understand. But there is no direct correlation between capabilities or context size of a model and its price, as the price is set by the model provider and can be influenced by many factors. However, especially the complexity of deploying and running the model, as well as any training costs, seem to have a big impact on the price, where size of the model has been shown to have some correlation with model performance [93]. This notion is somewhat challenged by groundbreaking releases like Deepseek's R1 model [22], which shook up the stock market as they claim to require drastically lower computational resources to train and run [23], making investors question the moats of the big LLM makers.

For Data Wizard, the cost per token is important to keep in mind when selecting the right model, as some models can be much more expensive to run than others. For example, using Claude 3 Opus on a large document can easily result in a bill of tens of dollars per extraction run [94], while using a Gemini model might only cost a few cents [95]. This is especially important when running multiple extraction processes in parallel, as the costs can quickly add up. To track the cost of extraction runs, Data Wizard is aware of the cost per token of most models and can provide cost calculations for each extraction run in the backend.

Model	Cost per 1M input tokens	Cost per 1M output tokens
gpt-4.5-preview	$75.00	$150.00
gpt-4-32k	$60.00	$120.00
gpt-4	$30.00	$60.00
claude-3-opus	$15.00	$75.00
gpt-4-turbo	$10.00	$30.00
o1	$15.00	$60.00
claude-3.7-sonnet	$3.00	$15.00
claude-3.5-sonnet	$3.00	$15.00
gemini-1.5-pro	$2.50	$10.00
gpt-4o	$2.50	$10.00
mistral-large	$2.00	$6.00
o1-mini	$1.10	$4.40
claude-3.5-haiku	$0.80	$4.00
gemini-2.0-flash	$0.10	$0.40
claude-3-haiku	$0.25	$1.25
gemini-1.5-flash	$0.15	$0.60
gpt-4o-mini	$0.15	$0.60
gemini-2.0-flash-lite	$0.075	$0.30

Table 3: Cost per one million tokens of some popular LLMs

By comparing Table 2 and Table 3, it becomes apparent that the cost per token is not necessarily tied directly to the model's performance. The GPT-4 32k model is one of the most expensive to run, but is generally outperformed by newer models like Claude 3 Haiku or GPT-4o mini. Since LLM usage has grown massively in the last years, LLM makers have to optimize the cost of running their models as well as the performance. As such, some older, weaker models are still in use but more expensive to run than newer, more powerful models.

Since the cost itself is not a technical problem, but rather a business one, it is not something that can be solved in the application itself. However, by providing the user the ability use a variety of models, the user can choose the model that best fits their budget requirements.

4.4.2 Prompting the LLM

The way an LLM prompt is structured is one of the most crucial aspects of ensuring high-quality extraction data. How an LLM understands the task to perform will greatly influence the quality of the extracted data. The system prompt is what most LLMs have been trained with to take their configuration from. While LLMs certainly also listen to instructions given in messages later on, most models have been reinforced for system instructions to take precedence over user instructions. This also has security reasons, as a system prompt may contain safety instructions that the user should not be able to override. While this does not mean that LLMs cannot be tricked into ignoring their system instructions [96], it does mean the system prompt is also very relevant for extracting data from documents, as any instructions in there are more likely to be followed by the model. A well-defined system prompt should explicitly define the AI's role and enforce strict adherence to the specific task at hand. By doing so, the model avoids making assumptions, generating unnecessary content or deviating from the intended response format.

In our case, the system prompt should guide the LLM to act as a strict data extractor, without making assumptions or filling in gaps with information that it made up. The system prompt needs to clearly instruct the model to extract data only according to the defined JSON schema, never adding or removing properties. This kind of instruction can help to prevent the model from inventing new properties, maintaining data integrity and potentially reducing the validation error rate. Furthermore, the system prompt includes rules for handling missing data by using `null` instead of dropping the property. During testing without this instruction, LLMs kept using nonsensical values for missing data, which makes it much harder to merge the data back together in a meaningful way. Explicitly using `null` makes it clear that no suitable data was found. Obviously, the output data schema also needs to allow this.

Depending on the strategy, we may want the model to reason about the data for a couple of sentences, which can improve the quality of the extraction results as potential difficulties are mentioned before starting the JSON output. Since LLMs cannot backtrack or correct themselves during the inference process, once output has started, the model cannot go back and change it. For example, when a JSON object is opened with `{`, the model will need to fill this object with data before closing it again with `}`, even if the JSON object itself is illogical. Letting the model reason beforehand could reduce these kinds of errors as any such issues are thought about before the JSON output starts. This approach is partly what makes newer models like OpenAI's o1 perform so well on various benchmarks [97].

In addition to understanding the data extraction task, the system prompt should also include information on the data that we provide it with. While just dumping the document contents into the context will result in some output, adding metadata about the document like page numbers or image IDs helps the model to understand the general shape of the document. Telling the model that it is being given

only a part of the document lets it know that it should not expect the full context to be present. Adding explanations about how the input data has been prepared can also help the model to understand why some sections of text may be missing or end abruptly, as they are contained in a different split.

While it is technically down to each strategy how the file contents are included in the prompt, the built-in strategies all use the same approach by wrapping the contents in XML tags. The following is an example of how an artifact is represented in the prompt:

```
<artifact id="<ARTIFACT_UUID>">
    <page num="1">
        <text>This is the text on the first page of the document. Lorem ipsum…</text>
        <image name="images/image1.png" />
    </page>
    <page num="2">
        <text>This is the text on the second page of the document. Lorem ipsum…</text>
        <image name="images/image2.png" />
    </page>
</artifact>
```

Code Snippet 17: Artifact representation within an LLM prompt

Using XML tags in the prompt has the advantage of being easily readable and understandable by both humans and machines. The explicit closing tags also reduce ambiguity, as the model can clearly see where a page or document starts and ends. Anthropic even mentions in their documentation that using XML tags can improve the quality of the output, as Claude-3 family models are trained on XML-based prompts for their chatbot application [39].

To let the model assign images as data points, we need to specifically instruct the model how this works. Since we want to be able to show the images in the output UI as well, it is necessary to define a specific way to reference the images in the output data. Each embedded image in an artifact will be available via an API so they can be displayed in the UI or downloaded by a program using the output data. To accomplish this, we use a specific ID structure for these images, that can then just be included in the extracted data as a string:

```
artifact:<ARTIFACT_ID>/images/image1.png
```

By instructing the model to always reference files in the format above, we can ensure that we can always find the correct image for a given data point later. Generally, the models are instructed to always use this format when referencing any images that it has been provided. It helps to also explicitly mention this in the provided JSON schema, as the locality and a reminder of this format can help the model to remember it better. For Data Wizard to show the images in the UI, some custom JSON Schema properties are required to define the property as containing a reference to an image:

```
{
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "avatar": {
      "type": "string",
      "format": "artifact-id"
    }
  }
}
```

Code Snippet 18: Example of how Artifact IDs can be included in JSON Schema

4.4.3 Simple Extraction Strategy

The simple extraction strategy is the most fundamental approach provided by Data Wizard. It involves a single LLM call, where the model is given as much of the document as possible within the token limit. This strategy is straightforward and efficient for small documents, as it provides the LLM with the maximum context available. However, it is not suitable for larger documents that exceed the token limit, as the excess content is simply ignored, resulting in incomplete extraction results.

Despite its limitations, the simple extraction strategy is valuable for scenarios where the document size is manageable, and the entire context can be processed in one go.

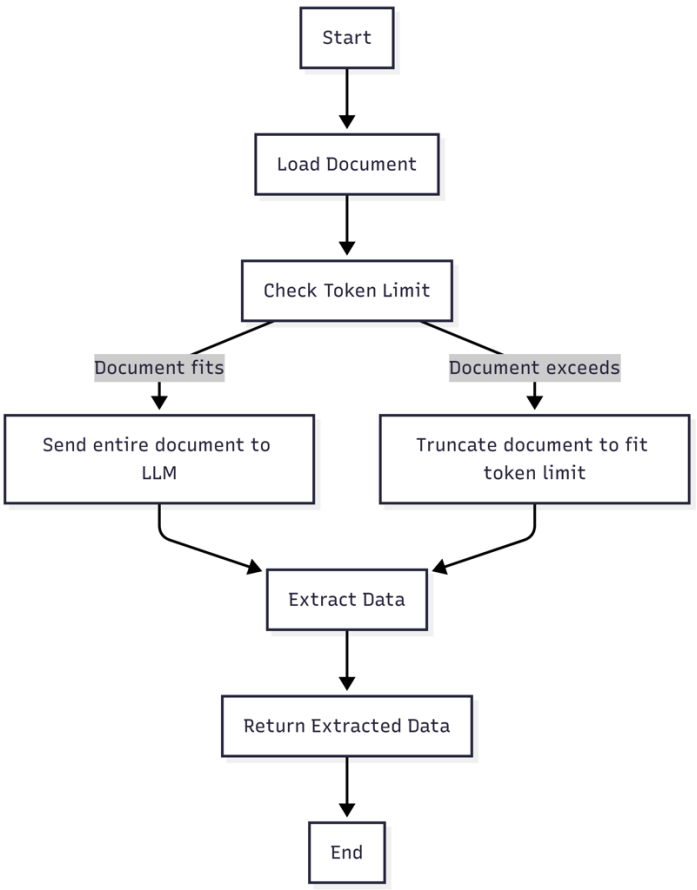


Figure 20: A flowchart of the simple extraction strategy

4.4.4 Sequential Extraction Strategy

The sequential extraction strategy is designed to handle larger documents more effectively than the simple strategy. In this approach, the document is split into smaller, manageable parts based on the conigured chunk size or the token limit of the LLM. Each part is processed sequentially, with the results of the previous extraction included in the prompt for the next part. This allows the LLM to build upon the previously extracted data, maintaining contextual continuity throughout the document.

This strategy is particularly useful for documents where information is spread across multiple pages, such as real estate exposés or detailed reports. By incorporating the results of previous extractions, the LLM can better understand the overall structure and context of the document. For example, in a real estate exposé, the initial pages might contain general information about the property, while subsequent pages detail specific rooms or floors. The sequential strategy ensures that the LLM has access to the general information when processing the detailed sections, leading to more accurate and coherent extraction results.

However, the sequential strategy can be time-consuming, as each part of the document must be processed one after the other. Additionally, the accumulation of results from previous extractions can lead to increased token usage, potentially limiting the amount of new content that can be processed in each call. Also, the chance that an LLM drops or looses some data increases the more it has to "*carry around*" from call to call, as it can just omit previous data, even if doing so by accident.

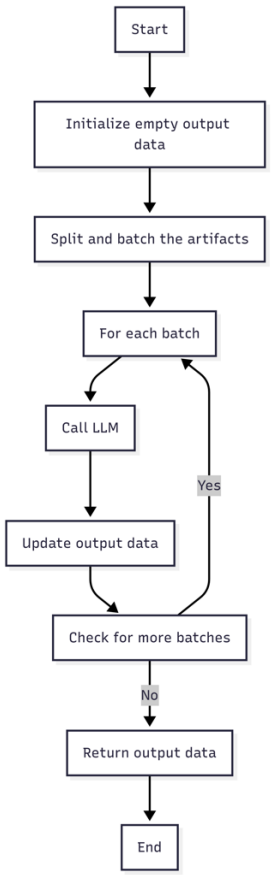


Figure 21: A flowchart of the sequential strategy

4.4.5 Parallel Extraction Strategy

The parallel extraction strategy is designed to process large documents efficiently by splitting the document into independent parts and processing each part in isolation. Unlike the sequential strategy, the parallel strategy does not pass the results of previous extractions to subsequent calls. Instead, all parts are processed independently, and the final data is merged at the end.

This strategy is well-suited for documents containing multiple independent data points, such as product brochures or catalogs with numerous products per page. Each product can be extracted independently, as there is no direct relation between them. This approach also allows the extraction process to be parallelized, massively shortening the execution time. Since the LLM requests do not need to wait for each other, they can be sent at the same time. This is only being hindered by API rate-limits and other processing constraints. Currently, the implementation executes up to 3 requests in parallel, but this is configurable.

The parallel strategy offers the advantage of reduced token usage compared to the sequential strategy, as there is no need to include previous extraction results in each call. However, it lacks the contextual continuity provided by the sequential strategy, which can be a disadvantage for documents where information is interconnected across pages.

A limitation of this approach is that the final data must be merged at the end using another LLM call, which can be challenging for complex data structures. The LLM no longer has access to the individual parts of the document, making it difficult to resolve conflicts or inconsistencies in the extracted data. In addition, if all the separate parts together exceed the output token limit of the LLM, the final merge call will fail. This practically only really occurs when extracting from very large documents, as the extracted data is usually much smaller than the input data, but it is still a potential issue.

An improvement to the strategy could be by running repeated merge calls, each time with a smaller subset of the data in a pyramid-like structure to avoid the output token limit. However, this is quite complex and currently not implemented as other strategies offer themselves as potential alternatives instead.

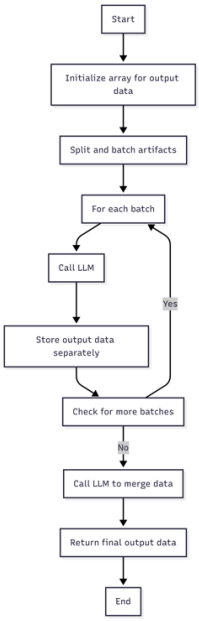


Figure 22: The flowchart of the parallel strategy

4.4.6 Sequential and Parallel Strategies with Auto-Merging

Both the sequential and parallel strategies are available as auto-merging variants. The main aspects of the strategies work in the same way as described above, with the difference being in the way that the data is merged back together at the end. They also only work for data schemas that consist solely of array properties at the top-level. For schemas that describe a single entity, these strategies do not work.

There is no separate data merging step at the end of the original sequential strategy, but its auto-merging variant continuously combines the generated output data with the previous input data. It does so by concatenating together the items of the top-level properties. Unfourtunately, this can potentially result in duplicate items being present. To remedy this, the strategy automatically removes duplicates that show a 100% match using hash-based data comparisons. However, this will not find duplicates if things like string representations of labels differ slightly. For this reason, the strategy runs performs a final LLM call at the end which is tasked to deduplicate the final results. This LLM does not have to return the full data again, and is instead asked to return a list of dot-notated keys to remove from the dataset. For example, it may request to remove `products.3` and `products.12`. Dot-notated keys are used instead of only the indices to support multiple array properties being present in the same schema, so it can remove `products.3` and `categories.5` at the same time.

The auto-merged parallel strategy works practically exactly like the sequential one, with the difference being that there is only a single auto-merge step at the very end. However, the same deduplicating logic is used, as the risk of having duplicates is similarly high.

These strategies work around both the output token limit problem and the issue of data potentially being dropped. Since data is continuously saved, it does not matter if an LLM "*forgets*" to include a certain entity in between inference calls. Neither is it possible for the LLM to overflow its output token window during the merging step, as it only has to output small amounts of data to drop the duplicates.

4.4.7 Double-Pass Extraction Strategy

The last strategy that is implemented out of the box is the double-pass strategy, which combines elements of both the sequential and parallel strategies together to enhance the accuracy and completeness of the extraction process. In the first pass, the document is processed using the parallel strategy, extracting data from each part independently. In the second pass, the results from the first pass are reviewed and refined using the sequential strategy, including the already generated output data while going through the data another time.

This approach aims to leverage the strengths of both strategies while mitigating their weaknesses. The initial parallel pass ensures that all parts of the document are processed efficiently, while the subsequent sequential pass provides the contextual continuity needed for accurate extraction. This strategy is particularly useful for complex documents where information is both independent and interconnected across different sections or where data accuracy is critical. During the second pass, the model can correct any errors or inconsistencies of the initial extraction results, leading to higher-quality output data overall.

Obviously, this strategy is the most resource-intensive of all, as it requires two full passes over the document, doubling the number of LLM calls needed. But in some cases, it can be worth the extra time and resources to ensure the highest quality extraction results.

While the first two strategies profit a lot from using a very capable LLM, the double-pass strategy with its inherent error correction allows for using a less capable but cheaper LLM instead. This can potentially offset the increased costs of the strategy, as a cheaper model can correct its own errors in the second pass. But whether or not this makes sense largely depends on the document and output data schema.

Since this strategy potentially suffers from the same output token limits, it also has an auto-merging version. The strategy itself is no different, but instead the auto-merging variants of both the sequential and parallel strategy are used.

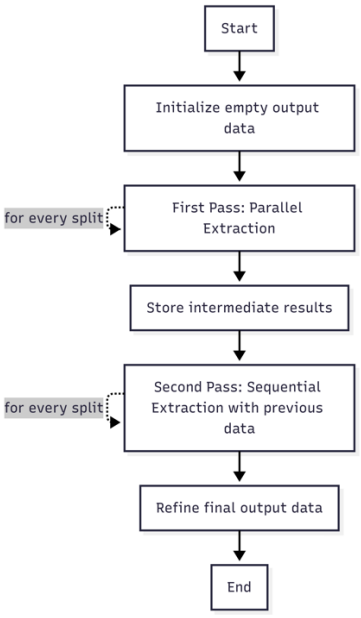


Figure 23: A flowchart of the double-pass strategy

4.4.8 Strategy Options

Each of the built-in strategies can be customized using a set of options that define how the extraction process should be run. In particular, it can be configured how to include the document contents in the prompt.

By default, only the text and the embedded images are included in the prompts. But the user can also choose to include the both the unmodified page screenshots or the marked page screenshots in the prompt. This is configurable, as the page screenshots can be quite large and may not be necessary for all extraction scenarios. Also, the marked screenshots may actually reduce the output quality, as the inserted numbers can potentially cover up important parts of the image.

Figure 24: An example where the markings cover important data

Also configurable is the chunk size, which defines how large each split of the document should be. Normally, this is determined by the model's context length limit, with the split being allowed to take up about two-thirds of the context window. This should allow for a good amount of data to be sent to the LLM, while still leaving enough space for the model to generate a meaningful response.

The token calculation is not very accurate, as Data Wizard is not actually using the correct tokenizers to determine the token counts and instead uses a rough estimate of around ~4.5 characters per token. This number has been determined by experimenting with OpenAI and Anthropic tokenizers and calculating an average character count per token. But the actual token count can vary greatly depending on the language and the text itself.

For the usecase that Data Wizards needs the token count for, this rough estimate is good enough, as its only used to split the documents. Since the context window is not being used filled all the way to the maximum anyway, some inaccuracies in the token count are acceptable.

4.4.9 Benefits of Building Custom Strategies

The strategies provided by Data Wizard are designed to cover a wide range of document types and extraction scenarios and are designed to be applied independently of the document contents. But there are cases where a custom strategy may be necessary to achieve the desired results.

An extraction strategy is inherently just a self-contained class that calls an LLM with some input data it is given and is eventually expected to return some output data. What happens within it is actually very flexible. Thus, custom strategies can technically implement any custom logic that may be required. While this seems like a small sidenote, this is actually quite powerful, as the strategy can rely on all of the abstractions of the rest of the application to perform its job. If a developer decides to write a custom strategy, they will not have to worry about implementing file loading or chunking logic unless they absolutely want to. This makes Data Wizard a great platform to implement custom data extraction pipelines.

While a custom strategy can certainly be used to optimize the quality of the extracted data, another potential use-case for writing one is to optimize the costs of the extraction process. Since the strategy does necessarily have to use the model that is configured in the SavedExtractor model, different models can be used for different parts of the process. While this is difficult to implement in a generic and widely usable way, it can be a powerful tool for reducing costs in specific scenarios. For example, the double-pass strategy could be optimized by using a cheaper model for the first pass and a more expensive model for the second pass only.

4.5 Using and Integrating with Data Wizard

4.5.1 REST and GraphQL API

Since Laravel is a robust and battle-tested web framework, building a REST API with it is at the core of its functionality. Laravel's controllers [98] and routes [99] make it easy to define the endpoints and the logic for handling requests and responses. However, an even easier way to build REST APIs than to define every route and controller manually is to use the API Platform integrations for Laravel [54].

API Platform is very powerful library for both the Laravel and Symfony [100] web frameworks that allows building APIs not by manually defining endpoints, but rather by defining data structures and how they should be exposed. In turn, API Platform dynamically routes and processes the requests to interact with the data. This reduces the amount of boilerplate code that needs to be written and makes the API more maintainable and easier to extend. Additionally, the API is automatically up-to-date with the data model, as changes to the data structure are automatically reflected in the API. While one should be aware of introducing breaking changes without noticing, it simplifies API development enough to make this trade-off worth it.

In addition to building a functional API, API Platform also generates a fully-featured API documentation using the OpenAPI specification, which can be used to test the API interactively in the browser and to even generate client libraries in other languages. In fact, the API documentation for Data Wizard is available at `https://DATA_WIZARD_URL/api/` and which can be used to explore the available endpoints and interact with the API directly.

In addition to an HTTP API, API Platform also supports accessing its endpoints via GraphQL [101], which is a modern query language for APIs that allows clients to request only the data they need. While I'm personally not very experienced with GraphQL, there are powerful open-source libraries for working with GraphQL and its very popular in the React ecosystem. Since there is no additional work required to expose the API via GraphQL, enabling these makes Data Wizard even more accessible to other applications.

Functionality-wise we only really want to expose a couple of endpoints, namely to create buckets as well as to start extraction runs and to retrieve their results. While the API could certainly be expanded to also include endpoints for managing the models and the extractors, this is not a priority for now.

Defining endpoints with API Platform is as easy as adding an attribute to the model class:

```
#[ApiPlatform]
class ExtractionRun extends Model
{
    // ... model code ...
}
```

Code Snippet 19: Example of how easily API Platform can be added to existing Laravel models

The `ApiPlatform` attribute also allows for some options to modify API settings like validation rules and access control. Securing the API is handled by Laravel's first-party API authentication system Sanctum, which allows for easy token-based authentication for APIs [102]. The API is then configured to use the `auth:sanctum` middleware, after which all requests require API tokens to be present. Tokens can be generated in the backend settings page and can then be passed to the API using the `Authorization` header.

Using the API, applications can build completely custom UIs, either completely hiding the fact that Data Wizard is used in the background or only embedding the UI once it is in the correction phase.

4.5.2 Installing and Configuring Data Wizard

The easiest way to run Data Wizard is by just pulling the Docker image and running it with a few environment variables set. This ease of use is one of the main advantages of using Docker, as it allows for a consistent and reproducible environment across different systems. However, Data Wizard can also be run like any other Laravel application, be it through something like Nginx or Apache or using something more modern and performance-optimized like Swoole or FrankenPHP.

```
docker run \
  -p 9090:80 \
  -p 4430:443 \
  -p 4430:443/udp \
  -v storage:/app/storage \
  -v sqlite_data:/app/database \
  -v caddy_data:/data \
  -v caddy_config:/config \
  -e APP_KEY=<REPLACE_WITH_APP_KEY> \
  mateffy/data-wizard:latest
```

Code Snippet 20: The short command needed to launch a Data Wizard instance

The `APP_KEY` environment variable is required for the server to be able to run securely, as this key is used for any encrypted values, including the session. Additionally, the server requires certain volumes to be configured so that data can be persisted across executions.

Once the server is running, navigating to `https://localhost:4430` will open the application. The browser might warn about invalid HTTPS certificates, but this can be safely ignored for now, as the server has to use local signing keys because it is running on `localhost`. When launching for the first time, Data Wizard will show the setup UI, where the user can create a new superadmin account. After this step, Data Wizard is ready to be used and API tokens for LLM providers can be set in the application settings. When not using Docker to run Data Wizard, it is important to disable the queue worker timeout, as extraction runs can take a long time to complete and the worker should not exit before the run is finished.

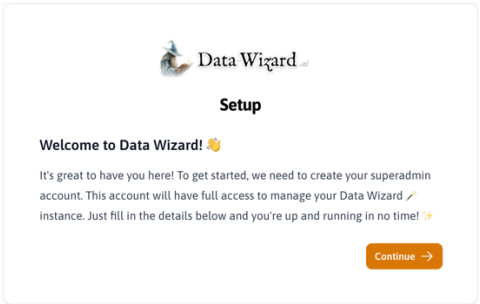


Figure 25: Data Wizard's welcome screen

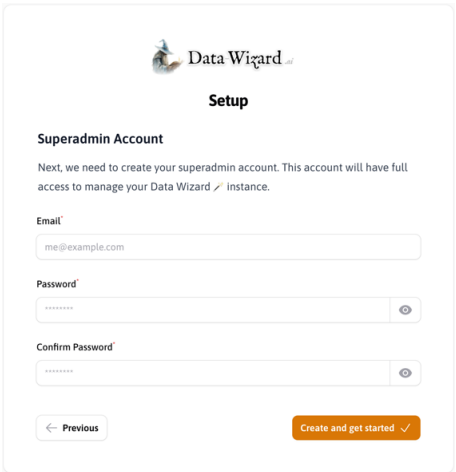


Figure 26: Creating the initial user account

4.5.3 Writing Custom Strategies

While writing the strategies themselves is not difficult, as they are just classes that implement a simple interface, adding them to Data Wizard is a bit more involved. The strategies themselves are part of the extraction logic of the separate LLM Magic package, and not directly part of Data Wizard application itself.

```
class SimpleStrategy extends Extractor
{
    /** @param Artifact[] $artifacts */
    public function run(array $artifacts): array
    {
        [$limitedArtifacts] = $this->getBatches(artifacts: $artifacts);

        $prompt = new ExtractorPrompt(
            extractor: $this,
            artifacts: $limitedArtifacts,
            contextOptions: $this->contextOptions
        );
        $threadId = $this->createActorThread(llm: $this->llm, prompt: $prompt);

        return $this->send(threadId: $threadId, llm: $this->llm, prompt: $prompt);
    }

    public static function getLabel(): string
    {
        return __('Simple');
    }

    public function getEstimatedSteps(array $artifacts): int
    {
        return 1;
    }
}
```

Code Snippet 21: The actual implementation of the `SimpleStrategy`, demonstrating the ease of writing them

At the moment the source code of Data Wizard application needs to be forked in order to add custom strategies. Then however, placing the following code in the `AppServiceProvider` is enough to add the class to the list of available strategies.

```
class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        Magic::registerStrategy('my-custom-strategy', MyCustomStrategy::class);
    }
}
```

Code Snippet 22: Example showing how easily new strategies can be added to LLM Magic

After this, the strategy is ready to be used and can even be selected in the Data Wizard settings UI. In the future, the forking requirement could be removed by adding the possibility to inject code into the Docker container.

5 Evaluation

Data Wizard contains some evaluation functionality that makes it easy to compare strategies and models across extractors and datasets. It can spawn multiple extraction runs with different configurations at once and then has interactive charts and tables to compare the results of these runs.

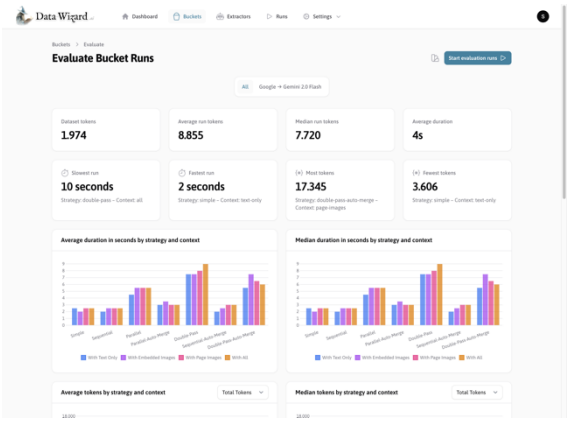


Figure 27: Built-in evaluation functionality

To accurately compare the performance and quality of the data extraction process, a set of both qualitative and quantitative metrics is used to measure the quality of the extracted data.

Since different strategies call the LLM in different ways, both the token usage of each strategy and the duration of the extraction process are important metrics that are used to evaluate the relative performance. Both also directly relate to the usability and cost of the extraction process, as more LLM calls and more tokens used mean higher costs and longer extraction times. One factor to consider for both of these metrics, is that they are not directly comparable between different LLMs. Internally, each LLM uses its own tokenizer which will lead to different token counts for the same prompts. Also, since the context window size differs, some LLMs simply require more LLM calls to process the same amount of data.

These tokenization and context window differences make it difficult to compare the extraction strategies themselves. For this reason, all evaluations will be run using two models of the same model family: Google's `gemini-2.0-flash` and `gemini-2.0-flash-lite` models, which are both cheap to run and still cutting-edge in terms of performance and quality. Using two models with different performance characteristics allows for more meaningful results. The Gemini 2.0 family models have a large context window of up to 1 million tokens, which would result in most of our example datasets fitting into a single LLM call. While this would be great for production use-cases, it diminishes the differences between the strategies, as they all can process the whole document in one go. For this reason the context window used in the evaluations is artificially limited to 15.000 tokens. Also, while getting these evaluation results, the tokenization counts are created using the internal LLM Magic tokenizer, which uses averaged character counts to estimate the token count. This is not accurate enough to calculate exact costs, but results in consistent token counts, which would make these results directly comparable if executed with different models. In addition to quantitative metrics like token count and execution speed, the quality of the extracted data is evaluated as best as possible. For this, quality indicators are identified, calculated and compared.

5.1 An Example With Many Entities: Supermarket Brochures

The example brochure used spans 38 pages, all of which contain information about the current product offers available at LIDL (© LIDL Schweiz AG) [103]. But not all of these products are generally discounted. While some of the discounts are only available for customers with a membership card, some producs are only listed as *"Highlights"* and are still listed at their original price. Product photos are also available and directly embedded into the PDF as separate files.

Figure 28: LIDL brochure cover

The specific LIDL brochure used contains a total of 173 products, of which 71 have some kind of discount attached. 102 products are simply listed with their regular price. This amounts to 144 products which would be relevant, as discounts for 29 products only apply to customers with memberships. Since the prompt includes a mention of not including the latter, the expected number of producs resulting from the extraction is thus around 144 products.

The schema used on this dataset includes a `products` array, each with a `name`, `original_price` and `discounted_price`, with the first two being required properties.

Not included in the evaluation results of this example is the `simple` strategy. Since we have (artificially) limited the chunk size to 15.000 tokens, the strategy does not have access to the entire dataset, thus heavily skewing the evaluation results. Additionally, the strategy does not perform any better if the limit is removed and the entire document is included. While large context models like Gemini 2.0 can handle the entire file, the amount of products is too large to fit into the output context window, with the execution of the strategy failing almost every time.

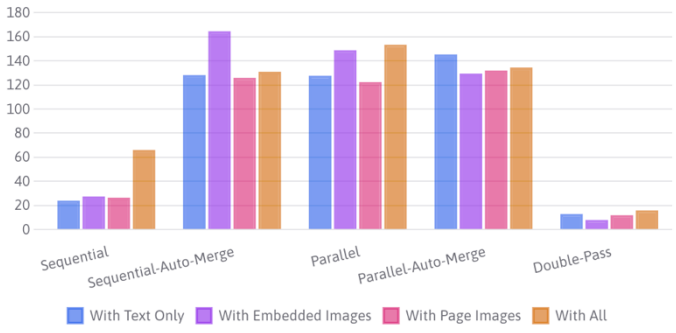


Figure 29: Average total number of products extracted (LIDL Brochure)

The evaluation results from Figure 29 indicate that strategies with auto-merging functionality appear to handle the dataset really well, whereas both the `sequential` and `double-pass` strategies seem to struggle with this extraction task.

A possible explanation for this lies in the nature of the `sequential` strategy, which is relevant to both. The LLM is prompted to output the entire existing data in addition to any new data that it has found. This instruction appears to not be followed very effectively, with the models only returning a subset of the previous data. For an extraction task such as this, it makes them effectively useless, as the final list of products will mostly contain items from the last few pages, mainly ignoring the rest of the document. Since the auto-merging and `parallel` strategies keep track of any data that has already been created, they still have the full list of products at the end, regardless of the intermediate LLM output.

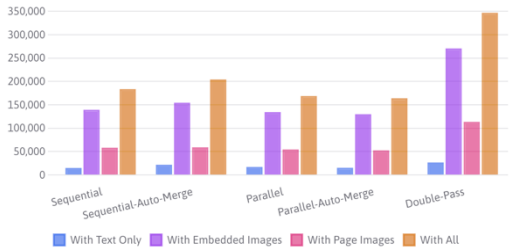


Figure 30: Average token usage (LIDL Brochure)

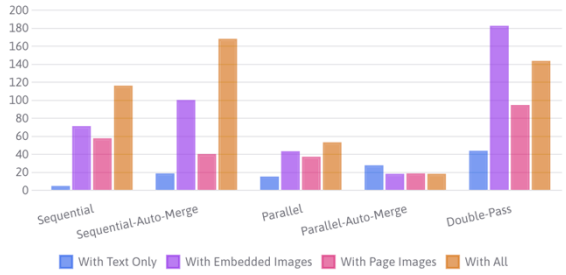


Figure 31: Average duration in seconds (LIDL Brochure)

The token usage appears to be mostly the same across strategies, with the acutal difference lying in the kind of context that is being sent along. The double-pass strategy has the highest token consumption, which is expected, as it performs both the `parallel` and `sequential` strategy as part of it, effectively doubling the tokens used.

Since the document contains lots of small images, including the embedded images results in much higher token usage than only using the page screenshots instead. While tokens used are more than doubled, the number of returned products is only slightly better, indicating that this may be a place where costs can be saved with only a slight reduction in output quality. In fact, wether images are included at all only appears to only have minimal effect on the number of products, sugesting they may even be unnecessary to begin with. The remaining quality difference might even be fixable just by changing the prompt or LLM used.

Looking at the extraction duration a clear pattern emerges. While strategies only containing text are quick regardless of the strategy, both `parallel` strategies result in quick execution times. While the default strategy takes an average of around 40 seconds to complete, enabling auto-merging reduces this number even further to an average of ~20 seconds. As the name suggests, the overall effectiveness can be explained by the chunks being executed in parallel instead of in sequence. The difference between the two is explained by the method used to merge the chunks together, with the latter not having to use another LLM call.

Concluding, the `parallel` strategy with auto-merging enabled appears to be the most effective strategy to extract product data from these kind of supermarket brochures.

5.2 An Example Including Images: Real Estate Exposés

Real Estate Exposés contain data about buildings and any rentable units available within them. They are used for marketing purposes, so that potential tenants can inspect a property before planning an in-person visit. Traditionally being made in Word or Powerpoint, they are often available as PDF files, with estate and space data being spread across multiple pages. Additionally, they contain images of the property at hand. The publicly-available exposé (© BEOS AG) [104] we are evaluating with contains 3 available spaces, which are entire floors in this case. Additionally, the file contains 22 images consisting of photos and floorplans that the model could assign.

Accessing this data without manual work can be useful for real estate agents, as they deal with lots of different kinds of properties. These often belong to different owners, thus a standard layout for these exposés cannot be expected. Data Wizard could be used to quickly import these kind of documents into any marketing software they work with.

The resulting data structure includes a property `name` and `address`, as well as a short `description_text`. An array of `units` is also expected to be returned, each expecting the following properties: `usages`, `label`, `floor` and `rent_per_m2`. Both the property itself and each unit has an `images` and `floorplans` property, which is expected to be filled with `Artifact` IDs.

Figure 32: The cover page of the exposé example [104]

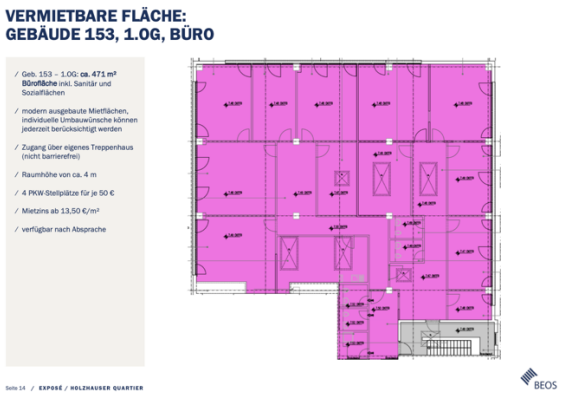


Figure 33: A page detailing an available rental space

Looking at the results, it appears most of the strategies find the same number of rental units in the data with some outliers. These could be explained due to the document itself actually mislabling a unit, which might throw off the LLM during extraction. Data values like `name` and `address` were filled correctly regardless of which strategy was used.

When observing the total length of the generated descriptions, extraction runs where the page screenshots were included tend do be longer than others. This is mainly the case with the two parallel strategies. The two double-pass strategies also seem to result in longer text, especially when including both the embedded images themselves as well as the page screenshots.

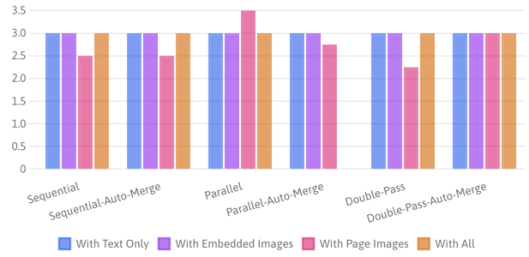


Figure 34: Average number of rental units

Figure 35: Average length of description text in words

The number of images the extractor can assign to either the property or single units is much better when the images themselves are sent along the request. Providing just the marked page screenshots does not appear to result in a lot of images associated. The results become even better when both the images themselves and the screenshots are made available to the model. This makes sense, as the original images let the model observe their contents in more detail, while the page screenshots help with finding relations between data and images.

Duration wise, a similar pattern to the supermarket brochure can be seen, where the `parallel` strategies run the fastest with an average of around 5 seconds, outperformed only by text-only requests which can be even quicker. As expected, the `double-pass` strategy takes the longest.

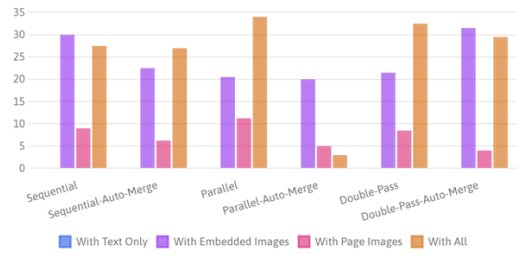


Figure 36: Total number of images associated

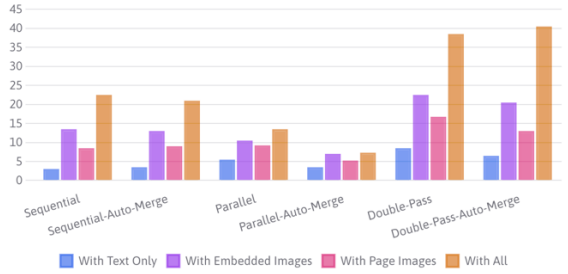
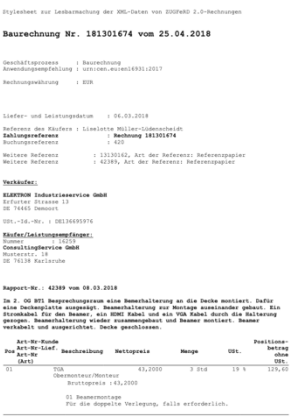


Figure 37: Average execution duration (Exposé)

To conclude, basically all the tested strategies are usable for the exposés. Depending on the user's priorities, different trade-offs can be made. If we equate longer descriptions to more detailed and better text, then it could make sense to use a double-pass strategy to generate better prose. This would lead to longer execution times, which would not be optimal if the task is performing lots of repeated data entry. However, it is clear that including just the original images or both kinds is essential for good image associations.

5.3 An Example Requiring Precision: Invoices

While the first two examples have some leeway in the correctness of the resulting data, an example where precision is an absolute priority is reading data from invoices. In this field, having the correct numbers is essential for the solution to be considered acceptable, as any errors can go undetected and lead to extensive problems down the line. There may even be laws in some jurisdictions that impose penalties for certain data errors, so looking for any faults is the main factor worth looking at in this case.


Stylesheet zur Lesbarmachung der XML-Daten von ZUGFeRD 2.0-Rechnungen

Baurechnung Nr. 181301674 vom 25.04.2018

Geschäftsprozess : Baurechnung
Anwendungsempfehlung : urn:cen.eu:en16931:2017

Rechnungswährung : EUR

Liefer- und Leistungsdatum : 06.03.2018

Referenz des Käufers : Lieselotte Müller-Lüdenscheidt
Zahlungsreferenz : Rechnung 181301674
Buchungsreferenz : 420

Weitere Referenz : 13130162, Art der Referenz: Referenzpapier
Weitere Referenz : 42389, Art der Referenz: Referenzpapier

Verkäufer:

ELEKTRON Industrieservice GmbH
Erfurter Strasse 19
DE 76465 Demnort

USt.-Id.-Nr. : DE136695976

Käufer/Leistungsempfänger:
Nummer : 10259
Consultingservice GmbH
Musterstr. 18
DE 76138 Karlsruhe

Rapport-Nr.: 42389 vom 09.03.2018

Im 2. OG BT1 Besprechungsraum eine Beamerhalterung an die Decke montiert. Dafür eine Deckenplatte ausgesägt. Beamerhalterung zur Montage auseinander gebaut. Ein Stromkabel für den Beamer, ein HDMI Kabel und ein VGA Kabel durch die Halterung gezogen. Beamerhalterung wieder zusammengebaut und Beamer montiert. Beamer verkabelt und ausgerichtet. Decke geschlossen.

Pos	Art-Nr-Kunde Art-Nr-Lief. Art-Nr (Art)	Bezeichnung	Nettopreis	Menge	Ust.	Positionsbetrag ohne USt.
01		VGA Obermonteur/Monteur Bruttopreis : 43,2000	43,2000	3 Std	19 %	129,60

01 Beamermontage
Für die doppelte Verlegung, falls erforderlich.

Figure 38: Example invoice

Invoices typically include information about both the buyer and the seller, including their `address` and `tax_id`. In addition to the final `net` and `gross` totals, they contain the `unit_price` and `quantity` of all the invoiced items. Some items are also taxable, so their `tax_rate` is also provided.

Since 2024, German companies need to provide machine-accessible endpoints to receive invoices in the E-Invoice format. However, companies are not forced by law to actually provide valid E-Invoices yet, which may become a requirement in the future. Until then, lots of non-machine-readable invoices will still need to be processed, possibly leading to some confusion when normal invoices are not accepted by the published endpoints. Using a Data Wizard extractor, functionality to also parse these could be implemented, improving the customer experience.

While comparing the amount of errors observed would normally qualify as a good way to differentiate the strategies, the results of the evaluation runs make this more difficult than expected. Surprisingly, none of the strategies resulted in any numerical data errors. There were some inconsistencies with the labelling of certain items, however these were only slight variations in the strings themselves. For example, some results included new-line characters in the seller names, which is technically also how it is written in the original invoice. Such smaller issues could probably be resolved with some modifications to the prompt.

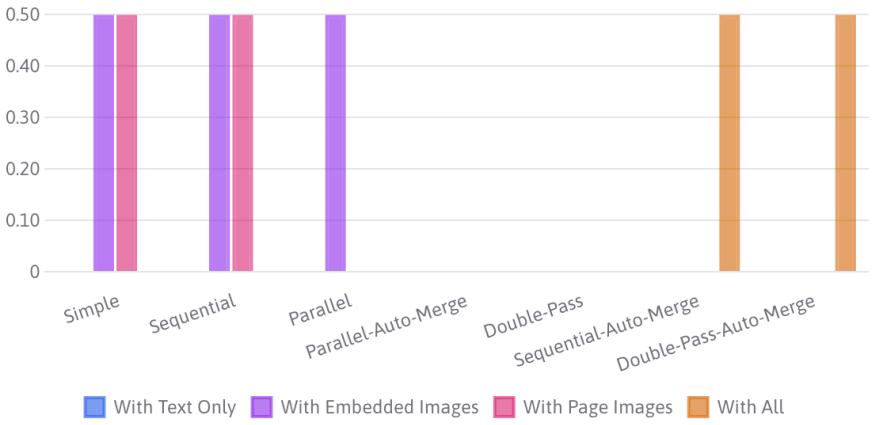


Figure 39: Average number of data inconsistencies to the base truth

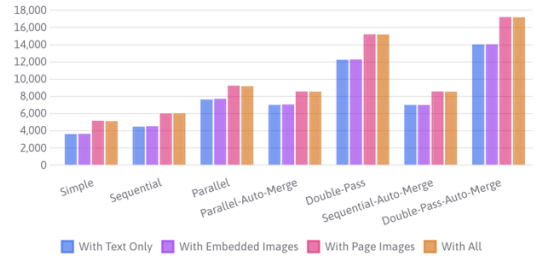


Figure 40: Average token counts (Invoices)

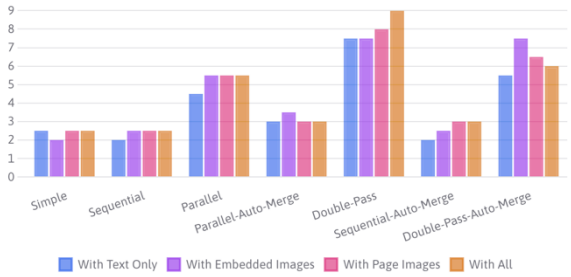


Figure 41: Average duration in seconds (Invoices)

Since the error rate is too low to meaningfully compare, only duration and token count can be considered now. These, however, also show relatively predictable results. The token differences are directly tied to the amount of LLM calls made as part of the strategy, since the invoice contains no images. Thus, including them is unnecessary to begin with. If anything, including images appears to make the results worse, as label inconsistencies only occurred when contents other than text were sent to the LLM, as Figure 39 demonstrates. The same applies to the execution duration, which is almost the same for all strategies only making a single LLM call. Only the `parallel` and `double-pass` strategies actually take longer.

Since invoices are normally relatively short, the `simple` strategy is worth considering for this use-case. In fact, it actually appears to perform the best, both in execution time and token count. This makes sense, as the amount of data invoices contain is relatively small, which makes it a prime candidate for fitting the entire document into the context window. With this perspective and the observed results, the other strategies could even be considered wasteful.

To be absolutely certain that the data is correct, extracting from invoices could be a great example where a custom strategy makes a lot of sense. By including some logical data validation and error detection before returning the final data, correctness can be ensured by summing up the line items and verifying the totals. Since the validation possibilities are almost endless, it is something that would be difficult to implement in a generic and reusable way, which is why Data Wizard cannot really provide it out of the box. But Data Wizard's developer friendly extendability makes this easy to accomplish on a case-by-case basis.

5.4 Interpretation of Evaluation Results

The evaluation results show that Data Wizard's extraction strategies perform well across diverse data types with unique extraction goals. For datasets characterized by multiple independent entities, such as supermarket brochures, the `parallel` strategy with auto-merging provided superior performance in entity retrieval and extraction speed. In contrast, the `sequential` strategy's limitations in data retention across LLM calls became apparent. Data Wizard's image association abilities make it a great fit for tasks that rely on visual-heavy data, especially prevalent in marketing sectors. For tasks prioritizing precision, the simple strategy proved most efficient and accurate, suggesting that for structured, concise documents, more complex strategies offer only diminishing returns.

Ultimately, the evaluation results show the importance of strategy selection being tailored to the unique characteristics of the input data and desired extraction outcome. The built-in strategies cover a wide range of possibilities already, with the escape hatch of a custom implementation always being ready for the taking. The evaluation capabilities being built right into the software itself allows future improvements to be measured against a baseline of performance, and even encourages the data-based development of a user's own extraction strategies. This transforms Data Wizard from merely being a tool that accomplishes the extraction itself, to being a platform that actively supports the development of new extraction pipelines.

6 Current Limitations and Improvements

While the scope of this thesis focuses on the core task of enabling reusable data extraction logic using Data Wizard and doing so without a specific LLM provider in mind, there are still some limitations and potential improvements that could be made to the system that either go beyond the scope of this thesis or require more advancements in the field of LLMs.

6.1 Output Token Limit

One of the fundamental limitations Data Wizard has at the moment is the output token limit of the LLMs. While input token limits continue to improve into the millions of tokens [105], the output token limit of most models is still around 4.000-8.000 tokens. If the data that needs to be extracted exceeds this limit, the extraction process will fail and the data will be incomplete. When using the parallel strategy without auto-merging, this problem is very apparent, as the full data size is only known after all parts have been extracted. The extration will then fail at the very last step, which can result in a terrible user experience.

This problem shows itself in weird ways, depending on the LLM provider. For Gemini models, the tool calling internals are exposed, showing a markdown code block with some abruptly cut-off python code. It can be assumed that this is how Gemini models have implemented tool calling, and the token limit simply stops the output immediately, leading to unfinished code.

```
```tool_code
print(default_api.extract(products=[
  {"image_artifact_id": "artifact:2956862347/images_marked/image1.jpeg", "name": "Festbier",
"original_price": 0.89, "discounted_price": None},
  {"image_artifact_id": "artifact:2956862347/images_marked/image2.jpeg", "name": "Rindsfilet am
Stück", "original_price": 4.99, "discounted_price": None},
  {"image_artifact_id": "artifact:2956862347/images_marked/image5.jpeg", "name": "Douro DOC Reserva
2019", "original_price": 5.99, "discounted_price": 3.99},
```

Code Snippet 23: Raw Gemini output when the output token limit is exceeded during a tool call

With time, the output token limits will most likely increase as well. The limit will then become less of a problem and will allow even bigger datasets to be processed with more quality. For now, the auto-merging strategies are a good workaround for this issue.

6.2 Chain-of-thought Reasoning

Chain-of-thought reasoning is a concept that has allowed LLMs to pass even more advanced benchmarks. Models like OpenAI's o1 or Deepseek's R1 spend some time reasoning about a problem in output that is hidden to the user before giving their final answer [97] [22]. This has resulted in a significant increase in output quality for tasks where logical thinking is required [106].

While Data Wizard does not currently prompt the models to reason before giving an answer, based on the performance improvements observed in math and logic benchmarks, using a reasoning step before letting the model output extraction data could lead to better results. Data Wizard could also just use models with built-in reasoning like o1 or R1 through the unified LLM interface, but at the moment, none of these models support tool calling, thus requiring other ways to extract the data.
```

6.3 Add Dynamic Data-Loading With Tool Calls and Embeddings

There are lots of existing tools for "*chatting*" with a PDF. These tools face similar context problems as Data Wizard, as most documents will not fit into a single LLM call. To work around this, tools like these use a process called "*Retrieval Augmented Generation*" (RAG) to load parts of the document dynamically as they are needed [107]. This works by generating vector embeddings for document chunks and then including relevant chunks in LLM calls with user prompts. What data to include is based on vector similarity to the user's query, for which embeddings must also be created.

LLM Magic already has basic support for generating embeddings as most LLM providers offer this through the same API as the LLMs themselves. This could be used in the future to implement a dynamic data-loading strategy that loads only the parts of the document that are relevant to the expected data schema. This would potentially allow extracting data from massive documents without having to process the whole thing. However, it is an open question how well vector similarity works when comparing embeddings of the data schema with content-based embeddings and some experimentation would be needed to find out. Additionally, the potential to miss out on details is present, as we are now dealing with a subset of the input dataset, instead of processing the entire document.

6.4 Improve or Replace the File Pre-Processing

The current logic for pre-processing files is relatively straightforward in that it only extracts the raw PDF text and the images from the document. If the data quality of the original document is poor, for example, because older, less-reliable OCR software was used or there was no textual data to begin with, the extraction results will also suffer.

Since this problem is not new and is required for all sorts of LLM integrations, there are some tools available that can help with this. Just recently in March 2025, Mistral AI introduced their new `mistral-ocr` model, which is specifically designed to extract text from images and PDFs [108]. Available via API, this model can be used to take in documents and give out structured contents in a surprisingly similar way to how Data Wizard works: split into text and media blocks. While this certainly shows that the approach taken by Data Wizard was the correct one, it might be an overall improvement to replace the internal pre-processing with a more capable tool like this.

Since Mistral OCR itself uses LLMs internally to process the files, it has additional features that the internal Data Wizard pre-processing cannot recreate, namely the ability to also extract images from documents, even if the document itself is only provided as a scan or photo.

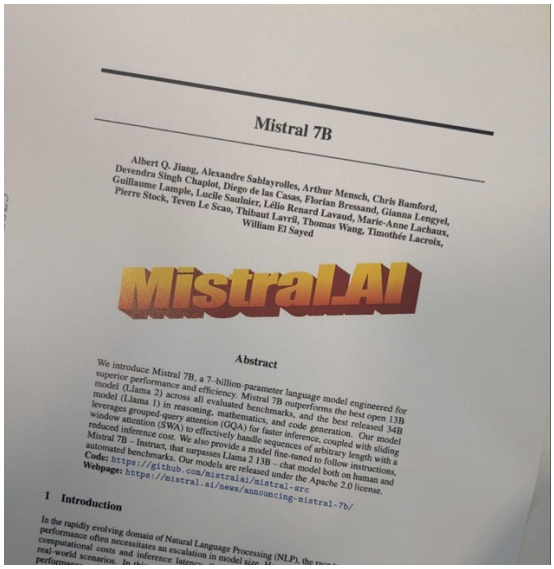


Figure 42: A photo of a document used as input, which contains text as well as the Mistral AI logo (© 2025 Mistral [102])

Figure 43: The generated output, including a cropped version of the logo layouted into the text contents (© 2025 Mistral [108])

What's interesting is how similar the output of Mistral OCR is to the Data Wizard `contents.json` format. Instead of using a flat array with page numbers in each slice, Mistral OCR uses a slightly more nested structure with contents and images being properties of a page object. The underlying concept of how data is represented appears to be the same. This would allow for an easy implementation of Mistral OCR, as the output given by the API can be converted into the Data Wizard format without any data loss. This would make Mistral OCR a prime candidate for an addition or even a replacement of the internal pre-processing logic, with the only downside being the additional costs of using the API and the reliance on an external service. However, since Data Wizard is already reliant on LLM APIs to begin with, this might be an acceptable trade-off, especially if the quality of the extracted data can be significantly improved in most cases.

```
[
    {
        "page": 1,
        "type": "text",
        "text": "VW Golf & Jetta..."
    },
    {
        "page": 2,
        "type": "text",
        "text": "Ferrari F40..."
    },
    {
        "page": 2,
        "type": "image",
        "mimetype": "image/png",
        "path": "images/image1.png",
        "x": 123.80999755859375,
        "y": 62.32989501953125,
        "width": 100.7958984375,
        "height": 101.035888671875
    }
]
```

Code Snippet 24: Data Wizard output

```
{
    "pages": [
      {
        "index": 1,
        "markdown": "# LEVERAGING UNLABELED..."
        "images": [
          {
            "id": "img-0.jpeg",
            "top_left_x": 292,
            "top_left_y": 217,
            "bottom_right_x": 1405,
            "bottom_right_y": 649,
            "image_base64": "..."
          }
        ],
        "dimensions": {
          "dpi": 200,
          "height": 2200,
          "width": 1700
        }
      }
    ]
}
```

Code Snippet 25: Mistral OCR output

6.5 Deep LLM Integration Into the Laravel and Filament Ecosystem

Laravel is a very powerful and flexible tool that can be used to build a wide range of applications. But it is also quite opinionated when it comes to how the data model is defined. Eloquent models are the standard way to interact with the database in Laravel and are used by most Laravel applications. Tools like Filament or API Platform make great use of this by building atop this assumption and are thus able to provide a lot of functionality out of the box and without much configuration.

This gives Laravel an edge over other popular tech stacks, especially those in the JavaScript ecosystem, where there is a lot of fragmentation and different ways to do things. LLM Magic has the opportunity to do the same and more directly integrate with the Laravel ecosystem, allowing developers to focus even more on their business logic, instead of having to spend a lot of time on moving data around.

6.5.1 Controlled Access to the Database via Eloquent Models

For obvious reasons, giving an LLM production access to a database via raw SQL queries can lead to all kinds of security and data integrity issues. However, Eloquent was made to deal with these issues and provides a safe and secure way to interact with stored data and even with data relations [109]. Combined with Laravel validation rules [110], Eloquent models can ensure that only valid data is stored in the database.

LLM Magic could provide pre-built tools to both query, create and modify data in the database through Eloquent models. All a developer would need to do is to add the tool into the LLM call, pass the model class and the tool would take care of the rest. This can have certain implications based on the application, but it could still be a powerful tool giving LLMs read and/or write access to the database.

```
Magic::chat()
    ->tools([
        EloquentTools::list(Product::class),
        EloquentTools::create(Product::class, rules: [
            'name' => 'required|string',
            'price' => 'required|numeric',
            'discounted_price' => 'nullable|numeric'
        ]),
        EloquentTools::update(Product::class, rules: [
            'name' => 'required|string',
            'price' => 'required|numeric',
            'discounted_price' => 'nullable|numeric'
        ]),
    ])
```

Code Snippet 26: Example code showing how a direct Eloquent integration could look

6.5.2 Integrated Chatbot UI with Filament Forms and Tables

Most LLM interactions take place in a chatbot-style interface, where the user can ask questions and the model responds with answers. This is how ChatGPT debuted and is still the most common way to interact with LLMs in a user-facing way. While there are already some libraries that provide UI components for chatbots, these mainly focus on the frontend part and are not functional on their own.

Using the backend-driven nature of Livewire, LLM Magic could go a step further and provide reusable components that already have actual chatbot functionality already built-in. This would allow developers to build chatbot interfaces using just a few lines of code, as the components would already know how to interact with the LLMs and how to handle the responses.

The advantage of this is even more apparent when considering tool calling, where a use case could be to ask a user to verify or fix some data that is meant to be stored in the database or sent somewhere else. In this case, the deep integration with Eloquent could be taken a step further, by also supporting Filament resources directly. Filament resources directly hook into Eloquent logic themselves and then define table and form schemas on top to be able to list and edit data in a declarative way. A tool that has access to a Filament resource could then automatically embed these forms and tables into the chatbot UI. A user could then see the data that an LLM sees in a good-looking, filterable table or make changes to LLM-generated data in a validated form before confirming the action and saving it to the database. Since Filament forms already include validation logic, the developer can skip writing validation rules for the form and just use the ones that are already defined in the resource [111].

```php
class MyCustomChatbot extends ChatbotComponent
{
    // Allow the LLM to list, view, create and edit products
    protected function getTools(): array
    {
        return [
            // Limit the scope that the LLM has access to for sensitive data
            FilamentTools::list(
                ProductResource::class,
                modifyQuery: fn($query) => $query->where('secret', 'abc')
            ),
            FilamentTools::view(ProductResource::class),
            FilamentTools::create(ProductResource::class),
            FilamentTools::edit(ProductResource::class, requireConfirmation: true),
        ];
    }
}
```

Code Snippet 27: Example code showing how LLM Magic's future chatbot UI can be configured

In fact, some of this is already experimentarily implemented and the code example above is actually how it works. But it is currently part of an experimental `llm-magic-ui` package and is not part of the scope of this thesis [112]. But it already works well enough to prove the concept and is planned to be included in a future release of LLM Magic.

6.6 Test Coverage

Currently, both the Data Wizard and LLM Magic code have few if any automatic unit or feature tests. Due to the complexity of the project, there was no time to implement these tests during development. However for long-term maintainability and stability, it is important to have a comprehensive test suite that covers the functionality of both the application and the library.

Especially the LLM interface needs to be thoroughly tested, as it is the most important aspect to work correctly. The LLM API response decoders are crucial for receiving the correct output from the LLMs and are also the most brittle part of the application. Since they deal with external APIs, they are subject to change and need to be tested regularly to ensure they still work correctly. Fundamentally, testing this requires two testing modes to be implemented. One for actually testing if the API integrations still work and another one for testing the more complex tool calling functionality. The latter should not

result in actual LLM calls being made, as this would incur costs. Instead, this functionality should be tested with mocked responses from the LLM providers and in complete isolation from the APIs.

For this to work, some response collection functionality needs to be added so that up-to-date responses can be collected automatically and then used for testing. PHP and Laravel provide fantastic testing tools like PHPUnit [113] and Pest [114] which make writing the tests themselves very easy. Pest even provides extra features like great mocking support [115] or snapshot testing [116] which may prove useful for testing the LLM interface. In general, the Laravel service container and dependency injection make it easy to mock dependencies and test classes in isolation. Both the app and the library already use this pattern, so adding mocked implementations where needed should be straightforward.

6.7 Improve Validation Error Messages for Humans

Currently, the error messages that are shown to the user are directly taken from the Opis JSON Schema library and are not very user-friendly [52]. Depending on the type of error, the message can be quite cryptic and not very helpful in understanding what went wrong. For example, if a property is required and to be a number but is currently empty, the error message will say that the type of the property is supposed to be a number instead of null. Stating that the property is required would be more helpful.

Luckily, Opis JSON Schema allows for custom error messages to be defined for each rule, so it would be possible to improve the error messages by defining custom messages for each rule. This can easily be added in a future update to the software and does not require any major changes to the existing codebase.

7 Conclusion

In conclusion, the results of this project show that Large Language Models are a powerful tool for data extraction, capable of transforming unstructured data into structured formats that can be utilized across various applications. Data Wizard, the tool developed in this thesis, exemplifies this potential by offering a flexible and extensible solution for extracting data from documents. It can answer the stated research question, wether the implementation of such a tool is technically feasable, with a positive response. Its ease of integration, user-friendly interface, and robust API make it a valuable solution for companies seeking to automate and implement a data extraction processes. The tool's open-source nature and self-hosting capabilities provide users with complete control over the data extraction process, ensuring data privacy and security.

Furthermore, Data Wizard's architecture and implementation directly address the need for flexibility. The tool's ability to adapt to multiple document types, extraction strategies, and LLM providers, as evidenced by its performance across diverse use cases and document types, confirms its broad applicability. The inclusion of custom strategies further empowers developers to tailor the extraction process to specific domain requirements, ensuring its continued viability across a wide spectrum of data extraction tasks. Additionally, by using a well-abstracted subsystem for LLM interaction, it can easily add support for newly released LLMs from various providers without requiring changes to the integrations that use Data Wizard or the extraction logic itself. This abstraction can even be reused in other software projects, adding another layer of usefulness to the project. This inherent adaptability directly answers the question of whether such a tool can be sufficiently flexible for real-world applicability.

Finally, the design and deployment of Data Wizard prioritize usability and simplicity. Its straightforward integration process, embeddable user interface, and readily accessible API are all geared towards minimizing configuration and facilitating seamless adoption within existing software ecosystems. The choice of Laravel, Livewire, and Filament as the technology stack has been instrumental in achieving this simplicity, providing mature and readily available tools that significantly reduced development complexity and enabled rapid UI creation. Data Wizard's ease of deployment and integration, therefore, affirms its potential to be a genuinely useful and accessible alternative to bespoke data extraction solutions, directly answering the question of practical usability.

Looking ahead, further improvements and deeper integrations with the PHP, Laravel and Filament ecosystems hold the promise of making Data Wizard even more powerful. The potential for turning Data Wizard into a business through managed hosting and support is also exciting, and its open-source nature may even be able to disrupt the existing market by being available for free.

In summary, Data Wizard represents a significant step forward in leveraging LLMs for data extraction, offering a robust, flexible, and user-friendly solution. The journey of developing this tool has been both challenging and rewarding, and I am eager to see where future advancements will take it. It has become somewhat of a passion project, and the possibilities for further development are endless. As we continue to push the boundaries of what's possible with Large Language Models, I am thrilled to now be a part of this innovative journey, and I cannot wait to see if Data Wizard can become a solution that’s more widely adopted.

8 References

[1] Statista, *Spending on digital transformation technologies and services worldwide from 2017 to 2027sa.* [Online]. Available: https://www.statista.com/statistics/870924/worldwide-digital-transformation-market-size/ (accessed: Mar. 10 2025).

[2] Crem Solutions GmbH & Co. KG, *iX-Haus: Property Management Software.* [Online]. Available: https://www.crem-solutions.de/softwareloesungen/ix-haus/ (accessed: Mar. 10 2025).

[3] I. 6Sense Insights, *DATEV Market Share: Market Share, Competitor Insights in Tax And Compliance Software.* [Online]. Available: https://6sense.com/tech/tax-and-compliance-software/datev-market-share (accessed: Mar. 10 2025).

[4] DATEV eG, *Ablauf einer Cloud Integration: DATEV Developer Portal.* [Online]. Available: https://developer.datev.de/de/guides/cloud-integration-workflow (accessed: Mar. 10 2025).

[5] DATEV eG, *DATEV API Products: DATEV Developer Portal.* [Online]. Available: https://developer.datev.de/de/products (accessed: Mar. 10 2025).

[6] I. 6Sense Insights, *Microsoft Office Marketshare: Market Share, Competitor Insights in Office Suites.* [Online]. Available: https://6sense.com/tech/office-suites/microsoft-office-market-share (accessed: Mar. 10 2025).

[7] I. 6Sense Insights, *Microsoft Office 365 Market Share: Market Share, Competitor Insights in Office Suites.* [Online]. Available: https://6sense.com/tech/productivity/microsoft-office-365-market-share (accessed: Mar. 10 2025).

[8] I. 6Sense Insights, *Google Workspace Market Share: Market Share, Competitor Insights in Office Suites.* [Online]. Available: https://6sense.com/tech/office-suites/google-workspace-market-share (accessed: Mar. 10 2025).

[9] Microsoft Inc., *Microsoft Power Automate: Process Automation. Microsoft Power Platform.* [Online]. Available: https://www.microsoft.com/en-us/power-platform/products/power-automate (accessed: Mar. 10 2025).

[10] Microsoft Inc., *Microsoft Power BI: Data Visualization. Microsoft Power Platform.* [Online]. Available: https://www.microsoft.com/en-us/power-platform/products/power-bi (accessed: Mar. 10 2025).

[11] Freshworks Inc, *SURVEY: Nine in 10 Employees are Frustrated by their Workplace Technology: Freshworks survey of nearly 9,000 workers around the world reveals software crisis despite pandemic-driven tech spend surge*, 2022. [Online]. Available: https://www.freshworks.com/press-releases/survey-nine-in-10-employees-are-frustrated-by-their-workplace-technology

[12] M. Marfatia, "The (Not So) Hidden Costs of Delaying Application Modernization," *Solutions Review*, 12 May., 2023. https://solutionsreview.com/business-process-management/the-not-so-hidden-costs-of-delaying-application-modernization (accessed: Mar. 10 2025).

[13] R. Smith, "An Overview of the Tesseract OCR Engine," in *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, 2007, pp. 629–633. Accessed: Mar. 10 2025. [Online]. Available: https://ieeexplore.ieee.org/document/4376991

[14] R. Smith, S. Weil, Z. Podobny, and Open Source Contributors, *Tesseract OCR 4.* [Online]. Available: https://github.com/tesseract-ocr/tesseract/releases/tag/4.0.0 (accessed: Mar. 10 2025).

[15] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, "SystemT: a system for declarative information extraction," *SIGMOD Rec.*, vol. 37, no. 4, pp. 7–13, 2009, doi: 10.1145/1519103.1519105.

[16] R. Smith, S. Weil, and Open Source Contributors, *Overview of the new neural network system in Tesseract 4.00: tessdoc.* [Online]. Available: https://tesseract-ocr.github.io/tessdoc/tess4/NeuralNetsInTesseract4.00.html (accessed: Mar. 10 2025).

[17] I. Amazon Web Services, *Amazon Textract: Amazon Textract is a machine learning (ML) service that automatically extracts text, handwriting, layout elements, and data from scanned documents.* [Online]. Available: https://aws.amazon.com/textract/ (accessed: Mar. 10 2025).

[18] T. Wu, "Reducing the Cost of Test Data Labelling for Deep-Learning Systems: An Empirical Study," Université d'Ottawa / University of Ottawa, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://ruor.uottawa.ca/items/155921fa-fd72-407a-a67d-ce182c5e0e75

[19] Y. Lin *et al.*, "Towards Accurate and Efficient Document Analytics with Large Language Models," 2024. Accessed: Mar. 10 2025. [Online]. Available: https://arxiv.org/abs/2405.04674

[20] X. Jin and Y. Wang, "Understand Legal Documents with Contextualized Large Language Models," 2023. Accessed: Mar. 10 2025. [Online]. Available: https://arxiv.org/abs/2303.12135

[21] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," May. 2020. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2005.14165v4

[22] DeepSeek-AI *et al.*, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," 2025. Accessed: Mar. 10 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[23] K. Buchholz, "DeepSeek-R1 Upsets AI Market With Low Prices," 28 Jan., 2025. https://www.statista.com/chart/33839/prices-for-processing-one-million-input-output-tokens-on-different-ai-models/ (accessed: Mar. 10 2025).

[24] I. Vercel, J. Palmer, S. Ding, and M. Leiter, *Introducing the Vercel AI SDK: An interoperable, streaming-enabled, edge-ready software development kit for AI apps built with React and Svelte.*, 2023. Accessed: Mar. 10 2025. [Online]. Available: https://vercel.com/blog/introducing-the-vercel-ai-sdk

[25] I. LangChain and Open Source Contributors, *LangChain v0.3: LangChain is a composable framework to build with LLMs.* [Online]. Available: https://langchain.com/ (accessed: Mar. 10 2025).

[26] I. Rudnytskyi, *openai 0.4.1: R Wrapper for OpenAI API.* [Online]. Available: https://cran.r-project.org/web/packages/openai/index.html (accessed: Mar. 10 2025).

[27] N. Mündler, J. He, S. Jenko, and M. Vechev, "Self-contradictory Hallucinations of Large Language Models: Evaluation, Detection and Mitigation," 2023. [Online]. Available: https://arxiv.org/abs/2305.15852

[28] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards Mitigating LLM Hallucination via Self Reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, Singapore, pp. 1827–1843. Accessed: Mar. 10 2025. [Online]. Available: https://aclanthology.org/2023.findings-emnlp.123/

[29] JSON Schema Contributors, *JSON Schema 2020-12: The JSON Schema specification.* [Online]. Available: https://json-schema.org/specification (accessed: Mar. 10 2025).

[30] Anthropic Inc., *Tool use with Claude – Anthropic Documentation: Tool use (function calling).* [Online]. Available: https://docs.anthropic.com/en/docs/build-with-claude/tool-use/overview (accessed: Mar. 10 2025).

[31] OpenAI Inc., *Function calling - OpenAI API: OpenAI Platform Documentation.* [Online]. Available: https://platform.openai.com/docs/guides/function-calling (accessed: Mar. 10 2025).

[32] JSON Schema Contributors, *Get Started with JSON Schema: JSON Schema Documentation* (accessed: Mar. 10 2025).

[33] Docker Inc., *What is a Container? | Docker: Use containers to Build, Share and Run your applications.* [Online]. Available: https://www.docker.com/resources/what-container (accessed: Mar. 10 2025).

[34] OpenAI Inc., *Text generation | OpenAI API Documentation: Learn how to generate text from a prompt.* [Online]. Available: https://platform.openai.com/docs/guides/text-generation

[35] Google Inc., *Gemini OpenAI compatibility: Gemini API. Google AI for Developers.* [Online]. Available: https://ai.google.dev/gemini-api/docs/openai (accessed: Mar. 10 2025).

[36] Ollama Contributors, *Ollama OpenAI Compatibility: Ollama GitHub Repository.* [Online]. Available: https://github.com/ollama/ollama/blob/main/docs/openai.md (accessed: Mar. 10 2025).

[37] OpenAI Inc. and Cookbook Contributors, *OpenAI Cookbook.* [Online]. Available: https://cookbook.openai.com/ (accessed: Mar. 10 2025).

[38] Anthropic Inc., *Anthropic Cookbook.* [Online]. Available: https://github.com/anthropics/anthropic-cookbook (accessed: Mar. 10 2025).

[39] Anthropic Inc., *Use XML tags to structure your prompts: Anthropic Documentation.* [Online]. Available: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/use-xml-tags (accessed: Mar. 10 2025).

[40] S. Campbell, *Better LLM Prompts Using XML.* [Online]. Available: https://www.aecyberpro.com/blog/general/2024-10-20-Better-LLM-Prompts-Using-XML/ (accessed: Mar. 10 2025).

[41] I. Amazon Web Services, *Sharing objects with presigned URLs: Amazon Simple Storage Service User Guide.* [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html (accessed: Mar. 10 2025).

[42] Laravel Inc., *Laravel 11 Signed URLs: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/urls#signed-urls (accessed: Mar. 10 2025).

[43] M. Jones, J. Bradley, and N. Sakimura, "RFC 7519: JSON Web Token (JWT)," 2015. Accessed: Mar. 10 2025. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519

[44] OnlinePHP.io, *PHP 4.1 - 8.3 Benchmarks.* [Online]. Available: https://onlinephp.io/benchmarks (accessed: Mar. 10 2025).

[45] T. Otwell, Laravel Inc., and Open Source Contributors, *Laravel 11: Laravel provides a complete ecosystem for web artisans. Our open source PHP framework, products, packages, and starter kits offer everything you need to build, deploy, and monitor web applications.* [Online]. Available: https://laravel.com/docs/11.x (accessed: Mar. 10 2025).

[46] T. Otwell, *Accel Invests 57M into Laravel*, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://blog.laravel.com/accel-invests-57m-into-laravel

[47] J. Reinink and Open Source Contributors, *Inertia.js: Build single-page apps, without building an API.* [Online]. Available: https://inertiajs.com/ (accessed: Mar. 10 2025).

[48] C. Porzio and Laravel Inc., *Laravel Livewire v3: Powerful, dynamic, front-end UIs without leaving PHP.* [Online]. Available: https://livewire.laravel.com/ (accessed: Mar. 10 2025).

[49] D. R. Hipp and Open Source Contributors, *SQLite.* [Online]. Available: https://www.sqlite.org/

[50] The PostgreSQL Global Development Group, *PostgreSQL 17: The World's Most Advanced Open Source Relational Database.* [Online]. Available: https://www.postgresql.org/ (accessed: Mar. 10 2025).

[51] A. Kane and Open Source Contributors, *pgvector v0.8.0: Open-source vector similarity search for Postgres.* [Online]. Available: https://github.com/pgvector/pgvector (accessed: Mar. 10 2025).

[52] Zindex Software, *Opis JSON Schema v2.4.1: JSON Schema validator for PHP.* [Online]. Available: https://github.com/opis/json-schema

[53] D. Bonsch and Open Source Contributors, *PhpSpreadsheet v4.1.0: A pure PHP library for reading and writing spreadsheet files.* [Online]. Available: https://github.com/PHPOffice/PhpSpreadsheet (accessed: Mar. 10 2025).

[54] K. Dunglas and Open Source Contributors, *API Platform for Laravel Documentation: API Platform for Laravel Projects.* [Online]. Available: https://api-platform.com/docs/laravel/ (accessed: Mar. 10 2025).

[55] F. Van der Herten and Open Source Contributors, *Laravel Media Library v11.12: Associate files with Eloquent models.* [Online]. Available: https://github.com/spatie/laravel-medialibrary (accessed: Mar. 10 2025).

[56] OpenAI Inc., *Vision: Learn how to use vision capabilities to understand images. OpenAI Platform Documentation.* [Online]. Available: https://platform.openai.com/docs/guides/vision (accessed: Mar. 10 2025).

[57] Google Inc., *Gemini Safety Settings: Gemini API. Google AI for Developers.* [Online]. Available: https://ai.google.dev/gemini-api/docs/safety-settings (accessed: Mar. 10 2025).

[58] I. DeepSeek, *Your First API Call | DeepSeek API Docs.* [Online]. Available: https://api-docs.deepseek.com/ (accessed: Mar. 10 2025).

[59] Ollama Contributors, *OpenAI compatibility: Ollama Blog*, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://ollama.com/blog/openai-compatibility

[60] I. OpenRouter, *Open Router: A unified interface for LLMs.* [Online]. Available: https://openrouter.ai/ (accessed: Mar. 10 2025).

[61] N. Maduro, S. Gehri, and Open Source Contributors, *OpenAI PHP Client v10.03: OpenAI PHP is a supercharged community-maintained PHP API client that allows you to interact with OpenAI API.* [Online]. Available: https://github.com/openai-php/client/releases/tag/v0.10.3 (accessed: Mar. 10 2025).

[62] T. Miller, *Prism: A unified interface for working with LLMs.* [Online]. Available: https://prism.echolabs.dev/ (accessed: Mar. 10 2025).

[63] G. Hunt, *Partial JSON Parser 1.1.0.* [Online]. Available: https://github.com/greghunt/partial-json (accessed: Mar. 10 2025).

[64] OpenAI Inc., *Structured Outputs: Ensure responses adhere to a JSON schema.* [Online]. Available: https://platform.openai.com/docs/guides/structured-outputs (accessed: Mar. 10 2025).

[65] The PHP Documentation Group, *PHP Reflection: A complete reflection API that adds the ability to introspect classes, interfaces, functions, methods and extensions.* [Online]. Available: https://www.php.net/manual/en/book.reflection.php (accessed: Mar. 10 2025).

[66] phpDocumentor and Open Source Contributors, *What is a DocBlock?: phpDocumentor Documentation.* [Online]. Available: https://docs.phpdoc.org/guide/getting-started/what-is-a-docblock.html (accessed: Mar. 10 2025).

[67] Artifex Software Inc, *PyMuPDF v1.25: PyMuPDF is a high performance Python library for data extraction, analysis, conversion & manipulation of PDF (and other) documents.* [Online]. Available: https://github.com/pymupdf/PyMuPDF/releases/tag/1.25.0

[68] E. Belval, *PDF2Image 1.17: A python (3.7+) module that wraps pdftoppm and pdftocairo to convert PDF to a PIL Image object.* [Online]. Available: https://pypi.org/project/pdf2image/ (accessed: Mar. 10 2025).

[69] Artifex Software Inc, *MuPDF 1.25: The ultimate library for managing PDF documents.* [Online]. Available: https://mupdf.com/ (accessed: Mar. 10 2025).

[70] R. Munroe, *xkcd: Python Environment.* [Online]. Available: https://xkcd.com/1987/ (accessed: Mar. 10 2025).

[71] C. Marsh, *uv 0.6: Python packaging in Rust*, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://astral.sh/blog/uv

[72] C. Marsh and Astral Software Inc., *uv 0.6: An extremely fast Python package and project manager, written in Rust.* [Online]. Available: https://github.com/astral-sh/uv (accessed: Mar. 10 2025).

[73] M. Deeming, I. M, and Blaspsoft, *doxswap: A Laravel package for seamless document conversion using LibreOffice.* [Online]. Available: https://github.com/Blaspsoft/doxswap (accessed: Mar. 10 2025).

[74] C. Fahlgren, *Hugging Face Hub Stats: Cumulative Hub Growth; Models, Datasets, Spaces created per month.* [Online]. Available: https://huggingface.co/spaces/cfahlgren1/hub-stats (accessed: Mar. 10 2025).

[75] H. Liu *et al.*, "Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning," in *Advances in Neural Information Processing Systems*, 2022, pp. 1950–1965. Accessed: Mar. 10 2025. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/0cde695b83bd186c1fd456302888454c-Paper-Conference.pdf

[76] M. J. J. Bucher and M. Martini, "Fine-Tuned 'Small' LLMs (Still) Significantly Outperform Zero-Shot Generative AI Models in Text Classification," Jun. 2024. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2406.08660v2

[77] A. Grattafiori *et al.*, "The Llama 3 Herd of Models," Jul. 2024. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2407.21783v3

[78] A. Zou *et al.*, "DOCBENCH: A Benchmark for Evaluating LLM-based Document Reading Systems," Jul. 2024. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2407.10701v1

[79] C. White *et al.*, "LiveBench: A Challenging, Contamination-Free LLM Benchmark," Jun. 2024. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2406.19314v1

[80] C. White, S. Dooley, M. Roberts, and A. Pal, *LiveBench Leaderboard.* [Online]. Available: https://livebench.ai/

[81] OpenAI Inc. and S. Altman, *Planning for AGI and beyond*, 2023. Accessed: Mar. 10 2025. [Online]. Available: https://openai.com/index/planning-for-agi-and-beyond/

[82] A. Vaswani *et al.*, "Attention Is All You Need," Dec. 2017. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/1706.03762v7

[83] OpenAI Inc., *GPT-4 Model Card: OpenAI Platform Documentation.* [Online]. Available: https://platform.openai.com/docs/models/gpt-4 (accessed: Mar. 10 2025).

[84] Anthropic Inc., *Introducing the next generation of Claude*, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://www.anthropic.com/news/claude-3-family

[85] Google Inc., *Gemini models: Gemini API. Google AI for Developers.* [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini

[86] *OpenAI O1 Model*.

[87] *OpenAI O1 Mini Model*.

[88] OpenAI Inc., *GPT-4-Turbo Model Card: OpenAI Platform Documentation.* [Online]. Available: https://platform.openai.com/docs/models/gpt-4-turbo (accessed: Mar. 10 2025).

[89] OpenAI Inc., *GPT-4o Model Card: OpenAI Platform Documentation.* [Online]. Available: https://platform.openai.com/docs/models/gpt-4o (accessed: Mar. 10 2025).

[90] PromptHub, *Llama 3.2 90B Model Card: Multimodal model, ideal for visual intelligence in image analysis, document processing, multimodal chatbots, and autonomous systems.* [Online]. Available: https://www.prompthub.us/models/llama-3-2-90b

[91] ContextAI, *Mistral Large 2 Model Card.* [Online]. Available: https://context.ai/model/mistral-large-2 (accessed: Mar. 10 2025).

[92] Mistral AI, *Models Overview: Mistral AI Large Language Models.* [Online]. Available: https://docs.mistral.ai/getting-started/models/models_overview/ (accessed: Mar. 10 2025).

[93] J. Kaplan *et al.*, "Scaling Laws for Neural Language Models," Jan. 2020. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2001.08361v1

[94] Anthropic Inc., *Anthropic API Model Pricing.* [Online]. Available: https://www.anthropic.com/pricing#anthropic-api (accessed: Mar. 10 2025).

[95] Google Inc., *Gemini Developer API Pricing: Gemini API. Google AI for Developers.* [Online]. Available: https://ai.google.dev/gemini-api/docs/pricing (accessed: Mar. 10 2025).

[96] X. Shen, Z. Chen, M. Backes, Y. Shen, and Y. Zhang, ""Do Anything Now": Characterizing and Evaluating In-The-Wild Jailbreak Prompts on Large Language Models," 2023. Accessed: Mar. 10 2025. [Online]. Available: https://arxiv.org/abs/2308.03825

[97] OpenAI Inc., A. El-Kishky, D. Selsam, and F. Song, *Learning to reason with LLMs: We are introducing OpenAI o1, a new large language model trained with reinforcement learning to perform complex reasoning. o1 thinks before it answers—it can produce a long internal chain of thought before responding to the user.*, 2024. Accessed: Mar. 10 2025. [Online]. Available: https://openai.com/index/learning-to-reason-with-llms/

[98] Laravel Inc. and Open Source Contributors, *Laravel 11 Controllers: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/controllers (accessed: Mar. 10 2025).

[99] Laravel Inc. and Open Source Contributors, *Laravel 11 Routing: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/routing (accessed: Mar. 10 2025).

[100] Symfony SAS and SensioLabs, *Symfony, High Performance PHP Framework for Web Development.* [Online]. Available: https://symfony.com/ (accessed: Mar. 10 2025).

[101] The GraphQL Foundation and Facebook Inc., *GraphQL Spec October 2021: A query language for your API.* [Online]. Available: https://graphql.org/ (accessed: Mar. 10 2025).

[102] Laravel Inc. and Open Source Contributors, *Laravel 11 Sanctum: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/sanctum (accessed: Mar. 10 2025).

[103] Lidl Schweiz DL AG, *LIDL Werbeprospekte als PDF: Unsere Aktionsprospekte.* [Online]. Available: https://www.lidl.ch/c/de-CH/werbeprospekte-als-pdf/s10019683 (accessed: Aug. 28 2024).

[104] *Holzhauser Quartier in Berlin-Reinickendorf: BEOS AG.* [Online]. Available: https://www.beos.net/objekte/holzhauser-quartier-in-berlin-reinickendorf (accessed: Mar. 10 2025).

[105] Gemini Team Google *et al.*, "Gemini: A Family of Highly Capable Multimodal Models," Dec. 2023. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2312.11805v4

[106] T. Zhong *et al.*, "Evaluation of OpenAI o1: Opportunities and Challenges of AGI," Sep. 2024. Accessed: Mar. 10 2025. [Online]. Available: http://arxiv.org/pdf/2409.18486v1

[107] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," 2020. Accessed: Mar. 10 2025. [Online]. Available: https://arxiv.org/abs/2005.11401

[108] Mistral AI Team, *Mistral OCR: Introducing the world's best document understanding API.*, 2025. Accessed: Mar. 10 2025. [Online]. Available: https://mistral.ai/fr/news/mistral-ocr

[109] Laravel Inc. and Open Source Contributors, *Laravel 11 Eloquent: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/eloquent (accessed: Mar. 10 2025).

[110] Laravel Inc. and Open Source Contributors, *Laravel 11 Validation: Laravel 11.x Documentation - The PHP Framework For Web Artisans.* [Online]. Available: https://laravel.com/docs/11.x/validation (accessed: Mar. 10 2025).

[111] D. Harrin and Open Source Contributors, *Validation | Form Builder: Filament 3.x Documentation.* [Online]. Available: https://filamentphp.com/docs/3.x/forms/validation (accessed: Mar. 10 2025).

[112] L. Mateffy, *LLM Magic UI Experiments: Experimental Livewire components for use with mateffy/llm-magic.* [Online]. Available: https://github.com/capevace/llm-magic-ui (accessed: Mar. 10 2025).

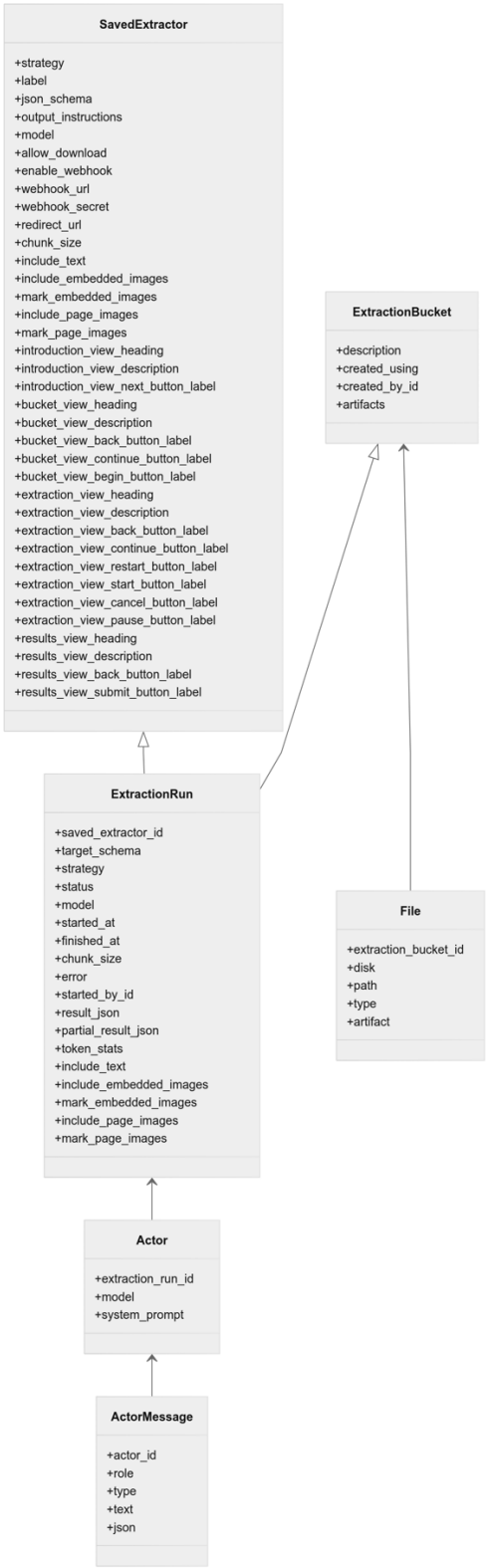
[113] S. Bergmann, *PHPUnit 12: PHPUnit is a programmer-oriented testing framework for PHP. It is an instance of the xUnit architecture for unit testing frameworks.* [Online]. Available: https://phpunit.de/

[114] N. Maduro and Open Source Contributors, *Pest PHP 3.0: The elegant PHP testing framework.* [Online]. Available: https://pestphp.com/

[115] N. Maduro, *Mocking: Pest 3.0 - The elegant PHP Testing Framework.* [Online]. Available: https://pestphp.com/docs/mocking

[116] N. Maduro and Open Source Contributors, *Snapshot Testing: Pest 3.0 - The elegant PHP Testing Framework.* [Online]. Available: https://pestphp.com/docs/snapshot-testing (accessed: Mar. 10 2025).

9 Appendix



Appendix 1: A class diagram of the core data model of Data Wizard

10 List of Figures

11 List of Code Snippets

12 List of Tables

13 Complete Source Code

The entire source code, including all libraries and other files that are part of this thesis submission, are available in the following GitHub repository:

`https://github.com/Capevace/bachelor-thesis-submission`

This repository will be archived on the 13th of March 2025 and will remain in that state. Refer to this repository for the state of the code at the time of submission.

Since two open-source projects were developed as part of this thesis, their respective repos will be continuously updated and are subject to change in the future. They can be found at the following two URLs:

`https://github.com/Capevace/data-wizard`

`https://github.com/Capevace/llm-magic`

In addition to the source code being available, a hosted version of the software is running at the following URL:

`https://data-wizard.ai`

During the evaluation period of this thesis, the following login credentials can be used to access the backend of the software:

Email: bachelor@example.com

Password: bachelor2025!

14 Open-Source Dependencies

14.1 PHP Dependencies

akaunting/laravel-money
Version: ^5.2
https://packagist.org/packages/akaunting/laravel-money

api-platform/graphql
Version: ^4
https://packagist.org/packages/api-platform/graphql

api-platform/laravel
Version: ^4.1
https://packagist.org/packages/api-platform/laravel

blade-ui-kit/blade-heroicons
Version: ^2.3
https://packagist.org/packages/blade-ui-kit/blade-heroicons

blade-ui-kit/blade-icons
Version: ^1.6
https://packagist.org/packages/blade-ui-kit/blade-icons

davidhsianturi/blade-bootstrap-icons
Version: ^1.5
https://packagist.org/packages/davidhsianturi/blade-bootstrap-icons

filament/filament
Version: ^v3.2
https://packagist.org/packages/filament/filament

filament/spatie-laravel-media-library-plugin
Version: ^3.2
https://packagist.org/packages/filament/spatie-laravel-media-library-plugin

guzzlehttp/guzzle
Version: ^7.2
https://packagist.org/packages/guzzlehttp/guzzle

laravel/framework
Version: ^11.0
https://packagist.org/packages/laravel/framework

laravel/octane
Version: ^2.5
https://packagist.org/packages/laravel/octane

laravel/prompts
Version: ^0.3
https://packagist.org/packages/laravel/prompts

laravel/pulse
Version: ^1.4
https://packagist.org/packages/laravel/pulse

laravel/sanctum
Version: ^4.0
https://packagist.org/packages/laravel/sanctum

laravel/tinker
Version: ^2.8
https://packagist.org/packages/laravel/tinker

livewire/livewire
Version: ^3.5
https://packagist.org/packages/livewire/livewire

mateffy/color
Version: ^1.0
https://packagist.org/packages/mateffy/color

mateffy/llm-magic
Version: dev-main
https://packagist.org/packages/mateffy/llm-magic

novadaemon/filament-pretty-json
Version: ^2.2
https://packagist.org/packages/novadaemon/filament-pretty-json

nyholm/psr7
Version: ^1.8
https://packagist.org/packages/nyholm/psr7

spatie/laravel-medialibrary
Version: ^11.8
https://packagist.org/packages/spatie/laravel-medialibrary

spatie/laravel-webhook-server
Version: *
https://packagist.org/packages/spatie/laravel-webhook-server

spatie/pdf-to-image
Version: ^3.0
https://packagist.org/packages/spatie/pdf-to-image

swaggest/json-schema
Version: ^0.12.42
https://packagist.org/packages/swaggest/json-schema

tempest/highlight
Version: ^2.8
https://packagist.org/packages/tempest/highlight

wire-elements/wire-extender
Version: ^1.0
https://packagist.org/packages/wire-elements/wire-extender

14.2 JavaScript Dependencies

@babel/eslint-parser
Version: ^7.25.1
https://www.npmjs.com/package/@babel/eslint-parser

@codemirror/lang-javascript
Version: ^6.2.2
https://www.npmjs.com/package/@codemirror/lang-javascript

@codemirror/lang-json
Version: ^6.0.1
https://www.npmjs.com/package/@codemirror/lang-json

@formkit/auto-animate
Version: ^0.8.2
https://www.npmjs.com/package/@formkit/auto-animate

@tailwindcss/forms
Version: ^0.5.7
https://www.npmjs.com/package/@tailwindcss/forms

@tailwindcss/typography
Version: ^0.5.14
https://www.npmjs.com/package/@tailwindcss/typography

@types/alpinejs
Version: ^3.13.10
https://www.npmjs.com/package/@types/alpinejs

alpinejs
Version: ^3.14.1
https://www.npmjs.com/package/alpinejs

autoprefixer
Version: ^10.4.20
https://www.npmjs.com/package/autoprefixer

caniuse-lite
Version: ^1.0.30001703
https://www.npmjs.com/package/caniuse-lite

cm6-theme-basic-light
Version: ^0.2.0
https://www.npmjs.com/package/cm6-theme-basic-light

cm6-theme-nord
Version: ^0.2.0
https://www.npmjs.com/package/cm6-theme-nord

codemirror
Version: ^6.0.1
https://www.npmjs.com/package/codemirror

postcss
Version: ^8.4.41
https://www.npmjs.com/package/postcss

postcss-nesting
Version: ^13.0.0
https://www.npmjs.com/package/postcss-nesting

tailwindcss
Version: ^3.4.9
https://www.npmjs.com/package/tailwindcss

tailwindcss-animated
Version: ^1.1.2
https://www.npmjs.com/package/tailwindcss-animated

15 Acknowledgements

I would like to thank my friends and family for everything they have done to assist me during the development of Data Wizard and the writing of this thesis. While balancing both commercial and academic work at the same time, they provided much needed calm during a time of great stress. I would like to wholeheartedly thank my partner, Ande Eitner, for alleviating a lot of pressure during the final stages of the writing process and for providing highly effective emotional support. I thank my more technical friends, specifically Kamran Vatankhah, for being great sparring partners during the development of the software, offering great feedback throughout the process. I would also like to thank both of my parents for leading me on a path in life that let me do the things I enjoy doing today. Their parenting and advice has made me the person I am now and their unwavering guidance throughout my life has pushed me to accomplish things I have not thought possible.

I also appreciate the opportunity given to me by both Leuphana University and Prof. Dr. Guido Barbian, to write my bachelors thesis about a topic that I am deeply interested in, allowing me to contribute something to a new and exciting field of research. I am thankful about the fact that I get to continue working on and developing the software solutions built as part of this thesis in the future.

This thesis benefited from the use of Grammarly, an AI-based proofreading tool, which aided in identifying and correcting grammatical errors and in improving textual clarity. The thesis inherently relies on the use of AI models, specifically Large Language Models, to provide the technical abilities described within it and would not be possible without them. GitHub Copilot was used to improve the inline autocomplete functionality of the code editor used. Nonetheless, all ideas and concepts are my own and of my own making.

16 Declaration of Authorship

I hereby certify that I have written this thesis independently and have not used any sources or aids other than those specified. I have correctly cited all passages that I have quoted or paraphrased. I have not yet submitted the thesis in the same or a similar form for any other course or examination authority.

Lukas Mateffy, 13.03.2025