Master Thesis

CIExMAS: Closed Information Extraction using a Multi-Agent-System

Max Lautenbach
(matriculation number 1980683)

18.08.2025

Submitted to
Data and Web Science Group
Dr. Sven Hertling
University of Mannheim

Abstract

Closed information extraction has so far relied mainly on fine tuned transformer models that require large training datasets. Large language models have been used for open information extraction in this context. However, their application to closed information extraction has not yet been explored, nor has the use of artificial intelligence agents in this domain. In this work, I introduce CIExMAS, a collection of modular architectures based on multi agent systems for closed information extraction without the need for fine tuning. The best architecture comprises specialized agents for triple extraction, uniform resource identifier mapping and validation, supported by tools for integration with knowledge graphs. CIExMAS is designed as an exploratory framework that tests a broad range of architectural setups to evaluate the general feasibility of multi agent systems in this domain. In the evaluation, CIExMAS achieved up to 80% of the performance of current state of the art models, despite requiring no fine tuning. This result highlights the effectiveness of approaches based on artificial intelligence agents in closed information extraction and points to their potential as flexible, scalable and lightweight alternatives. As an initial study in this direction, CIExMAS lays the foundation for further research at the intersection of artificial intelligence agents and closed information extraction.

Contents

List of Figures

List of Tables

List of Abbreviations

1 Introduction

LM!s (**LM!**s) and especially modern **LLM!**s (**LLM!**s) have significantly improved the ability of machines to understand and generate human language. These models have enabled progress in various downstream tasks ranging from question answering to summarization and reasoning (**?**). However, their static training data limits their ability to incorporate new or domain-specific knowledge, a limitation known as the knowledge cutoff problem. Most **LLM!**s are trained once on large corpora and as a result they have no built-in access to information that emerged after the end of their training period and cannot retrieve up-to-date or external knowledge out of the box (**??**).

As a result there is a growing interest in structured external knowledge sources, especially **KG!**s (**KG!**s), to provide reliable and semantically rich context for generative models (**?**). Knowledge graphs are a concept used to store knowledge in a form that allows machines to retrieve and interpret semantic relationships (**?**). They are typically modeled as directed graphs where nodes represent real-world entities and edges define the relationships between these entities (**?**). Ontologies specify the types of entities and relationships and additionally provide logical constraints that govern how instances can be interpreted within a domain (**??**). The increasing use of **LLM!**-based applications has therefore renewed attention on **KG!**s as critical infrastructure for trustworthy and explainable **AI!** (**AI!**) (**?**).

At the same time this demand creates a practical challenge: many enterprise or domain-specific datasets have not yet been transformed into a knowledge graph. Most relevant knowledge is still locked in unstructured text documents, reports or communication logs (**?**). To make this information accessible to downstream applications it must be transformed into structured representations that align with a predefined ontology or knowledge schema.

This transformation process leads directly to the research field of **cIE!** (**cIE!**). As a specialized task within the broader area of **IE!** (**IE!**), **cIE!** focuses on extracting subject–predicate–object triples from text such that all components can be linked to valid entries in a target knowledge graph (**?**). This makes **cIE!** a crucial enabling technology for knowledge base population (**?**).

1.1 Motivation and Problem Statement

The automatic extraction of structured information from text is a long-standing goal in natural language processing. This task, generally referred to as **IE!**, includes subtasks such as named entity recognition, entity linking, and relation extraction. These components aim to identify real-world entities in text, assign them canonical identifiers, and detect semantic relationships between them (**?**).

cIE! is a constrained subtask of **IE!**, where the goal is to extract subject–predicate–object triples that exactly match predefined entries in a **KG!**. In contrast to open **IE!** approaches, where relations may be freely expressed in natural language, **cIE!** requires strict alignment with the vocabulary and structure of a target ontology. This constraint makes **cIE!** especially relevant for applications that involve structured knowledge integration or automated **KG!** construction (**?**).

This tight coupling reduces reusability and makes it difficult to adapt systems to new domains or ontologies without significant retraining. Consequently, researchers are exploring more modular alternatives that promise better generalisation without the need for task-specific fine-tuning (**?**).

State-of-the-art models such as GenIE and synthIE model have shown that **cIE!** can be performed with very high accuracy, particularly when the models are fine-tuned on a high-quality dataset. However, (**??**) demonstrate a limitation that models trained on one dataset do not generalize well to others (e.g., synthIE) even when the task and **KG!** structure remain similar. This lack of generalization highlights a key drawback of current **IE!** systems, which often rely on supervised fine-tuning and are closely coupled to a specific domain or knowledge graph.

To address such challenges, large language models have been explored for information extraction tasks in general, leveraging their strong **ICL!** (**ICL!**) capabilities. These capabilities allow **LLM!**s to adapt to new tasks and domains based on a few examples or instructions provided at inference time, rather than requiring retraining. This property makes them a promising candidate for overcoming both the limited generalization and the need for extensive fine-tuning in current **cIE!** models.

While LLM-based approaches have shown promise for **IE!**, their potential for the more constrained setting of **cIE!** has not yet been systematically investigated. This opens up the possibility of solving **cIE!** through prompting alone, without any task-specific fine-tuning.

At the same time, recent work in agentic AI explores how **LLM!**s can be used as decision engines in **MAS!**s (**MAS!**s) that interact via tool use, planning, and role-based coordination (**???**). These systems have shown promise for complex reasoning tasks and dynamic workflows. **?** further demonstrate that even single-agent architectures can achieve state-of-the-art performance on **IE!** tasks.

These developments motivate the hypothesis that **MAS!**s with **LLM!**s as decision engines may offer an effective and generalizable approach to **cIE!**. Unlike monolithic pipelines or fine-tuned models, multi-agent architectures can decompose the task into modular roles such as entity extraction, **URI!** (**URI!**) disambiguation, and triple validation while combining the flexibility of prompting with the ability to access external tools. This is especially relevant when generating **KG!**-ready triples, as these often require lookups or validation against the underlying graph structure. Such tool use is increasingly common in modern agentic frameworks (**?**) and forms the foundation of the CIExMAS architecture proposed in this thesis.

1.2 Research Objectives and Questions

This thesis investigates whether **cIE!** can be solved effectively by **LLM!**-based **MAS!**s without relying on model fine-tuning. The central idea is to test such systems as an alternative to traditional supervised fine-tuned **cIE!** models. These agents operate in distributed, role-based settings and coordinate through a shared plan or workflow, performing tasks such as entity and relation extraction, **URI!** matching, or validation by means of prompting and tool use.

The following research questions guide this investigation:

RQ1 To what extent can **LLM!**-based **MAS!**s perform closed information extraction on unstructured text?

RQ2 Which agentic architectures (e.g., single-agent, supervisor-agent, or agent networks) perform best for this task in terms of accuracy and robustness?

1.3 Methodological Approach

To address these questions, this thesis presents CIExMAS, a closed information extraction multi-agent system. CIExMAS was developed through an iterative process that emphasised modularity and extensibility, but followed no rigid pipeline. Rather, agents and prompts were initially tested on a small sample set and improved step by step, using qualitative trace inspection and targeted adjustments. This empirical, trial-and-error driven strategy was followed by evaluation on a larger benchmark subset.

The system was implemented using the LangGraph framework, which provides abstractions for **LLM!**-based agent flows and tool orchestration, and served as the foundation for developing and testing several configurations. These configurations ranged from a single-agent to supervisor–worker hierarchies and decentralized agent networks. In all setups, the goal was to extract valid triples from unstructured documents using only prompt engineering and tool usage, without any model fine-tuning.

To evaluate this agentic approach, pre-trained snapshots of existing comparison models GenIE and the synthIE model were used. Both were run on the same subset of the synthIE dataset as the CIExMAS configurations, enabling a fair comparison. Performance was assessed using macro-F1 as well as entity-level precision, recall, and coverage to capture both overall and component-specific effectiveness.

The evaluation considered both absolute performance and the ability of the systems to generalise without retraining. All CIExMAS configurations and the comparison models were run on the same subset of the synthIE dataset to ensure comparability, using only tool access and prompting for task execution.

1.4 Contribution of this Thesis

This thesis introduces a novel approach to **cIE!** by proposing several **MAS!** architectures that leverage large language models in combination with external tools. The proposed

systems incorporate agents specialized in **URI!** retrieval, semantic validation, and transformation of triples from `Turtle` notation to human-readable labels. They systematically evaluate the performance of multiple architectural patterns, including single-agent baselines, supervisor–worker models, customized pipelines, and decentralized networks.

Furthermore, the work empirically demonstrates that incorporating knowledge graphs and using tool-supported **URI!** matching are essential for producing correct **cIE!** output. The results also show that recent open-source large language models are capable of solving the task of **cIE!** without the need for model fine-tuning. The developed framework is designed to be both extensible and generalizable, thus laying the groundwork for future research in agentic systems for knowledge-based reasoning and extraction.

1.5 Limitations

Despite its promising results, this thesis has several limitations. The evaluation is restricted to a single synthetic benchmark dataset and does not include multilingual scenarios or real-world enterprise data. The primary focus is on validating the functionality and coordination of the **MAS!**, and as a result deeper semantic modeling, ontology alignment, or integration into production-level knowledge graph infrastructure remains outside the scope of this work.

In addition, this work does not explicitly address ethical risks such as knowledge bias or error propagation through incorrectly generated triples. The use of pre-trained **LLM!**s may introduce unintended biases, and the extraction process lacks mechanisms for ensuring fairness or factual correctness in sensitive domains. These aspects should be carefully considered in downstream applications.

1.6 Structure of the Thesis

The thesis unfolds over six chapters and one appendix, each building on the previous to guide the reader from basic concepts to empirical evidence and, finally, to broader implications.

Chapter **??** lays the conceptual groundwork. After introducing essential terminology and formalisms of knowledge graphs (Section **??**), it narrows the focus to **cIE!** (Section **??**) and explains the functioning and training of modern **LM!** (Section **??**). Subsequent sections explore sentence-level representations for semantic similarity (Section **??**), the paradigm of retrieval-augmented generation (Section **??**), and the principles of AI agents and **MAS!**s (Sections **??**–**??**). The chapter concludes with best practices for agent design (Section **??**), which later inform the CIExMAS architecture.

Building on this foundation, Chapter **??** surveys prior research. Section **??** catalogues publicly available datasets that target entity and relation extraction, while Section **??** analyses state-of-the-art approaches ranging from end-to-end transformer models to pipeline-based systems. Particular attention is paid to generative AI methods and their reported generalisation gaps, an observation that motivates the agentic direction taken in this thesis.

Chapter **??** details the proposed solution. Section **??** compares five agent architecture patterns, including a baseline, several supervisor variants, a ReAct implementation, and a decentralised network. Section **??** lists the bespoke tools such as **URI!** retrieval, network traversal, and semantic validation that are required by the agents, whereas Section **??** explains the iterative prompt engineering strategy adopted. Error handling routines are summarised in Section **??**.

The empirical setup appears in Chapter **??**. Section **??** describes the dataset split, competing models and knowledge graph access rules. Evaluation metrics are justified in Section **??**, followed by a comparison of the best-performing CIExMAS model with the comparison models in Section **??**. Section **??** presents the results for nine configuration stages and variations in the underlying **LLM!**s, describing their design and analysing their impact on performance. Section **??** discusses the quantitative results, highlighting the influence of architectural choices and language model variation.

Chapter **??** closes the thesis by summarising the contributions, acknowledging limitations, and highlighting where further research could build on this work, including multilingual extension, domain adaptation, and ontology-driven refinement. Appendix **??** contains supplementary material such as extended result tables and a detailed category-wise performance analysis that supports the claims made in the main text.

2 Background

This chapter provides the theoretical foundations necessary to understand the core concepts addressed in this work. It begins with an introduction to knowledge graphs (Section **??**), which serve as the foundational data structure for this approach. Building on this foundation, Section **??** introduces the concept of **cIE!**, which constitutes the central problem addressed in this thesis.

Subsequently, the chapter outlines the theoretical background related to the **MAS!** approach. It first presents the relevant core technologies, including language models, sentence embeddings, and retrieval-augmented generation (Sections **??**, **??**, and **??**). Based on these, the concepts of **AI!** agents and **MAS!**s are introduced (Sections **??** and **??**). The chapter concludes with a discussion of best practices in agent design (Section **??**).

2.1 Knowledge Graphs

Knowledge graphs are a concept used to store knowledge in a form that allows machines to retrieve and interpret semantic relationships (**?**). To achieve this, knowledge graphs are modeled as directed graphs, where nodes represent real-world entities and edges define the relationships between these entities (**?**). Ontologies specify the types of entities and relationships, and additionally provide logical constraints that govern how instances can be interpreted within a domain (**??**). They can be understood as the rulebooks of a knowledge graph, defining what types of entities and relations are allowed and how they can be combined. Based on these definitions, the knowledge graph ecosystem can be divided into three fundamental building blocks: knowledge graph construction, storage, and consumption (**?**).

The construction of knowledge graphs involves specification, modeling, and data lifting (**?**). In the specification phase, the requirements of the knowledge graph are defined, followed by the creation of an ontology. Subsequently, the data sources that will provide the content of the knowledge graph must be processed (**?**).

A crucial step within construction, with implications for later storage, is selecting a suitable representation format. Knowledge graphs are typically represented using the **RDF!** (**RDF!**) data model, the standard for data interchange in the context of knowledge graphs (**?**). Therefore, the original data sources must be transformed into the **RDF!** format (**?**).

RDF! is composed of triples, where each triple describes a relationship and consists of three parts: a start node, an edge, and a target node. All components of a triple must be either a resource identifier, a blank node, or a literal. In the context of this work, the

resource identifiers used are **URI!**s, such as *http://example.org/entity/Angela_Merkel*. Blank nodes are used when a resource cannot be explicitly specified, for example in triples describing concepts such as *someone* or *something*. Literals represent simple values such as integers or strings (**?**).

Each triple can also be interpreted as a simple English sentence consisting of a subject (start node), predicate (edge), and object (target node). There are standard rules regarding the format of each triple component. Subjects can be either a **URI!** or a blank node. Predicates (hereinafter referred to as properties) must be **URI!**s. Objects may be a **URI!**, a blank node, or a literal. RDF, like other semantic web technologies, supports multiple syntactical representations of triples. One widely used format is `Turtle` (**?**).

`Turtle` syntax consists of triple statements separated by spaces, tabs, or other whitespace. Each statement ends with a period. **URI!**s are enclosed in angle brackets, blank nodes are represented with an underscore followed by a colon and an arbitrary name, and literals are enclosed in quotation marks. For literals, the `Turtle` parser may infer the data type (e.g., integers, decimals, booleans) or it can be explicitly specified by the user. The following example illustrates how four RDF triples can be represented using `Turtle` (**?**):

```
# Angela Merkel is member of the CDU.
<http://example.org/entity/Angela_Merkel> <http://example.org/property/
    member_of> <http://example.org/entity/CDU>.

# Angela Merkel is written "Angela Merkel".
<http://example.org/entity/Angela_Merkel> <http://www.w3.org/2000/01/rdf-schema
    #label> "Angela␣Merkel".

# Angela Merkel is greeted by someone.
<http://example.org/entity/Angela_Merkel> <http://example.org/property/
    greeted_by> _:b1.

# This someone works for Friedrich Merz.
_:b1 <http://example.org/property/works_for> <http://example.org/entity/
    Friedrich_Merz>.
```

Listing 2.1: Example of a Knowledge Graph in `Turtle` Format

To simplify `Turtle` statements, prefixes can be defined. Prefixes correspond to a namespace, which is a common path in a **URI!** under which related resources are grouped. A prefix is defined using the `@prefix` keyword, followed by a name and the corresponding namespace. Using prefixes, the previous example can be rewritten more compactly (**?**):

```
@prefix ex: <http://example.org/entity/>
@prefix exp: <http://example.org/property/>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

ex:Angela_Merkel exp:member_of ex:CDU.
ex:Angela_Merkel rdfs:label "Angela Merkel".
ex:Angela_Merkel exp:greeted_by _:b1.
_:b1 exp:works_for ex:Friedrich_Merz.
```

Listing 2.2: Example of a Knowledge Graph in `Turtle` Format with Prefixes

Beyond describing entities and their relationships, `Turtle` is also used for defining ontologies. Ontologies can be modeled using the **RDFS!** (**RDFS!**) or the **OWL!** (**OWL!**), where OWL extends the expressiveness of RDFS.

RDFS! provides a standardized way to define classes and properties within a knowledge graph. It also allows the specification of semantic constraints, such as class restrictions. One commonly used mechanism is the definition of domains and ranges for properties. Domains and ranges restrict the type of the subject and object, respectively. For example, if the property `exp:member_of` is defined with the domain `ex:human` and the range `ex:organisation`, then every subject of a triple using this property must be of type `ex:human`, and every object must be of type `ex:organisation`.

Since **RDFS!** has limited expressiveness, **OWL!** was developed to support additional constructs such as symmetric properties. For instance, defining `exp:married_to` as symmetric implies that if A is married to B, then B is also married to A. **OWL!** supports a wide variety of constraints and enables the definition of complex rules for both classes and properties (**?**).

Following the construction phase, the next component is knowledge graph storage. This task is handled by so-called triple stores, which are systems designed to persist and retrieve RDF triples. Triple stores serve as the central component that links data construction with data consumption, and they must comply with the **RDF!** specification (**?**). One widely used implementation of a triple store is Apache Jena. Jena provides a comprehensive software stack for storing and querying RDF data. It offers direct API and I/O access in Java and supports all relevant standards, including **RDFS!** and **OWL!** (**?**). To enable access via **SPARQL!** (**SPARQL!**), the de facto standard query language for RDF data, Apache Jena includes the Fuseki module, which functions as a **SPARQL!** endpoint (**?**).

Within the consumption building block of the knowledge graph ecosystem, **SPARQL!** is a W3C recommendation and thus effectively the standard language for querying and modifying data in RDF knowledge graphs (**?**). Its syntax closely resembles `Turtle`, which facilitates its adoption in RDF-based systems. Commonly used query forms include

`SELECT`, `ASK`, and `INSERT`. While `SELECT` queries are used to retrieve specific data from the graph, `ASK` queries return a boolean value indicating whether at least one match for the given pattern exists in the graph. In contrast, `INSERT` queries are used to add new data into the graph (**?**).

A typical **SPARQL!** query begins with the query type, followed by the `WHERE` clause that specifies the graph pattern to be matched. In `SELECT` queries, variables are identified by a leading question mark (e.g., `?organisation`). Similar to `Turtle`, **SPARQL!** supports the use of prefixes to simplify long **URI!**s. In addition, queries can include filtering conditions and modifiers such as `LIMIT` or `OFFSET` to constrain or paginate the results (**?**). The following example illustrates a query that retrieves all organisations Angela Merkel is a member of:

```
@prefix ex: <http://example.org/entity/>
@prefix exp: <http://example.org/property/>

SELECT ?organisation WHERE {
  ex:Angela_Merkel exp:member_of ?organisation
}
```

Listing 2.3: Example of a **SPARQL!** Query

With the core components of knowledge graphs defined, one practical use case can help illustrate their capabilities and associated challenges. The project HAVAS 18 aimed to monitor start-up activity, tech trends, and tech talent using knowledge graphs. It relied on existing relationships between such entities in social networks to reveal hidden connections and gain deeper insights into the evolution of start-ups and technologies. Since social networks already provide structured relational data, they serve as a valuable data source for such graphs (**?**). However, **?** also emphasize key challenges such as entity resolution, disambiguation, and processing of unstructured data. These challenges are also central to the present work.

In addition to specific applications, broader initiatives such as Wikidata and DBpedia aim to represent Wikipedia as a knowledge graph. Wikidata was created to serve as a structured, multilingual knowledge base for factual content in Wikipedia. It eliminates redundancy and enables machine-readability of facts. As a community-driven project, it allows users to edit and maintain its content, and it now serves as the central factual backbone of Wikipedia (**?**). In contrast, DBpedia focuses on extracting structured data from Wikipedia using syntactic patterns in MediaWiki. The extracted data is then transformed into **RDF!**, making it usable in semantic web applications (**?**).

In conclusion, knowledge graphs provide semantically rich data by combining graph structures with standardized frameworks and languages. Publicly available at scale, they offer significant potential in many domains. Nevertheless, implementations often face challenges such as processing unstructured data and resolving ambiguous entities.

2.2 Closed Information Extraction

Closed information extraction refers to the task of extracting triples from unstructured text, where each component of the triple must correspond to elements within a predefined knowledge graph (**?**). This means that the extracted entities and properties must be mappable to existing entries in the knowledge graph, even if their surface forms do not exactly match. In contrast, open information extraction generates free-form triples directly from text, without requiring alignment to a specific schema or vocabulary (**?**).

A related concept within the broader field of information extraction is relation extraction, which focuses on identifying relationships between entities and often includes the detection of the entities themselves (**?**). Relation extraction differs from open relation extraction in that it relies on a predefined set of relation types and structures (**?**). All of these tasks are subfields of information extraction, which broadly refers to the process of extracting entities and their relations from natural language text (**?**).

A primary challenge in **cIE!** is the identification and extraction of entities from text. This step is closely tied to entity linking, as entities must be disambiguated and mapped to corresponding entries in the knowledge graph. This disambiguation is necessary because multiple entities may share identical surface forms. Once entities are identified and resolved, the next step is to detect relationships between them and classify these relations according to a predefined ontology. In a closed setting, these extracted relations must then be mapped precisely onto the structure of the knowledge graph (**??**).

To illustrate this, consider the sentence *"Angela Merkel is the chancellor of Germany."* A possible **cIE!** result, using Wikidata as the target knowledge graph, could be the triple `wd:Q567 wd:P39 wd:Q14211`, where `wd:Q567` represents Angela Merkel, `wd:P39` denotes the property *position held*, and `wd:Q14211` corresponds to the entity *Chancellor of Germany*.

To address these tasks, two principal approaches are commonly used: pipeline-based methods and joint models. Pipeline approaches decompose the task into sequential stages, such as entity recognition, disambiguation, relation detection, and triple generation. In some implementations, individual stages may be further subdivided or combined depending on the system design. In contrast, joint models aim to perform all subtasks simultaneously, often using shared representations. This can help reduce error propagation across pipeline stages by optimizing for the overall goal of producing accurate and coherent triples (**??**).

While the foundational goals of information extraction remain consistent across its subfields, **?** emphasize that techniques developed for open information extraction are generally not applicable to closed settings. The reason lies in the stricter requirements of **cIE!**, which is constrained by the structure and vocabulary of an existing knowledge base. Consequently, the output must be structured in a way that aligns precisely with predefined entities and properties (**?**).

Because **cIE!** outputs can be matched directly against the contents of the target knowledge graph, it is possible to determine true positives, false positives, true negatives, and false negatives. This, in turn, allows the performance of **cIE!** systems to be assessed using standard evaluation metrics such as precision, recall, and F1-score (**??**). These

metrics capture how accurately and completely a system can extract triples that are both correct and mappable to the knowledge graph. A detailed discussion of these metrics is provided in Section **??**.

In summary, **cIE!** represents a highly constrained subclass of information extraction. In addition to the standard challenges of entity recognition, disambiguation, and relation extraction, it introduces the additional requirement of aligning outputs with a knowledge graph. Each step in the process presents opportunities for error, which is why pipeline approaches often emphasize robustness at each stage. Joint approaches, in contrast, seek to optimize the process holistically in order to improve overall accuracy.

2.3 Language Models

A **LM!** is a **ML!** (**ML!**) model that, at its core, predicts the probability distribution over all possible tokens given the preceding tokens in a text sequence (**?**). Since the original Transformer architecture was proposed by **?**, most modern **LM!**s consist of two main components, a tokenizer and a Transformer model. The tokenizer converts text into sequences of integers, each representing a token. These integers serve as indices into a vocabulary that maps character sequences to token IDs (**?**).

In models like GPT, the tokenizer is typically based on **BPE!** (**BPE!**) (**?**). **BPE!** starts with a vocabulary of individual characters, for example `a`, `b`, `c`, and iteratively merges the most frequent adjacent pairs, for example `a` and `t` to form `at`, creating new tokens at each step. This process continues until a predefined vocabulary size is reached, allowing the tokenizer to balance between representing frequent words as single tokens and breaking rare or novel words into smaller, reusable units (**?**).

A **LM!** can be understood as a model that predicts the next token in a sequence based on all preceding tokens (**?**). This prediction can be expressed formally as the prediction of the conditional probability of a token t_i given all previous tokens in the sequence:

$$P(t_i|t_1, ..., t_{i-1}). \tag{2.1}$$

By learning this probability distribution, a language model becomes capable of generating coherent text (**??**).

The core of most modern **LM!**s is the Transformer architecture, which is composed of layers of multi-head self-attention and feed-forward networks. Self-attention allows the model to compute contextualised representations of each token by attending to all previous tokens in the sequence. This requires a learned embedding for each token and three learnable weight matrices W^Q for queries, W^K for keys, and W^V for values (**?**).

These matrices are used to compute attention scores that determine how much focus the model should place on other tokens when encoding a given token. For example, in the phrase "the former German chancellor Angela Merkel", attention mechanisms may associate the token "chancellor" with modifiers such as "former" and "German", thereby encoding contextual meaning (**?**).

To enrich this representation, **?** introduced multi-head attention, which computes multiple attention distributions in parallel, each with separate parameter sets. This

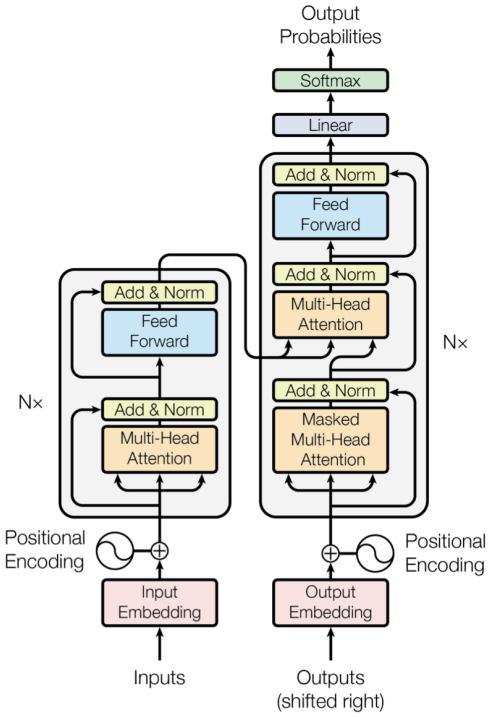

Figure 2.1: Architecture of a Transformer model. (**?**)

allows the model to capture different types of relationships or focus on different positions in the sequence simultaneously, thereby representing information in multiple subspaces at once. The outputs of these attention heads are concatenated and passed through a projection layer (**?**).

In addition, positional encodings are added to token embeddings to inject information about token order. Each layer also includes residual connections and layer normalization to stabilize training. The final layer maps the hidden representation back onto the vocabulary space, yielding a score for each token candidate (**?**). Figure **??** summarises these components and shows how they are arranged within the overall Transformer architecture.

These scores are referred to as logits, which are real-valued outputs representing the model's raw, unnormalized preferences for each possible next token. Let $\mathbf{z} \in \mathbb{R}^n$ denote the vector of logits for all n tokens in the vocabulary, where z_i is the logit assigned to the i-th token and z_j the logit for the j-th token. To convert these logits into a probability distribution over the vocabulary, the softmax function is applied. A temperature parameter T can be used to control the randomness of the sampling process (**?**). The softmax function is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_{j=1}^{n} \exp\left(\frac{z_j}{T}\right)}. \tag{2.2}$$

After applying the softmax function, the model obtains a probability distribution over the entire vocabulary. A decoding method then selects the next token from this distribution.

A common approach is stochastic sampling, where a token is drawn at random according to its probability in the distribution (**?**). The temperature parameter T in Eq. (**??**) controls the sharpness of this distribution: $T < 1$ makes it more peaked, increasing the chance of selecting the most likely token, while $T > 1$ flattens it, making lower-probability tokens more likely. Setting $T = 0$ (interpreted as the limit $T \to 0$) effectively disables randomness, resulting in greedy decoding, where the token with the highest probability is always chosen (**?**).

State-of-the-art models such as OpenAI's **GPT!** (**GPT!**) series or Meta's **LLaMA!** (**LLaMA!**) family are decoder-only architectures that predict each token t_i in an autoregressive manner (**??**). These models are typically trained in an unsupervised fashion by maximizing the likelihood of the next token over large-scale text corpora. In earlier versions, such as GPT-1, fine-tuning on specific tasks was still necessary (**?**).

Subsequent research demonstrated that scaling model size and training data leads to emergent capabilities. **?** showed that their 175-billion parameter model, `GPT-3`, could perform a variety of tasks without task-specific fine-tuning, using only **ICL!**. In this setting, the model receives task instructions and a few examples directly in the input prompt. For instance, to perform translation, the prompt might include *house* → *Haus* and *light bulb* → *Glühbirne*. Using one, several, or no examples is referred to as one-shot, few-shot, or zero-shot prompting, respectively.

ICL! is particularly effective in large models because their greater parameter capacity allows them to capture more complex patterns and relationships from the training data. As a result, they can apply this knowledge to novel tasks with minimal guidance. Even without any examples in the prompt (zero-shot), models like `GPT-3` can achieve results that approach or, in some cases, match those of smaller models that have been fine-tuned for the task. To support such performance, `GPT-3` was trained on a filtered version of the Common Crawl corpus, along with other high-quality datasets, totaling over 500 billion tokens (**?**).

However, general web text corpora are not optimized for instruction-following tasks, and the original `GPT-3` was not fine-tuned to handle such prompts out of the box. **?** point out that real-world usage often involves natural language instructions. To address this gap, they fine-tuned `GPT-3` using a two-step process. First, the model was trained on prompts paired with high-quality human-written completions. Second, human annotators ranked model outputs to train a reward model, which was then used in a reinforcement learning setup to further refine the base model.

The prompts for evaluation came from a held-out set of real user queries submitted to the `GPT-3` API. The resulting model, `InstructGPT`, achieved superior performance on these user-style prompts. Notably, even the smallest `InstructGPT` variant (1.3B parameters) outperformed the much larger base `GPT-3` (175B) in human evaluations, illustrating that instruction-fine-tuned models can be far more effective for instruction-based tasks than models trained purely to predict the next token in a sequence.

To evaluate and compare the performance of modern **LLM!**s, a variety of benchmarks

have been developed. One example is MMLU-Pro (**?**), which tests multitask language understanding across diverse domains such as mathematics, physics, law, and engineering. Another benchmark is the Chatbot Arena[1], a human preference evaluation platform where users compare the outputs of two models on the same prompt. From these comparisons, aggregated preferences are used to compute both a win rate and an overall ranking (**?**).

The impressive performance of large models on these benchmarks is linked to their high complexity. Models such as `GPT-3`, with 175 billion parameters, require significant computational resources (**??**). To make deployment more accessible, two main strategies are commonly pursued. The first is to use smaller models, often based on newer architectures that achieve prior state-of-the-art results with fewer parameters, thereby reducing computational demands while aiming to preserve accuracy (**??**).

The second strategy is quantization, which compresses a model by reducing the bit-width used to store each parameter, thereby lowering memory requirements and increasing inference speed. Naive quantization often leads to large accuracy drops, so methods like GPTQ and AWQ introduce techniques to minimize this effect. GPTQ reduces quantization error during weight transformation, while AWQ preserves salient weights to maintain output quality. These techniques can achieve up to threefold speed-ups with minimal loss in performance compared to standard 16-bit precision (**??**).

In conclusion, modern **LM!**s are Transformer-based models that estimate the probability of the next token in a sequence. Their effectiveness is enabled by large-scale training, advanced architectural designs, and optimization techniques such as quantization. The evaluation of these models relies on benchmarks like MMLU-Pro and Chatbot Arena to assess their capabilities across diverse domains.

2.4 Sentence Embeddings

Sentence embeddings aim to represent textual input, such as full sentences or short passages, as dense vectors in a continuous vector space (**?**). These vectors encode semantic information, allowing the similarity between different pieces of text to be measured mathematically. A common metric for this is cosine similarity, which computes the cosine of the angle between two vectors. A cosine similarity of 1.0 indicates identical direction (i.e., maximum similarity), while a value near 0.0 suggests minimal or no semantic similarity (**?**). Figure **??** illustrates how this measure reflects the angular relationship between two vectors.

Such embeddings are crucial for large-scale semantic similarity tasks, such as retrieving documents or knowledge passages relevant to a user query (**??**). One recent approach is the M3-Embedding model, also referred to as `bge-m3`, which was developed to support multilingual and multi-purpose semantic retrieval across varying input granularities. To achieve this, it leverages the encoder architecture of Transformer-based models, as described in Section **??** and illustrated in Figure **??** (**?**).

[1] Accessible at https://lmarena.ai

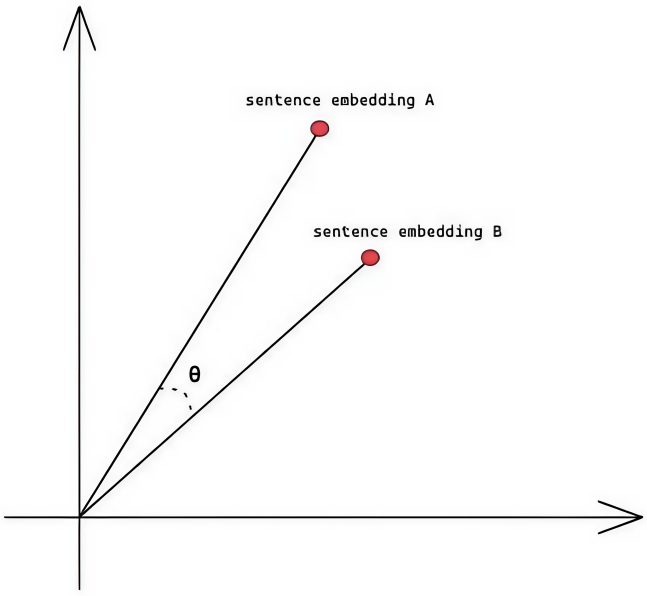

Figure 2.2: Visualization of cosine similarity between two vectors showing how the angle between vectors determines their similarity score. (**?**)

According to (**?**), the training strategy for M3-Embedding follows a combination of pre-training and fine-tuning. During pre-training, the model was exposed to unlabeled corpora using weak semantic signals such as title–body or title–abstract pairs, and translation-based datasets were incorporated to support multilingual capabilities. Fine-tuning was performed using labeled data from multiple languages, as well as synthetic question–answer pairs generated by `GPT-3.5` from multilingual paragraphs. The model was optimized to determine whether a given query semantically matches a target passage, enabling it to perform robust retrieval across languages and domains (**?**).

To evaluate the performance of sentence embedding models across a wide range of downstream tasks, **?** introduced the **MTEB!** (**MTEB!**) [2]. **MTEB!** consists of 58 datasets covering diverse task categories such as classification, clustering, semantic textual similarity, reranking, retrieval, and summarization. It enables standardized comparisons between embedding models across languages, domains, and input types. Models such as M3-Embedding have achieved strong results on the **MTEB!** benchmark, particularly in multilingual retrieval and reranking tasks. This underlines **MTEB!**'s role as a comprehensive and practical framework for assessing the effectiveness of sentence embeddings across a wide range of real-world applications (**?**).

In summary, sentence embeddings enable the representation of sentences in a vector space while preserving their semantic content. By applying vector operations such as cosine similarity, texts related to user queries can be identified efficiently. To achieve high performance on such tasks, recent embedding models like M3-Embedding com-

[2] available under https://huggingface.co/spaces/mteb/leaderboard

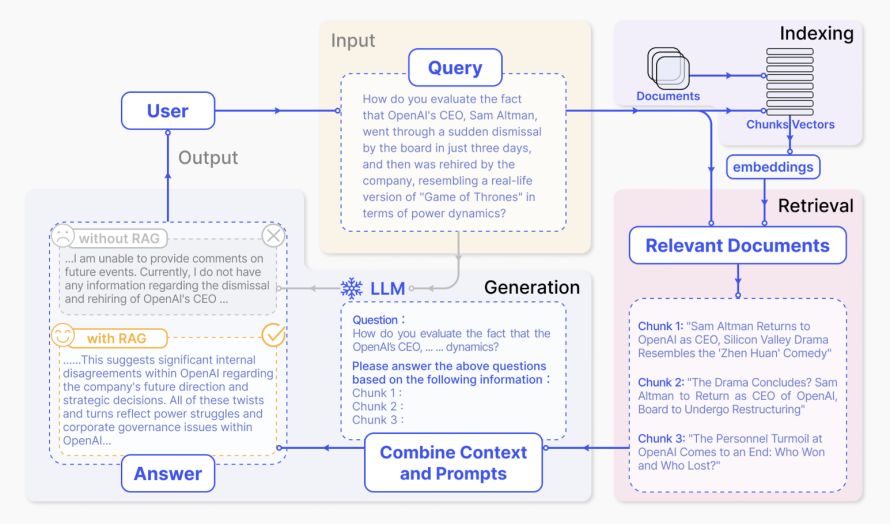

Figure 2.3: Overview of the Retrieval-Augmented Generation (RAG) architecture showing the indexing, retrieval, and generation pipeline. (**?**)

bine Transformer-based architectures with large-scale pre-training and multilingual fine-tuning.

2.5 Retrieval-Augmented Generation

As **LM!**s are trained on static snapshots of large-scale corpora, they are inherently limited in their ability to access up-to-date or domain-specific information that lies outside their training data. **RAG!** (**RAG!**) addresses this limitation by integrating external knowledge sources at inference time. In its simplest form, a **RAG!** system consists of three main components: indexing, retrieval, and generation (**?**).

In the indexing step, unstructured text documents are encoded into dense vector representations using embedding models such as M3-Embedding (see Section **??**) (**?**). These vectors are then stored in a vector database that supports efficient similarity search using vector algebra (**??**).

Figure **??** illustrates the architecture of a typical RAG pipeline. While indexing is performed offline, the retrieval and generation steps occur at runtime. During retrieval, the user query is embedded into the same vector space using the same embedding model. Based on this embedding, a similarity search is performed on the vector database, retrieving the most relevant documents. The retrieved documents are then combined with the original user prompt and, if applicable, a system instruction to form the final prompt. The resulting prompt is passed to the **LM!**, which generates a response informed by the retrieved context (**?**).

A key component of the **RAG!** pipeline is the **VDBMS!** (**VDBMS!**). Similar to traditional database systems, a **VDBMS!** comprises a query processor and a storage

manager. The query processor handles vector-based similarity search and optimizes query execution. One example **VDBMS!** is Qdrant, which employs filtering strategies and adaptive execution paths depending on the size of the vector collection. For large collections, Qdrant reduces the search space through pre-filtering mechanisms, after which a brute-force search is performed on the remaining candidate vectors. The storage manager handles data persistence and access (**??**).

Qdrant organizes data into collections, which are logical groupings of vectors that often represent a specific domain or project. Each vector can be associated with metadata, called payloads, which support structured filtering during retrieval. The system provides integration with frameworks like Langchain, which facilitates the development of LLM-powered applications (**?**). Due to its optimization strategies, Qdrant enables efficient retrieval at scale even for large and complex datasets (**??**).

Despite its advantages, RAG systems are not without limitations. **?** identify several typical failure points in the RAG pipeline. These include cases where relevant information is missing from the indexed content, retrieved documents fail to rank highly enough to be included in the context, or documents are retrieved but discarded during context consolidation. Even when the relevant information is present, the language model may fail to extract it or ignore formatting constraints specified by the user (**?**). Such issues highlight the importance of careful system design, particularly regarding retrieval accuracy, context management, and prompt construction.

In summary, **RAG!** provides a scalable and updatable mechanism for incorporating external knowledge into language model outputs. By combining dense semantic embeddings with similarity search in a **VDBMS!**, it enables the generation of context-aware responses grounded in up-to-date and domain-specific information.

2.6 AI Agents

? defines an agent as "(a)n entity which is placed in an environment and senses different parameters that are used to make a decision based on the goal of the entity. The entity performs the necessary action on the environment based on this decision" (**?**, S. 28574). The environment describes the setting in which the agent operates and is shaped by factors such as data availability and quality, predictability of outcomes, the degree of change over time, and the continuity of the system state. Parameters refer to the data perceived by the agent, and actions are the set of operations it can perform in response (**?**).

Agents typically operate based on a decision engine. In the case of **AI!** agents, this decision engine is commonly implemented using **LLM!**s (**??**). These agents are characterized by a high degree of independence, are tailored to specific tasks, and are capable of adapting their behavior to the current context (**??**). This distinguishes them from traditional workflow automation, where the sequence of steps is rigidly predefined (**?**). **LLM!**s enable agents for human like behavior due to their internal representation of linguistic and behavioral patterns derived from large scale training corpora (**?**).

An **AI!** agent system typically consists of three main components, namely an orches-

tration layer, a language model for decision making and reasoning, and a collection of external tools (**??**). These components are coordinated by a runtime environment. The orchestration layer manages the agent's memory, prepares prompt templates, and controls the message flow, thereby determining which information is passed to the agent and how the resulting actions are executed (**?**).

According to **?**, tools can be grouped into three categories: data tools, action tools, and orchestration tools. Data tools are designed to retrieve information from external systems, for example through database queries or web searches. Action tools allow agents to perform operations in external systems such as modifying records in a **CRM!** (**CRM!**). Orchestration tools include other agents, enabling recursive structures which are further discussed in Section **??**. Tools enable agents to interact directly with their environment, thereby enhancing their autonomy and utility.

Each **AI!** agent is typically responsible for a clearly defined task and is equipped with a matching tool set. In such scenarios, **?** suggest that an agent should be capable of processing complex inputs, using tools reliably, recovering from errors, and performing task level reasoning and planning. While this setup allows for independence within a task boundary, it limits flexibility beyond that scope unless multi agent orchestration is used (**?**).

One example of an **AI!** agent framework is ReAct, proposed by **?**. ReAct combines stepwise reasoning and tool use by prompting the language model to alternate between generating thoughts and performing actions. This process allows agents to maintain a structured internal state composed of thoughts, actions, and observations. Based on this state, the model can iteratively reason about the next step and either call a tool, continue thinking, or terminate execution. The approach is motivated by the observation that human reasoning often interleaves mental reflection with real world interaction (**?**).

To build agents in practice, frameworks such as LangGraph are available. LangGraph is an open source framework for defining agent logic, memory, state tracking, and tool integration using either Python or JavaScript (**?**). LangGraph supports high customizability and is part of the broader LangChain ecosystem. This ecosystem provides additional capabilities that are relevant for agent development, including direct LLM calling and tracing of execution flows (**?**).

While **AI!** agents provide a high degree of autonomy and adaptability, they are not without limitations. One key challenge lies in their limited ability to model causal relationships, as **LLM!**s are primarily trained for next-token prediction rather than understanding cause and effect. Additionally, their performance is often highly dependent on the exact wording of the prompt, making outcomes less predictable. **?** further note that current agents struggle with long-horizon planning in complex, multi-stage tasks, particularly when extended temporal consistency or contingency planning is required. Taken together, these issues can lead to failures in completing complex tasks reliably (**?**).

In summary, **AI!** agents are autonomous systems that make decisions using language models and interact with their environment through structured tools. They rely on orchestration mechanisms to manage memory and state. Frameworks such as ReAct illustrate one way to guide language model behavior in a controlled reasoning and action

loop, but many architectures and implementations exist to serve different use cases.

2.7 Multi-Agent Systems

As stated in the previous section, agents are often limited to a single task, making them inflexible when addressing broader problems. A solution to this limitation can be found in **MAS!**. The idea is to develop a cost efficient, flexible and reliable system by using multiple agents instead of a single one. In this setup, a complex task can either be decomposed into multiple simpler tasks or the same task can be executed multiple times in parallel. The underlying theory is that such task distributions lead to more efficient solutions, which in turn compensates for the overhead associated with managing a **MAS!** (**?**).

To make a general **MAS!** work, different agents are defined for specific subtasks. In some cases, several agents may handle the same task to increase redundancy or scalability. This requires a well-defined routing strategy within the **MAS!**, which can follow either a static or dynamic topology. In a static topology, a leader agent coordinates the other agents and serves as the central point of communication. In contrast, a leaderless **MAS!** allows each agent to decide autonomously which agent to invoke next (**?**).

Leaderless systems introduce greater flexibility, but they also increase the complexity of coordination. Ensuring that all agents contribute effectively toward a shared goal becomes more challenging. Moreover, organizing inter-agent communication and assigning the correct tasks with appropriate inputs becomes increasingly difficult. Fault propagation and fault detection are also harder to manage when multiple agents contribute to a single outcome (**?**).

These concepts can be extended to **LLM!**-based **MAS!**s. Although single **AI!** agents powered by **LLM!**s can already solve complex tasks, increasing complexity often leads to failure modes such as looping or tool misuse (see Section **??**) (**?**). Therefore, decomposing goals into smaller subtasks managed by specialized agents can be beneficial (**?**).

Each **LLM!**-based agent generally shares the same architectural components discussed in Section **??**. In a **MAS!**, however, these components are further extended. In particular, the agent state is transformed into a shared state accessible across agents (**?**). Each agent must also define a handoff mechanism, which determines the next tool or agent to invoke and the corresponding input or state update. In frameworks like LangGraph, handoffs specify the destination and include a payload that updates the shared state. This shared state is typically implemented as a Python dictionary with predefined fields. Consequently, the orchestration layer becomes more complex, and this complexity is managed by agent frameworks like LangGraph (**?**).

? describe several agentic architecture patterns for **LLM!**-based **MAS!**s, which follow classic **MAS!** designs (see Figure **??**). The supervisor pattern represents a leader-follow structure in which a central supervisor agent coordinates the communication with specialized expert agents. These expert agents do not communicate with each other, but only with the supervisor. The hierarchical pattern extends this idea by allowing expert agents to act as intermediate supervisors themselves, enabling a multi-level structure

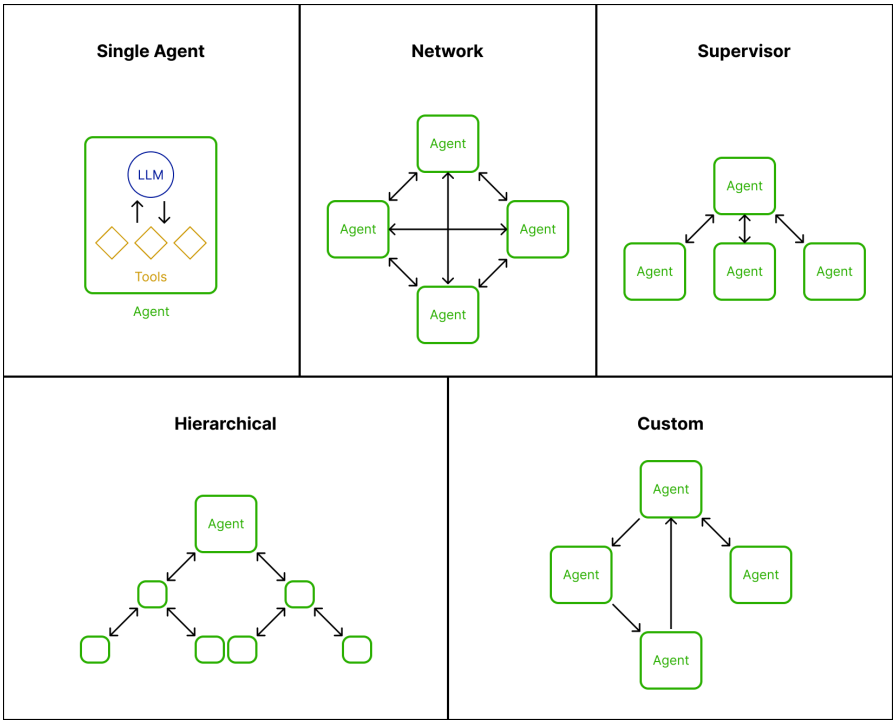

Figure 2.4: Example multi-agent architecture patterns showing supervisor, hierarchical, network, and custom patterns for LLM-based **MAS!**s (self-created after (**?**))

with distributed decision-making responsibilities (**?**).

LLM!-based **MAS!**s can also follow decentralized, leaderless designs (**??**). In the network pattern, each agent decides independently which other agent to pass control to, effectively managing the process flow itself (**??**).

Custom patterns, as described by **?**, do not strictly belong to either leaderless designs or supervisor-based patterns. They enable bespoke configurations, such as limiting communication between specific agents or implementing fixed workflows. Such designs enable highly individualized and adaptable agent roles, particularly when implemented with frameworks like LangGraph (**?**).

Despite their advantages, **LLM!**-based **MAS!**s also introduce new challenges. First, the lack of causal reasoning in **LLM!**s becomes more problematic as agents must interact multiple times. Each **AI!** agent already introduces uncertainty due to prompt sensitivity and a tendency to enter infinite loops. Using multiple agents compounds this uncertainty and increases the overall system fragility (**?**). Furthermore, coordinating communication between agents exposes additional bottlenecks, particularly given the limited context windows of **LLM!**s, both model-wise and infrastructure-wise (**?**). Maintaining a shared context and ensuring alignment across agents becomes increasingly difficult (**??**).

Another critical issue is non-composability. Because **LLM!**s may respond unpredictably, introducing a new agent into the system does not guarantee that it will be

utilized effectively. Instead, it may increase prompt complexity without improving performance. As the number of agents grows, tracing the source of errors also becomes harder due to the sensitivity of **LLM!**s to prompt changes. In addition, the field of **LLM!**-based **MAS!**s remains relatively immature, which contributes to the lack of standardized solutions (**?**).

Nonetheless, recent work has begun to address some of these problems. For instance, integrating retrieval-augmented generation can help mitigate reasoning gaps by supplying agents with relevant external knowledge. The use of deterministic tools within the agent loop also improves reliability. Monitoring and auditing tools can support root cause analysis, and simulation tools may enhance causal awareness. Communication challenges may be addressed through improved memory architectures and coordination mechanisms (**?**).

In summary, **LLM!**-based **MAS!**s offer a promising approach to solving complex problems through the collaboration of multiple specialized **AI!** agents. These systems can be organized hierarchically, networked, or in fully custom configurations. While they provide flexibility and scalability, they also introduce challenges such as error amplification, coordination overhead and reduced traceability. As research in this area progresses, new techniques and tools continue to emerge to address these limitations and improve the robustness of such systems.

2.8 AI Agent Design

As the field of **AI!** agent research matures, so does the understanding of when and how to use such agents effectively. In general, **?** recommend employing **AI!** agents when deterministic rules fall short, particularly when the problem cannot be easily formalized into a ruleset. This is especially true in cases involving complex decision-making or unstructured data, where **AI!** agents are likely to outperform rule-based systems (**?**).

As described in Section **??**, **AI!** agents are typically designed for a single task and constrained by the context they can process. These limitations become apparent when agents begin to fail at following instructions or when their decision-making performance degrades. Some problems are inherently too complex in terms of context or tool usage, leading to tool overload. In such cases, task and tool responsibilities can be decomposed across a **MAS!**. This approach is particularly useful when problems require complex logic that benefits from being broken into subtasks (**??**). However, **LLM!**-based **MAS!**s should be used thoughtfully, as their architecture entails high cost and complexity (**?**).

Regarding agent design, **?** emphasize prioritizing simplicity over complexity. They argue that agentic systems should strike a balance between latency and task performance. Similarly, **?** propose treating agent systems like human teams. Tasks should be concise, actions clearly defined, and the context understandable to humans, given that **LLM!**s are trained on human-generated text. Context compression is another recommended technique to keep prompts manageable and more interpretable (**?**). These principles support the hypothesis proposed by **?**, which suggests that if a human can solve a task, an **AI!** agent should be able to solve it as well, provided the problem is stated clearly.

Aligned with these principles, **?** advise capturing edge cases early to improve agent robustness and keeping components modular and composable, particularly in **LLM!**-based **MAS!**s.

Since **LLM!**s remain prompt-sensitive, effective prompt engineering is essential. **?** suggest considering the training background of the model. As **LLM!**s are primarily trained on prose and optimized to generate coherent text, prompts should closely resemble the structure and format of natural language. Unnecessary formatting should be avoided, as it may reduce performance. Furthermore, **?** recommend teaching agents how to orchestrate the **MAS!**, including a guided reasoning process. Overall, prompt clarity and structure are key to agent effectiveness (**?**).

As outlined in Section **??**, one effective prompting strategy is **ICL!**. When applying one-shot or few-shot prompting, decisions must be made regarding the number, order, format, and similarity of the examples (**?**). Various prompting styles such as role prompting, style prompting, or emotion prompting can also influence outcomes. Another popular method is **CoT!** (**CoT!**) prompting, which encourages the model to reason step by step. Few-shot **CoT!** extends this by incorporating worked-out examples, such as complete reasoning traces leading to correct answers. Automatic prompt optimization can further improve agent performance (**?**).

? evaluate various prompting strategies using the MMLU benchmark, which is closely related to the MMLU-Pro benchmark discussed in Section **??**. Their findings show that few-shot **CoT!** prompting performed best, while zero-shot **CoT!** performed worst. These results align with earlier findings by **?**, who described **LM!**s as effective few-shot learners.

In addition to prompt engineering, well-designed tools are crucial for tasks requiring interaction with external systems or knowledge (**??**). According to **?**, tool descriptions must be as carefully engineered as the prompts themselves. **LLM!**s must be able to generate structured handoffs from unstructured input based on the tool descriptions. Tools should be tested thoroughly, and their usage logged for traceability. Making tools foolproof can reduce the likelihood of errors and support more robust performance (**?**).

Once agents are deployed, performance iteration is key. **?** recommend beginning evaluation with small examples and incorporating human oversight to identify errors not easily caught through automation. In particular, tool execution should be traced and debugged to identify weak points. One example of a platform for this kind of engineering is Langfuse (**?**).

In summary, given the limitations of **LLM!**s, careful agent design is essential. Effective **AI!** agents and **LLM!**-based **MAS!**s must be modeled after human problem-solving processes and supported through robust prompt and tool engineering. When well designed, such systems can effectively address complex tasks (**?**).

3 Related Work

This chapter presents the current state-of-the-art training and benchmarking datasets as well as various approaches in the field of information extraction, with a particular focus on solutions in the area of **cIE!**. Section **??** discusses the REBEL dataset, one of the first large-scale datasets for open and closed information extraction, followed by the synthIE dataset, a synthetically generated resource based on models from the `GPT-3.5` family. Section **??** outlines different methodological approaches, ranging from transformer fine-tuning and prompt tuning to the use of **AI!** agents for open and closed information extraction.

3.1 Related Datasets

Datasets are crucial for training, and especially for evaluating, approaches in the domain of **cIE!**. This requires not only high-quality but also large-scale datasets (**?**). The REBEL dataset was created by analyzing Wikipedia abstracts and extracting all hyperlinks. Additionally, a `RoBERTa`-based **LM!** was used to filter whether a text entails the extracted triples by leveraging `RoBERTa`'s entailment prediction capabilities. This results in a large-scale, automatically created dataset in which noise is accepted. Accordingly, the REBEL dataset can be classified as a silver-standard dataset. Such datasets typically contain a higher level of noise and lower reliability compared to human-annotated (gold-standard) datasets (**?**)[1].

To address issues with noise and predicate frequency skewness in the REBEL dataset, **?** introduced the synthIE dataset. The idea is to leverage the text generation capabilities of **LLM!**s from the `GPT-3.5` series to generate texts that correlate with expected triples (**?**). **?** created an unskewed set of triples based on a subset of Wikidata and prompted these with corresponding instructions into `text-davinci-003`, which produced the dataset `synthIE-text`, or `code-davinci-002`, a base `GPT-3.5` model that produced the dataset `synthIE-code` (**??**). Human-annotated samples revealed that, in contrast to the REBEL dataset, most of the triples within the generated texts of the synthIE dataset were detectable (**?**)[2].

Both the REBEL dataset and the synthIE dataset provide pairs of natural language text and corresponding knowledge triples. These triples include not only the surface forms of entities and predicates as they appear in the text, but also an identifier for the

[1] The REBEL dataset is publicly available on Hugging Face: https://huggingface.co/datasets/Babelscape/rebel-dataset

[2] The dataset is publicly available on Hugging Face: https://huggingface.co/datasets/martinjosifoski/SynthIE

corresponding entry in Wikidata. This enables evaluation for **cIE!** approaches on both datasets.

In practical evaluation scenarios, however, the REBEL dataset has shown significant limitations. Many of the provided triples represent only a fraction of the relations that are actually present in the input text. For example, in the sentence "Richard H. Weisberg is a professor of constitutional law at the Cardozo School of Law at Yeshiva University in New York City, a leading scholar on law and literature", the expected triples are limited to *(Cardozo School of Law; headquarters location; New York City)* and *(Yeshiva University; located in the administrative territorial entity; New York City)*. Other relevant facts such as *profession*, *employer*, or *field of work* are not included, despite being inferable from the text.

Furthermore, a notable portion of the documents in the REBEL dataset spans multiple paragraphs, which increases the complexity of **cIE!** without offering a proportional benefit in evaluation quality. Combined with the reduced coverage of semantically relevant information, these issues significantly impair the usefulness of the REBEL dataset as a benchmark for evaluating system performance. As a result, subsequent development and evaluation in this work primarily rely on the higher-quality synthIE dataset.

In summary, **?** and **?** provided large-scale datasets. The synthIE dataset is of higher quality and thus yields better results when used for evaluation. Both the REBEL dataset and the synthIE dataset offer extensive training, validation, and test splits, serving as a foundation for the development of approaches in **cIE!**.

3.2 Related Approaches

As outlined in Section **??**, open and closed information extraction methods follow either pipeline or joint approaches. These rely on a broad variety of techniques, ranging from probability learning approaches to LSTM-based models and encoder-decoder architectures (**???**). With the rise of Transformer-based architectures, recent approaches predominantly implement Transformer models (**???**).

Even before the emergence of **LLM!**s, Transformer models demonstrated superior performance compared to other state-of-the-art methods. A prominent example is the REBEL model trained on the REBEL dataset (**?**). By fine-tuning a `BERT` **LM!** with texts as input and triples as output, **?** achieved superior results on benchmarking datasets, presenting a flexible relation extraction model.

? introduced GenIE, adapting the REBEL framework for **cIE!**. In their approach, **?** constrained the decoding process in the `BART`-based **LM!** so that it would generate only outputs aligned with entities and relations from the underlying knowledge base Wikidata. They applied output linearization by introducing special tokens to delimit subject, predicate, and object. Furthermore, a constrained beam search was implemented to ensure that only valid tokens from Wikidata could be generated. This method outperformed a traditional pipeline consisting of named entity recognition, entity disambiguation, relation classification, and triplet classification (**?**).

Building upon GenIE and the quality of the synthIE dataset, **?** proposed the synthIE

model. While following GenIE's architecture, the synthIE model replaces `BART` with `Flan-T5` due to broader availability of pre-trained configurations. The best-performing synthIE model, trained on the training split of the synthIE dataset, surpassed a Flan-T5-based GenIE variant trained on the REBEL dataset by over 0.8 macro-F_1 on the `synthIE-text` test set. This result highlights the strong correlation between the quality of training and benchmarking datasets and overall model performance (**?**).

Exploring foundation models that can be fine-tuned or prompt-engineered, **?** proposed a pipeline approach using fine-tuned **LLM!**s for relation extraction, subject identification, and triplet fact extraction. In contrast, **?** introduced a prompt-tuning method for joint optimization. The approach by **?** achieved a 10% F1-score improvement on the document-level open information extraction dataset Re-DocRED compared to prior models. Moreover, they demonstrated that base models like ChatGPT were clearly outperformed by fine-tuned models (**?**).

?, on the other hand, applied prompt tuning for open information extraction across sentence boundaries rather than full documents. Their results showed that well-designed prompt tuning strategies can match the performance of fine-tuned approaches. **?** did not compare their model to other approaches presented in this work. It should be noted that both **?** and **?** used datasets focused on open information extraction rather than **cIE!**. As a result, their performance metrics are not directly comparable to approaches evaluated on the REBEL dataset or the synthIE dataset. In general, comparing models trained and tested on different datasets provides limited insight into their relative strengths, particularly in task-specific contexts like **cIE!**.

The most similar approach to this work is AgentRE by **?**. Leveraging advances in AI agents, they proposed a single-agent framework for open information extraction. In their design, the agent had access to both a retrieval module and a memory module. The retrieval module provided relevant samples and context from a knowledge graph, including sample triples and relation descriptions. Meanwhile, the memory module stored correct and incorrect extractions and supported self-reflection with the help of the **LLM!**. By orchestrating these components, the agent was able to extract valid triples from input texts (**?**).

AgentRE was evaluated on several benchmark datasets unrelated to the REBEL dataset or the synthIE dataset. The main dataset, SciERC, is limited and outdated (**?**). Experiments using `GPT-3.5 Turbo` and a reasoning-enhanced `LLaMA-2-7B` model showed that AgentRE outperformed other methods on most of the datasets evaluated by **?**, highlighting the potential of AI agents for information extraction (**?**). However, as with other approaches, the implications for **cIE!** remain limited.

In summary, recent advances in **LLM!** research have demonstrated improvements in both open and closed information extraction. A range of approaches such as fine tuning, prompt tuning, and agent based frameworks can address the problem. Furthermore, the importance of high quality datasets for training and especially for benchmarking in **cIE!** has been clearly demonstrated. However, many of the approaches discussed either do not use the latest **LLM!**s or do not directly address **cIE!**, and closed information extraction has so far relied mainly on fine tuned Transformer models that require large training datasets. While large language models have been applied to open information

extraction, their use in closed information extraction and the integration of artificial intelligence agents in this domain remain unexplored.

4 Approach

This work introduces CIExMAS, a novel approach to **cIE!** based on **MAS!**s. The core idea is to evaluate both the pipeline and the joint approach to **cIE!** through multiple specialized agents. To fulfil this task, agents must be capable of accessing and retrieving data directly from the knowledge graph. For this reason, the concept of tools is employed.

Instead of relying on model fine tuning, the proposed **MAS!** is designed entirely around prompt tuning and **ICL!**. The overarching goal is to create a system that can be applied in real world scenarios. In this context, real world applicability means that the system must be able to handle diverse text types and operate with different knowledge graphs. The decision to focus on **ICL!** is informed by findings from **?**, who showed that, within the `GPT-3` language model, this method could be used effectively and, in certain cases, even surpass domain specific fine tuned models. Moreover, recent advancements in large language models demonstrate that newer architectures can match or surpass the performance of their predecessors, sometimes with fewer parameters (**??**). Details on the language models used in this work are provided in Section **??**.

An iterative development process was followed to design and evaluate a range of **MAS!**s, drawing on the best practices discussed in Section **??**. To the best of the author's knowledge, no prior research has proposed a **MAS!** specifically for **cIE!**, which made the iterative process essential as it enabled the identification of limitations in early designs and informed subsequent refinements. As a result, five core architectures emerged.

The *baseline architecture*, described in Section **??**, follows a strict pipeline-oriented design. The **SSA!** (**SSA!**) and **SSSA!** (**SSSA!**) architectures, outlined in Sections **??** and **??**, introduce task-splitting mechanisms based on the baseline. In contrast, the *ReAct architecture* in Section **??** simplifies the system by consolidating all responsibilities into a single agent while retaining tool usage. Finally, the *network architecture*, detailed in Section **??**, demonstrates how expert agents can autonomously define and manage the system's flow.

In addition to agent structures, several tools were developed. The most important tool for **cIE!** is the ***URI!** retrieval tool* (Section **??**), which performs a similarity search to support the mapping of search terms to corresponding **URI!**s. To further increase knowledge graph integration, additional tools were designed, including the *network traversal tool* (Section **??**), the *semantic validation tool* (Section **??**), and the *turtle-to-label tool* (Section **??**). However, the presence of a tool does not imply that it was integrated into every architecture.

Agents are implemented as calls to **LLM!**s using a prompt template, which is dynamically filled with selected input fields. In principle, many **LLM!**s supported in LangChain offer structured output parsing, where responses can be automatically mapped to pre-

defined schemas (**?**). Since this functionality was not consistently supported across all used models, this work instead relies on parsing structured data directly from the model-generated text. To enable this, the **LLM!** is instructed to output specific XML tags containing information about the next agent, the final output, instructions for subsequent agents, or the input for tools.

It is important to note that all architectures were iteratively refined with regard to prompt design and tool usage, especially those that demonstrated promising results. As previously mentioned, the development of tools was driven by the iterative process chosen for this work. Consequently, certain tools were designed exclusively for specific architectures. Details on which tools were used with which configurations are provided in Section **??**.

4.1 Agent Architectures

This section presents the different agent architectures developed in this work. Each architecture is described with respect to the specific agents involved, the overall system flow, and the processes for transferring control or data between agents. Furthermore, the definition of agent state and the process of passing control or data from one agent to another are outlined for each architecture. Understanding these architectural variations is essential for evaluating the trade-offs between modularity, control, and performance in **MAS!**s for **cIE!**.

4.1.1 Baseline Architecture

The *baseline architecture* represents the initial implementation of a **MAS!** for closed information extraction in this work. It follows a classical pipeline approach to **cIE!** by introducing two specialized agents: an *entity extractor* and a *triple extractor*. This reflects the multi-agent principle of decomposing complex tasks. Closed information extraction is commonly considered such a task (**?**). Additionally, the architecture implements a *supervisor* agent that aligns with the supervisor pattern and coordinates the workflow and communication between agents. In this configuration, the ***URI!*** *retrieval tool* is already integrated to map the free-text triples to corresponding **URI!**s. An overview of this architecture is shown in Figure **??**.

At the core of the *baseline architecture* lies the concept of state, represented as a dictionary containing key information exchanged between agents and tools. Table **??** summarizes its main fields.

Because this is a supervisor-based architecture, orchestration and routing are fully managed by the *supervisor agent*. It is the only agent with access to the complete system state. In addition to coordinating the sequence of operations, the *supervisor agent* is responsible for validating intermediate results and formatting the final output. After invoking the specialized agents for *entity extraction* and *triple extraction*, the *supervisor agent* assesses the quality of the returned results. If needed, it can initiate additional reasoning steps or repeat certain parts of the process. This includes re-running agents to improve disambiguation. The *supervisor agent* uses the `instruction` field to send

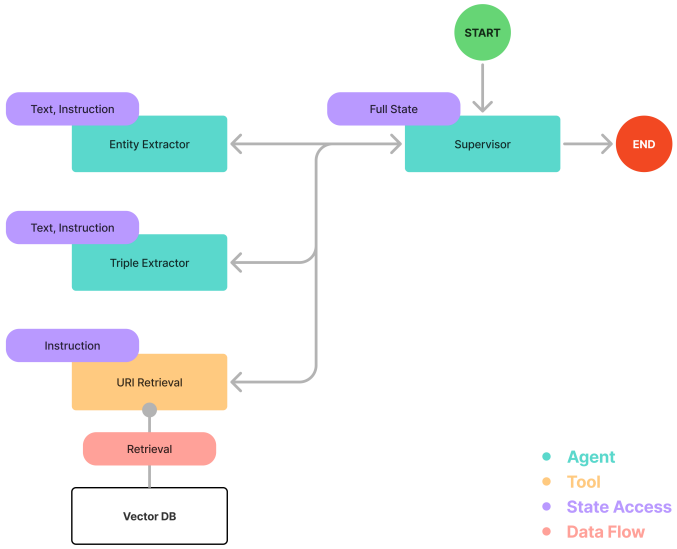

Figure 4.1: Baseline architecture showing the pipeline approach with specialized agents for each step of the closed information extraction process (self-created)

Table 4.1: State fields in the *baseline architecture*

Name	Description
`text`	The document or sentence being processed.
`messages`	List of exchanged messages between agents, enabling history tracking and reasoning.
`instruction`	Task-specific instructions for agents or tools.

specific prompt inputs to other agents and tools. When a triple is produced by the *triple extractor*, the *supervisor agent* constructs search terms and queries the ***URI!*** *retrieval tool*. Based on the retrieved **URI!**s, the *supervisor agent* composes a valid `Turtle` output.

The *entity* and *triple extraction agents* are entirely based on prompt engineering. Initially, both agents were prompted with simple instructions requesting entity or triple extraction respectively. Over the course of the iterations, these agents are expected to perform increasingly well on their narrow tasks as prompts are refined iteratively. Details about the overall prompt engineering process are described in Section **??**.

Overall, this baseline design results in a pipeline, in case any result is error-free. First, entities are extracted. Then, triples are constructed. The supervisor retrieves matching **URI!**s and generates a final `Turtle` output. If this process fails, for example due to missing or incorrect entities or predicates, the supervisor can repeat parts of

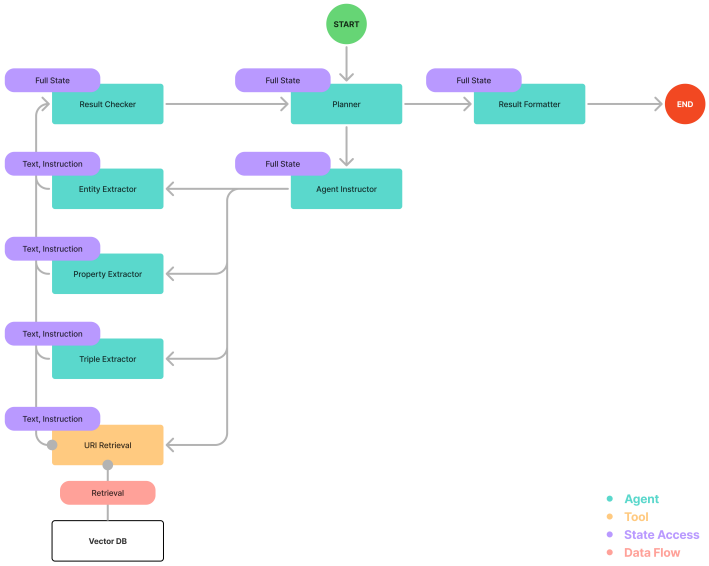

Figure 4.2: Splitted supervisor architecture showing the decomposition of the supervisor agent into specialized sub-agents for better task management (self-created)

the workflow. This flexibility allows the architecture to extend beyond a purely linear structure and reduces the risk of error propagation.

4.1.2 Splitted Supervisor Architecture

Building on the *baseline architecture*, the *splitted supervisor architecture* (**SSA!**) extends the supervisor pattern by decomposing the central *supervisor agent* into a series of smaller, specialized agents. In both setups, the main responsibilities remain the same: planning the task, routing it to the appropriate agent or tool with the correct instruction, evaluating the outputs, and either refining the plan or producing a final `Turtle` string. However, this process proves to be complex, particularly because the idea of using a **MAS!** is to enable iterative adjustments to elements such as triples or **URI!** mappings.

The **SSA!** addresses the limitations of **LLM!**s that struggle with processing many instructions at once or face context window restrictions as prompt sizes grow. To overcome this, the *supervisor agent* role is split into four distinct agents: *planner*, *agent instructor*, *result checker*, and *result formatter*. Additionally, a new *property extractor* for **PropEx!** (**PropEx!**) complements the existing specialized agents for *entity extraction* and *triple extraction*. Because the *property extractor* was added later in the iterative process, its evaluation is discussed separately in Section **??**. Figure **??** illustrates the overall structure of this architecture.

To support the expanded supervision logic in the **SSA!**, the system state was extended to include additional fields beyond `text` and `instruction`, while the `messages` field

from the *baseline architecture* was removed. This structured representation separates core **cIE!** outputs from orchestration-related data and facilitates reasoning over prior steps. Table **??** summarizes the fields used in this architecture.

Table 4.2: State fields in the **SSA!** architecture

Name	Description
`text`	The document or sentence being processed.
`instruction`	Task-specific instructions for agents or tools.
`call_trace`	Log of which agent or tool was invoked with which input, used by the *agent instructor* to track system flow.
`results`	Collected outputs from specialized agents and the ***URI!*** *retrieval tool*.
`comments`	Internal notes written by the *result checker* and *planner agent*.

At the core of the **SSA!** is the *planner agent*. It initiates the overall process, develops a step-by-step plan, and decides at each stage whether to proceed or terminate by delegating final output creation to the *result formatter*. If the next action requires agent or tool involvement, the *agent instructor* translates the planner's intention into a concrete call with a matching instruction. This modularity allows each agent to specialize: planning agents can focus on sequencing, while instructors handle syntactic formatting. The *result formatter* follows the same philosophy and is responsible for generating the final output in valid `Turtle` syntax. In doing so, it combines all partial outputs such as triples and **URI!**s.

Functioning as the quality control unit, the *result checker agent* plays a critical role in this architecture. This agent evaluates whether the specialized agents are producing meaningful results and identifies missing or incorrect elements. Based on this evaluation, the *result checker agent* provides feedback to the *planner agent* and encourages iterative improvement over the extraction process.

The specialized agents for *entity extraction* and *triple extraction* behave similarly to those in the *baseline architecture*, with adjustments made regarding where their results are stored in the updated state. The new *property extractor* agent was introduced in line with design practices discussed in Section **??**. Its task is to extract properties from the text, which were previously identified as part of *triple extraction*. The ***URI!*** *retrieval tool* remains conceptually the same as in the *baseline architecture*, though its outputs are now written to the centralized `results` field.

In summary, the **SSA!** is an expanded interpretation of the supervisor pattern, aimed at breaking down complex orchestration into smaller, more specialized components. The state has been enhanced to support this increased granularity by providing more transparency and structure, enabling a greater degree of control and adaptability across the agent system.

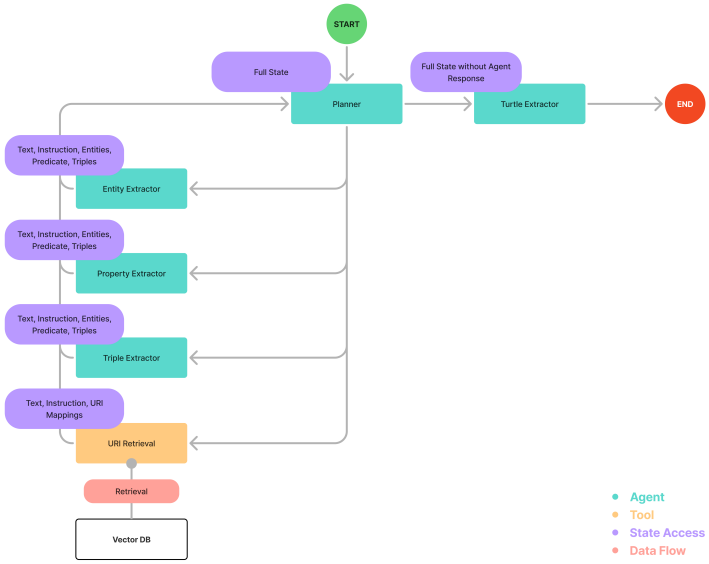

Figure 4.3: Simplified splitted supervisor architecture showing a streamlined version of the supervisor decomposition with optimized agent interactions (self-created)

4.1.3 Simplified Splitted Supervisor Architecture

Continuing the principles established in the *splitted supervisor architecture* (**SSA!**), the *simplified splitted supervisor architecture* (**SSSA!**) streamlines the agent structure for more efficient interaction and reduced complexity. This design assumes that one agent can still manage all supervision tasks as long as it does not have to generate valid `Turtle` output or operate over an unstructured state. To accommodate this, the architecture retains all specialized agents from the **SSA!**, while merging the roles of the *result checker* and *agent instructor* back into the *planner agent*, with output formatting still handled separately. The *planner agent* in this setup functions similarly to the original *supervisor agent* introduced in the *baseline architecture*. A more structured and targeted state is introduced to support these adjustments. Figure **??** provides an overview of the architecture.

A central change of the **SSSA!** lies in its redesigned state. The general-purpose `results` field from the **SSA!** is replaced by dedicated fields that store the core outputs of specialized agents and tools. This explicit separation improves transparency and enables the *planner agent* to access relevant information directly. Because this streamlined structure removes access to full agent outputs, the state also includes a new `agent_response` field. It captures the raw responses from specialized agents and tools, allowing the *planner agent* to verify task completion, detect errors, and initiate corrective actions. Table **??** summarizes the fields in this architecture.

The actual formation of RDF triples remains delegated to the *turtle extractor*, which

Table 4.3: State fields in the **SSSA!** architecture

Name	Description
`entities`	Output of the *entity extraction* process.
`properties`	Output of the *property extractor* agent.
`triples`	Output of the *triple extraction* process.
`uri_mapping`	Mapped **URI!**s from the ***URI!** retrieval tool*.
`agent_response`	Raw responses from specialized agents and tools, used by the *planner agent* for quality checks and corrections.
`messages`	Dialogue history of the *planner agent* to avoid unintended loops.

assembles entities, properties, and **URI!**s into valid `Turtle` syntax. This separation of concerns ensures that the planner agent remains focused on flow control and state refinement, rather than performing complex string transformations.

This architectural simplification is motivated primarily by practical considerations. As context window limitations increasingly influence **LLM!** performance, minimizing unnecessary content in the state becomes essential. At the same time, the system retains the ability to iterate over the **cIE!** process when needed. In doing so, the **SSSA!** balances agent autonomy with prompt efficiency. It represents a targeted evolution of the supervisor pattern that reduces resource demands while maintaining modular clarity within the **MAS!**.

4.1.4 ReAct Architecture

Unlike the previously presented solutions, which all rely on a **MAS!** to address the complexity of **cIE!**, the *ReAct Architecture* takes a contrasting approach. It is designed to evaluate the capabilities of a single agent equipped with reasoning and tool-use abilities (**?**). By eliminating inter-agent communication, the system can focus entirely on tool interactions, reducing coordination overhead and minimizing points of failure.

Another advantage of this joint approach is that it avoids the error propagation that can occur when agents are used to mirror the task-splitting of a traditional **cIE!** pipeline. As discussed in Section **??**, joint optimization benefits from a single decision point, enabling more coherent outputs. Additionally, reducing the number of agent calls leads to lower inference latency and operational costs.

The state in the *ReAct architecture* is kept intentionally simple. Since only a single agent is involved, the original baseline structure from Section **??** is reused. Outputs from tools are written to the `messages` field, allowing the agent to keep track of prior actions and intermediate results.

To support its reasoning capabilities, the agent is equipped with the ***URI!** retrieval tool* from the outset. As the architecture proved effective during iterative evaluations, additional tools were developed to extend its capabilities. These include the *network traversal tool*, the *semantic validation tool*, and the *turtle-to-label tool*, each facilitating

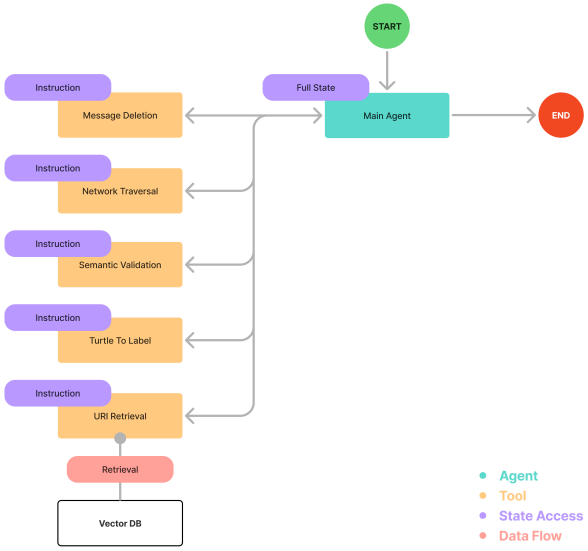

Figure 4.4: ReAct architecture showing the single-agent approach with reasoning and action capabilities for closed information extraction (self-created)

deeper integration with the underlying knowledge graph. Moreover, a *message deletion tool* was introduced to allow the agent to manage its limited context window by pruning irrelevant or outdated information. The overall tool integration is depicted in Figure **??**.

While the *ReAct architecture* does not qualify as a **MAS!**, it plays an important role in contrasting complex **MAS!** configurations. It represents a simplified single-agent approach that highlights how much can be achieved without multi-agent orchestration. Due to its efficiency and simplicity, it is particularly well-suited for early-stage development and tool testing, making it a logical step within the broader iterative design process. All development steps, including the incremental addition of tools and their evaluated effectiveness, are discussed in detail in Section **??**.

4.1.5 Network Architecture

Building on the network pattern introduced in Section **??**, the *Network Architecture* represents a culmination of insights gathered from the design and evaluation of all preceding architectures, tools, and state configurations. It aims to combine the reasoning and validation capabilities developed in the **SSA!** and **SSSA!** with the direct triple extraction process featured in the *ReAct Architecture*. Furthermore, this architecture distributes decision-making across all participating agents rather than centralizing it within a single supervisory entity. An overview of the setup is shown in Figure **??**.

The underlying state in this architecture extends the structured design of the **SSSA!** while introducing new elements to better support routing and control logic. It retains

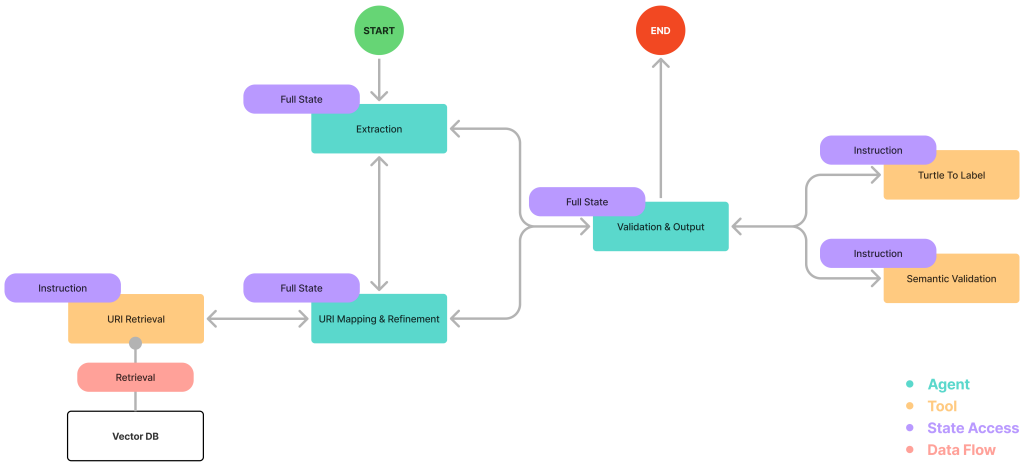

Figure 4.5: Network architecture showing the fully connected **MAS!** with dynamic routing and reasoning capabilities (self-created)

`triples` and `uri_mapping`, the latter now enriched with original labels, along with Wikidata-provided labels and descriptions. To facilitate reasoning over prior steps, a `call_trace` is reintroduced. Additional fields such as `last_call`, `last_response`, `agent_instruction`, and `tool_call` capture the most recent interactions and instructions, enabling efficient forwarding and decision-making. Unlike previous designs, this architecture avoids cumulative state growth, except for the `call_trace`, thereby reducing context size and emphasizing concise, high-value information for agent reasoning. The `messages` field is preserved solely to store the final output string for interoperability with the evaluation framework.

From an agent perspective, the architecture employs three distinct agents with full read access to the complete state. In contrast, tools only receive the input specified in the `tool_call` field. Granting agents full-state visibility allows them to make more informed decisions about subsequent actions within the overall flow.

The initial entry point for processing is the *Extractor Agent*, which mirrors the core responsibilities of the main agent in the *ReAct Architecture*, while delegating **URI!** mapping, validation, and `Turtle` output to separate components. It is the only agent permitted to write to the `triples` field.

Handling the **URI!** mapping is the responsibility of the ***URI!** Mapping and Refinement Agent*. This agent serves as the central unit for generating and updating **URI!** mappings and is the only component allowed to write to the `uri_mapping` field. It is tightly coupled with the output of the *Extractor Agent* and utilizes the ***URI!** Retrieval Tool* to resolve surface forms into structured Wikidata entities and relations.

Validation and output formatting are conducted by the *Validation and Output Agent*. This agent prevents hallucination and ensures semantic correctness by validating triples

Table 4.4: State fields of the network architecture

Name	Description
`triples`	Extracted triples from the text.
`uri_mapping`	**URI!**s with corresponding original labels, plus Wikidata labels and descriptions.
`call_trace`	Log of all agent and tool calls, including inputs.
`last_call`	Most recent agent or tool invocation.
`last_response`	Result of the most recent call.
`agent_instruction`	Instruction received by the agent, forwarded if needed across tool calls.
`tool_call`	Input for the most recent tool invocation.
`messages`	Final output string for compatibility with the evaluation framework.

based on features retrieved from the knowledge graph. To do so, it employs both the *Semantic Validation Tool* and the *Turtle-to-Label Converter*. If any extracted triple fails validation, the agent attempts to refine or regenerate it, either by re-engaging the *Extractor Agent* or by forwarding the task to the ***URI!** Mapping and Refinement Agent*, depending on the assumed source of error.

Overall, the network architecture reflects the lessons learned from previous iterations. It omits unpromising tools, makes use of a highly structured and minimal state, and adheres to a fully distributed decision-making strategy. While still grounded in the **MAS!** paradigm, it reduces coordination overhead and context consumption to a minimum.

4.2 Agent Tools

Agents across the different architectures form the backbone of the approach to **cIE!**. However, since **LLM!**s are inherently limited in their up-to-date knowledge and may hallucinate information, as discussed in Section **??**, additional support is required. In this work, such support is provided exclusively through *data tools* (see Section **??**), as all tools operate by accessing external data sources. These tools address both reliability concerns and performance limitations, enabling agents to retrieve accurate, contextually relevant information during the extraction process.

Two general categories of tools are introduced in this work. The first category focuses on integrating knowledge graph information. This includes the *uri retrieval tool* (Section **??**), the *network traversal tool* (Section **??**), the *semantic validation tool* (Section **??**), and the *turtle-to-label tool* (Section **??**). These tools enable agents to interact with and reason over structured external knowledge.

The second category supports system-level coordination. In this work, only one such tool is introduced: the *message deletion tool* (Section **??**), which is used to manage context size and prevent prompt overflow in long-running agent interactions.

4.2.1 URI! Retrieval

The ***URI!** retrieval tool* remains a central component in solving the **cIE!** task. While various **LLM!**-based agents are capable of extracting free-form triples, the core challenge lies in mapping these triples correctly to the underlying knowledge graph. This mapping capability is provided through the ***URI!** retrieval tool*, which enables interaction with the knowledge graph.

Its core function is to receive search terms from an agent and perform similarity-based retrieval using sentence embeddings and cosine similarity. Search terms can be iterated and varied by the agent, for example by changing the phrasing, the search mode, or the filter settings. This flexibility allows for multiple retrieval attempts, potentially improving mapping performance.

In order to support this mechanism, the tool maintains three search indices, each generated by embedding specific textual representations from the knowledge graph. These include the `rdfs:label`, the `rdfs:description`, and example triples for properties. For the latter, Wikidata provides example triples for many properties, which are reformulated as complete sentences to support semantic encoding of relational context. The construction of these indices is based on Wikidata, the only knowledge graph used throughout this work's evaluation.

The creation of the search indices follows a structured multi-step pipeline. First, documents are loaded from a selected dataset split, such as the test set of the synthIE dataset. Second, all entities and properties referenced in these documents are extracted by identifying the **URI!**s of subjects, properties, and objects. For each subject and object, the corresponding `rdfs:label` and `rdfs:description` are retrieved from Wikidata. For each property, in addition to the label and description, an example triple that demonstrates the property's usage is also retrieved.

In the next step, these textual representations are embedded and stored in separate search indices. The `rdfs:label` values are embedded into a label index, the `rdfs:description` values into a description index, and the example triples into an example index. Alongside the embeddings, relevant metadata is stored with each entry. This includes the corresponding **URI!**, the type of the resource (entity or property), the `rdfs:label` (for entries in the description and example indices), the `rdfs:description` (for entries in the label and example indices), and the example triple itself (for properties in both the label and description indices).

To control retrieval behavior, the tool interprets two parameters, *search mode* and *filter mode*. The *search mode* selects the semantic space used for similarity retrieval, that is, whether to search on the labels index, the descriptions index, or the examples index. The *filter mode*, in contrast, restricts retrieval to either entities or properties. This distinction enables more accurate matching and reduces noise, especially when agents provide ambiguous input.

Through the iterations, the tool is tested in multiple configurations, initially without any search mode, then with search modes enabled, followed by a combination of search and filter modes, and finally in a setup that uses only filter modes together with a search mode for example-based search for properties. Details on these configurations are

provided in Section **??**.

Internally, the ***URI!** Retrieval Tool* parses the agent's instruction, determines the appropriate search and filter modes per term, and executes a cosine similarity search using the term embedding over the selected index. The top result includes the matching entry's `rdfs:label`, `rdfs:description`, and **URI!**.

Architectures sensitive to context limitations may encounter issues when the tool returns extensive candidate lists, which is primarily caused by an excessive number of search terms. To address this, the tool supports a compact output mode. In this mode, an additional **LLM!** call is used to evaluate whether a specific result matches the query. The response is reduced to a simplified **URI!** mapping that includes only the selected **URI!** and a brief justification, omitting alternative matches and detailed descriptions. This compact output is used in configurations of the *Baseline Architecture*, the **SSA!**, and the **SSSA!**, with the exception of the initial baseline version.

In summary, the ***URI!** Retrieval Tool* equips agents with retrieval-augmented access to structured knowledge. By indexing key components of Wikidata and offering fine-grained retrieval controls, it supports precise **URI!** mapping in a variety of architectures. Its flexibility and accuracy-critical role led to extensive iterations throughout this work, as discussed in Section **??**.

4.2.2 Network Traversal

The *network traversal tool* represents a further step in incorporating knowledge graph information into the agent workflow. Its main purpose is to support agents in addressing the challenge of selecting appropriate properties, particularly in cases where the input sentence does not contain the exact label of the target property. This issue could lead to the selection of super-properties or sub-properties. By querying structural information from the knowledge graph, the tool enables agents to explore property hierarchies and make more informed decisions.

To achieve this, the tool executes *SPARQL* queries on Wikidata, focusing on the *subproperty of* relation. These queries retrieve both direct super-properties and sub-properties of a given property. To enhance the output's usability, labels are also queried for each retrieved property. Where available, example triples are included for additional context. The resulting output presents the agent with related properties and supporting examples to assess semantic proximity.

An illustrative example can be found in the property hierarchy involving `member of political party` (`P102`). This property is a subproperty of `member of` (`P463`), which in turn is a subproperty of `part of` (`P361`). An agent that extracts a triple such as "Angela Merkel — part of — CDU" might therefore benefit from exploring this hierarchy to identify the most specific and semantically accurate property. The network traversal tool enables this by surfacing both super-properties and sub-properties along with contextual information, such as labels and example triples.

In some cases, individual properties are connected to a large number of related properties, which may result in long and unordered outputs. Since **LLM!**s often struggle to prioritize relevant information in such cases, additional filtering mechanisms were intro-

duced. Later iterations of the tool included support for inspecting property constraints to assess type compatibility. In Wikidata, these constraints are accessible through qualifiers attached to properties.

Qualifiers in Wikidata are expressed as triples consisting of a predicate and an attribute (**?**). For the purpose of this tool, the *property type constraint* is used. This can be further specified as a *subject type constraint* or a *value type constraint*. The actual constraint is represented by the **URI!** of a specific class that the subject or object of a property must conform to.

Using this information, the tool extends its *SPARQL* query to determine whether the subject and object types in a candidate triple align with the constraints of a given property. To increase flexibility, the system allows for one level of abstraction by including the direct superclass of each type. This choice is made to avoid the risk of infinite expansion when traversing class hierarchies, which may occur in unbounded queries against Wikidata.

Through the network traversal tool, agents gain access to structural and semantic context that allows them to refine extracted triples. The tool supports identification of overly broad or overly narrow property selections. However, its usage can introduce considerable overhead due to the potentially large number of returned results. Agents must therefore balance constraint alignment with semantic appropriateness, avoiding overfitting to constraints while maintaining accuracy in property selection.

4.2.3 Semantic Validation

Language models are prone to hallucinations and typically lack the built-in capability to verify their own outputs, as discussed in Sections **??** and **??**. To address this limitation, the semantic validation tool was introduced. Similar to the approach used in the network traversal tool, it focuses on type restrictions associated with properties in extracted triples. However, it relies on a pre-filtered, loop-free class hierarchy derived from Wikidata, which allows efficient validation across the full ontology.

The tool receives a complete `Turtle` string as input. This string is parsed and split into individual triples, and each triple is further decomposed into subject, property, and object. As described in Section **??**, qualifiers are then used to determine whether type constraints exist for the subject and object. These constraints are matched against the expected types defined in Wikidata. The output of the tool indicates whether the triple passes validation. In the case of a failed check, the tool also returns the expected subject and object types.

Despite its simplicity, the semantic validation tool provides a valuable mechanism for improving result quality. It helps agents detect cases where a chosen property may be semantically inappropriate, even when descriptions or examples retrieved by the **URI!** retrieval tool appear correct. In this way, the tool serves as a knowledge-aware safety net and provides a foundation for iterative improvement.

4.2.4 Turtle to Label

Throughout development, agents occasionally produced incorrect **URI!**s due to copy failures or internalized associations from prior knowledge. Given that Wikidata is one of the largest publicly available knowledge graphs, it is likely that parts of it were included in the training data of various **LLM!**s. As a result, agents may generate **URI!**s based on memorized content rather than verified information. This introduces a particular risk in CIExMAS, where precise usage of **URI!**s is essential.

Wikidata makes use of a consistent structure across entities, properties, and their relationships. However, it distinguishes between the namespace used to define metadata (such as labels or descriptions) and the one used for actual property assertions. Since other tools in this work, particularly those relying on *SPARQL* queries, depend on correctly defined property **URI!**s, namespace mismatches can lead to tool failures or incorrect results.

To address this issue, the *turtle to label* tool processes the final `Turtle` output and attempts to retrieve the human-readable `rdfs:label` for each part of every triple. This is done by executing *SPARQL* queries on Wikidata. If a query fails, it typically indicates that the corresponding **URI!** is invalid or refers to an incorrect namespace. In addition to flagging invalid **URI!**s, the tool detects whether a **URI!** exists but lacks an English label, and communicates this to the calling agent.

When hallucinated **URI!**s are generated, the tool highlights these by re-outputting the free-form triples derived from the text, allowing agents to fall back on reasoning over textual content. In doing so, the tool helps expose incorrect or unresolvable references and supports downstream correction.

Overall, the *turtle to label* tool offers a lightweight mechanism for integrating with the knowledge graph and validating agent outputs. Used in tandem with the semantic validation tool, it can strengthen the agent's ability to detect errors and improves the final quality of extracted triples.

4.2.5 Message Deletion

Context bloating has been addressed repeatedly across chapters, particularly through the careful design of agents, states, and tools. An additional strategy introduced in this work is to provide agents with the ability to manage the context proactively. To support this, the *message deletion tool* was implemented.

This tool allows agents to delete individual messages from the state by their identifier. Its development was motivated especially by the needs of the *ReAct Architecture*, where a single `messages` field stores all outputs from the main agent and tools. Since components such as the *network traversal tool* or the ***URI!** retrieval tool* often produce verbose outputs, this field can quickly become a bottleneck in terms of context size. At the same time, agents are capable of reusing or summarising content. Consequently, the tool enables them to remove outdated or redundant messages and optionally replace them with more concise summaries of the knowledge extracted.

When used alongside agents that can directly edit structured fields, such as the

`triples` field in the **SSSA!** or the *network architecture*, this tool supports self-managed context pruning. While this flexibility helps avoid context length limitations entirely, it introduces new challenges for the agents. They must now make reliable decisions about which information to retain, discard, or summarise. The practical implications of using this tool are further discussed in Section **??**.

4.3 Iterative Prompt Engineering

As outlined in Sections **??** and **??**, **LLM!**s are highly sensitive to prompt design. Finding suitable prompts is therefore a non-trivial and often challenging task. Since this work focuses primarily on developing a range of multi-agent architectures and demonstrating that knowledge graph incorporation improves performance, the prompt engineering process was conducted manually and iteratively.

The general strategy follows the principles discussed in Section **??**. Prompts were written and refined to be solvable by a human, aiming to leverage the reasoning capabilities of **LLM!**s without exceeding their limitations. In addition, most prompts include examples to support **ICL!**. Depending on the use case, these examples either illustrate tool instructions or are drawn directly from the dataset used in a specific task, such as entity extraction. In the latter case, original text passages were combined with the expected entities to show the agent which types of triples are considered relevant.

Another technique applied throughout this process is guided **CoT!**. Agents were explicitly instructed to follow a predefined and prompted chain of thought. Most prompts consist of a consistent structure, including a role introduction, a description of available tools (if applicable), a list of guidelines, illustrative examples, and finally the input state. The guidelines serve to handle edge cases and known failure modes, whereas the role introduction typically remains unchanged across different prompt iterations.

As discussed in Section **??**, clear tool descriptions are essential to enable proper tool usage. In CIExMAS, tools are embedded directly in the prompt. Each is presented with an identifier, a name, and a descriptive text. This description also defines the expected instruction format. This is particularly relevant for tools such as the **URI!** retrieval component, where specific search and filter modes must be communicated to the agent through the prompt.

In summary, prompt engineering in CIExMAS is realized as a manual, iterative process. The design principles are aligned with best practices observed in prior research, emphasizing human-readability, structured layout, and the inclusion of well-selected examples—either for tool usage or for solving the core extraction tasks.

4.4 Error Handling

As **LLM!**s are designed to generate unstructured text, problems can arise when a specific output format or the use of predefined tags is expected. One solution lies in leveraging the reasoning and looping capabilities of agent systems to incorporate error messages into

the agentic flow. This allows agents to detect and potentially correct faulty behavior, particularly when error messages are well-defined and understandable.

As described in Section **??**, the consistent parsing strategy across CIExMAS is the use of XML tags to mark key elements in agent responses, such as the identifier of the next agent or the corresponding instruction. Therefore, each agent is prompted to return information in an XML-tagged format, which is subsequently parsed during execution. For example, the agent output is scanned for a `<goto>` tag to determine which agent or tool should be called next. Similarly, the `<instruction>` tag is used to extract task-specific input. These parsing routines are vulnerable to failures if tags are missing, malformed, or empty.

To manage such failures, all parsing steps in the agent execution loop are enclosed in `try-except` blocks. If an expected tag is not found or cannot be interpreted correctly, the system generates a specific error message describing the missing or malformed element. This error is then returned to the calling agent, allowing it to revise its output in the next reasoning step.

Similar mechanisms are implemented for tool inputs. If an agent provides a malformed instruction, such as an undefined search mode for the **URI!** retrieval tool, the tool parser triggers an exception and returns a descriptive error message back to the agent. This enables iterative corrections without terminating the execution.

Another source of errors involves incorrect or hallucinated agent outputs. As discussed in Sections **??** and **??**, dedicated tools are used to validate the semantic and syntactic correctness of extracted triples and **URI!**s. In the final execution step, all generated `Turtle` outputs are parsed using a `Turtle` parser to verify compliance with syntax rules. Errors at this stage include undefined prefixes, incorrect namespaces, or invalid characters. These issues are again transformed into structured error messages that can be interpreted by the system.

Through this layered error handling, CIExMAS can detect and respond to a broad range of issues—from missing XML tags to malformed `Turtle` syntax. Error messages are returned in a machine-readable format and reintegrated into the agent flow, enabling agents to revise and repeat failed steps. External issues such as context overflows or GPU memory errors remain outside the scope of this system-level handling. Nevertheless, within the agentic loop, this approach increases robustness and helps ensure that a valid output can be generated for each datapoint.

5 Evaluation

This chapter presents the evaluation of the effectiveness of **MAS!**s in solving the **cIE!** task. All architectures and configurations were implemented in Python and evaluated within a standardized setup. Section **??** outlines the core components of this setup, including the datasets, the integration of Wikidata, the language models used, and the supporting frameworks. Section **??** explains the evaluation metrics that serve as the foundation for comparing different configurations.

The presentation of the results are spread across two sections. Section **??** presents a benchmark comparison of the best-performing CIExMAS configuration against existing approaches such as the synthIE model (**?**) and GenIE (**?**), based on a test split of the synthIE dataset (**?**). Section **??**, in contrast, dives into the different CIExMAS configurations and highlights how individual architectural and tool-based iterations influenced performance. Finally, Section **??** synthesizes the results and provides an interpretation in the context of multi-agent design and knowledge graph incorporation.

5.1 Evaluation Setup

This section outlines the experimental setup used to evaluate the effectiveness of the proposed **MAS!** architectures for **cIE!**. The structure comprises four main components.

First, Section **??** introduces the datasets and comparison models used in the evaluation. It outlines which splits were selected and provides relevant context for how the evaluation is structured. Second, the process of accessing and interacting with the knowledge graph is described in Section **??**. This includes technical details regarding how Wikidata was queried or indexed to enable efficient retrieval of relevant information during agent execution.

Section **??** then presents the language models and providers employed throughout the experiments. These models form the foundation for agent behavior and reasoning across the evaluated architectures. Finally, Section **??** details the agent framework used to orchestrate interactions between agents and tools. This includes how prompts are assembled, how state transitions are handled, and how tools are invoked. Large parts of the execution and traceability of CIExMAS were monitored using the open-source platform *LangFuse*[1], which was integrated to support reproducibility and debugging.

5.1.1 Dataset and Comparison Models

The evaluation is based on the synthIE dataset, with one experiment conducted using the REBEL dataset. Both datasets, which are described in Section **??**, contain triples that

[1] Available at https://langfuse.com

are extracted from a given text. Each extracted triple contains unique identifiers from Wikidata for each part of the triple, which makes them ideal for **cIE!**. Each identifier can be resolved within the entity namespace `https://www.wikidata.org/entity/`.

Both datasets follow a similar structure. They are stored in JSON Lines format, which is a line-separated format consisting of objects that represent entities, relations, and triples. Each entity and relation, which can also be referred to as a property, is further decomposed into a `surfaceform` and a `uri`, the latter being a unique identifier. The `surfaceform` corresponds to the textual appearance of the entity or relation in the input text, while the `uri` denotes the local part of the Wikidata Qname. A Qname itself consists of a prefix (i.e., the namespace) and a local identifier (**?**). For example, the entity "June" is represented as `Q120` and can be resolved under the namespace `https://www.wikidata.org/entity/`. Each triple is composed of subject, property, and object, each following the same structure.

As outlined in Section **??**, two variants of the synthIE dataset exist: `synthIE-code` and `synthIE-text`. These versions differ in the `GPT-3.5` model used for text generation. Since the `synthIE-text` version demonstrated higher performance, it is exclusively used for the final evaluation in this work.

However, all evaluation configurations prior to Section **??** were developed using the `synthIE-code` variant. The larger amount of available data in this variant allowed for a more extensive development set, increasing the likelihood of identifying edge cases. Specifically, the training split of `synthIE-code` was used and sorted so that Document ID 1 appears first in the JSON Lines file.

Due to limitations in data quality and the practicality of developing on small example sets, the later stages of development were shifted to the validation split of `synthIE-text`. This split, which is also sorted by ID, contains the same underlying triples but with different textual phrasing. Since `synthIE-text` does not provide a training split, the validation set served as a basis for prompt iteration. Its higher text quality also facilitates the resolution of edge cases that may be difficult to detect in more ambiguous examples.

The final evaluation, applied to CIExMAS configurations and the comparison models, was conducted on the first 50 documents of the sorted test split of `synthIE-text`. This number was chosen due to limitations in computational resources, particularly relevant for architectures with frequent agent calls, such as the **SSA!** or **SSSA!**. The use of `synthIE-text` is further justified by the findings of **?**, who observed improved performance compared to `synthIE-code` without changes to the underlying triple set. Empirical testing during this work also confirmed a clearer alignment between text and triple content.

To illustrate the distinction between `synthIE-code` and `synthIE-text`, the first document from both datasets is shown in the following example. In the `synthIE-code` version, the sentence reads: *"'Groovin' Blue' is an album by Curtis Amy, released on Pacific Jazz Records."* In contrast, the `synthIE-text` version states: *"Groovin' Blue was released on Pacific Jazz Records and performed by Curtis Amy."* While both versions encode the same structured triplets, shown in Table **??**, the surface form differs. The `synthIE-text` version more clearly hints at the intended predicate, which is harder to identify in `synthIE-code`.

Subject	Property	Object
Groovin_Blue	record label	Pacific_Jazz_Records
Groovin_Blue	performer	Curtis_Amy

Table 5.1: Surfaceform Triplets – DocID: 0 – `synthIE-code` & `synthIE-text`

To benchmark the performance on the REBEL dataset as well, the first 50 samples from its sorted test split were evaluated. Since CIExMAS was primarily developed using the synthIE dataset, its generalization to REBEL was limited. In testing, none of the CIExMAS configurations achieved an F1 score above 0.1, which led to the exclusion of REBEL from further evaluation. The full test results are documented in Appendix **??**.

The synthIE dataset was ultimately used to evaluate both CIExMAS configurations and the comparison models. For the synthIE model, the pre-trained models from **?**[2] were used for comparison. These include the $synthIE_{T5\text{-}base}$, $synthIE_{T5\text{-}base\text{-}SC}$, and $synthIE_{T5\text{-}large}$ models, as well as the $GenIE_{T5\text{-}base}$ and $GenIE_{T5\text{-}base\text{-}SC}$ models. The abbreviation `SC` refers to the subject-collapsed variant, in which all triples sharing the same subject are grouped. All comparison models were evaluated on the same 50 documents from `synthIE-text` and use the `Flan-T5` language model, following the setup by **?**, although the original GenIE implementation in **?** was based on `BART`.

In summary, the sorted 50-sample test split of the `synthIE-text` dataset serves as the primary benchmark for both CIExMAS and the comparison models. This setup enables the evaluation of different configurations and allows a direct comparison of their respective performance levels.

5.1.2 Knowledge Graph Access

As outlined in multiple sections concerning knowledge graph incorporation tools (cf. Section **??**), it is necessary to provide the agent system with direct access to Wikidata as the underlying knowledge graph. To achieve lower latency, this is realized through a setup consisting of Qdrant as a vector database, Redis as a key-value store, and Apache Jena Fuseki as a local **SPARQL!** endpoint. All components are hosted on a virtual private server, which provides reduced latency compared to direct usage of Wikidata's public **SPARQL!** endpoint. Nonetheless, direct access to the public **SPARQL!** endpoint is required in the *network traversal tool* and the *turtle-to-label tool* to retrieve specific restrictions.

The ***URI!*** *retrieval tool* (cf. Section **??**) requires a search index to generate relevant mappings. In the present evaluation, each search index corresponds to one collection within the hosted Qdrant vector storage. It is important to note that a database supporting similarity search was necessary to enable the described approach.

To populate the Qdrant collections, relevant entity and property data were precollected from all datasets used during development and benchmarking. Specifically,

[2] Available at https://github.com/epfl-dlab/SynthIE

the first 50 samples from the sorted validation and test splits of the `synthIE-text` dataset, the sorted train and test splits of the `synthIE-code` dataset, and the sorted test split of the REBEL dataset (cf. Section **??**) were processed. From these samples, all referenced entities and properties were extracted and indexed into the Wikidata-based collections.

This approach ensures that the agent system can resolve all required **URI!**s regardless of which dataset is currently being used. Furthermore, it reflects the objective of preparing CIExMAS for practical deployment scenarios, where the system must operate over the entire knowledge graph. At the same time, it enables efficient switching between datasets in both development and benchmarking phases without the need to rebuild or reindex the knowledge base.

As previously stated, semantic validation requires a loop-free hierarchy graph of Wikidata's entities and properties. In this context, a loop-free hierarchy graph refers to a directed acyclic graph structure derived from the subclass and subproperty relations of Wikidata, where cycles have been removed to ensure consistency during hierarchical reasoning. To make these graphs accessible across multiple machines, an instance of Apache Jena Fuseki was hosted containing a class and a property hierarchy. Due to the reduced size of these graphs, this approach also improved performance compared to direct queries to the Wikidata **SPARQL!** endpoint.

The Redis key-value store is employed to accelerate data processing pipelines and debugging. Specifically, Redis stores which data from Wikidata have already been inserted into the Qdrant database. Additionally, it caches labels, descriptions, and types of elements from Wikidata, enabling faster label display. This becomes particularly relevant when benchmarking results are inspected in detail and each **URI!** must be transformed into a human-readable label.

In summary, the use of a vector store is essential for addressing the **cIE!** task within the CIExMAS framework. Moreover, components such as the triple store and key-value store facilitate distributed access to information and contribute to reduced latency. Together, they ensure that the overhead for preparing and executing the agent system remains low.

5.1.3 Language Models

The choice of a language model is a complex task in the context of a **MAS!**. For such a task, the **LLM!** must be able to follow prompts, plan, reason about tasks, extract `triples`, and understand how knowledge graph `triples` are typically constructed. Furthermore, from the start of the CIExMAS project until its conclusion, multiple models were released. These models had not previously been tested on approaches related to CIExMAS, which meant that the evaluation began from a foundational level.

An additional limitation concerned the benchmarking system and the providers used for development. All benchmarks were executed on two Nvidia A100 GPUs with 48 GB VRAM each. The **LLM!** ran using *vLLM*[3], while the embedding model ran using

[3] Available under https://docs.vllm.ai/en/latest/

Ollama[4]. This setup imposed a practical parameter limit of approximately 70 billion parameters for a 4-bit quantised **LLM!**, based on empirical observation, though the exact limit varies between models. Development and evaluation focused on open-source models to ensure traceability in CIExMAS and provide greater flexibility in model use. This also enabled running evaluations on local servers rather than consuming credits at AI vendors for token-intensive benchmarking.

During the course of the work, `Llama-3.3-70B` consistently demonstrated the strongest performance in the required capabilities, particularly in executing a **MAS!** and following instructions. For this reason, all CIExMAS benchmarks, with the exception of the model variation tests, were conducted using a 4-bit AWQ version of `Llama-3.3-70B`, namely `kosbu/Llama-3.3-70B-Instruct-AWQ`. The development process was conducted on unquantised versions of the models using the providers *SambaNova Cloud*[5] and *Cerebras*[6], which offered significantly faster inference. However, both providers imposed a context length limit of 8192 tokens during the course of the work, which is why a maximum context of 8192 tokens was used throughout this thesis.

The achieved performance of `Llama-3.3-70B` was in line with the benchmarks reported by **?**, showing results comparable to `GPT-4o` and `Claude-3.5-Sonnet`. These models were considered state-of-the-art at the time but required more parameters and were not open source. The `Llama-3` series follows a similar architecture to `GPT` models, yet benefits from training on a large volume of high-quality data. This approach allowed the parameter count to be reduced to 405 billion for `Llama-3` and 70 billion for `Llama-3.3` while maintaining competitive performance (**??**).

More recently, Google released `Gemma-3`, an open-source model ranked among the best in the human-evaluated LMSys Chatbot Arena (**?**), while maintaining a comparatively low 27 billion parameters (**?**). Nevertheless, in the experiments conducted for this work, `Gemma-3` scored slightly lower than `Llama-3.3-70B` in benchmarks such as *MMLU-Pro* (see Section **??**) and exhibited weaknesses in instruction following and logical agent flow execution.

Newer model architectures, such as *Mixture-of-Experts* and reasoning models, emerged with the release of `DeepSeek-R1`, `Kimi-K2` and `Llama-4` (**???**). These models have achieved high benchmark scores while maintaining lower parameter counts compared to top-end state-of-the-art models. In some cases, they outperformed larger models. However, their reasoning capability involves iterative processing over the **LLM!**, which results in increased token usage and higher resource demands. Due to hardware and time limitations, these models were not employed for development.

The evaluation of model variations (see Section **??**) included the unquantised `Llama-3.3-70B` via *SambaNova Cloud*, `Llama-4-Maverick` via *Groq*[7], `Kimi-K2-Instruct` also via *Groq*, and the unquantised `Gemma-3` via *DeepInfra*[8]. All prompts were optimised for `Llama-3.3-70B`. When a different model is used, prompt sensitivity (see Section **??**) can cause lower

[4] Available under https://ollama.com
[5] https://sambanova.ai
[6] https://cerebras.net
[7] https://groq.com
[8] https://deepinfra.com

performance compared to results obtained with a prompt specifically adapted to that model. The choice of `Llama-4-Maverick` and `Kimi-K2-Instruct` was based on their strong benchmark performance, while the hardware requirements of `Llama-4-Maverick` exceeded the combined 96 GB VRAM of the available GPUs, necessitating its evaluation via *Groq*. `Kimi-K2-Instruct` was selected for its architecture and reasoning capabilities. `Gemma-3` was chosen as a recent high-performing open-source alternative, with *DeepInfra* enabling flexible and cost-efficient testing of the unquantised model. The `Llama-3.3-70B` model was also used to compare the quantised version applied in the main evaluation with its unquantised counterpart, in addition to serving as a baseline in the model variation analysis.

Overall, `Llama-3.3-70B` proved ideal for rapid development using *SambaNova Cloud* in its unquantised form, while the main evaluation used the quantised model. This model represented a significant improvement over the **LM!** used by **?**, and its advancements were integrated into CIExMAS. To capture more recent developments, other models were also included in the evaluation.

5.1.4 Agentic Framework

Evaluating an agentic system requires an appropriate framework. In the case of CIExMAS, this framework needed to support the specific setup in which the **LLM!** and the embedding model run on *vLLM* and *Ollama*, while the **LLM!**s in development are accessed via cloud providers. In addition, it had to meet the core requirements of the project: enabling the construction of agents and tools, and managing a custom state for different architectures (see Section **??** and Chapter **??**).

A framework that fulfilled these requirements and was flexible enough for extension was *LangGraph*[9], as already noted in Section **??**. *LangGraph* primarily implements flow-related features and, in the form applied here, does not interact directly with the **LLM!**s. This separation kept access to the **LLM!** and the embedding model fully flexible. The integration of *LangGraph* with *LangChain*[10] provided additional advantages, such as the ability to use *Langfuse*[11] for seamless monitoring at the agent overview level. This made it possible to realise the traceability capabilities described in Section **??**. Another strength of *LangChain* is the wide availability of adapters for different **LLM!** providers. Furthermore, all three frameworks are open-source projects with large, active communities, which proved valuable for the development of CIExMAS.

Although many other agentic and **LLM!** frameworks could have been used, the combination of *LangChain*, *LangGraph*, and *Langfuse* was ultimately chosen because it met all technical requirements and is widely adopted in the community. Together, these tools enabled the application of best practices in agent design for the implementation of CIExMAS.

[9] https://www.langchain.com/langgraph
[10] https://www.langchain.com
[11] https://langfuse.com

5.2 Evaluation Metrics

Evaluation metrics are essential both for comparing CIExMAS with other models and for identifying optimisation potential within CIExMAS. The evaluation builds on the metric definitions of **?**, which specify how to classify predicted `triples` as true or false positives. Their approach, referred to here as *hard matching*, classifies a predicted `triple` as a true positive if, within the same document, there exists a gold standard triple whose subject, property, and object each match exactly by **URI!**. Predicted triples without such a correspondence are counted as false positives, while gold standard triples without a matching prediction are counted as false negatives. Based on these counts, standard measures such as F_1-score, precision, and recall can be computed.

While **?** define their metrics only at the triple level, CIExMAS extends the evaluation to four additional categories: Subject, Property, Object, and Entity. For each category, the set of all distinct **URI!**s appearing in the respective role within a document is considered. This broader scope provides finer-grained insight into which components are handled well and which require improvement. Here, *Entity* denotes the union of the subject and object sets in a document, regardless of their position in the triple. The calculation of precision, recall, and F_1 for these categories follows the same procedure as for triples.

In addition to exact matches, this work introduces *soft matches* to capture cases where predictions are close to the gold standard but not identical. The motivation arose from the observation that a prediction might not use the exact expected property **URI!**, yet still choose one that is ontologically related in the Wikidata hierarchy. Without accounting for such relationships, evaluation would classify these as entirely wrong, even though they may be only one level of abstraction away from the expected result.

Two categories of soft matches are distinguished:

- **Parental soft match:** The predicted property is a direct super-property of the expected property.
- **Related soft match:** The predicted property is either a super-property or a sub-property of the expected property.

Both types can be applied at the predicate or triple level. By recognising these relationships, the metrics can award partial credit for predictions that are semantically close to the target.

Soft matches are computed using a greedy search that iterates over all predicted and gold standard triples for a document. For each predicted–gold predicate pair, the algorithm checks their ontological relationship. If the pair is linked by a `parentPropertyOf` relation, it is counted as both a parental and a related soft match. If the pair is linked by any other super-/sub-property relation, it is counted as a related soft match only. This matching is performed in both directions, predicted to gold and gold to predicted, to ensure symmetric detection of true positives, false positives, and false negatives.

Because multiple predicted triples can match the same gold triple, naive counting could produce precision or recall values greater than 1. To avoid this, recall is defined as

the number of gold standard triples or properties detected, divided by the total number of gold standard triples or properties. Precision is defined analogously on the prediction side, following standard definitions exactly.

For each document, preliminary counts of true positives, false positives, and false negatives are computed for triples, subjects, properties, objects, and entities, including their respective soft match variants. Together with the corresponding `Turtle` representation, these results are stored in an Excel sheet for further processing into F_1-scores, precision, and recall. Because the metrics are computed per document, both macro and micro averages are possible. Unless otherwise stated, macro averages are reported, as they assign equal weight to each document regardless of its number of triples.

To illustrate the computation of the metrics, a compact example is constructed using three triples about Angela Merkel. Table **??** presents a set of illustrative "predicted" triples, which are constructed for this example, alongside a manually curated gold standard. The selection is deliberately minimal so that the complete evaluation process, from strict matching to the inclusion of soft matches, can be followed in detail.

Predicted triple	**Gold standard triple**
Angela Merkel ; *profession* ; Politician	Angela Merkel ; *profession* ; Politician
Angela Merkel ; *is member of* ; CDU	Angela Merkel ; *is member of political party* ; CDU
Angela Merkel ; *hobby* ; Tennis	Angela Merkel ; *lives in* ; Berlin

Table 5.2: Predicted versus gold standard triples for Angela Merkel.

Under hard matching, only the first triple, Angela Merkel ; *profession* ; Politician, is counted as a true positive. The second prediction is semantically close but still incorrect under hard matching. In the gold standard, the predicate is *is member of political party*, which the ontology declares as a sub-property of the more general *is member of*. The third prediction is a full mismatch because it shares neither predicate nor object with the gold standard.

Soft matching addresses such cases by incorporating ontological relationships between predicates. In the second row above, the predicted property is recognised as a *parental soft match* (direct super-property) and therefore also as a *related soft match* (any super-/sub-property link). This reclassification turns the second triple from a false negative into a partial success, improving both precision and recall in the triple and property categories.

Table **??** reports the impact of soft matches. Under hard matching, exactly one triple is correct, which yields low precision and recall in the triple and property categories. When parental and related matches are taken into account, the number of true positives increases in both categories and the F_1 score rises from 0.33 to 0.67. Subject recognition remains perfect. Object and entity detection also improve under the ontology-aware evaluation. In summary, this example shows that soft matches reduce overly harsh penalties on near-correct predictions and provide a more informative assessment of model

Category	TP	FP	FN	Precision	Recall	F_1
Triples	1	2	2	0.33	0.33	0.33
Triples – Parental	2	1	1	0.67	0.67	0.67
Triples – Related	2	1	1	0.67	0.67	0.67
Subjects	1	0	0	1.00	1.00	1.00
Properties	1	2	2	0.33	0.33	0.33
Properties – Parental	2	1	1	0.67	0.67	0.67
Properties – Related	2	1	1	0.67	0.67	0.67
Objects	2	1	1	0.67	0.67	0.67
Entities	3	1	1	0.75	0.75	0.75

Table 5.3: Per-category evaluation metrics for the example in Table **??**. "Parental" and "Related" rows include soft matches where the predicted predicate is, respectively, a super-property or any super-/sub-property of the expected predicate.

performance.

Taken together, the combination of fine-grained categories and soft matching extends traditional triple-level evaluation into a richer diagnostic tool. This methodology not only quantifies performance but also reveals specific strengths and weaknesses of **cIE!** approaches such as CIExMAS, guiding targeted improvements in model design and prompting strategies.

5.3 Evaluation Overview

Model	***Triples***				
	Precision	Recall	F_1	Parental F_1	Related F_1
synthIE$_{\text{T5-large}}$	**0.835**	**0.832**	**0.831**	**0.841**	**0.858**
synthIE$_{\text{T5-base-SC}}$	0.789	0.798	0.789	0.819	0.833
synthIE$_{\text{T5-base}}$	0.784	0.769	0.775	0.812	0.826
GenIE$_{\text{T5-base}}$	0.527	0.344	0.391	0.406	0.419
GenIE$_{\text{T5-base-SC}}$	0.527	0.320	0.372	0.391	0.404
CIExMAS	0.728	0.635	0.665	0.728	0.728

Table 5.4: Performance comparison of CIExMAS against state-of-the-art models on the synthIE dataset.

The best CIExMAS configuration on the 50-sample `synthIE-text` dataset adopts the network architecture described in Section **??**, and the exact setup is documented in Section **??**. Unless noted otherwise, scores are macro-averaged as defined in Section **??**.

With this configuration, CIExMAS achieves a relative F_1 improvement of approximately 62% over the strongest GenIE baseline reported by **?**, while remaining within about 25% of the leading synthIE models. All results are computed on the same set of dataset samples.

Models fine-tuned on the synthIE dataset generally outperform models without synthIE-specific fine-tuning. Subject-collapsed variants, introduced in Section **??**, do not provide a consistent advantage on this evaluation. On the triple metric, $\text{synthIE}_{\text{T5-base}}$ exceeds CIExMAS by roughly 16% in F_1, and $\text{synthIE}_{\text{T5-large}}$ adds a further margin of about 7% over the *T5-base* variants.

An analysis of precision and recall clarifies these differences. CIExMAS lags the synthIE models by at least eight percentage points in precision, which nevertheless indicates that many predicted triples coincide with the gold standard. The recall gap is larger: relative to $\text{synthIE}_{\text{T5-large}}$, recall is lower by about 31%, indicating that a substantial portion of expected triples remains undetected. This imbalance between precision and recall is pronounced in models without synthIE-specific fine-tuning, whereas $\text{synthIE}_{\text{T5-base-SC}}$ exhibits a more balanced profile.

Soft matching adds nuance to the comparison. When soft matching is applied rather than strict matching, CIExMAS improves by approximately 10%. For $\text{synthIE}_{\text{T5-large}}$, parental and related F_1 increase by 1% and 3%, respectively. As a result, the gap between CIExMAS and $\text{synthIE}_{\text{T5-large}}$ narrows to about 15% in parental F_1 and about 18% in related F_1. The differences to $\text{synthIE}_{\text{T5-base}}$ follow the same trend and shrink under soft matching, while the distance to the GenIE baselines widens.

Beyond triples, results for the categories Subject, Property, Object, and Entity are reported in the appendix. CIExMAS demonstrates state-of-the-art performance in subject and object detection, as detailed in Section **??** and in Tables **??** and **??**. Strong precision in these categories translates into competitive F_1 for undifferentiated entities. In contrast, GenIE achieves comparable precision but systematically lower recall, leading to lower F_1 scores in all categories.

With respect to property prediction, this category is the primary source of error of the tested **cIE!** approaches. In this area, $\text{synthIE}_{\text{T5-large}}$ achieves an F_1 which is 24% higher than the best CIExMAS configuration. Among the non-triple categories, properties yield the lowest scores across approaches. The triple metric, which requires the subject, property, and object to be correct simultaneously, remains the most demanding overall.

In summary, CIExMAS delivers near state-of-the-art performance on the **cIE!** task. It is approximately 62% relatively better than GenIE, whereas *synthIE models* remain about 25% ahead of CIExMAS. The dominant gap lies in recall for expected triples, a limitation shared with the non-synthIE dataset-fine-tuned GenIE baseline.

5.4 Evaluation Configurations

This section outlines the successive configurations explored during the iterative development of CIExMAS and summarises their outcomes. Detailed descriptions of each configuration, including the motivation for specific design choices, are provided in the

corresponding subsections.

Configuration	***Triples***					
	Prec.	Rec.	F_1	Par. F_1	Rel. F_1	Err.
Initial Baseline	0.294	0.380	0.322	0.358	0.358	0.200
Task Modularization (**SSA!**)	0.220	0.306	0.237	0.277	0.277	0.180
Error Incorporation (**SSA!**) + **PropEx!**	0.205	0.264	0.222	0.262	0.272	0.020
Error Incorporation (**SSA!**)	0.293	0.406	0.325	0.365	0.365	0.120
Error Incorporation (Baseline)	0.431	0.507	0.456	0.525	0.525	0.000
Agent System Simplification (**SSSA!**)	0.376	0.410	0.385	0.423	0.423	0.020
ReAct architecture	0.485	0.438	0.456	0.511	0.511	0.060
URI! Retrieval Filtering (**SSSA!**)	0.455	0.451	0.449	0.495	0.503	0.040
URI! Retrieval Filtering (ReAct)	0.473	0.419	0.440	0.484	0.491	0.020
Knowledge Graph Integration	0.390	0.362	0.373	0.438	0.448	0.240
Network architecture	**0.728**	**0.635**	**0.665**	**0.728**	**0.728**	**0.000**
Advanced Adaptation (ReAct)	0.667	0.484	0.545	0.584	0.584	0.020
Advanced Adaptation (Baseline)	0.538	0.578	0.552	0.619	0.627	0.000

Table 5.5: Performance comparison of CIExMAS development iterations on the `synthIE-text` dataset. The *Err.* column reports the fraction of runs per sample that resulted in an error and were therefore scored as zero in macro averages.

Table **??** reports the same set of metrics used in previous sections and adds an error rate. The error rate denotes the share of runs per sample that failed and were scored as zero in the macro averages. The results indicate that iteration does not guarantee monotonic improvement. Task modularisation and error incorporation under **SSA!** with and without **PropEx!** reduce the triple F_1 from 0.322 for the initial baseline to 0.237 and 0.222. Later configurations recover and surpass the baseline: error incorporation on the baseline and the *ReAct* architecture each reach $F_1 = 0.456$. Variants with **URI!** retrieval filtering using **SSSA!** or *ReAct* achieve F_1 between 0.440 and 0.449. The knowledge graph integration step yields $F_1 = 0.373$ and an error rate of 0.240.

A marked improvement occurs with the *network architecture* described in Section **??**. It achieves $F_1 = 0.665$ with an error rate of 0.000, which corresponds to a relative gain of approximately 107% over the initial baseline and about 46% over the best pre-network configuration at $F_1 = 0.456$. With various configurations, CIExMAS also exceeds the F_1 of the GenIE model *T5-base-SC* at 0.372 on the `synthIE-text` dataset. For readability,

detailed per-category results are presented in the appendix **??**.

5.4.1 Initial Baseline Setup

As a starting point, the baseline architecture followed the design described in Section **??**. The initial baseline implemented a simplified variant of this architecture. Prompts were intentionally minimal, particularly for the extraction agents, and the output of ***URI!*** *retrieval* was forwarded directly to the supervisor agent.

This configuration[12] executed a fixed sequence comprising entity and relation extraction, **URI!** mapping via the tool, and final result generation. Execution traces showed no iteration over `triples`. Although iterative refinement was supported by design, no iterations were triggered in the observed runs most likely due to the prompt of the supervisor agent.

At this early stage, the target output consisted of `QNames` rather than full **URI!**s. When the baseline was re-run within the current CIExMAS agent framework, `QNames` were parsed into `Turtle` strings. The evaluation logs revealed occasional outputs of full **URI!**s instead of `QNames`. Because this behaviour was unintended, such cases were counted as errors.

Already in this state, the supervisor agent exhibited flaws that motivated the later prompt architecture based on role description and guidelines. The supervisor sometimes acted without delegating to the extraction agents and, in particular, without calling ***URI!*** *retrieval*. Since Wikidata is likely represented in pretraining corpora (see Section **??** on Common Crawl), the **LLM!** can infer **URI!** patterns and entity mappings from internal knowledge. The supervisor therefore tended to extract `triples` and fill in **URI!**s from memory, which led to hallucinated identifiers.

The supervisor also omitted required outputs. In several runs no `goto` tag was produced to indicate the next agent, which triggered a `NoneType` error because the tag was missing. To ensure reliable control flow, the supervisor required strict rules. Based on this behaviour, the prompt design with role description and explicit guidelines, as outlined in Section **??**, was introduced. The guidelines instructed the supervisor to delegate tasks rather than solving them independently and to follow the specified procedure.

The extractors adopted the same prompt design, including explicit handling of edge cases. For the entity extractor, this concerned composite entities such as *2022 Winter Olympics*, which the dataset expects to be split into *[2022, 2022 Winter Olympics, Winter Olympics]*. Both extractors also had to be guided towards the expected output, since `synthIE-code`, which was used for the development of this configuration, frequently required information that is not directly visible in the text. For example, the compound name *Corfe Castle railway station* is expected to yield the triple *(Corfe Castle railway station; named after; Corfe Castle)*.

With these adjustments, the initial baseline attained F_1 scores close to the GenIE models and even exceeded their recall. Assuming that the observed results on 80% of

[12] Snapshot available at: https://github.com/maxlautenbach/CIExMAS/blob/b75556c727e208f048e00caa327778d80cfc9934/baseline.ipynb

the data extend to the remaining samples, the baseline would have achieved an estimated F_1 of 0.402, which would surpass both GenIE models.

The baseline showed weaknesses in subjects and objects, with $F_1 = 0.646$ and $F_1 = 0.665$, respectively, and higher recall than precision. The principal limitation, as in all configurations, lay in property detection and thus constrained the overall F_1. The baseline missed many hard matches, which is reflected in a property-level parental F_1 that is 22% higher (relative) than the hard-matched F_1, see Section **??** for definitions.

In sum, the initial baseline already demonstrated the potential of an agentic approach. Its scores approached those of comparison models with similar prerequisites that were not fine-tuned on the synthIE dataset, while the identified shortcomings provided a clear direction for further development. The baseline architecture was therefore extended in subsequent iterations.

5.4.2 Modularisation of Supervisor Agent Tasks

The supervisor agent in the initial baseline setup exhibited significant shortcomings that could not be remedied by prompt engineering alone. Execution traces revealed a predominantly linear workflow, which motivated the modularisation of the supervisor's responsibilities into the **SSA!**[13].

In this configuration, the supervisor's functions were divided into four specialised agents: *planner*, *agent instructor*, *result checker* and *result formatter*. Furthermore, the ***URI!*** *retrieval* tool was extended with an **LLM!**-based summary of the retrieved results. This design made the overall process more transparent and controllable, as each step could be instructed with a clearly defined scope. Planning was elevated to an explicit and separate stage, with the planner instructed to adapt as necessary. Unlike the initial baseline, the **SSA!** produced `Turtle` string outputs to match CIExMAS's real-world integration requirements, which is a constraint applied to all subsequent configurations.

To give the agentic system greater flexibility in searching for **URI!**s, two retrieval modes were introduced. The *label mode* performed similarity search against the `rdfs:label` of entities and properties, while the *description mode* prompted the system to generate a description of the target entity or property and used this to retrieve suitable candidates.

Despite these refinements, the first **SSA!** implementation showed a marked decline in triple-level performance, achieving $F_1 = 0.237$, which is approximately 74% relative to the initial baseline's score. Most errors arose from context-length limitations. At the same time, the failure rate decreased slightly to 18% of executions, indicating a somewhat more stable basis for further optimisation.

The modular structure led to an increase in iteration over the **cIE!** task. For example, the result checker once produced the following instruction: "To further refine the extraction, consider calling the **URI!** Detection Agent to find associated entities and relations in the Knowledge Graph for the extracted entities "Bluegogo", "bicycle-sharing system", and "bankruptcy". This step can help validate the extracted information and potentially identify additional relationships." Such behaviour was absent in the initial

[13] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/tree/ad061f1fd93d8e22ca9046769580e46e8ff2365d

baseline. Nevertheless, some runs revealed degenerate loops in which the same entities were revisited without producing additional value.

The intended benefit of the two retrieval modes was not fully realised. For instance, the agent instructor issued the search terms "`Bluegogo[LABEL]`, `bicycle-sharing system[LABEL]`, `bankruptcy[LABEL]`, `a product or material produced[DESCR]`, `history[DESCR]`", showing a tendency to apply the description mode mainly to properties and the label mode mainly to entities.

In other evaluation categories, the **SSA!** also underperformed. The most pronounced gap occurred in property detection, while entity detection showed the smallest shortfall. Moreover, the snapshot revealed a considerable disparity between soft and hard matches, particularly for properties, with the soft-matched F_1 being 30% higher (relative) than the hard-matched F_1.

Overall, the first **SSA!** implementation exposed additional failure modes and demonstrated how errors can propagate, even when progressing through the **cIE!** task in a more structured manner. The increased complexity introduced new limitations related to context length, without exceeding baseline performance. Nonetheless, it provided clear priorities for subsequent development steps aimed at matching and ultimately surpassing the baseline.

5.4.3 Error Incorporation and Agent Engineering

Both the initial baseline configuration and the first **SSA!** implementation showed notable potential based on their error rates. However, several refinements were required to improve the **SSA!**'s performance over the previous configuration. These refinements included the integration of more effective feedback mechanisms, referred to as *Error Incorporation*, and the addition of a dedicated property extractor agent (**PropEx!**) to the **SSA!**[14]. In parallel, the baseline architecture was extended with Error Incorporation and with a ***URI!*** *retrieval* summary, mirroring the improvements introduced in the **SSA!**.

As outlined earlier, looping and missing outputs were significant issues in the first **SSA!** implementation. These loops arose primarily when an agent was called without sufficient instructions to produce a response or returned an indecisive answer. In addition to context-length overflows, missing `goto` tags contributed to the error rate. From this configuration onwards, explicit error messages were incorporated into the agent flow. If a `goto` tag was missing, the system re-invoked the same agent until a valid `goto` was provided, subject to a global recursion limit. Instructions were also validated to ensure that the specified agent existed and that the instruction was not empty. No dedicated semantic check was performed; instead, agents themselves could detect implausible inputs and return corresponding error responses. Context overflows were addressed through prompt design, but agents were not given the capability to alter the context autonomously.

To address the **SSA!**'s persistent weakness in property extraction, similar to the baseline's earlier shortcomings, the **PropEx!** agent was introduced. This agent followed the

[14] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/tree/38c72a951ad513bb980b40f432b1958b796d7e4b

same prompt structure as the entity and triple extractors and was tasked with extracting properties directly from the input text. In practice, **PropEx!** generated a greater number of property candidates without improving their overall quality, which degraded triple-level F_1 while still yielding high scores in other categories.

The Error Incorporation mechanism was also implemented in the baseline architecture, along with the adoption of ***URI!*** *retrieval* summaries. As with the **SSA!**, the baseline was updated to produce `Turtle` strings instead of `QNames`.

These updates reduced error rates and improved performance relative to the first **SSA!** implementation. The updated **SSA!** without `PropEx` achieved a triple-level F_1 of 0.325, while the updated baseline reached $F_1 = 0.456$, surpassing the initial baseline. By contrast, the **SSA!** with `PropEx` scored only $F_1 = 0.222$—32% lower (relative) than the updated **SSA!** without this agent—despite reducing the error rate to 2%. The baseline architecture with Error Incorporation recorded no errors across the evaluation set, enabling it to exceed the potential F_1 of the initial baseline. The prompts for all configurations were adapted to support the added functionality.

Although triple extraction performance declined with **PropEx!**, all configurations improved their F_1 scores for subject, object and entity extraction. Notably, the **SSA!** with **PropEx!** outperformed the initial baseline in subject and entity extraction and matched it in object extraction. Its main weakness remained property extraction, a limitation amplified by the additional agent. The updated baseline, by contrast, outperformed both the initial baseline and the GenIE models in all individual categories.

With these changes, most remaining errors shifted from the agentic control flow to the extraction stage. A typical case is the sentence "Daniel Johannsen is a tenor.", which should yield the triple *(Daniel Johannsen; voice type; Tenor)*. The updated baseline instead extracted the property *occupation*, a semantically reasonable but contextually incorrect choice.

In summary, Error Incorporation functioned as intended: it reduced the error rate and eliminated most looping behaviours observed in earlier configurations. Nonetheless, overall results remained constrained by the system's limited ability to select contextually appropriate properties.

5.4.4 Agent System Simplification

The main issue in the **SSA!** remained the high incidence of context-length errors. Combined with the observed looping behaviour between the *result checker*, *planner* and *agent instructor* agents, this motivated the implementation of the **SSSA!**[15]. The key idea was *state simplification*, meaning that the overall state was split into simpler and more granular parts.

As described in Section **??**, the planning, instruction and result-checking tasks were recombined into a single *planner* agent, while the *result formatter* remained unchanged. Manual prompt engineering was applied to make the **SSSA!** operate effectively, and the new, more granular state information had to be parsed from the **LLM!**'s responses.

[15] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/tree/91e14e7561e2ed6ba85e944f8063aca813d173e6

This simplified structure, despite introducing a small amount of additional parsing complexity, yielded improvements over the **SSA!**. Most notably, the **SSSA!** reduced token usage by involving fewer agents. The central *planner* also exhibited parts of the behaviour previously handled by the *result checker*. For example, it stated: "The triple extraction agent has successfully formed additional triples from the extracted entities and predicates, but we should verify if the entities and predicates are correctly extracted and if the triples are accurate." It then instructed the *entity extractor*: "Re-examine the text to extract entities, focusing on locations such as villages, districts, and counties, and verify the accuracy of the previously extracted entities." This illustrates that the **SSSA!** retained the ability to iterate over extracted triples.

In overall performance, the **SSSA!** achieved a triple-level F_1 of 0.385, the second-highest score among all configurations to this point. The error rate dropped to 2% of all runs, entirely attributable to a context-limit overflow, which created the potential for even higher scores. By linear extrapolation, the updated **SSA!** with its 12% error rate could have reached a triple-level F_1 of 0.369, while the **SSSA!** could have achieved 0.393, indicating a modest but measurable advantage for the **SSSA!**.

The **SSSA!** outperformed both the **SSA!** and the initial baseline across all evaluation categories. However, traces also revealed a characteristic trade-off. The *planner* agent was designed to maximise recall by extracting every possible triple from a sentence, which in some cases reduced precision. For example, from the sentence ""No More" by Jamelia is a Baroque pop single. It is performed by Jamelia.", the **SSSA!** generated triples stating that *No More* was an instance of rock music, Internet Archive or Daemosan station. That alternative classifications were not supported by the given context.

In summary, the **SSSA!** preserved the **SSA!**'s iterative triple-extraction capability while significantly reducing the context length, thereby lowering the error rate and improving the triple-level F_1. It also delivered better results in subject, property, object and entity extraction. Nonetheless, the recall-focused design occasionally introduced hallucinated triples, which negatively affected precision and overall scores.

5.4.5 ReAct Architecture

In contrast to the **SSA!** and **SSSA!**, the ReAct architecture took a radically simplified approach. It embedded the original ***URI!*** *retrieval* tool from the initial baseline directly into a single ReAct agent, thereby consolidating the entire agentic system into one component[16]. This design was intended both to test a joint-optimisation strategy and to explore the most minimal form of an **LLM!**-driven agent. The single agent was responsible for all stages of **cIE!**, including triple extraction, planning, **URI!** mapping and final output generation. In addition, it was used to evaluate how many tasks could be handled effectively within one model.

With this configuration, the ReAct agent matched the triple-level F_1 of the updated baseline with Error Incorporation, scoring $F_1 = 0.456$. It also reduced token consumption by nearly 25% compared with that baseline, using 453,000 tokens instead of 606,000

[16] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/tree/91e14e7561e2ed6ba85e944f8063aca813d173e6

in the benchmark setting. The single-agent design simplified tuning by concentrating adjustments in one place, but also introduced challenges: the `Llama-3.3-70B` model exhibited limitations in instruction following, making it harder to steer reliably towards the target behaviour.

Across evaluation categories, the ReAct agent displayed a markedly higher precision than recall, the opposite balance to earlier configurations, indicating a focus on fewer but more accurate triples. Unlike the **SSSA!**, it generated fewer triples overall. It also achieved a new high score among all previous configurations for subject extraction, while matching the **SSSA!**'s performance in the other categories. The error rate increased slightly to 6% of all runs, again due solely to context-limit overflows. Based on linear extrapolation, this suggests a potential triple-level F_1 of 0.485, which is 6.4% higher (relative) than the achieved score.

A persistent limiting factor, shared with all earlier models, was the lack of contextual sensitivity in property selection. As in previous examples, the sentence "Daniel Johannsen is a tenor." should yield the triple *(Daniel Johannsen; voice type; Tenor)*. The ReAct agent instead produced the property *instance of*, which in correct usage would denote *instance of human*. This misclassification illustrates both an inaccurate property choice and an incomplete application of the intended predicate semantics. Furthermore, while the ***URI!** retrieval* tool likely retrieved correct **URI!**s for subjects and objects, it often failed to retrieve correct **URI!**s for properties.

In summary, combining the ***URI!** retrieval* tool with a ReAct agent demonstrated strong potential. The joint-optimisation approach achieved near-updated-baseline performance in triple-level F_1, which remains the best result among all configurations tested so far, while improving efficiency and reducing system complexity. This minimalistic approach could be extended with additional tools and specialised strategies to further improve results.

5.4.6 URI! Retrieval Filtering

As outlined in Section **??** for the first **SSA!** implementation, one issue with **URI!** search terms was the suboptimal use of search modes: the label mode was applied almost exclusively to entities, while the description mode was used for properties. To give the agent more flexibility in applying search modes for different purposes, an additional filtering mechanism was introduced. This allowed the agent to specify whether the search should target entities or properties, independently of whether it was run in label or description mode[17].

The filtering mode worked by adding a keyword suffix to the search term. For example, `[LABEL-Q]` triggered a similarity search on the `rdfs:label` field restricted to entities, while `[LABEL-P]` restricted the search to properties. This mechanism was implemented in both the ReAct architecture and the **SSSA!**, as these had shown the most promising results in earlier experiments.

[17] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/commit/35ff8fc370103a9453aaf7babb8e62f576fd8650

The adaptation yielded positive results for the **SSSA!**, while the ReAct architecture achieved results similar to its previous version. The updated **SSSA!** recorded a slightly lower error rate of 4% and achieved a triple-level F_1 of 0.449, which is 16% higher (relative) than the original **SSSA!** implementation. The updated ReAct agent achieved a 0% error rate but saw its triple-level F_1 decrease from 0.456 to 0.440. As a result, the **SSSA!**, ReAct agent and updated baseline implementations scored the same score.

Looking at the evaluation categories individually, the ReAct agent with **URI!** filtering reached top or near-top scores among all prior models in subject and entity extraction, with $F_1 = 0.885$ and $F_1 = 0.906$, respectively. In the property and object categories, it was outperformed by the baseline architecture with error incorporation. The updated **SSSA!** also came close to the ReAct agent in these metrics, which explains its improved triple-level F_1. Property-level F_1 increased by 5% for the **SSSA!** and by 6% for the ReAct agent, suggesting that while filtering may have supported property retrieval, the effect was not large enough to substantially raise triple-level performance.

Since subject and entity extraction are already close to perfect in these configurations, the main limiting factors remain property extraction and the merging of all components into correct triples. Trace analysis showed that **URI!** filtering was used, but sometimes in unexpected ways. For example, in the sentence "Bluegogo was a bicycle-sharing system", the updated ReAct agent searched for `bicycle-sharing system[LABEL-P]`, which filters the search to *properties*. As a result, the agent did not retrieve the entity and failed to detect it as such. Nevertheless, most extractions demonstrated a higher degree of control in **URI!** retrieval. Furthermore, in both the ReAct architecture and the **SSSA!**, the gap between parental and hard-matched F_1 exceeded 10% (relative) for properties and for triples overall, highlighting substantial headroom for improvement through more accurate matching.

In summary, **URI!** retrieval filtering has clear potential to improve triple accuracy. The ReAct agent achieved near-perfect F_1 scores for subjects and entities, and the **SSSA!** delivered slightly lower but still competitive results. However, the consistently high category-level scores did not translate into significant triple-level gains, and the filtering mechanism has yet to produce a marked effect on property or triple F_1.

5.4.7 Knowledge Graph Integration

In previous configurations, a dominant issue was the considerable gap between parental and hard-matched F_1 scores, particularly in properties. The parental F_1 alone accounted for most of this gap, indicating that the extraction results tended to be overly general. One idea to address this problem and narrow the gap was to integrate the underlying knowledge graph directly into the agentic system[18].

Property extraction, whether performed by a dedicated agent or not, showed the largest disparity, with F_1 scores up to 41.2% lower (relative) than in the entity category. The expectation was that, if the extracted properties were correct but only soft

[18] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS/tree/9ae8a7b8b5d5aa54c0875684a0f42a41cf740e04

matches, their corresponding exact matches could be found within the property hierarchy of Wikidata, as used for the synthIE dataset. In addition, this hierarchy could provide a semantic indication of the intended meaning of the original property.

In this configuration, the *network traversal* tool described in Section **??** was used to establish this connection. As this was an early implementation of knowledge graph integration, Wikidata was accessed directly and restricted to retrieving only direct subproperties and superproperties. This limitation served to avoid cycles in Wikidata's property graph and to prevent excessive context growth, since the large number of possible properties could easily overflow the available context window. Context overflows were already a known issue in earlier configurations.

Given the strong performance and simpler tuning process of the ReAct architecture, it was selected as the basis for testing knowledge graph integration with the network traversal tool. The tool was implemented as a recommended step in the prompt sequence towards producing the final result. For example, the agent produced the output: "Next, I will use the Network Traversal Search Tool to find super- and sub-properties for "located in" and "capital of" to ensure the best match for the context."

The results did not meet expectations. The triple-level F_1 decreased by 18% (relative) to 0.373, and all other evaluation categories also declined. Two effects stood out in the detailed results. First, although the scores in all categories were well below those of the best previous models, the triple-level F_1 was not as low as the category-level gaps might have suggested. Second, the error rate rose to 24% of all runs, caused mainly by context overflows when the agent attempted to use the ***URI!*** *retrieval* tool with multiple search terms and modes. Based on linear extrapolation, this error rate implies a potential triple-level F_1 of 0.490, which would have been the highest among all CIExMAS configurations tested.

The *message deletion* tool, described in Section **??**, was also tested in small-scale experiments but could not be successfully integrated. A central issue was that the agent sometimes deleted messages essential for downstream processing, such as ***URI!*** *retrieval* outputs. Without prior summarisation, this led to the loss of relevant context and degraded the final results. Consequently, the focus shifted to maintaining precise and concise context rather than pruning it dynamically.

The configuration also exhibited similar errors to previous setups. **URI!** mapping remained a weak point, and traces showed the agent often generating multiple alternative search terms for the same target. While intended to improve recall, this behaviour sometimes reduced the likelihood of finding the correct **URI!** because relevant results were not prioritised.

In summary, the knowledge graph integration experiment demonstrated that such tools do not automatically improve results. While the approach confirmed that knowledge graphs can be leveraged by agents, it also showed that the *network traversal* tool increases system complexity and context size, which limits prompt and context flexibility. For these reasons, the tool was not carried forward into subsequent configurations.

5.4.8 Network Agent Architecture

Across the development of CIExMAS, a range of insights emerged. More complex systems such as the **SSA!** did not necessarily produce better results. With substantial improvements, the **SSA!** and later the **SSSA!** could match the initial baseline's performance. In contrast, the most minimal approach, a single agent combined with the ***URI!*** *retrieval* tool, matched the *baseline architecture* while reducing complexity. Between these extremes, the baseline itself consistently delivered one of the best scores, using a simple pipeline process implemented in a supervisor architecture.

The *network architecture*, introduced in Section **??**, was designed to merge the most effective elements from earlier configurations into a system of three specialised agents[19]. Key insights from prior experiments shaped its design: the ReAct architecture showed that a single agent can extract complete triples in one step; the **SSA!** and **SSSA!** demonstrated that iterative result review often adds complexity without improving performance; and the network traversal tool highlighted the theoretical potential of knowledge graph integration, but without delivering practical benefits.

Additional lessons were equally important. Earlier approaches struggled with property retrieval and sometimes hallucinated **URI!**s. Knowledge graph integration did not resolve property semantics, and centralised planning in supervisor architectures frequently underperformed. Even with filtering modes, the ***URI!*** *retrieval* tool was occasionally misused, for example by performing description searches with label inputs. By contrast, **URI!** summaries, which were used in several approaches except the ReAct-based ones, produced more concise and relevant results. The **SSSA!** also demonstrated that simpler states and contexts reduce error rates.

Building on these findings, the *network architecture* introduced three specialised agents with the ability to control agentic flow: an *triple extraction* agent, a ***URI!*** *mapping & refinement* agent, and a *validation and output* agent. Inspired by the result reviewer's iteration behaviour, the *validation and output* agent was equipped with the *semantic validation* and *turtle-to-label* tools to incorporate semantic checks and reduce hallucinations. The *semantic validation* tool used a filtered, locally hosted Wikidata dump that was loop-free, in contrast to the network traversal tool.

Several prompt design features from earlier stages had not yet been fully exploited. In particular, **?**'s **ICL!** was applied to the triple extractor. Its prompt included three manually filtered samples from the `synthIE-text` validation split. The extractor was also instructed to output an entity type to support disambiguation during search, as the embeddings would encode semantic information. For prompt development, the *synthIE-text* dataset was now used explicitly for development, ensuring that optimisation and testing drew on representative task data.

For **URI!** search, the search and filter modes were simplified. The ***URI!*** *retrieval* tool accepted only entity and property filtering, both using `rdfs:label` for similarity. Property filtering gained an additional example-based mode, where the agent could input a triple and retrieve properties that semantically matched the intended meaning. The

[19] Snapshot of this configuration: https://github.com/maxlautenbach/CIExMAS

***URI!** mapping and refinement* agent was instructed to use both modes to find the most suitable property.

Qualitatively, the improvements performed as expected. The *validation and output* agent could return control to the ***URI!** mapping and refinement* agent when type restrictions were violated. For example, it produced the reasoning: "The semantic triple checking tool has identified a type restriction mismatch for the object of the triple, so the uri_mapping_and_refinement agent is called to refine the triple and resolve the mismatch."

Quantitatively, these changes delivered the highest CIExMAS score to date. The *network architecture* achieved a triple-level F_1 of 0.665, with precision exceeding recall, indicating that it produced fewer but more accurate triples. The error rate dropped to 0%, showing that the simplified context, containing all necessary information, was effective. Neither the updated baseline nor the improved ReAct agent from Section **??** came close to the *network architecture* in triple-level F_1. However, the soft-match F_1 remained about 10% higher (relative) than the hard-match F_1, primarily due to properties that were parent properties of the intended targets.

Category-level results were also notably strong: subject and entity extraction achieved F_1 scores above 0.900, object extraction reached 0.867, and precision in all categories was 0.93 or higher, indicating best-in-class or near-best-in-class performance. The most significant gain over previous models was in property extraction. The achieved property F_1 of 0.710 was 33% higher (relative) than the best previous result, achieved by the *ReAct* architecture with ***URI!** retrieval* filtering. This highlights how limiting property extraction had been and how much potential could be unlocked by improving it.

The *network architecture* also revealed long-standing issues in CIExMAS. **URI!** mapping remained a weak point. For example, the search term "took place in" produced:

> took place in — Search mode applied [P|X] — Applicable to [P1] → **URI!**: http://www.wikidata.org/entity/P2408, **URI!**-Label: set in period

The new semantic validation tool also had limitations. For example, the triple *(Vapiano; cuisine; Italian cuisine)* was rejected because it determined that the object type constraint was not satisfied, as Italian cuisine is not listed as an instance of cuisine in the filtered dataset used for the tool[20], even though Wikidata contains the relevant relationship.

In conclusion, the *network architecture* proved to be the most performant configuration. By combining a small number of capable agents, it merged the strengths of earlier designs: extracting all triples in one step, while retaining a validation stage that could reject erroneous triples or trigger iteration. Although not error-free, it delivered the highest overall performance and highlighted remaining opportunities for improvement.

[20] The underlying datasets for the semantic validation tool are available at: Wikidata class hierarchy https://github.com/maxlautenbach/CIExMAS/blob/main/infrastructure/classes.nt and Wikidata property hierarchy https://github.com/maxlautenbach/CIExMAS/blob/main/infrastructure/all_properties.ttl

5.4.9 Integration of Tool Improvements to ReAct and Baseline Architecture

The integration of validation tools and refined ***URI!*** *retrieval* modes had shown high potential in subject and object extraction, and especially in property extraction. These improvements were therefore adapted to simpler architectures. The *ReAct architecture* was a prime candidate for this experiment, as it had achieved high scores while using fewer tokens and being easier to optimise. In particular, the updated ***URI!*** *retrieval* search modes was applied to both the *ReAct* and *baseline* architectures, with the *ReAct* architecture additionally extended by the semantic validation tools from the *network architecture*.

Concretely, both implementations received the new example search mode, while the label and description modes were removed. The *baseline architecture* retained its original set of agents and tools, whereas the validation tools were added only to the *ReAct architecture*. In this case, the prompt was adapted to make use of the *semantic validation* and *turtle-to-label* tools.

The results placed both models between their previous implementations and the *network architecture*. The *ReAct* architecture achieved a triple-level F_1 of 0.545, while the updated *baseline* reached 0.552. These correspond to relative increases of nearly 20% over their earlier versions. Both retained their characteristic strengths: the *baseline* detected more gold-standard triples, whereas the *ReAct* approach produced a larger total number of correct triples.

In the category-level evaluation, the *ReAct* approach again revealed the general limitations in property extraction. With more properties detected, property F_1 of 0.616, other scores such as subject and object F_1 moved toward the middle of the CIExMAS range. Nevertheless, the *ReAct* architecture exceeded previous top scores in several categories. The updated *baseline* achieved best-in-class performance in object and entity detection, while also maintaining higher property scores than the *ReAct* approach.

Overall, these results highlight the potential that can be unlocked even in simpler, easier-to-maintain architectures by applying the same tools and ideas used for the *network architecture*. Both the *baseline* and *ReAct* configurations surpassed their previous versions. However, in all configurations, including these, ***URI!*** *retrieval* remained the limiting factor for property F_1, and by implication, for triple-level F_1.

5.4.10 LLM Variation

Building on the strong results of the `Llama 3.3 70B` model, which matched the performance of OpenAI's GPT-4 models from late 2023 (**?**) and was established as the primary model in Section **??**, it was a logical next step to investigate whether newer large language models could deliver further improvements. As outlined in Section **??**, Meta has since introduced the `Llama 4` family, led by `Llama 4 Maverick`, Moonshot AI has released `Kimi-K2`, and Google has introduced `Gemma-3`, all with strong results on the LMSys Chatbot Arena benchmark. Given their benchmark performance and architectural differences from `Llama 3.3 70B`, these models appeared promising candidates to potentially surpass it, motivating a direct comparison in the CIExMAS setting.

Model (Provider)	*Triples*					
	Prec.	Rec.	F1	Par. F1	Rel. F1	Err.
Llama 3.3 70B 4-bit AWQ (vLLM)	**0.728**	**0.635**	**0.665**	**0.728**	**0.728**	**0.000**
Llama 3.3 70B (SambaNova)	0.650	0.599	0.608	0.647	0.647	0.000
Llama 4 Maverick (Groq)	0.346	0.306	0.321	0.391	0.391	0.200
Kimi-K2 Instruct (Groq)	0.060	0.043	0.049	0.049	0.049	0.060

Table 5.6: Performance comparison of **LLM!** variations against the original `Llama 3.3 70B` model, evaluated on the synthIE dataset.

The evaluation used the *network architecture* configuration from Section **??**, with all tools and prompts unchanged. The selection of models followed the rationale in Section **??**: testing high-ranking newer models (`Llama 4 Maverick`, `Kimi-K2`, `Gemma-3`) alongside an unquantised `Llama 3.3 70B` to compare directly with the 4-bit quantised `Llama 3.3 70B` model. The alternative models ran via the Groq and SambaNova providers, since their parameter sizes exceeded the available infrastructure for local inference. `Gemma-3` was evaluated via *DeepInfra* for efficient and convenient testing of the unquantised model. For reference, an unquantised `Llama 3.3 70B` was also tested on *SambaNova Cloud*, complementing the quantised baseline used throughout CIExMAS.

Results in Table **??** show that all alternative models underperformed the quantised `Llama 3.3 70B` across every metric. `Kimi-K2` achieved a triple-level F_1 of only 0.049, which is 7.4% (relative) of the quantised `Llama 3.3 70B` score. `Llama 4 Maverick` performed better, with $F_1 = 0.321$, roughly matching the result of the *initial baseline* configuration on the quantised `Llama 3.3 70B`, but still 52% lower than the *network architecture* on the quantised version. Both models showed higher error rates, with `Llama 4 Maverick` at 20% and `Kimi-K2` at 6%. The unquantised `Llama 3.3 70B` scored 8.5% lower than its quantised counterpart. While this suggests a benefit from quantisation in this experiment, the effect is likely specific to the **cIE!** setup, the exact `Llama 3.3 70B` model variant, and the prompt–tool configuration used, rather than being a general property of quantisation.

Trace analysis explains these discrepancies. Many of `Llama 4 Maverick`'s errors stemmed from recursion limits caused by syntax incompatibilities with the *semantic validation* tool — for example, outputting triples such as "@prefix wd: <http://www.wikidata.org/entity/>.\nwd:P1546 wd:Q877781", where the unexpected sequence `\nwd` triggered unrecoverable syntax errors and repeated looping. In addition, the model often extracted incomplete or incorrect triples, reaching only 48% (relative) of the `Llama 3.3 70B` score even in successful runs.

`Kimi-K2` exhibited fewer outright errors but showed severe hallucination and poor tool usage. In most traces, the ***URI!*** *retrieval* tool was not called at all, despite being crucial for correct mapping. In its rare successful runs, the tool was used correctly, producing some valid triples. Its 6% error rate came from repeated loops through the *turtle-to-*

label tool, sometimes triggered by syntax mismatches (e.g., DBpedia **URI!**s) but often occurring despite correct input and output.

An additional experiment with `Gemma3`, a smaller yet competitive open-source model, failed for a different reason: the model ignored the required output format, producing JSON instead of the expected XML-tagged structure. This incompatibility with the CIExMAS processing pipeline led to the model being excluded from further evaluation.

In summary, none of the tested **LLM!**s improved over the 4-bit quantised `Llama 3.3 70B`. Several, including `Kimi-K2` and `Gemma3`, struggled to produce any valid triples. Given these results, along with higher inference costs and latency for larger models, the 4-bit quantised `Llama 3.3 70B` remained the model of choice for CIExMAS.

5.5 Discussion

- Probleme im Prozess (Problem für interne, externe validität) -¿ Welche Gefahren gibt es bei der Generalisierbarkeit
- Externe Validität -¿ Modell/Scope sehr eng -¿ nur 2 Modell
- Auf Validität nochmal mehr achten
- Risken vorher betrachten und wie denen begegnet werden kann

To investigate the effectiveness of **MAS!**s for the task of **cIE!** and to identify the most effective agentic architecture patterns, CIExMAS was developed through an iterative design process. This process evaluated supervisor, network, and single-agent configurations, along with modified variants such as a splitting supervisor and a simplified splitting supervisor. All configurations were compared against the GenIE and *synthIE models* (**??**), using the first 50 samples of the `synthIE-text` dataset.

Evaluation results indicate strong performance of the CIExMAS approach, achieving an overall F_1 score of 0.665. Among the tested configurations, the network architecture consistently delivered the highest scores, while the custom design with a splitting supervisor performed worst. On the same dataset, comparison models achieved up to 0.831 when fine-tuned, whereas CIExMAS relied solely on **ICL!**.

A more granular analysis revealed that CIExMAS outperformed all other approaches in subject, object, and entity extraction, reaching F_1 scores above 0.93 in each of these categories. Several CIExMAS configurations also achieved the top score in at least one category. The only notable weakness appeared in property extraction, where the best CIExMAS configuration lagged 19% behind the top-performing synthIE model.

These results must be interpreted in the context of prior work. **?**, for example, trained their **LM!** on the synthIE-code training split containing more than 1.8 million documents, while CIExMAS was developed using manual prompt engineering based on only five examples from the `synthIE-text` validation split. GenIE, another non-fine-tuned model, performed even worse than CIExMAS, reaching only 52% of synthIE's performance, compared to CIExMAS's 81%.

The design of CIExMAS aimed to remain dataset-agnostic. To familiarise the system with the expected triple format, prompt engineering was combined with **ICL!** rather than fine-tuning or extensive example-based adaptation. Notably, the initial baseline already produced competitive results compared to more complex configurations such as the **SSA!** or those incorporating knowledge graph integration. Some approaches faced challenges because the output specification required properties to be expressed in the Wikidata entity namespace. In contrast, the baseline's use of QNames rather than full **URI!**s simplified the output process.

It is also worth noting that neither GenIE nor *synthIE models* addressed the problem of **URI!** retrieval. Both systems were constrained by a fixed token set for output generation, avoiding hallucination by design. CIExMAS, in contrast, required hallucination prevention and output validation through dedicated agents to maintain result quality. This difference in design likely contributed to performance differences.

Taken together, the configuration comparison reinforces a recurring principle: for **cIE!**, simplicity should be prioritised over unnecessary complexity. As discussed in Section **??** and supported by **?**, a **MAS!** should be introduced only when a single agent can no longer reliably follow instructions.

The *baseline* and *ReAct architectures* proved easier to optimise than more complex, highly customised solutions such as the **SSA!** or **SSSA!**. CIExMAS initially adopted a multi-agent setup because the tested models demonstrated limited instruction-following capabilities. However, the **SSSA!**, and particularly its more elaborate variants, did not prove to be the right remedy. The most effective configuration ultimately emerged from systematically probing the capabilities of the chosen **LLM!**. The strong performance of the *ReAct* agent, combined with the iterative problem-solving approach seen in human reasoning and reflected in both **SSA!** and **SSSA!**, naturally converged in the network architecture.

Low instruction-following skills, even when supported by well-crafted prompts, placed a ceiling on the simplest configurations. Additional guidance within prompts did not resolve certain edge-case errors, making alternative solutions necessary. The inherent complexity of mapping triples to Wikidata **URI!**s was one of the main drivers for introducing a **MAS!**. Within this framework, the network configuration demonstrated the advantage of clearly scoped agents, distributing the operational overhead evenly across the system. This aligns well with the agent design principles outlined in Section **??**, which emphasise human-readable and concise agent roles.

Another design choice balancing simplicity against complexity concerned the **URI!** search mechanism. Experiments showed that more elaborate search and filtering modes did not lead to higher performance. In practice, the **LLM!** must be able to determine the appropriate search and filter mode, which was not consistently achieved. Search modes based on label or description fields were rarely used as intended. The most reliable approach remained a straightforward entity-and-property filter, optionally combined with a property example search. This outcome further underlines that **cIE!** requires only the minimum level of complexity necessary, with label-to-**URI!** mapping being one of the most persistent challenges.

The use of **LLM!**s introduced additional difficulties, particularly hallucination in

URI! mapping—even when the **URI!** retrieval tool was employed. **?** argue that hallucination is an inherent property of **LLM!**s. Nevertheless, the hallucination-prevention methods and verification steps integrated into CIExMAS proved effective, particularly in the network architecture and in refined versions of the ReAct and baseline setups. For example, the turtle-to-label tool reliably exposed hallucinated results, enabling **LLM!**-based detection and filtering.

In contrast, the baseline architecture relied solely on one supervisor, two extractors, and the **URI!** retrieval tool with an **LLM!**-generated summary—without advanced hallucination-prevention mechanisms. Even so, targeted prompt optimisation, error incorporation, and effective **URI!** search modes yielded strong results. This suggests that much of the performance improvement observed in the network and updated ReAct architectures can be attributed to changes in **URI!** retrieval strategies. Validation tools likely contributed additional gains by preventing nonsensical triples from entering the output, boosting precision.

The experiments with knowledge graph integration tools further indicate that data quality is critical. The network traversal tool, operating on the full Wikidata dataset, suffered from limitations in SPARQL query depth for type restriction checks, resulting in performance drops. By contrast, the semantic validation tool, built on a filtered and loop-free subset of Wikidata, improved results. For such tools to work efficiently, datasets must be both loop-free and fast to traverse; otherwise, queries risk becoming unacceptably slow or timing out.

A recurring weakness across all configurations was property extraction. While error incorporation and context-size reduction by design helped lower error rates, subject, object, and entity extraction were generally easier to optimise—often reaching consistent success rates when paired with the **URI!** retrieval tool. Property extraction remained challenging because properties in the source text often did not match their exact Wikidata labels. For example, the sentence "Daniel Johannsen is a Tenor" would typically lead to the extraction of *is a* as the property, which, when searched directly, often returns *wd:P361* (*part of*). To address this, agents were equipped with a full-triple search mode that used the complete sentence as context. However, traces and experiments showed that this approach recovered only a fraction of the missing properties.

Finally, the interpretation of these findings must consider the dataset constraints. Only 50 samples were used for testing, with even fewer for manual training. While this ensures comparability to the baseline models—since all were evaluated on the same subset—it may also have suppressed maximum achievable performance. **?** reported a macro F_1 score of 0.93 for triple extraction with their best synthIE model on the full dataset, suggesting that these 50 samples may be inherently more challenging. The decision to limit the sample size was driven by runtime constraints: for instance, the **SSA!** and **SSSA!** architectures required around six hours to process the 50 samples on a vLLM `Llama 3.3 70B` setup with 2×Nvidia A100 GPUs. Given the 12-hour total runtime limit, expanding the test set was not feasible.

The question of **URI!** mapping remains a relevant point for discussion. It is likely that the Wikidata knowledge graph was included in one or more datasets used to train `Llama 3.3 70B` and other models. If so, these models may have memorised **URI!**s and

their corresponding mappings, potentially providing an advantage in triple formation and **URI!** retrieval. At the same time, trace analysis revealed that outputs without any interaction with the **URI!** retrieval tool often contained hallucinated **URI!**s, which significantly degraded results. Another challenge arose from the Wikidata integration: properties were frequently produced in the `wdt:` namespace, which is technically correct for properties, but evaluation—particularly the detection of soft matches in the property hierarchy—required them to be expressed in the `wd:` namespace.

A similar uncertainty applies to the synthIE dataset itself. Parts of it may be present in the training data of various **LLM!**s, and examples were also used for **ICL!**. Given that CIExMAS performance is still far from perfect, any such overlap appears to have had little to no practical effect on extraction quality.

Most agent engineering was performed on the `synthIE-code` variant rather than the text version. Since `synthIE-code` generally has lower data quality, prompt tuning may have been biased towards edge cases specific to that version. For example, the code dataset contains the sentence "Herman Heijermans was born in the Netherlands.", whereas the text version reads "Herman Heijermans was a citizen of the Kingdom of the Netherlands." Both are intended to produce the triple *(Herman Heijermans; country of citizenship; Kingdom_of_the_Netherlands)*. In practice, the same models performed better on `synthIE-text`, likely because fewer edge cases needed explicit handling and prompts could be generalised more effectively.

The datasets used by the knowledge graph integration tools also varied. The network traversal tool relied on the full Wikidata dataset but was limited in **SPARQL!** query depth for type-restriction checks, which hurt performance. By contrast, the semantic validation tool was built on a filtered, loop-free subset of Wikidata, allowing unrestricted traversal depth and yielding performance gains. Since the dataset used for semantic validation was not part of the CIExMAS development process, it is not described in detail here. Nonetheless, these results suggest that network traversal might have performed better on a similarly optimised dataset. Due to time constraints, optimising the traversal tool was deprioritised in favour of integrating validation tools into the ReAct architecture and refining the search modes.

Prompt engineering was carried out entirely by hand, with occasional assistance from ChatGPT 4o and Cursor. Manual prompt design does not guarantee optimal results, and prompts had to be updated frequently to accommodate new tools and tool implementations. This process could inadvertently favour one architecture over another in F1-scores. However, the evaluation results show consistent architectural trends, suggesting that design elements such as search mode and state had a greater impact than prompt wording. Automated prompt optimisation was considered but deemed too resource-intensive for the expected benefit.

The same time and resource limitations that constrained sample size also limited model choice. As described in Section **??**, **LLM!**s with up to 70 billion parameters achieve near state-of-the-art results on benchmarks. Within CIExMAS, `Llama 3.3 70B` consistently outperformed more recent or ostensibly stronger models when applied to this task. **LLM!**s remain sensitive to prompt design and to expected output formatting, which, if mishandled, can sharply reduce performance. Modern models such as `Llama 4`

offer very large context windows—up to 10 million tokens—but CIExMAS was limited to 8,192 tokens due to provider and infrastructure constraints. While this restriction caused challenges during development, it also enforced precision in context design.

For the embedding component, only open-source models were considered. Two options—`nomic-embed-text` and `bge-m3`—were tested, with `bge-m3` showing slightly better results in small-scale experiments and ranking well in embedding benchmarks (**?**). More advanced models exist but were excluded due to higher memory requirements, later publication dates, or lack of availability on Ollama. As embeddings generally worked as intended, focus remained on **LLM!** selection and agent engineering.

Finally, CIExMAS was designed for multi-dataset compatibility and potential extension to generate new entities in a knowledge graph, rather than only relying on existing ones. Unlike synthIE, triples were not limited to predefined tokens. Multiple entities from various datasets and splits were imported into the vector database used for **URI!** retrieval. While this broader scope could negatively impact retrieval precision, it reflects a more realistic scenario—especially since neither synthIE nor REBEL were fully ingested. Using a combined vector store allowed quick switching between datasets without managing separate collections or dataset filters. It also introduced natural redundancy, enabling the retrieval mechanism to prioritise the most relevant **URI!**s according to search term and search mode.

Furthermore, CIExMAS was deliberately designed with a strong emphasis on generalisability to datasets beyond synthIE. Despite this, extensive evaluation was only carried out on a single dataset. Many configuration changes were already required to push performance within synthIE, and alternative datasets such as REBEL were ultimately rejected due to insufficient data quality. As a result, while the architecture is theoretically generalisable, this remains unverified in practice. The switch from `synthIE-code` to `synthIE-text` did not change the expected triples, only the input text format, and thus did not serve as a full cross-dataset validation.

The development process for CIExMAS should also be considered in its entirety. Iterative experimentation demonstrated that insights gained from one architecture could improve others. For example, experience with the ReAct agent informed the consolidation of extraction agents in the network architecture. However, the project did not strictly progress from simple to complex designs. Starting with the baseline architecture, the work moved directly to more complex approaches such as the **SSA!**, skipping a minimal single-agent configuration. Nonetheless, each iteration was based on empirical findings, and even complex configurations yielded lessons that improved the final system.

Some concepts were further refined before being discarded. The property extraction agent, which underperformed in the **SSA!**, was also tested in the **SSSA!** due to the persistent challenge of property mapping across all configurations. Tool development generally targeted specific weaknesses in the pipeline, and problematic areas like property extraction motivated repeated optimisation attempts. A more granular comparison of individual agent contributions could have improved understanding of their impact. In practice, CIExMAS was optimised as an integrated whole, rather than perfecting each step in isolation. This made training and tuning more efficient, as all agents and tools were aligned towards a common objective.

Overall, CIExMAS has proven that **cIE!** can be performed effectively using a multi-agent architecture. Achieving a triple-level F_1 of 0.665 without fine-tuning or heavy **ICL!** demonstrates competitive performance, surpassing systems such as GenIE. Although the development path was not always linear, the process revealed both the strengths and limitations of each configuration. A clear conclusion is that simplicity should be favoured over unnecessary complexity—yet in **cIE!**, a certain degree of complexity remains essential, especially when working with **LLM!**s whose capabilities in instruction-following and structured output are still imperfect.

6 Conclusion and Outlook

- Wo ist RQ1 beantwortet? Keine klare Antwort in Conclusion. -¿ Subject, Object, Entity gut, Properties nicht, mainly freeform extraction errors, die dann nicht mehr zugeordnet werden können
- RQ2 -¿ konkretisieren
- Vllt. Discussion noch besser in Conclusion zusammenfassen

This thesis developed CIExMAS to address two research questions: to what extent **LLM!**-based **MAS!**s can perform **cIE!** on unstructured text, and which agentic architectures (e.g., single-agent, supervisor-agent, or agent networks) perform best in terms of accuracy and robustness. An evaluation on a 50-sample subset of `synthIE-text` showed that the best CIExMAS configuration achieved a triple-level F_1 of 0.665. This corresponds to 80% (relative) of the performance of $\text{synthIE}_{\text{T5-large}}$, which was fine-tuned on the train split. By comparison, the model of **?**, trained on a different and lower-quality dataset, reached only $F_1 = 0.392$ for the same metric on the same samples.

More detailed analyses indicate that **MAS!**s, especially when combined with the chosen embedding-based similarity search for ***URI!*** *retrieval*, excel at extracting subjects, objects, and entities overall. Across configurations, however, property extraction remained the prevalent limitation. This bottleneck is also apparent for synthIE and GenIE models. Accordingly, the **LLM!**-based **MAS!**s tested here can already perform **cIE!** to a high extent, though not to a near-perfect level.

The explored architectures varied widely in number of agents and tools, state handling, and overall capabilities. A clear trajectory emerged: simple and precise designs, prompts, and tools tend to achieve the best results. Complex architectures require complex prompts and extensive optimisation, often inflate the context, and thereby increase error rates. In the thesis evaluations, more complex systems did not yield better results than simpler ones even when executions were error-free. Consequently, the *ReAct* and textitbaseline architectures ranked among the strongest approaches, surpassed only by the *network architecture*. The findings further show that combining **LLM!**s with ***URI!*** *retrieval* benefits from additional agents for **URI!** mapping and for validation. Hence, the most accurate and robust solution for **cIE!** was not the absolutely simplest design; rather, it was a streamlined network that introduced targeted specialisation where it paid off.

CIExMAS presents multiple **MAS!** architectures capable of processing **cIE!** tasks and thus contributes to the broader objective of generating knowledge graphs from unstructured text. More generally, it adds to current research on **LLM!**s for **IE!** by providing

a proof of concept for **MAS!** applied to **cIE!**, which can serve as a basis for further development.

In total, five architectural patterns were proposed for **cIE!**. Among these, the supervisor, ReAct, and network patterns proved most useful in testing. The study also catalogues advantages and disadvantages of different agents and tools employed for **cIE!**. In particular, knowledge graph incorporation and ***URI!** retrieval* via `rdfs:label` as well as property-example search were found to be beneficial.

Because **LLM!**s, sentence embeddings, and agentic AI evolve rapidly, the work cannot include every recent improvement. Moreover, **MAS!** development opens a broad optimisation space. The choice and configuration of agents depend heavily on design goals, and the solution space could not be explored exhaustively within the timeframe of a master's thesis. The focus was therefore placed on architectural patterns and knowledge graph incorporation.

This focus entails several limitations. First, prompt engineering was performed manually. Second, CIExMAS did not employ any fine-tuned models. Both constraints were driven by the limiation of a master thesis. In line with **?**, **ICL!** was preferred as a near-equivalent alternative to fine-tuning.

Finally, evaluation was conducted on 50 samples, which limits the precision of point estimates such as the exact F_1. Even so, the results indicate consistent behaviour across configurations. The generalisation objective, baked into all architectures, was not tested extensively; therefore, no firm claims can be made about the generalisation capability of CIExMAS or of the compared *synthIE models*.

In addition, as the field of **LLM!**s is rapidly evolving, driven by major technology companies, new models are released at high velocity (**????**). Consequently, models more capable than Llama 3.3 70B are already available. The same applies to the use of bge-m3 as the embedding model for **URI!** retrieval. Other embedding models, including larger and state-of-the-art variants, were tested but not exhaustively; none were found to consistently outperform the chosen configuration in this setting.

All aspects of creating CIExMAS, from the overall system architecture to the individual prompts for each agent, were developed manually. While automated design and optimisation approaches have demonstrated notable success, they typically require multiple **LLM!** calls and substantial computational resources. As this was not the focus of the present work, an iterative trial-and-error process was adopted instead.

The limitations encountered in this work suggest several directions for future research. The strong emphasis on generalisation, including the preference for **ICL!** over fine-tuning, should be validated on alternative datasets. Currently, REBEL and REDFM offer the closest comparability to the synthIE dataset (**??**), yet both lack either quality or suitable approaches for comparison. It may therefore be advisable to await the release of new, high-quality **cIE!** datasets. In addition, document-level datasets could be evaluated, and free-form triple extraction could be tested across multiple corpora to facilitate broader comparison with **IE!** approaches. Datasets without Wikidata as the underlying knowledge graph would also help to rule out potential bias from test data appearing in model training data.

Alongside dataset changes, the models used for agents and for **URI!** retrieval could

be replaced or adapted. Given that **LLM!**s are highly prompt-sensitive, it is plausible that other models, optimised with model-specific prompts, could achieve competitive or superior results on **cIE!** tasks. As new and more capable models emerge, they could be incorporated into various CIExMAS configurations.

Alternative runtime setups could also be explored. While Llama 3.3 70B supports context lengths of up to 128,000 tokens (**?**), the present configuration was constrained to 8,192 tokens by the inference provider, and further limited by the available `2×A100` GPU setup. Future work might accept slower inference in exchange for a significantly larger context window.

Beyond switching between **LLM!**s, other **NLP!** (**NLP!**) models could be introduced for specialised subtasks such as relation extraction or entity recognition. Furthermore, Llama 3.3 70B could be fine-tuned for each subtask, which might particularly improve property extraction. However, such fine-tuning would require re-evaluating the generalisation capabilities of the system.

The emphasis on generalisation in CIExMAS also meant allowing fully free-form triples. While this improves flexibility, it made **URI!** retrieval, particularly for properties, more difficult. In contrast, *synthIE models* are using a controlled output space, which simplifies downstream mapping. Adopting a similar controlled output approach could be beneficial.

Output formatting represents another area for improvement. Transitioning to a standardised structured output format could eliminate parsing errors observed with the custom format. Generating `Turtle` strings remains challenging in some configurations, whereas outputting QNames, as in the initial baseline, proved more reliable and in some cases outperformed more complex designs. Adopting an alternative structured output format might simplify the generation of valid, machine-readable triples.

Property extraction remains the principal challenge. Adding dedicated agents for this subtask did not yield the expected benefits in CIExMAS. Alternative strategies, such as instructing agents to describe the property rather than label it directly, could be explored. Since properties often appear implicitly, embedded in simple phrases linking subjects and objects, decomposing the task may help. For example, one agent could establish the subject–object connection, while another derives the descriptive property, potentially leading to more accurate **URI!** searches.

Finally, other architectural paradigms could be tested for **cIE!**. All CIExMAS configurations followed a largely sequential, pipeline-style process, in which triples were constructed step by step. While effective, pipeline approaches have often underperformed compared to joint models. A joint approach could first gather all relevant **URI!**s, entities, and relations, and then assemble them into triples in a second stage. The flexibility of **LLM!**s makes it possible to design such hybrid workflows that integrate the strengths of both agentic reasoning and broader information aggregation.

Future work could also explore alternative document processing strategies. In the current CIExMAS configuration, documents, typically no longer than three sentences, are processed in their entirety. This could be replaced by a sentence-by-sentence approach, potentially reducing complexity and improving extraction accuracy. Moreover, the original text retains the free-form properties that consistently challenge CIExMAS. As a

pre-processing step, an **LLM!** could rewrite sentences to explicitly highlight potential properties, thereby making them easier to detect.

Property extraction might also be improved by refining the **URI!** retrieval process. Both the sample search mode and the property label search mode have already demonstrated their impact on retrieval quality. For example, CIExMAS could first generate more detailed property descriptions and then search based on those descriptions. As this approach has never been exhaustively tested, it offers a promising avenue for future research. Currently, **URI!** retrieval in CIExMAS relies on embeddings. Alternative systems, such as GraphRAG (**?**), leverage the knowledge graph directly and allow the **LLM!** to assess the usefulness of retrieved subgraphs. Integrating such methods could strengthen retrieval accuracy.

Knowledge graph incorporation itself could be further intensified. CIExMAS has already demonstrated the value of *semantic validation* and *turtle-to-label* tools. A definitive assessment of the *network traversal* tool, however, remains open. Future work could grant agents direct access to the knowledge graph via a **SPARQL!** endpoint, enabling them to perform textitnetwork traversal or *semantic validation* autonomously when prompted appropriately.

Another core optimisation target is the prompt design. While the current prompts follow best practices, such as a structured format and few-shot examples for **ICL!**, they remain manually engineered. Automated prompt optimisation techniques, such as those proposed by **?**, have shown high effectiveness and could reduce subjectivity by replacing human-driven iterations with more reproducible, model-assisted processes. Since **LLM!**s can analyse both the prompt and its outputs, they are well-suited to iteratively refine the instructions.

Beyond automation, prompt design could also be enhanced through more advanced strategies. For instance, the triple extractor could be given additional few-shot examples or a more explicit chain-of-thought structure. Reworking prompts from scratch with a focus on clarity and readability might also lead to more robust performance across configurations.

Ultimately, CIExMAS serves as a starting point for further research. Owing to its modular **MAS!** design, it could be narrowed to focus on specific subtasks — such as property extraction or entity disambiguation — or extended to construct complete knowledge graphs. While the current **cIE!** task still presents substantial limitations and challenges, overcoming these and achieving near-perfect F1 scores would pave the way for CIExMAS to generate entirely new entities or properties absent from the reference knowledge graph. This would move the system into a different research domain, one that currently lacks suitable datasets but has strong potential for real-world applications.

In conclusion, CIExMAS demonstrates a subset of the overall potential of **MAS!**-based **cIE!**. It achieves 80% of state-of-the-art comparison models without fine-tuning and while producing output in `Turtle` format. The findings indicate that simpler yet precise and capable architectures can achieve strong performance. The best results were obtained with a *network architecture* featuring agents for triple extraction, **URI!** mapping, validation, and structured output, supported by corresponding tools. Nevertheless, CIExMAS covers only a focused portion of the possible combinations of **MAS!** and **KG!**

techniques, leaving substantial room for improvements that could surpass current state-of-the-art results. Future research should especially investigate novel property extraction methods and strategies to optimise the overall agentic system.

A Additional Material

A.1 REBEL Testing

Model	***Triples***					
	Precision	Recall	F1	Parental F1	Related F1	Err.
Baseline	0.080	0.189	0.100	0.081	0.081	0.400
ReAct	0.107	0.115	0.089	0.085	0.085	0.100
Network	0.063	0.121	0.076	0.059	0.059	0.120

Table A.1: Triple recognition performance on the REBEL dataset for different CIExMAS configurations.

Configuration	***Subjects***		
	Precision	Recall	F1
Baseline	0.417	0.507	0.414
ReAct	0.577	0.410	0.447
Network	0.310	0.305	0.280

Table A.2: Subject recognition performance on the REBEL dataset for different CIExMAS configurations.

Configuration	***Properties***				
	Precision	Recall	F1	Parental F1	Related F1
Baseline	0.176	0.362	0.214	0.276	0.300
ReAct	0.253	0.247	0.217	0.259	0.318
Network	0.148	0.194	0.149	0.184	0.214

Table A.3: Property recognition performance on the REBEL dataset for different CIExMAS configurations.

Configuration	*Objects*		
	Precision	Recall	F1
Baseline	0.163	0.384	0.208
ReAct	0.198	0.200	0.161
Network	0.122	0.186	0.134

Table A.4: Object recognition performance on the REBEL dataset for different CIExMAS configurations.

Configuration	*Entities*		
	Precision	Recall	F1
Baseline	0.311	0.582	0.382
ReAct	0.383	0.370	0.342
Network	0.211	0.309	0.241

Table A.5: Entity recognition performance on the REBEL dataset for different CIExMAS configurations.

A.2 Detailed Comparison Model Performance Analysis by Category

Model	*Subjects*		
	Precision	Recall	F1
SynthIE$_{T5-large}$	0.927	0.943	0.929
SynthIE$_{T5-base-SC}$	**0.930**	**0.945**	**0.934**
SynthIE$_{T5-base}$	0.923	0.938	0.926
GenIE$_{T5-base}$	0.710	0.742	0.689
GenIE$_{T5-base-SC}$	0.691	0.698	0.665
CIExMAS	**0.930**	0.915	0.907

Table A.6: Subject recognition performance comparison of CIExMAS against state-of-the-art models on the synthIE dataset.

Model	***Properties***				
	Precision	Recall	F1	Parental F1	Related F1
SynthIE$_{T5-large}$	**0.882**	**0.890**	**0.882**	**0.894**	**0.912**
SynthIE$_{T5-base-SC}$	0.850	0.868	0.854	0.885	0.907
SynthIE$_{T5-base}$	0.853	0.848	0.849	0.888	0.905
GenIE$_{T5-base}$	0.646	0.395	0.459	0.501	0.527
GenIE$_{T5-base-SC}$	0.625	0.357	0.424	0.455	0.478
CIExMAS	0.784	0.672	0.710	0.782	0.790

Table A.7: Property recognition performance comparison of CIExMAS against state-of-the-art models on the synthIE dataset.

Model	***Objects***		
	Precision	Recall	F1
SynthIE$_{T5-large}$	0.926	**0.930**	**0.921**
SynthIE$_{T5-base-SC}$	0.891	0.915	0.895
SynthIE$_{T5-base}$	0.912	0.915	0.909
GenIE$_{T5-base}$	0.858	0.484	0.583
GenIE$_{T5-base-SC}$	0.845	0.449	0.551
CIExMAS	**0.960**	0.823	0.867

Table A.8: Object recognition performance comparison of CIExMAS against state-of-the-art models on the synthIE dataset.

Model	***Entities***		
	Precision	Recall	F1
SynthIE$_{T5-large}$	0.936	**0.937**	**0.933**
SynthIE$_{T5-base-SC}$	0.915	0.929	0.918
SynthIE$_{T5-base}$	0.931	0.929	0.928
GenIE$_{T5-base}$	0.932	0.606	0.712
GenIE$_{T5-base-SC}$	0.937	0.587	0.696
CIExMAS	**0.966**	0.876	0.907

Table A.9: Entity recognition performance comparison of CIExMAS against state-of-the-art models on the synthIE dataset.

A.3 Detailed Configuration Performance Analysis by Category

Configuration	***Subjects***			
	Prec.	Rec.	F1	Err.
Initial Baseline	0.601	0.768	0.646	0.200
Task Modularization (SSA)	0.487	0.767	0.564	0.180
Error Incorp. (SSA) + PropEx	0.697	**0.952**	0.771	0.020
Error Incorp. (SSA)	0.564	0.815	0.633	0.120
Error Incorp. (Baseline)	0.764	0.932	0.803	0.000
Agent System Simplification (SSSA)	0.764	0.880	0.791	0.020
ReAct Architecture	0.827	0.863	0.825	0.060
URI! Retrieval (SSSA)	0.837	0.882	0.841	0.040
URI! Retrieval (ReAct)	0.898	0.900	0.885	0.020
Knowledge Graph Integration	0.671	0.700	0.675	0.240
Network Architecture (Gen2)	**0.930**	0.915	0.907	**0.000**
Advanced Adapt. (ReAct)	0.870	0.833	0.837	0.020
Advanced Adapt. (Baseline)	0.800	0.948	0.840	0.000

Table A.10: Subject recognition performance across CIExMAS development iterations.

Configuration	***Properties***					
	Prec.	Rec.	F1	Par. F1	Rel. F1	Err.
Initial Baseline	0.351	0.393	0.363	0.442	0.450	0.200
Task Modularization (SSA)	0.281	0.331	0.290	0.364	0.378	0.180
Error Incorp. (SSA) + PropEx	0.301	0.294	0.286	0.345	0.358	0.020
Error Incorp. (SSA)	0.378	0.445	0.390	0.469	0.487	0.120
Error Incorp. (Baseline)	0.522	0.564	0.531	0.617	0.625	0.000
Agent System Simplification (SSSA)	0.498	0.462	0.471	0.545	0.556	0.020
ReAct Architecture	0.551	0.484	0.503	0.597	0.604	0.060
URI! Retrieval (SSSA)	0.529	0.482	0.495	0.577	0.585	0.040
URI! Retrieval (ReAct)	0.597	0.497	0.533	0.611	0.623	0.020
Knowledge Graph Integration	0.495	0.420	0.449	0.539	0.546	0.240
Network Architecture (Gen2)	**0.784**	**0.672**	**0.710**	**0.782**	**0.790**	**0.000**
Advanced Adapt. (ReAct)	0.746	0.548	0.616	0.676	0.683	0.020
Advanced Adapt. (Baseline)	0.645	0.640	0.631	0.724	0.735	0.000

Table A.11: Property recognition performance across CIExMAS development iterations.

Configuration	***Objects***			
	Prec.	Rec.	F1	Err.
Initial Baseline	0.625	0.745	0.665	0.200
Task Modularization (SSA)	0.524	0.702	0.568	0.180
Error Incorp. (SSA) + PropEx	0.793	0.851	0.802	0.020
Error Incorp. (SSA)	0.675	0.777	0.705	0.120
Error Incorp. (Baseline)	0.860	0.922	0.880	0.000
Agent System Simplification (SSSA)	0.804	0.797	0.789	0.020
ReAct Architecture	0.877	0.815	0.834	0.060
URI! Retrieval (SSSA)	0.899	0.861	0.871	0.040
URI! Retrieval (ReAct)	0.910	0.800	0.843	0.020
Knowledge Graph Integration	0.705	0.641	0.665	0.240
Network Architecture (Gen2)	**0.960**	0.823	0.867	**0.000**
Advanced Adapt. (ReAct)	0.862	0.637	0.714	0.020
Advanced Adapt. (Baseline)	0.906	**0.949**	**0.918**	0.000

Table A.12: Object recognition performance across CIExMAS development iterations.

Configuration	***Entities***			
	Prec.	Rec.	F1	Err.
Initial Baseline	0.695	0.767	0.724	0.200
Task Modularization (SSA)	0.619	0.741	0.657	0.180
Error Incorp. (SSA) + PropEx	0.875	0.905	0.880	0.020
Error Incorp. (SSA)	0.778	0.814	0.784	0.120
Error Incorp. (Baseline)	0.944	0.960	0.947	0.000
Agent System Simplification (SSSA)	0.882	0.892	0.879	0.020
ReAct Architecture	0.918	0.878	0.891	0.060
URI! Retrieval (SSSA)	0.927	0.898	0.908	0.040
URI! Retrieval (ReAct)	0.943	0.878	0.906	0.020
Knowledge Graph Integration	0.731	0.706	0.716	0.240
Network Architecture (Gen2)	**0.966**	0.876	0.907	**0.000**
Advanced Adapt. (ReAct)	0.895	0.718	0.785	0.020
Advanced Adapt. (Baseline)	0.963	**0.970**	**0.964**	0.000

Table A.13: Entity recognition performance across CIExMAS development iterations.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelor-, Master-, Seminar-, oder Projektarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und in der untenstehenden Tabelle angegebenen Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Declaration of Used AI Tools

Tool	Purpose	Where?	Useful?

Unterschrift
Mannheim, den XX. XXXX 2024