universität wien

MASTERARBEIT | MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„A Framework for the Specification of Similarity Functions and Precision Metrics“

verfasst von / submitted by

Paul Jodok von Braitenberg, BA, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2025 / Vienna, 2025

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr.Wolfgang Klas

Acknowledgements

I would like to thank Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas for supervising this thesis and for providing guidance throughout the process. The support was very helpful in shaping the direction and the quality of this thesis.

I also want to thank the MIS (Multimedia Information Systems) research group at the University of Vienna. Their work on the FactCheck system provided a solid foundation for this thesis and made it possible to explore the topic in a practical and focused way. This project has been both challenging and rewarding, and I am grateful for all the support and opportunities that made it possible.

Abstract

Many websites use structured embedded data, which search engines leverage to enhance search result quality and relevance. However, as more and more information on the Web is available in the form of structured data, data describing the same entity more often appears inconsistently across different websites, leading to conflicting information. The FactCheck system, developed by the MIS research group at the University of Vienna, provides an approach for detecting and resolving such conflicting structured data on the Web.

FactCheck relies on a comparison logic that identifies inconsistencies between datasets, but its current implementation is rather simplistic, primarily based on direct string comparisons. This often results in misclassification of equivalent data as conflicting due to differences in formatting, representation, or linguistic variations.

This master's thesis proposes an approach to significantly enhance FactChecks comparison logic by developing new comparison functions and precision metrics, which are combinations of similarity predicates and enable more precise and sophisticated fact comparison. These advanced metrics allow FactCheck to function more effectively across multiple domains, improving its adaptability and accuracy in detecting true conflicts. A framework has been designed to test, evaluate, and develop new functions and precision metrics, providing a structured foundation for refining FactChecks data reconciliation capabilities.

Kurzfassung

Viele Webseiten nutzen strukturierte Daten, die von Suchmaschinen verwendet werden, um die Qualität und Relevanz von Suchergebnissen zu verbessern. Allerdings treten diese strukturierten Daten, selbst wenn sie dieselbe Entität beschreiben, oft in inkonsistenter Form auf verschiedenen Webseiten auf, was zu widersprüchlichen Informationen führt. Das FactCheck-System, entwickelt von der MIS-Forschungsgruppe der Universität Wien, bietet einen Ansatz zur Erkennung und Auflösung solcher widersprüchlichen strukturierten Daten im Web.

FactCheck basiert auf einer Vergleichslogik, die Inkonsistenzen zwischen Datensätzen identifiziert, deren aktuelle Implementierung jedoch recht simpel ist und hauptsächlich auf direkten String-Vergleichen beruht. Dies führt oft zur Fehlklassifizierung äquivalenter Daten als widersprüchlich, da Unterschiede in Formatierung, Repräsentation oder sprachlichen Varianten nicht berücksichtigt werden.

Diese Masterarbeit schlägt einen Ansatz zur signifikanten Verbesserung der Vergleichslogik von FactCheck vor, indem neue Vergleichsfunktionen und Präzisionsmetriken entwickelt werden. Letztere bestehen aus Kombinationen von Vergleichsfunktionen und ermöglichen eine präzisere und differenziertere Faktenabgleichung. Diese fortschrittlicheren Metriken erlauben es FactCheck, in mehreren Anwendungsdomänen effektiver zu arbeiten und verbessern die Anpassungsfähigkeit sowie die Genauigkeit bei der Erkennung echter Konflikte. Es wurde ein Framework entwickelt, das das Testen, Evaluieren und Entwickeln neuer Vergleichsfunktionen und Präzisionsmetriken ermöglicht und somit eine strukturierte Grundlage für die Verfeinerung der Datenabgleichsfähigkeiten von FactCheck schafft.

Contents

Acknowledgements i

Abstract iii

Kurzfassung v

List of Figures ix

Listings xi

1. Introduction 1
- 1.1. Motivation . . . 3
- 1.2. Goal . . . 4

2. Background 7
- 2.1. Fact-Checking and Fake News Detection . . . 8
- 2.2. Structured Data on the Web . . . 9
- 2.3. Similarity Functions for Structured Data Comparison . . . 11
  - 2.3.1. Basic Similarity Functions . . . 11
  - 2.3.2. Advanced and Domain-Specific Similarity . . . 12
  - 2.3.3. Use in Precision Metrics . . . 13
- 2.4. Summary . . . 13

3. Related Work 15
- 3.1. Automated Fact-Checking Systems . . . 15
- 3.2. Similarity Functions and Comparison Techniques . . . 19
- 3.3. Conflict Resolution in (Semi-)Structured Data . . . 22
  - 3.3.1. Problem Formulation and Modeling Assumptions . . . 22
  - 3.3.2. Voting-Based Methods . . . 23
  - 3.3.3. Source Reliability Estimation . . . 24
  - 3.3.4. Similarity-Driven Fusion . . . 26
  - 3.3.5. Probabilistic Inference . . . 27
  - 3.3.6. Metadata, Constraints, and Contextual Signals . . . 28
- 3.4. Existing Frameworks and Tools . . . 29
  - 3.4.1. General-Purpose Integration and Fusion Frameworks . . . 29
  - 3.4.2. Truth Discovery Frameworks . . . 30
  - 3.4.3. Libraries for Similarity Computation . . . 30
- 3.5. FactCheck . . . 31

3.6. Summary and Research Gap 34

4. Design **37**
4.1. System Architecture Overview 37
4.1.1. Predicate Integration via `predicate.py` 40
4.1.2. Precision Metric Configuration via `pm_data.json` 42
4.1.3. Property Annotation via `propertydata.json` 42
4.1.4. Semantic Comparison via LLM and SKOS 43
4.2. Design Requirements 44
4.3. API Endpoints and Data Format Requirements 46
4.4. Web-Based Editor for Predicate and Metric Management 47
4.5. Design Summary 49

5. Implementation **51**
5.1. Predicate Function Implementation 52
5.2. Precision Metric Configuration 58
5.3. Property Mapping via `propertydata.json` 59
5.4. Integration of Semantic Methods 60
5.4.1. LLM-Based Semantic Comparison 60
5.4.2. SKOS-Based Concept Matching 62
5.5. Frontend Editor Implementation 64
5.5.1. Overview and Architecture 64
5.5.2. Similarity Function Creator 64
5.5.3. Precision Metrics Tab 65
5.5.4. API Playground 66
5.5.5. Runtime Integration and Deployment 67
5.5.6. Conclusion 67
5.6. Similarity Editor and Safe, Concurrent Code Updates 68
5.7. Summary and Conclusion 70

6. Experimental Evaluation **71**
6.1. Evaluation Objectives 71
6.2. Test Data 72
6.3. Coverage Analysis 74
6.4. Error Analysis and Limitations 75
6.5. Performance Considerations 78
6.6. Summary of Findings 79

7. Conclusion **81**

Bibliography **85**

A. Appendix: Frontend editor screenshots **89**

List of Figures

3.1. High-level architecture of the FactCheck system 34

4.1. System architecture of the FactCheck system showing interactions, with a focus on the FactServer. 39

A.1. Similarity Function Creator Overview (Part 1/3). 90
A.2. Similarity Function Creator Overview (Part 2/3). 91
A.3. Similarity Function Creator Overview (Part 3/3). 92
A.4. Precision Metrics Overview (Part 1/2). 93
A.5. Precision Metrics Overview (Part 2/2). 94
A.6. API Playground Request/Response 95

Listings

4.1. Predicate Function for Date Comparison 41
4.2. Precision Metric Definition for Date Matching 42
4.3. Mapping Schema Properties to Precision Metrics 42
4.4. Assigning a Precision Metric to a Property-Type Combination 43
4.5. JSON-LD input example for /compare 46

5.1. Predicate for Comparing Wind Speed 53
5.2. Predicate for Comparing Temperature Values 55
5.3. Minimal precision metric definition 58
5.4. Precision metric with logical OR operator 58
5.5. Global mapping of properties to metrics 59
5.6. Type-specific property mapping 60
5.7. Semantic comparison using a local LLM 61
5.8. SKOS-based synonym matching using altLabels 62
5.9. Example Fact set Request Body 66
5.10. Structure of a UI-generated predicate method 69

6.1. Normalization map for common genre variants 76

1. Introduction

The rise of all-encompassing digitalization and the ubiquitous use of the internet in society has led to a paradigm shift in the way people use media and consume news. According to Pew Research Center, in 2017 online social networks have outpaced print newspapers as a news source in the United States[1]. For young people between 18 and 24, social media has even overtaken television as the main source of news consumption in the UK and the US already back in 2016[2]. This is particularly noteworthy because the mechanisms behind information made available online and shown on television are fundamentally different. Anyone can gain reach on the internet and disseminate news that is not subject to validation under the pretence of neutral reporting, and new communication channels are spreading at high speed. The spread of fake news can be considered one of the major challenging issues in that regard. Whether this is information that is deliberately disseminated incorrectly in order to stir up discord or information that is disseminated incorrectly without malicious intent due to a lack of expertise plays a rather subordinate role. Vosoughi et al. [VRA18] have shown that online news that is proven to be false spread six times faster than news that is proven to be truthful, and that 70 percent of users cannot distinguish fake from real news. Due to this increasing volume of online content and the rising spread of fake news and misinformation and all its social or political consequences, automated fact-checking has evolved rapidly. Various technical approaches make it possible to scale and speed up the fact-checking process. Among the most prominent detection techniques are machine learning, deep learning (NN), natural language processing (NLP), crowd sourcing, blockchain and artificial intelligence (AI), which are employed to identify and validate statements. According to Aïmeur et al. [AAB23], fake news detection solutions can be classified into three different categories, namely the news content-based approaches, the social context-based approaches (that can be divided into network and user-based approaches), and hybrid approaches. These approaches will be further described in the next chapter.

Recent developments are moving toward hybrid models that combine, for instance, both AI-based and manual reviews [AAB23]. These models, for example, use machine learning to pre-select suspicious statements, which are then analyzed in greater detail by human fact-checkers. A growing trend is real-time fact-checking, especially useful on social media or during live broadcasts, to identify and correct misinformation immediately.

The rapid advancement in this field is primarily driven by progress in AI technologies and the constant growth in available data. Additionally, there is increasing awareness of

[1] https://www.pewresearch.org/short-reads/2018/12/10/social-media-outpaces-print-newspapers-in-the-u-s-as-a-news-source/, last access date 05.11.2024

[2] https://www.bbc.com/news/uk-36528256, last access date 05.11.2024

the importance of transparency and explainability in automated systems to build trust in fact-checking tools [GSV22].

However, the challenge of combating fake news is not limited to analyzing the text, multimedia, or social context of individual news articles. A growing portion of information available on the Web is structured in the form of machine-readable data, providing a standardized and semantically rich format that can be processed automatically. Structured data helps systems recognize relationships between entities, understand context, and integrate information from multiple sources. This shift towards structured data offers new opportunities for fact-checking and verifying the accuracy of information. For example, structured data can support automated verification of factual claims by cross-referencing multiple data sources, enabling scalable and accurate fact-checking solutions. Thus, the increasing use of structured data on the Web contributes not only to better search and data integration but also to the broader goal of mitigating misinformation.

The most widely applied format for structured data on the Web is considered to be *schema.org*, with over 12 million websites using that format [GBM16]. Schema.org provides a shared vocabulary for annotating web content in a way that can be understood by search engines and other automated systems. It offers a set of schemas for describing various types of entities - such as people, places, events, organizations, and products - using structured data formats like JSON-LD, RDFa, and Microdata. By embedding this structured data into HTML, websites enable search engines to generate rich snippets, enhance search result presentations, and improve content discoverability. As a result, schema.org plays a crucial role in enabling machines to understand the semantics of web content, which is a foundational aspect of the semantic web and linked data initiatives.

Despite these advancements, the field faces ongoing challenges. One major issue is the lack of precision in comparing structured data across various sources, especially when inconsistencies arise due to different data formats, units, or linguistic representations. Structured data on the Web is crucial, as it enables search engines to deliver meaningful and well-organized search results. Semantic data annotation technologies, such as RDFa, Microdata, and JSON-LD, have been widely adopted by major search engines, making them essential tools for navigating the semi-structured nature of the World Wide Web [GBM16]. Another issue is that, with the rising number of information-providing websites, also the amount of contradictory structured data increases.

For that reason, the MIS research group at the University of Vienna has developed *FactCheck*, a framework for the detection and resolution of conflicting structured data on the Web, with the general goal to improve the quality of information on the Web. The FactCheck system is a step forward in addressing these earlier mentioned issues. However, the current systems reliance on simple, generic similarity measures - such as string equality or basic numeric comparisons - limits its effectiveness, particularly when dealing with complex or domain-specific data. For example, it cannot adequately handle nuanced differences in data formats, measurement units, or semantic variations, that often arise in specialized domains like scientific publications, financial data, or multilingual datasets. As a result, the accuracy and reliability of the conflict resolution process suffer, and the system's ability to scale to diverse and heterogeneous datasets is constrained.

This thesis aims to address these shortcomings by introducing a robust framework of similarity functions and precision metrics to improve and expand the systems applicability. Specifically, it proposes a set of newly defined similarity functions and an extensible system that allows for the dynamic creation, configuration, and integration of customized similarity functions tailored to the characteristics of the data at hand. This includes the development of a web-based interface for defining and managing these functions, as well as mechanisms to connect these definitions directly to the FactCheck backend. In addition, the thesis focuses on designing new precision metrics to evaluate the accuracy and reliability of the similarity functions, thus providing a solid foundation for future enhancements of the FactCheck framework. The goal is not only to increase the accuracy of conflict resolution in structured data but also to make the system more transparent, adaptable, and user-friendly for both researchers and practitioners.

1.1. Motivation

The increasing prevalence of structured data on the Web has created new opportunities for enhancing the accuracy and efficiency of information retrieval and fact-checking systems. Standardized data formats, such as *schema.org*, enable systems to interpret information consistently across diverse sources, improving interoperability and making it easier for automated tools to access and process data. Automated fact-checking frameworks, such as *FactCheck*, already demonstrate how structured data can be leveraged to enhance the reliability of information and reduce the manual effort required for verification.

However, this development also brings new challenges, particularly in terms of data consistency and reliability. While structured data formats provide a standardized way to annotate information, they do not inherently guarantee accuracy or completeness. Conflicting data from multiple sources, differences in data formats, and inconsistencies in measurement units are just some of the obstacles that undermine the credibility of online information. Additionally, many systems, including the current version of *FactCheck*, rely on simple, generic similarity measures - such as string equality or basic numeric comparisons - which are insufficient for capturing subtle differences in complex or domain-specific data. This limitation can result in incorrect assessments of equivalence or inconsistency, reducing the systems overall effectiveness.

Moreover, systems that depend on basic similarity measures often struggle to handle the heterogeneity of real-world data. Differences in representation formats, the use of different units or scales, and variations in terminology or language are common issues that can lead to false negatives or false positives in data comparison. These issues are compounded by the scale and complexity of structured data available online, posing challenges for both performance and scalability.

Nevertheless, the potential benefits of enhancing structured data comparison and validation systems are substantial. More advanced frameworks that incorporate robust similarity functions and precision metrics can significantly improve the quality and reliability of fact-checking systems. They enable more accurate identification and resolution of data conflicts, support the integration of heterogeneous data sources, and increase trust

in the information presented to users. Furthermore, by accommodating domain-specific needs and providing greater flexibility in data interpretation, such systems can extend the applicability of automated fact-checking to a wider range of use cases.

The *FactCheck* system, developed by the MIS research group at the University of Vienna, represents a promising approach to addressing these challenges. However, its current limitations highlight the need for further research and development. This thesis is motivated by the ambition to overcome these challenges by developing a comprehensive framework that integrates advanced similarity functions and precision metrics to enhance the precision, scalability, and applicability of structured data comparison and validation in the *FactCheck* system.

1.2. Goal

The primary goal of this thesis is to significantly enhance the *FactCheck* system by introducing a sophisticated framework for similarity functions and precision metrics, capable of addressing the inherent complexities of structured data from diverse domains. To achieve this, the thesis sets out several interconnected goals:

- Develop and integrate domain-specific similarity functions that go beyond basic string and numeric comparisons, incorporating semantic analysis, unit conversion, and data normalization techniques, and leveraging local Large Language Models (LLMs) and SKOS (Simple Knowledge Organization System) labels for enhanced semantic understanding and contextual relevance.
- Design and implement precision metrics that allow for the fine-tuning of similarity thresholds, thus improving the accuracy of conflict detection and resolution.
- Create a modular and extensible framework that enables the dynamic addition of new similarity functions and precision metrics, facilitating adaptation to evolving data standards and requirements.
- Evaluate the effectiveness of the proposed framework using real-world datasets from various domains, demonstrating its applicability and scalability.

An ideal comparison framework for structured fact-checking must satisfy multiple, sometimes competing requirements. It must be **flexible** enough to support heterogeneous input formats, **expressive** enough to define domain-specific equivalence conditions, and **transparent** enough for users to understand and refine the logic that drives comparison decisions. These requirements exceed the capabilities of many existing similarity libraries, which primarily focus on syntactic string distances and offer limited support for semantic or configurable comparison strategies. For example, toolkits like *SimMetrics* or *FuzzyWuzzy* implement a variety of string similarity metrics, but do not account for contextual or semantic equivalence [CH13, GMIHF19]. On the other hand, many truth discovery systems such as *TruthFinder* rely on hard-coded conflict resolution strategies tailored to specific application domains [YHY07, LGM+16]. While probabilistic frameworks such

as *Probablistic Soft Logic* (PSL) offer a more flexible modeling approach, they are not designed for modular or user-definable similarity functions [BBHG].

The FactCheck system developed at the University of Vienna already provides a modular foundation for predicate-based comparison, but until now, this functionality was largely hidden within Python code and JSON configuration files. Users who wished to experiment with new predicates or metric combinations had to modify the source code, redeploy the backend, and revalidate changes manually. This technical barrier limited the system's applicability and hindered its adoption by non-programmers.

To overcome this gap, the present work introduces a web-based frontend that enables users to create, modify, and assign predicate functions and precision metrics in an interactive, real-time environment. While technically all configuration steps - such as mapping properties to metrics - could be expressed in JSON, doing so requires detailed knowledge of the systems internal data model. The frontend therefore serves not merely as a convenience layer, but as an essential component for democratizing access to comparison logic. It provides visual feedback, form-based editing, and an integrated API playground that together form a feedback loop for iterative refinement. In this way, the system becomes both more usable and more robust: usability is enhanced through immediate interaction, and robustness is improved by allowing testing and debugging without leaving the browser.

This thesis thus pursues not only algorithmic improvements in predicate design, but also a shift in how comparison logic is configured, extended, and validated - placing user control and explainability at the center of the system design.

The following chapters will delve deeper into the theoretical foundations, the design and development of the proposed framework, and its evaluation on representative datasets, outlining how each aspect contributes to achieving these goals. Chapter 2 introduces the fundamental concepts and theoretical background related to fact-checking, structured data, and similarity measurement. Chapter 3 provides a comprehensive review of existing approaches and relevant work in the field, highlighting their strengths and limitations. Chapter 4 describes the overall system architecture and design principles of the enhanced *FactCheck* framework. Chapter 5 details the technical implementation of the new similarity functions and precision metrics, including semantic methods such as local Large Language Models (LLMs) and SKOS-based approaches, and the implementation of the frontend environment that allows for the definition, editing, and testing of similarity functions and precision metrics using arbitrary data sets. Chapter 6 presents the experimental evaluation setup, describes the datasets used, and analyzes the performance and scalability of the proposed framework. Finally, Chapter 7 summarizes the main findings, discusses limitations, and outlines potential directions for future work.

2. Background

The increasing prevalence of fake news and misinformation in the digital era has sparked significant research interest and led to the emergence of a wide range of technical approaches aimed at mitigating their spread. Misinformation can have far-reaching consequences, from influencing political discourse and public health decisions to eroding trust in democratic institutions. As traditional media gatekeeping mechanisms have given way to algorithm-driven content distribution on social media platforms, users are increasingly exposed to unverified or even deliberately misleading information.

This growing challenge has motivated researchers to explore automated solutions for fact-checking - that is, the verification of factual claims made in public discourse. Fact-checking traditionally involves human experts evaluating the truthfulness of statements using reputable sources. However, due to the immense volume of online content and the real-time nature of digital communication, manual verification methods struggle to keep pace. Automated fact-checking seeks to alleviate this bottleneck by leveraging technologies such as natural language processing (NLP), machine learning (ML), knowledge graphs, and structured data to scale the verification process.

An especially promising approach for automated fact-checking is the use of structured data on the Web. Unlike unstructured text, structured data provides information in a standardized, machine-readable format, making it easier to retrieve, compare, and reason about factual content. Initiatives like schema.org, which offer a shared vocabulary for annotating web content, have significantly contributed to the proliferation of structured data online. Search engines, recommendation systems, and intelligent agents increasingly rely on structured data to interpret and present information more accurately.

At the same time, the growth of structured data introduces new challenges. As different sources independently annotate entities and attributes, inconsistencies and contradictions emerge: values may differ due to outdated information, measurements may use different units, or labels may vary across languages or schemas. These inconsistencies are particularly problematic for fact-checking systems that must decide whether two data values refer to the same fact.

To address this, systems require mechanisms for comparing structured data instances in a nuanced and context-aware manner. This is where similarity functions become essential. A similarity function computes the degree of equivalence between two values - be they strings, numbers, dates, or more complex objects. While many applications use basic functions such as exact string matching or Euclidean distance, these approaches fall short in more sophisticated scenarios. For example, determining whether 12.5řC and 54.5řF represent the same temperature requires not just numeric comparison but also unit normalization. In other cases, semantic similarity - such as recognizing that actor

and Schauspieler refer to the same role - necessitates external resources like ontologies or language models.

This chapter introduces the key concepts and technologies that underpin the rest of this thesis. Section 2.1 provides an overview of fact-checking and its automation, including major technological paradigms and challenges. Section 2.2 explores the nature and role of structured data on the Web, emphasizing the significance of standards like schema.org. Section 2.3 discusses the role of similarity functions in data comparison, presenting both foundational methods and more advanced techniques that enable robust conflict resolution. Together, these foundations frame the problem space and justify the design decisions made in the development of the similarity framework introduced in later chapters.

2.1. Fact-Checking and Fake News Detection

The proliferation of digital media and the widespread use of social networks have fundamentally changed how information is produced, distributed, and consumed. Traditional gatekeepers, such as journalists and editorial teams, have been increasingly bypassed by user-generated content and algorithm-driven news feeds. While this democratization of information allows for greater participation and immediacy, it also increases the risk of misinformation and disinformation spreading rapidly and uncontrollably. Studies such as Vosoughi et al. [VRA18] have shown that false information, for example on platforms like Twitter, spreads significantly faster and more broadly than true information, underscoring the urgent need for robust fact-checking mechanisms.

Fact-checking is the process of assessing the accuracy of factual claims made in public discourse, particularly in the context of news and social media. Traditionally, this task has been performed by professional journalists who manually verify claims using trusted sources. However, the volume and speed of information in todays media landscape make manual verification impractical as a sole solution. This has led to the emergence of automated fact-checking, which leverages computational methods to detect and validate factual assertions with minimal human intervention.

Automated fact-checking typically involves several core steps: claim detection, evidence retrieval, claim-evidence alignment, and veracity assessment. To support these steps, a wide range of technologies is employed, including natural language processing (NLP), information retrieval, machine learning (ML), knowledge graphs, and semantic analysis. More advanced systems also incorporate techniques from explainable AI and probabilistic reasoning to better handle uncertainty and justify decisions.

Guo et al. define automated fact-checking as the task of assessing the veracity of claims made in written or spoken language. They outline a three-stage framework consisting of claim detection (identifying check-worthy claims), evidence retrieval (locating relevant supporting or refuting information), and claim verification, which itself includes verdict prediction and, in more advanced systems, justification production [GSV22]. The authors emphasize that the rapid spread of information in modern media ecosystems has made manual fact-checking infeasible at scale, thereby increasing the urgency for reliable

automated solutions. Their survey provides a comprehensive overview of existing datasets and modeling strategies, and highlights key challenges such as cross-domain generalization, explainability, and the reliability of external knowledge sources.

A key area of development in the field of automated fact-checking is the classification of detection approaches. Aïmeur et al.[AAB23] identify three primary categories:

- News content-based approaches, which analyze the intrinsic properties of the news itself. These methods rely on features extracted from the textual and multimedia content of the article, such as writing style, sentiment, grammar, or image consistency. Techniques include supervised ML, DL (Deep Learning), and NLI (Natural Language Inference). Such models often operate independently of social context and are trained on labeled datasets of true and false claims.

- Social context-based approaches, which examine the extrinsic environment in which news is shared. These methods use metadata such as user engagement (likes, shares, comments), user profiles, and network dynamics (e.g., retweet patterns or propagation trees) to assess credibility. For example, news that spreads unusually quickly or is propagated by low-credibility accounts may be flagged as suspicious. This category is further subdivided into user-based and network-based methods.

- Hybrid approaches, which combine the strengths of content-based and context-based strategies. These models integrate linguistic features with contextual signals to form a more holistic assessment of a claim's veracity. Recent advances also include combining automated pre-filtering with human-in-the-loop verification to improve both scalability and accuracy. Hybrid systems are especially promising in dynamic environments like live events or breaking news scenarios, where real-time fact-checking is critical.

Despite these advances, challenges remain. False claims are often subtle, highly contextual, or deliberately crafted to evade detection. Moreover, cross-lingual misinformation and multimedia deepfakes introduce new complexities. Addressing these issues requires continuous innovation in both model development and data curation.

The evolution of fact-checking technologies is increasingly tied to the availability and quality of structured data. While much of the existing research focuses on unstructured text (e.g., news articles or tweets), structured representations - such as schema.org annotations or linked data - offer new opportunities for scalable, machine-readable fact verification. The next section explores how structured data on the Web plays a pivotal role in this context.

2.2. Structured Data on the Web

In the context of automated fact-checking and information validation, the increasing presence of structured data on the Web marks a fundamental shift. Unlike unstructured text, which requires extensive parsing and semantic interpretation, structured data is

encoded in machine-readable formats that explicitly define relationships between entities, attributes, and values. This facilitates automatic processing, integration, and reasoning across different datasets, making structured data a valuable asset for large-scale and domain-independent fact-checking systems.

Structured data is often embedded directly within web content using formats such as RDFa, Microdata, or JSON-LD. These formats allow web developers to annotate HTML elements with metadata that describes their semantic meaning. For instance, a product listing may include fields for name, price, currency, and availability, each tagged using a standardized vocabulary. Among these vocabularies, schema.org is by far the most widely adopted. According to Guha et al.[GBM16], schema.org annotations are used by more than 12 million websites, making it the de facto standard for structured data on the Web.

Schema.org was launched as a collaborative initiative by major search engines - including Google, Bing, Yahoo, and Yandex - with the goal of improving search quality by making web content more understandable for machines. It defines a comprehensive ontology for a wide range of domains, including people, organizations, places, products, events, and creative works. By providing a shared vocabulary, schema.org enables the Web to function as a vast, decentralized knowledge graph, where facts are explicitly stated and accessible to automated agents.

The practical benefits of structured data are evident in various real-world applications. Search engines use structured annotations to generate rich snippets, which display detailed search results with ratings, prices, availability, and other structured facts. Virtual assistants like Siri, Alexa, or Google Assistant rely on structured data to answer user queries. Furthermore, knowledge bases such as Wikidata or DBpedia aggregate structured information from diverse sources and use it to support tasks ranging from question answering to recommendation and reasoning.

However, the widespread adoption of structured data also introduces new challenges. Since structured data is published independently by different actors, inconsistencies frequently arise. For example, the same entity (e.g., a movie, person, or location) might appear on multiple websites with slightly differing values. These differences may result from updates over time, different measurement units, alternative naming conventions, or translation into different languages. Without a reliable mechanism to resolve such conflicts, automated systems may return contradictory information or fail to recognize equivalence between datasets.

Additionally, structured data is not inherently trustworthy. While the use of standard vocabularies helps with interoperability, it does not guarantee the correctness of the values provided. In fact, misinformation can be just as easily encoded in structured form as in unstructured form. This is particularly problematic for fact-checking systems that rely on structured inputs as sources of truth.

To overcome these limitations, systems need to employ robust comparison and validation mechanisms that can handle subtle variations in data representation. These include techniques such as unit normalization (e.g., converting Fahrenheit to Celsius), synonym

detection, temporal reasoning (e.g., recognizing outdated data), and semantic alignment using ontologies or language models. Moreover, the use of linked data principles - which encourage the interconnection of datasets via unique identifiers and relationships - can further support disambiguation and fact validation.

The next section explores how such comparison mechanisms can be formalized through the use of similarity functions, which serve as the computational core for structured data matching and conflict resolution in systems like FactCheck.

2.3. Similarity Functions for Structured Data Comparison

The task of comparing structured data across different sources is a core challenge in automated fact-checking systems such as *FactCheck*. At the heart of this comparison lies the concept of a similarity function - a computational method that evaluates how alike two values are. These functions are essential for identifying whether two pieces of information refer to the same real-world fact, even when they differ in form, language, units, or structure.

2.3.1. Basic Similarity Functions

Many existing systems rely on basic similarity measures, such as exact string matching or numeric equality. While these approaches are simple and fast to compute, they fail in cases where the same concept is expressed in slightly different ways. For example, *10.00 USD* and *10 dollars*, or *Schauspieler* and *actor*, may represent the same value but would not be considered equal by a naive comparison. These shortcomings are particularly problematic when working with data from diverse sources, written in different languages, or using different notations and standards.

To address this, more flexible measures such as Levenshtein distance, Jaccard similarity, or cosine similarity can be used[1]. These allow for approximate string matching by quantifying how many edits are needed to transform one string into another or how similar two sets of words are. Numeric comparisons can be improved by using relative differences or tolerance thresholds rather than requiring exact equality.

A comprehensive overview and evaluation of syntactic similarity measures is provided by a framework proposed in [GMIHF19], which systematically categorizes similarity methods based on three components: character-level similarity, string segmentation strategy, and the matching technique used. The study distinguishes between character-level measures (such as Levenshtein distance), token-level measures (which operate on word-like units), and soft variants that combine both types. These soft measures, which allow approximate token matches using character-level distances, consistently outperform their crisp counterparts in noisy or inconsistent datasets. The framework also demonstrates that no single similarity measure performs best across all domains, highlighting the importance of context-specific selection and configuration. This insight

[1]These concepts will be described in more detail in the next chapter.

supports the modular design of the similarity function framework proposed in this thesis, where different matching strategies can be dynamically combined to fit the structure and characteristics of the domain data at hand.

2.3.2. Advanced and Domain-Specific Similarity

More sophisticated similarity functions are often domain-specific, meaning they are tailored to the semantics and typical variation patterns of the data they compare. For instance, when comparing temperatures, the values *12řC*, *12 degrees Celsius*, and *53.6řF* should all be considered equivalent. Here, a proper comparison requires unit normalization, converting all values to a common reference (e.g., Celsius) before comparison.

Similarly, dates might require normalization to a common format, and names of entities may require canonicalization (e.g., recognizing that *NYC*, *New York City*, and *New York* refer to the same place).

To enable this kind of nuanced comparison, modern systems often integrate semantic similarity techniques. These include:

- Knowledge-based similarity, where ontologies or vocabularies (e.g., SKOS, WordNet) provide information about the meaning and relationships of terms. This approach utilizes structured lexical databases like WordNet to measure similarity based on the relationships between concepts. For instance, semantic similarity can be computed by analyzing the distance between concepts in a taxonomy or ontology. A comprehensive overview of such methods is provided by Harispe et al. [HRJM16] in their survey on semantic measures.

- Vector-based similarity, where text is represented as vectors using techniques such as word embeddings or transformer-based language models. This method represents words or texts as vectors in a high-dimensional space, allowing for the computation of similarity based on distance metrics like cosine similarity. Techniques such as Word2Vec and GloVe are prominent examples.

- Language model integration, where local Large Language Models (LLMs) are queried with prompts to assess whether two terms are semantically similar or refer to the same concept. Modern NLP systems often integrate large language models (LLMs) to assess semantic similarity. These models, such as BERT or GPT, can be queried with prompts to determine the semantic equivalence of terms or phrases. A discussion on the evolution of semantic similarity algorithms, including the role of LLMs, is presented by [Aya25].

For example, a local LLM can be asked: *Are director and Filmemacher semantically equivalent?* - and it can respond based on contextual knowledge and multilingual embeddings. Similarly, SKOS labels can be used to match entities via preferred labels and synonyms, even across languages.

2.3.3. Use in Precision Metrics

In the context of the *FactCheck* system, similarity functions are not used in isolation. They are combined into precision metrics - composite logic structures that apply similarity predicates to selected properties of structured data. For example, a precision metric might evaluate whether two movie records match based on the similarity of their titles, release dates, and director names. Each of these comparisons may use a different similarity function appropriate for the respective property.

By modularizing these functions and allowing them to be combined dynamically, the FactCheck system gains both adaptability and transparency. Users can define how strict or lenient the comparison should be and can select or implement custom similarity functions for specific domains such as weather observations, scientific measurements, or cultural content.

2.4. Summary

This chapter provided a conceptual foundation for the work presented in this thesis. It began by outlining the context of automated fact-checking in an age of widespread misinformation, highlighting the need for scalable and reliable verification systems. Fact-checking has evolved from manual journalistic practice to sophisticated technical systems using artificial intelligence, natural language processing, and network analysis to identify and validate claims. In particular, the combination of content-based and context-based methods has proven effective in detecting fake news on social media and other online platforms.

Subsequently, the role of structured data on the Web was discussed. Structured data enables machines to interpret and link information across websites, forming the technical basis for improved search, intelligent assistants, and automated fact-checking. Standards such as schema.org and formats like JSON-LD, RDFa, and Microdata have driven its widespread adoption. Yet the diversity of data sources and annotation practices gives rise to inconsistencies, making accurate comparison and validation a non-trivial challenge.

The final section explored similarity functions as a key mechanism for comparing structured data. It highlighted the limitations of simple measures like string or numeric equality and emphasized the need for advanced, domain-aware techniques. These include semantic similarity, unit conversion, normalization, and even the use of local language models or external knowledge bases such as SKOS. In the *FactCheck* system, such functions are composed into precision metrics that drive the decision logic for identifying conflicting information.

Together, these foundational concepts justify the development of a flexible and extensible framework for structured data comparison - a core contribution of this thesis. The following chapter builds upon this background by reviewing related work in structured data validation, similarity metrics, and conflict resolution.

3. Related Work

In recent years, the field of automated fact-checking has gained significant traction, resulting in the development of a wide range of approaches and tools. These advances span several interrelated domains, including fake news detection, semantic similarity measurement, and the reconciliation of structured or semi-structured data. A thorough understanding of the state of the art in these areas is essential for situating this thesis within the broader research landscape and for identifying the gaps it aims to address.

This chapter provides a structured overview of the most relevant literature. We begin by reviewing approaches to automated fact-checking and their integration with structured or semi-structured data. We then examine techniques for string similarity and data comparison, covering both syntactic and semantic methods. Subsequently, we explore existing frameworks and toolkits that support the development of extensible infrastructures for similarity functions. Finally, we introduce the *FactCheck* system developed by the MIS research group at the University of Vienna, outlining its functionality, concepts, and current limitations. *FactCheck* represents a practical approach to identifying and resolving conflicts in structured data on the Web and serves as the foundation for the framework proposed in this thesis.

3.1. Automated Fact-Checking Systems

Automated Fact-Checking (AFC) has emerged as a key research area in response to the increasing volume and velocity of (mis)information dissemination on the Web and social media. Manual fact-checking, while reliable, is time-consuming and cannot scale with the current rate of information flow. Automated approaches therefore aim to assist or replace parts of this process using techniques from natural language processing (NLP), machine learning, knowledge representation, and information retrieval [GSV22].

A widely adopted framework structures the AFC process into three core stages: *claim detection*, *evidence retrieval*, and *claim verification*. A fourth component, *justification production*, has gained increasing importance, particularly in user-facing applications where transparency and explainability are critical.

Claim Detection. The first step involves identifying statements that are worth verifying, typically referred to as check-worthy claims. These are often selected based on heuristics or learned models that assess factors such as potential public impact, novelty, or political relevance. Early approaches focused on manually curated inputs, but recent systems leverage binary classification or ranking models to determine check-worthiness, often relying on linguistic, contextual, and social features [GSV22].

Evidence Retrieval. After a claim is identified, relevant evidence needs to be retrieved to support or refute it. Sources include knowledge bases (e.g., Wikipedia, DBpedia), search engine results, scientific papers, and social media posts. Methods range from keyword-based retrieval and TF-IDF matching to dense vector representations and entity linking. A central challenge is ensuring the quality and trustworthiness of the evidence, particularly in open-domain scenarios [GSV22].

Claim Verification. In this stage, the veracity of a claim is determined based on the retrieved evidence. Common formulations include binary classification (true/false), multi-class labeling (e.g., supported, refuted, or not enough info), or textual entailment modeling. Systems may process claims in isolation or require complex reasoning over multiple pieces of evidence. Approaches range from pipeline architectures to end-to-end or joint models [GSV22].

Justification Production. A growing demand for explainable AI has led to the inclusion of justification generation as a distinct stage in fact-checking systems. Justifications can either quote relevant parts of the evidence directly (extractive) or summarize and explain the reasoning in their own words (abstractive) and should aim to be readable, plausible, and faithful to the models reasoning. Techniques include attention-based highlighting, logic-based reasoning, and natural language generation. This component is essential for user trust and system transparency, especially in critical domains such as health or politics [GSV22].

Together, these components form a modular architecture that allows for fine-grained control and evaluation of AFC pipelines. Despite significant advances, several challenges remain, including the integration of multimodal and multilingual data, robust evidence selection from noisy sources, and ensuring explanation faithfulness.

An overview is provided by [TV18]. They emphasize that automated fact-checking systems vary along three primary axes: the type of input, the nature of the output, and the kind of evidence used.

Common input formats include subject-predicate-object triples, textual claims, and entire documents. Triples are widely used in structured data contexts, while textual claims are extracted and condensed for clarity, as seen on platforms like PolitiFact. Document-level inputs pose the additional challenge of claim identification as a preliminary step [TV18].

The choice of evidence profoundly influences system design. Some systems rely solely on the linguistic features of the claim without external evidence - an approach that departs significantly from journalistic practice. Others retrieve structured evidence from knowledge graphs, or unstructured textual evidence from encyclopedic sources (e.g., Wikipedia), news outlets, or past fact-check repositories. Additional contextual signals such as claim originator metadata (e.g., political affiliation) or crowd behavior on social media can be incorporated, though they often serve as priors rather than direct evidence [TV18].

Fact-checking against structured knowledge graphs requires that relevant facts exist in the graph and can be reliably retrieved. Open-text sources present the challenge of retrieving and grounding multi-hop evidence. Some datasets (e.g., FEVER) include evidence sentence annotations, while others (e.g., HeroX) demand manual verification, making large-scale training difficult [TV18].

Different formulations yield different outputs. While binary classification (true/false) is the simplest, many systems adopt more nuanced scales such as those used by journalistic platforms (e.g., mostly true, half true). Other formulations define the task as stance detection (supported, refuted, or observed), ordinal regression, or regression-based triple scoring. Some datasets require systems to not only classify the claim but also identify or generate the evidence justifying the label, adding a retrieval or ranking component to the pipeline [TV18].

Early systems in automated fact-checking primarily relied on supervised learning techniques, often treating the task as a straightforward text classification problem [TV18]. However, these methods typically lacked the capacity to incorporate external evidence or perform deeper reasoning. Over time, more sophisticated architectures have emerged. Many recent systems model fact-checking as a problem of textual entailment or natural language inference (NLI), where a claim is assessed against retrieved evidence to determine whether the evidence supports, refutes, or is unrelated to the claim [TV18].

In structured data contexts, graph-based reasoning methods have been applied, using the topology of knowledge graphs to assess the plausibility of claims. Another widely adopted strategy is distant supervision for relation extraction, which allows systems to generalize from loosely labeled training data by leveraging patterns observed in large corpora [TV18].

A particularly influential architecture is the retrieval-augmented pipeline, exemplified by the FEVER challenge. These systems follow a multi-step process in which relevant documents are first retrieved, then narrowed down to key evidence sentences, before finally being fed into a verification model. Although effective, these pipelines require carefully tuned components and are often brittle when applied to less curated or noisier domains [TV18].

In addition to evidence-based approaches, some systems attempt to match incoming claims with previously fact-checked ones from repositories such as PolitiFact or Full Fact. While efficient, this method is inherently constrained to recurring or paraphrased claims. Other models incorporate speaker profiling or contextual metadata, such as the political affiliation of the claim's source or their prior credibility score. These features, although potentially useful, raise ethical questions related to fairness, bias amplification, and the generalizability of such models to novel or neutral speakers [TV18].

Research Challenges

In addition to architectural and algorithmic challenges, recent work has also highlighted the importance of aligning AFC systems with the workflows and expectations of professional fact-checkers. [NCH+21] argue that despite the growing sophistication of automated

methods, full automation of the verification process is neither realistic nor desirable in most current real-world contexts. Professional fact-checkers not only verify factual accuracy, but also provide nuanced interpretations, contextualization, and explanations - aspects that current AI systems still struggle to reproduce faithfully.

A central recommendation is the development of human-in-the-loop systems: tools that assist rather than replace fact-checkers. These systems support claim detection, retrieval of relevant evidence, and matching to previously fact-checked claims, while leaving final judgments and publication decisions to human experts. For example, tools that rank incoming claims by "check-worthiness" (e.g., ClaimBuster, ClaimRank) or help find paraphrased versions of known misinformation have shown promise in improving efficiency without compromising editorial standards [NCH+21].

The authors also highlight a key mismatch between research priorities and fact-checkers' actual needs: while academic work often focuses on automated verification and accuracy metrics, fact-checkers are more concerned with transparency, explainability, and usability. Fact-checking organizations have reported reluctance to adopt fully automated systems due to fears of false positives and reputational risks.

Furthermore, system design must account for practical constraints, such as time pressure during breaking news, multilingual evidence sources, or the need for rapid responses to viral misinformation. Nakov et al. emphasize that tools should aim for real-time responsiveness, multimodal retrieval (including images and videos), and cross-lingual matching to help fact-checkers handle claims that are recycled across countries and platforms.

Finally, fact-checkers increasingly express interest in interpretable models, especially ones that make their decision process transparent and modifiable. Feedback loops between system suggestions and human corrections are considered a promising direction for future development.

Despite considerable progress, the field of automated fact-checking continues to face significant challenges. One of the most fundamental limitations lies in the closed-world assumption underpinning many current systems, which presupposes that all relevant facts are available and accessible within predefined corpora. This assumption fails in open-world scenarios, where knowledge is incomplete, distributed, or ambiguous [TV18].

Another key difficulty is the handling of multi-modal or complex evidence. While most systems operate exclusively on textual inputs, real-world fact-checking often involves cross-referencing images, videos, or numerical data. The development of systems capable of reasoning over such heterogeneous inputs remains an open research frontier [TV18].

Additionally, the generation of human-like justifications - explanations that are not only correct but also understandable and persuasive - has proven to be a non-trivial task. Current systems tend to rely on extractive techniques or highlight spans of text but rarely produce explanatory rationales that reflect genuine reasoning processes [TV18].

A further challenge lies in the interplay between verification and fact-checking. In journalistic practice, verifying the credibility of sources is a crucial step that precedes claim assessment. However, many automated systems either ignore this aspect or conflate source trustworthiness with factual accuracy, leading to questionable inferences [TV18].

Lastly, systems must learn to cope with ambiguous, nuanced, or intentionally misleading language. Satirical content, vague formulations, or rhetorical strategies can obscure the factual core of a claim and complicate automated judgment. Addressing such subtleties requires models that can perform abstraction, disambiguation, and contextual interpretation - capabilities that remain underdeveloped [TV18].

While much of the existing work in automated fact-checking has focused on verifying textual claims against unstructured evidence, a growing number of systems now operate on structured or semi-structured data. In these contexts, accurate comparison of literal values - such as entity names, dates, or numerical properties - becomes crucial. This shift in focus introduces a distinct set of challenges, particularly related to string similarity and data reconciliation. The following section provides an in-depth overview of techniques used to compare textual expressions, with a particular emphasis on syntactic similarity functions and their role in structured fact-checking pipelines.

3.2. Similarity Functions and Comparison Techniques

Accurately comparing textual data is a foundational requirement in fact-checking systems, particularly when reconciling structured or semi-structured sources. A wide range of string similarity measures have been developed across disciplines such as natural language processing, information retrieval, and data integration. These methods are typically divided into syntactic and semantic categories, with syntactic approaches being especially relevant when semantic models or ontologies are unavailable. To structure the discussion of syntactic techniques, we draw on several existing taxonomies and comparative studies and discuss how they can be effectively composed and applied in practice.

A variety of syntactic string similarity measures have been proposed across different domains, many of which can be decomposed into reusable components. String similarity techniques aim to determine how similar two textual expressions are - for instance, whether Barack Obama and B. Obama refer to the same entity. They form the basis for many downstream tasks, such as duplicate detection, record linkage, or ontology alignment. [GMIHF19] propose a generic and extensible framework that analyzes similarity measures in terms of three core components: character-level similarity, string segmentation, and token-level matching techniques. These components can be systematically combined to construct both established and novel similarity measures.

Character-level similarity refers to comparisons made directly on the level of characters, without regard to word boundaries. A classic example is the **Levenshtein distance**, which computes the minimal number of character edits - insertions, deletions, or substitutions - required to transform one string into another. For example, the distance between *kitten* and *sitting* is 3. Other well-known character-level methods include the **Damerau-Levenshtein distance** (which also considers transpositions) and alignment-based methods like **Smith-Waterman**, which were originally developed for bioinformatics and emphasize local similarity within substrings.

String segmentation defines how input strings are divided into smaller units prior to

comparison. Two widely used segmentation strategies are **tokenization** (splitting at whitespace or punctuation into words) and the use of **q-grams**. Q-grams are contiguous substrings of length q extracted from a string by sliding a window over it. For example, the 3-grams (trigrams) of the word *data* are `dat`, `ata`. This technique captures partial overlaps and is particularly robust to minor typos or character swaps.

Token-level matching techniques then determine how these segments (tokens or q-grams) are compared across two strings. Token-level matching builds on the idea that two strings can be considered similar if they share many of the same words or components, even if their overall structure differs. Common approaches include treating tokens as sets and applying overlap-based measures like the Jaccard index (which divides the number of shared tokens by the total number of unique tokens across both strings), or computing vector-based similarity such as cosine similarity over token frequency vectors.

This modular view enables the construction of so-called soft similarity measures, where token-level comparisons incorporate approximate character-level matching instead of relying on exact equality. A well-known example is **Monge-Elkan** similarity, which compares each token in one string with the most similar token in the other string (according to a character-level metric like Levenshtein), and then averages the resulting scores. Similarly, Soft-TF-IDF combines traditional term weighting with fuzzy matching between near-identical tokens.

A typical configuration within this framework might first tokenize the input strings into q-grams, normalize case and punctuation, compare tokens using edit distance, and aggregate the resulting scores using set overlap or soft cosine similarity - a method that extends cosine similarity by allowing approximate matches between vector dimensions. This compositional strategy allows for both fine-grained tuning to specific tasks and the flexible combination of reusable components for broader applicability.

Beyond the structural decomposition of similarity pipelines, empirical studies help identify which metric configurations are effective in practice. [CH13] offer a large-scale empirical study on the performance of various string similarity metrics in ontology alignment. Their analysis covers metrics such as Jaccard, TF-IDF, Soft TF-IDF, Levenshtein, Jaro-Winkler, and Monge-Elkan across datasets of different characteristics - standard, multilingual, and biomedical ontologies. A key insight is that while many metrics perform comparably on standard English ontologies, performance diverges considerably in multilingual or domain-specific settings. In particular, Soft TF-IDF showed consistently strong results in difficult alignment problems.

Their study also demonstrates that preprocessing strategies can significantly affect similarity performance. Tokenization, normalization, and especially translation are helpful in multilingual contexts, while synonym expansion modestly improves recall for biomedical domains. Interestingly, some commonly used strategies such as stop-word removal and stemming showed limited or even negative impact in several settings.

A particularly useful classification introduced by Cheatham and Hitzler groups string similarity metrics along three axes: global vs. local, set-based vs. whole-string, and perfect-sequence vs. imperfect-sequence. Global metrics (e.g., TF-IDF) consider corpus-

wide statistics, whereas local metrics operate only on the compared pair. Perfect-sequence metrics require exact position matching, while imperfect-sequence metrics (e.g., Jaro, Soft TF-IDF) allow positional shifts. This classification helps clarify the trade-offs between precision, recall, and computational cost.

Moreover, they point out that class labels are generally easier to match than property labels due to the syntactic complexity and variability of verbs and function words in property names. Metrics such as Monge-Elkan and TF-IDF perform better on properties, but even here optimization of thresholds specific to classes vs. properties proved beneficial.

In many of the studies surveyed, the choice of similarity threshold - the minimum similarity score required to consider two strings a match - was shown to be a critical factor in performance. Cheatham and Hitzler systematically varied threshold values to optimize F1-score, revealing that even well-performing metrics require adaptation to the specific data and task. For hybrid or asymmetric metrics (e.g., Monge-Elkan), threshold optimization was often performed in both directions (AB and BA) before selecting the minimum or maximum value. Some systems, such as those based on the stable marriage algorithm, further leverage bidirectional similarity matrices to find consistent mappings.

Earlier work by Cohen et al. [CRF] provides a detailed evaluation of string similarity techniques in the context of name matching. Their findings reinforce the strength of hybrid approaches like Soft TF-IDF, which combine token-based frequency models with approximate string matching (e.g., using Jaro-Winkler as base metric). They also emphasize practical trade-offs: while more complex methods such as Monge-Elkan can achieve high accuracy, simpler methods often provide competitive results with significantly lower computational cost.

Beyond classical string similarity techniques, more structured approaches offer complementary perspectives that become especially relevant when dealing with hierarchical, ambiguous, or richly structured data. Hierarchical similarity metrics, such as the Rada distance or Resnik similarity, leverage taxonomic structures by measuring edge distances or shared information content between concepts in a hierarchy. Information-theoretic measures like the Normalized Compression Distance (NCD) model similarity through compression efficiency and can capture structural redundancy beyond token-level patterns.

Further, refinement-graph-based approaches conceptualize similarity in terms of generalization hierarchies, where distance is defined through sequences of refinement or abstraction operations. These methods allow for the formal decomposition of structural similarity in a way that is independent of the underlying data representation.

In this broader view, many similarity measures - whether string-based, set-based, or graph-based - can be interpreted as instantiations of a few core principles: quantifying shared structure, weighting information content, or representing instances as high-dimensional fingerprint vectors. Understanding the inductive biases behind these strategies (e.g., local vs. global focus, sensitivity to order or structure) is crucial when configuring similarity pipelines for fact-checking tasks, especially in the presence of heterogeneous or domain-specific data [Ont20].

Taken together, these studies underscore the importance of context-aware metric

selection, fine-tuned threshold optimization, and careful pre-processing. They also highlight the value of viewing similarity measures as modular pipelines that can be tailored to task-specific constraints and domains. In the context of automated fact-checking over structured data, such techniques form the backbone of entity comparison and provide interpretable, tunable components for higher-level reasoning tasks.

3.3. Conflict Resolution in (Semi-)Structured Data

As automated fact-checking systems increasingly operate on structured or semi-structured data, such as RDF triples, relational tables, or knowledge graph representations, they encounter not only the need to compare data points for similarity, but also to resolve conflicting information. When multiple sources provide differing values for the same property of an entity (e.g., varying birth dates for a historical figure or contradictory ratings for a product), it becomes necessary to determine which value(s) to trust or present as correct. This process is commonly referred to as *conflict resolution* or *truth discovery*.

Unlike in unstructured fact-checking, where the task often revolves around verifying individual claims against textual evidence, conflict resolution focuses on reconciling factual inconsistencies across data representations. These inconsistencies can result from errors, outdated information, differing granularities, or fundamentally incompatible assertions. The challenge is exacerbated in open-world and web-scale settings, where provenance, trustworthiness, and completeness of data can vary widely.

In this section, we review approaches to conflict resolution in structured and semi-structured contexts. We begin by introducing the core problem formulations and the underlying assumptions (e.g., closed-world vs. open-world). We then examine major strategies including voting-based methods, source reliability estimation, similarity-driven fusion, and probabilistic inference. We also discuss the role of external signals such as metadata, schema constraints, or temporal context. Finally, we highlight open challenges, particularly those related to transparency, explainability, and dynamic data updates - issues that are highly relevant for the integration of conflict resolution into explainable fact-checking pipelines like the one pursued in this thesis.

3.3.1. Problem Formulation and Modeling Assumptions

Before exploring specific methods of conflict resolution, it is essential to understand how the task is formally modeled and what assumptions underlie different resolution strategies. At its core, the conflict resolution problem arises when multiple sources provide different values for the same data item - typically a property of an entity. For example, two datasets might provide different birthdates for the same historical figure, or different prices for the same product. The goal of conflict resolution is to determine the most plausible value(s) among the conflicting ones.

According to Bleiholder and Naumann [BN09], data fusion is the process of integrating multiple data representations of the same real-world object into a single, consistent

view. This view may take the form of a resolved data record, a set of probable values, or a distribution over possible truths. The authors categorize conflicts into several types, including value-level conflicts (contradictory values), representation conflicts (e.g., different formats or null values), and structural conflicts (e.g., schematic or identity conflicts such as missing or additional attributes). Understanding the nature of the conflict is crucial for selecting an appropriate resolution strategy.

A central modeling choice in conflict resolution is the assumption about the *completeness of knowledge*. Most early systems adopt a **closed-world assumption** (CWA), which presumes that all relevant facts are known and that any missing information can be treated as false or irrelevant. This assumption simplifies reasoning and allows for deterministic fusion strategies like majority voting or rule-based resolution. However, as Dong et al. [DBES13] argue, the CWA does not hold in many web-scale or open-domain scenarios, where information is inherently incomplete and dynamically evolving.

In contrast, the **open-world assumption** (OWA) posits that the absence of information does not imply falsity. This view is more realistic for many knowledge integration tasks, especially those involving heterogeneous or crowd-sourced data. Under OWA, conflict resolution becomes a problem of estimating plausibility under uncertainty, often requiring probabilistic models or evidence aggregation techniques. As Dong and Srivastava [DS13] discuss, data fusion in Big Data settings must handle conflicting values, the possibility of multiple truths, and high uncertainty due to heterogeneous and incomplete sources.

Recent surveys distinguish between *single-truth* and *multi-truth* models in truth discovery tasks [LGM+16, WZS+25]. While single-truth approaches assume that only one correct value exists for each data item (e.g., a person's birthdate or a products manufacturer), multi-truth models allow for the coexistence of several valid values - such as a person having multiple nationalities or a book having multiple authors. The choice between these models affects not only the evaluation criteria but also the design of fusion strategies, determining whether conflicting values are resolved through consolidation or managed through coexistence and ranking.

Taken together, these assumptions - closed vs. open world, single vs. multi-truth - define the solution space of conflict resolution systems. They influence everything from the selection of features and evidence signals to the choice of learning or aggregation strategy. In this thesis, we operate primarily under an open-world, multi-truth perspective, in which conflicting values are not merely filtered out, but compared and evaluated based on configurable similarity functions and provenance-aware logic.

3.3.2. Voting-Based Methods

One of the earliest and most intuitive approaches to conflict resolution is to select the most frequently reported value - a strategy commonly referred to as *majority voting*. The core assumption underlying this method is that the correct value is the one most often asserted across sources. In its simplest form, each source is treated equally, and the value with the highest frequency is selected as the truth.

In the literature, simple frequency-based strategies - commonly referred to as majority voting - are often used as baseline methods when no information about source trustworthiness is available [DBES13, LGM+16]. However, Bleiholder and Naumann [BN09] emphasize that conflict resolution strategies must consider the application context and the nature of the data, including possible structural or representation conflicts.

Dong et al. [DBES13] formalize voting within their data fusion framework and highlight several of its limitations. First, they note that in web-scale settings, sources are not independent, and value popularity does not necessarily correlate with truthfulness. Second, majority voting assumes a *single-truth* world model, which fails in domains where multiple correct values may coexist. To address such issues, later models introduced source weighting and content-based similarity to refine the aggregation process.

Li et al. [LGM+16] categorize voting into several variants:

- **Simple Majority Voting**, where all sources have equal weight.
- **Weighted Voting**, where source reliability is estimated externally or iteratively.
- **Threshold-based Voting**, which introduces minimum agreement levels before accepting a value.

These methods offer increasing robustness in the presence of noisy or unreliable data, but still rely on the assumption that ground truth correlates with collective agreement.

Despite its limitations, majority voting remains a useful baseline and is often used as a benchmark for evaluating more sophisticated fusion models. It is particularly effective in high-density, low-conflict domains with a strong signal-to-noise ratio. Wang et al. [WZS+25] mention that majority voting is a commonly used strategy to aggregate conflicting information from multiple sources. However, they highlight that the effectiveness of such methods can be limited in scenarios with dependent sources, as often encountered in crowdsourcing settings.

In the context of fact-checking over structured data, voting strategies can serve as a fallback mechanism or a first-pass filter, especially when similarity between values is high. However, in more ambiguous settings, they are often insufficient without being combined with additional heuristics, source profiling, or semantic reasoning.

3.3.3. Source Reliability Estimation

A major limitation of majority voting is the implicit assumption that all sources are equally reliable. In practice, however, data sources differ greatly in terms of accuracy, consistency, and independence. To address this, a wide range of methods have been developed that estimate the reliability of each source - either beforehand (e.g., via metadata) or during the conflict resolution process itself.

One of the most influential approaches in this domain is *TruthFinder*, introduced by Yin et al. [YHY07]. *TruthFinder* applies an iterative algorithm that jointly estimates source trustworthiness and the confidence of individual claims. It initializes all source

trust scores equally and then refines them based on the consistency of their claims with the majority. Claims supported by highly trustworthy sources are given higher confidence, and vice versa, creating a feedback loop between source reliability and claim correctness.

A central question in truth discovery is how the reliability of data sources is determined. Some systems rely on external information to assess trustworthiness, while others infer it from the data itself. This leads to a conceptual distinction between two types of approaches, which can be broadly described as static and dynamic.

While Li et al. [LGM+16] do not explicitly use this terminology, their survey describes a range of methods that naturally fall into these two categories. The distinction arises from how a system obtains or estimates the trustworthiness of sources during the truth discovery process.

- **Static approaches** rely on external or prior knowledge about a sources reliability. These may include domain-specific heuristics, historical accuracy records, or platform-based indicators such as user reputation scores or verified status. For example, in some applications, government or institutional data sources may be assigned a higher trust level by default. Li et al. mention such strategies when describing initialization methods based on labeled truths or metadata (Section 3.2.6). These methods assume a fixed level of trust, independent of the current dataset under analysis.

- **Dynamic or data-driven approaches** estimate source reliability iteratively or probabilistically during the fusion process itself. This is the dominant strategy in most unsupervised truth discovery systems, including models like TruthFinder [YHY07], Investment [LGM+16], and CATD. These systems begin with uniform or heuristic initializations and refine source trustworthiness based on consistency with other sources and inferred truths. The idea is that reliable sources tend to agree with each other and with the emerging consensus, enabling feedback loops that mutually reinforce claim confidence and source trust.

This categorization captures a core design choice in truth discovery: whether source reliability is injected from outside the system or inferred from within. While static methods can leverage valuable domain expertise, dynamic approaches allow for more flexible, context-sensitive modeling - especially in open environments where prior reliability indicators may be unavailable or misleading.

Iterative methods are particularly prominent among dynamic approaches. One representative example is the Latent Truth Model (LTM) proposed by Zhao et al. [ZRGH12], which adopts a Bayesian framework to jointly infer both the reliability of sources and the correctness of individual claims. Unlike earlier models that use scalar trust scores, LTM models each sources error behavior via two independent parameters: sensitivity (true positive rate) and specificity (true negative rate). This two-sided modeling enables the system to distinguish between omission and fabrication, making it especially effective in domains with noisy, sparse, or biased input data.

Wang et al. [WZS+25] emphasize that accurate source reliability estimation is crucial in domains with uneven data quality or potential information cascades. They also note that advanced models often incorporate prior knowledge (e.g., topic-specific trust, time decay), and that combining source-based and claim-based features yields better results than relying on either in isolation.

In structured fact-checking systems, source reliability estimates can be used not only to resolve conflicts, but also to prioritize evidence, rank alternative values, or filter low-quality inputs. However, these methods also raise ethical and technical challenges - including fairness, bias amplification, and the transparency of trust scoring mechanisms. As such, explainability and user control are increasingly important design considerations.

3.3.4. Similarity-Driven Fusion

In many real-world settings, conflicting values are not strictly contradictory, but differ only slightly in format, spelling, or granularity. For instance, the strings *Barack Obama* and *B. H. Obama* may refer to the same entity, while *January 1, 2020* and *01/01/2020* represent the same date. Traditional fusion techniques that rely on exact matching would treat such values as incompatible. To overcome this limitation, similarity-driven fusion approaches incorporate string or semantic similarity into the conflict resolution process.

Such methods typically follow a multi-step pipeline: First, candidate values are compared using a similarity function. Then, similarity scores are used either to cluster near-duplicate values or to weight them in an aggregation scheme. This can improve conflict resolution by allowing semantically equivalent or syntactically similar values to reinforce each other, even if they do not match exactly.

Bleiholder and Naumann [BN09] describe how data fusion techniques can operate on clusters of duplicate records obtained through similarity-based detection methods. In their HumMer system, conflicting values within such clusters are resolved using rule-based aggregation functions such as majority vote, min, or concept-based generalization. While the paper does not explicitly address similarity-aware fusion in the narrow sense, the use of similarity metrics during the duplicate detection phase indirectly shapes which values are grouped together for fusion.

Ontañón et al. [Ont20] provide a comprehensive overview of similarity and distance functions for structured data, such as graphs, frames, or logical clauses. They highlight that, unlike propositional data, structured representations often require domain-specific or ad-hoc similarity functions. These choices inherently reflect assumptions about what constitutes similarity in a given context. As the authors note, such assumptions vary across domains and influence how structured inputs are interpreted and compared in AI systems.

In structured fact-checking scenarios, similarity-driven fusion is particularly important for resolving soft conflicts - that is, cases where values are not strictly equal or opposite, but exhibit partial agreement. Designing configurable similarity functions is therefore essential for supporting transparent and context-sensitive conflict resolution - a goal that is pursued in this thesis.

3.3.5. Probabilistic Inference

While voting and similarity-based methods provide valuable heuristics for conflict resolution, they often lack an explicit model of uncertainty. Probabilistic approaches address this gap by representing truth and source reliability as latent variables, and by formulating the fusion process as a statistical inference problem. These models aim not only to resolve conflicts, but also to quantify the confidence of their decisions, making them particularly suitable for noisy, ambiguous, or large-scale data integration tasks.

One of the foundational models in this space is the Latent Truth Model (LTM) proposed by Zhao et al. [ZRGH12], which uses a Bayesian framework to jointly infer the truth of claims and the sensitivity and specificity of each source. By modeling both false positives and false negatives, LTM captures asymmetries in source behavior that are invisible to simpler models. This allows the system to distinguish between sources that are overly inclusive (low specificity) and those that are overly conservative (low sensitivity).

Building on this idea, more recent models incorporate richer dependencies between data sources. Dong et al. [DBES13] extend truth discovery by modeling copying relationships through probabilistic inference. Their Bayesian framework estimates the likelihood that one source copies from another based on shared false values - a strong signal for dependence. This improves robustness against correlated errors and information cascades, which can otherwise mislead naive or majority-based methods.

Some variants, such as ACCUCOPY and ACCUCOPYSIM, further integrate source accuracy and copying detection into an iterative algorithm. This allows more accurate computation of confidence scores for candidate values and better reflects real-world dependencies between data providers.

Probabilistic methods can be extended with additional constraints or priors, including domain knowledge, ontological hierarchies, or temporal plausibility. Some systems integrate logic-based rules or soft constraints into probabilistic graphical models to encode expectations about consistency or co-occurrence. For example, if two conflicting values belong to mutually exclusive categories, this can be captured by structural priors in a Bayesian network.

Wang et al. [WZS+25] argue that probabilistic truth discovery is especially promising for dynamic and open environments, where evidence may be incomplete, noisy, or adversarial. However, these models also present practical challenges: they are often computationally intensive, sensitive to hyperparameters, and may require domain-specific tuning to avoid overfitting or instability.

In fact-checking systems that operate over structured or semi-structured data, probabilistic models offer a principled way to weigh conflicting evidence, account for uncertainty, and expose confidence scores that can inform downstream decisions. Their ability to integrate diverse sources and signals makes them a powerful complement to similarity-based and voting-based techniques.

3.3.6. Metadata, Constraints, and Contextual Signals

Beyond source reliability and value similarity, many systems incorporate external signals to guide conflict resolution. These signals include metadata about the sources, schema-level constraints, and contextual features that influence how information should be interpreted or weighted. While not always available, such auxiliary information can significantly improve both the accuracy and interpretability of fusion outcomes.

Metadata provides useful cues about data quality and relevance. For example, timestamps can help prioritize recent or temporally consistent values, especially in domains with time-sensitive facts. Provenance information - such as the publishing organization, geographic origin, or document type - may influence trust estimates. In structured datasets, attribute-specific features like update frequency or known value ranges can also inform conflict resolution. Wang et al. [WZS+25] emphasize that dynamic truth discovery requires accounting for temporal evolution and changing source quality. While not always explicitly modeled as metadata, features like timestamps, update frequency, and source behavior over time are crucial for achieving robustness in streaming or rapidly evolving environments.

In addition to metadata, schema- and domain-level constraints can play a central role in detecting and resolving conflicts. Structural constraints such as functional dependencies, cardinality restrictions, or domain constraints enable systems to identify implausible or contradictory values. For instance, if an attribute is known to be single-valued - such as a birth date - the occurrence of multiple conflicting entries can trigger resolution strategies or indicate data errors. Fan [Wen] demonstrates how classical constraints like functional and inclusion dependencies, as well as their conditional extensions (e.g., conditional functional dependencies), can be leveraged to detect and repair inconsistencies in real-world data. These formal mechanisms provide a principled foundation for evaluating the plausibility of conflicting claims, especially when domain-specific rules can be encoded declaratively.

Moreover, the use of such constraints is not only effective for validation but can also guide automated repairs and improve the confidence in selected values. In fact, Fan's work shows that even partial or conditional constraints - defined to apply only in certain subsets of the data - are valuable tools in maintaining high data quality in the presence of noisy, incomplete, or conflicting information.

Conflict resolution in (semi-)structured data is a multifaceted challenge that requires the integration of diverse signals - from statistical voting and reliability estimation to similarity metrics, probabilistic reasoning, and contextual metadata. While early methods relied on simple heuristics, modern approaches increasingly emphasize adaptivity, semantic awareness, and explainability. Importantly, the choice of resolution strategy depends not only on the data domain but also on the goals of the system - whether robustness, interpretability, or scalability is prioritized.

For fact-checking pipelines, these techniques are particularly relevant: source trustworthiness, claim similarity, temporal dynamics, and structural constraints all influence how conflicting claims are resolved. Understanding these foundations is essential for

integrating conflict resolution into transparent and effective fact-checking architectures - including the one developed in this thesis.

3.4. Existing Frameworks and Tools

The previous sections outlined key techniques for resolving conflicting information in structured and semi-structured data. However, effectively applying these methods in practice requires not only algorithmic strategies, but also tools and frameworks that enable modular development, configuration, and evaluation. This section surveys existing systems that support tasks such as entity resolution, truth discovery, and similarity-based reasoning - either as research prototypes or as reusable toolkits.

We structure the discussion into three categories: general-purpose toolkits for entity matching, specialized truth discovery frameworks, and libraries for similarity computation. For each, we examine their architectural scope, extensibility, and relevance to fact-checking tasks. The goal is to identify existing solutions that inform the design of fact-checking infrastructure, as well as current limitations that motivate the development of a new modular framework - presented in the following section.

3.4.1. General-Purpose Integration and Fusion Frameworks

Several systems have been developed to support data fusion tasks within broader data integration pipelines. These frameworks typically combine schema alignment, duplicate detection, and value-level conflict resolution, often in interactive or semi-automatic settings. Two representative examples are the *HumMer* and *Ajax* systems, both described in detail by Bleiholder and Naumann [BN09].

HumMer (short for Humboldt-Merger) supports the virtual integration of heterogeneous relational sources. The system incorporates schema matching, duplicate detection, and a flexible fusion operator framework. Fusion functions include simple heuristics like *Vote*, numerical aggregations such as *Min* and *Max*, and more advanced semantics like *Most Abstract Concept* based on external taxonomies. HumMer supports both interactive and batch-oriented workflows and allows users to express fusion operations using SQL extensions - enabling seamless integration with standard querying environments [BN09].

The *Ajax* system takes a declarative approach to data fusion. It provides a logical workflow language that extends relational algebra with fusion-specific operators for formatting, matching, and merging. These logical workflows are compiled into executable Java programs. Ajax emphasizes reusability and customizability: operators can be adapted or replaced, and exception-handling mechanisms enable human-in-the-loop feedback during the fusion process. This makes it especially suitable for complex or evolving data integration scenarios [BN09].

Both systems illustrate key architectural ideas - including modularity, operator extensibility, and integration with standard database technology - that continue to inform the design of modern fact-checking frameworks.

3.4.2. Truth Discovery Frameworks

A variety of frameworks have been proposed to implement and evaluate truth discovery algorithms in practice. These systems often provide reusable implementations of core models, support data preprocessing and evaluation, and enable customization for specific domains. Among them, several frameworks stand out for their architectural scope and methodological diversity.

One influential framework is *CATD* (Confidence-Aware Truth Discovery), introduced by Li et al. [LGM+16]. CATD addresses the challenge that many data sources provide only sparse or noisy claims, making point estimators for source reliability unreliable. Instead, CATD estimates *confidence intervals* for each sources reliability, allowing for more nuanced reasoning under uncertainty. This probabilistic modeling is especially valuable in settings where claim coverage is uneven and classical majority-based methods fail.

Another approach is the *Investment model*, which conceptualizes truth discovery as a resource allocation process. Sources invest their reliability across the values they provide, and the confidence of each claim grows based on the combined investments from all sources. In turn, sources collect credibility back from the identified truths, creating a feedback loop between claim accuracy and source weight. This non-linear reinforcement mechanism allows for robust convergence even in the presence of conflicting information.

The *Latent Truth Model (LTM)*, proposed by Zhao et al. [ZRGH12], is a probabilistic graphical model that introduces a two-sided view of source errors: it models both false positives and false negatives. This distinction allows LTM to perform well in multiple-truth scenarios, where a claim may be incomplete rather than incorrect. LTM has been widely adopted and extended, often serving as a baseline in evaluations of newer truth discovery systems.

3.4.3. Libraries for Similarity Computation

Similarity functions play a central role in conflict resolution, entity matching, and fact validation. While many frameworks embed domain-specific similarity logic, there also exist standalone libraries and toolkits that provide reusable, configurable similarity measures. These libraries often focus on syntactic or semantic comparisons between strings, concepts, or records and are crucial for building explainable and tunable fact-checking systems.

One of the most widely cited libraries is *SimMetrics*, an open-source Java library that implements a broad range of string similarity metrics, including Jaccard, Levenshtein, MongeElkan, SmithWaterman, and others[1]. Its modular design and extensibility make it useful for quick experimentation or integration into larger pipelines. Cheatham and Hitzler [CH13] evaluated such measures extensively in the context of ontology alignment, showing how the choice of metric can significantly affect downstream matching performance.

In the context of semantic similarity, libraries such as *WordNet::Similarity* [PPM04] and *Sematch* [ZIF15] offer concept-level comparison using lexical databases or knowledge graphs. These tools compute path-based, information-theoretic, or embedding-based

[1] https://github.com/Simmetrics/simmetrics

distances between entities and are particularly useful for aligning partially structured claims, such as named entities or types.

Other libraries, like *FuzzyWuzzy* (Python), offer user-friendly APIs for fuzzy string matching using token sorting and ratio comparisons - useful for noisy or unnormalized inputs. More recently, embedding-based libraries like *Sentence-BERT* provide similarity estimates between full sentences or claims, supporting context-aware matching in NLP-driven fact-checking settings.

While many of these tools focus on pairwise similarity, few support explainable composite functions or modular predicate composition. This motivates the need for configurable predicate frameworks, such as the one developed in this thesis, that bridge the gap between low-level metrics and higher-level reasoning in structured fact verification.

In summary, existing frameworks offer valuable tools and abstractions for various facets of data fusion and truth discovery. However, they often fall short in supporting modular, explainable, and predicate-based integration tailored to fact-checking needs. The following section introduces a novel framework that addresses these gaps through configurable predicate families and precision-aware metrics.

3.5. FactCheck

While the previous section highlighted frameworks that support entity matching, truth discovery, or similarity-based reasoning, these tools often focus on creating consistent fused representations or assessing claim veracity, without addressing the correction of data at its source. The FactCheck framework, developed at the University of Vienna by Kalchgruber et al. [KKJM18], offers a distinct approach: it not only detects conflicting structured data across web sources but also enables users to identify, annotate, and suggest corrections for inconsistencies directly within the web ecosystem.

FactCheck targets semantically rich embedded data formats like JSON-LD and Microdata, which are increasingly prevalent due to the adoption of schema.org standards. Unlike traditional fusion approaches that generate a new clean representation (thus leaving conflicts in the original sources unaddressed), FactCheck emphasizes the recognition and rectification of conflicts at the point of consumption. This is achieved through a modular architecture combining automated similarity predicates, a formalized logic for conflict resolution, and interaction with web users and content providers.

The framework consists of several interdependent components - including the FactServer, FactBase, FeedbackBase, Precision Metric Manager, and an Information Dashboard - and is operationalized through IdaFix, a browser-integrated tool designed to highlight, explain, and gather feedback on conflicting facts. This integration of formal semantics, predicate-based similarity functions, and user-facing feedback mechanisms positions FactCheck as a pioneering infrastructure for maintaining data consistency on the web, complementing existing fact-checking and data integration systems.

Information Model and Similarity Predicates

At the core of FactCheck lies a formalized information model that captures structured data from web sources as subjectpredicateobject triples. Each data source s provides information about a set of objects o, each associated with a set of facts f, such as `ReleaseDate` or `Author`. Facts take on values v from a valid domain (e.g., date, string, or number), and may conflict across sources describing the same entity.

To identify such conflicts, FactCheck introduces the notion of *similarity predicates*, which determine whether two fact values are considered equivalent, conflicting, or compatible under certain conditions. These predicates are organized into five predefined categories: generic (e.g., string equality), abstraction (e.g., taxonomic hierarchy), interval (e.g., numerical proximity), date (e.g., tolerable temporal deviations), and language (e.g., cross-lingual equivalence).

For example, the predicate $p_{4.3} \in c_4$ (date category) may specify that two dates within a three-day range are non-conflicting. This allows for fine-grained, interpretable definitions of similarity that reflect both syntactic and semantic closeness. Each fact f can be assigned a logical combination of such predicates - a *precision metric* - which encodes the acceptable variation for that fact in a given context.

This formalization enables FactCheck to go beyond exact matching and instead reason about contextual or domain-specific equivalence. The flexibility of defining and combining predicates makes the system highly adaptable to different use cases and quality requirements.

System Architecture

The FactCheck framework is designed as a modular architecture that supports the detection and resolution of conflicting structured data across web sources. It consists of five main components, each with a well-defined responsibility in the processing and reconciliation pipeline:

- **FactBase**: A central repository that stores all extracted facts and associated metadata. It serves as a long-term cache that grows over time and enables cross-source comparisons across previously visited pages or entities. FactBase forms the central database of the FactCheck framework.
- **FactServer**: The core reasoning engine of the system. It performs comparisons between facts using the defined similarity predicates and precision metrics. The FactServer determines whether two values are conflicting or not and prepares results for presentation to the user. The FactServer is the most important core component of the FactCheck system and is responsible for entity matching, data cleansing, data normalization and the comparison logic. It extracts entities from structured data, cleans processes them, and prepares them for comparison.
- **FeedbackBase**: This component captures and stores user feedback on proposed fact classifications. If a user confirms or rejects a suggested conflict, this information

is retained and used to improve precision metric configurations over time.

- **Precision Metric Manager**: A learning module that dynamically updates and refines precision metric classes based on user interactions. It enables the system to adapt to evolving definitions of similarity and tolerance in specific domains or use cases.

- **Information Dashboard**: A user-facing interface for data owners and maintainers. It displays conflict reports, provides diagnostics about structured data, and offers actionable suggestions for correcting or updating inconsistent information.

All components are implemented in Python and are designed to integrate with web infrastructure such as browsers and content management systems. The architecture supports both automated and interactive workflows, making FactCheck suitable for real-time assistance and offline analysis alike. The modular design also facilitates extensibility, allowing new predicate types, data domains, or user feedback mechanisms to be added with minimal effort.

While the FactCheck system comprises multiple components, the focus of this thesis lies specifically on the *FactServer*, as it constitutes the central module responsible for evaluating similarity and resolving factual conflicts. The FactServer implements the logical core of the system: it operationalizes the comparison of fact values based on configurable *similarity predicates* and *precision metric classes*, and decides whether values from different sources are deemed equivalent, compatible, or contradictory.

In the course of this master's project, the FactServer was substantially extended and refined. These enhancements include the design and implementation of new predicate categories, the formal specification of custom similarity functions, and the development of a flexible predicate framework that supports modular combination, runtime configuration, and user-driven adaptation. By concentrating on the FactServer, this work directly contributes to the core reasoning capabilities of FactCheck, laying the foundation for more accurate, explainable, and domain-sensitive conflict resolution in structured data.

IdaFix User-Facing Frontend

Although not the focus of this thesis, the FactCheck framework includes a frontend component called *IdaFix*, which serves as the primary user interface for interacting with conflicting structured data. IdaFix is implemented as a browser extension and acts as a visual layer that notifies users when structured data on a visited web page conflicts with equivalent data from other sources.

The extension supports several user-centric features: it identifies equal, conflicting, locally missing, and remotely missing facts, and displays them in an intuitive tabular view. Users can inspect individual fact pairs, provide feedback on proposed conflicts, and trigger notifications to data source owners. This feedback is captured and stored in the *FeedbackBase*, enabling a form of lightweight crowdsourcing that informs future similarity assessments.

While this thesis does not address the frontend layer in detail, the integration of user feedback into the FactCheck backend illustrates the systems commitment to explainability and adaptivity. The existence of IdaFix emphasizes that structured data quality is not merely a backend concern, but also a matter of user perception and interaction - a perspective that complements the architectural emphasis of this work on reasoning mechanisms within the *FactServer*.

To summarize, the FactCheck system essentially consists of three core components: **FactBase**, **FactServer**, and **IdaFix**. Each of these modules fulfills a distinct role in the overall fact-checking pipeline and interacts with the others via clearly defined interfaces. Figure 3.1 provides a high-level visualization of the basic interaction between these components.

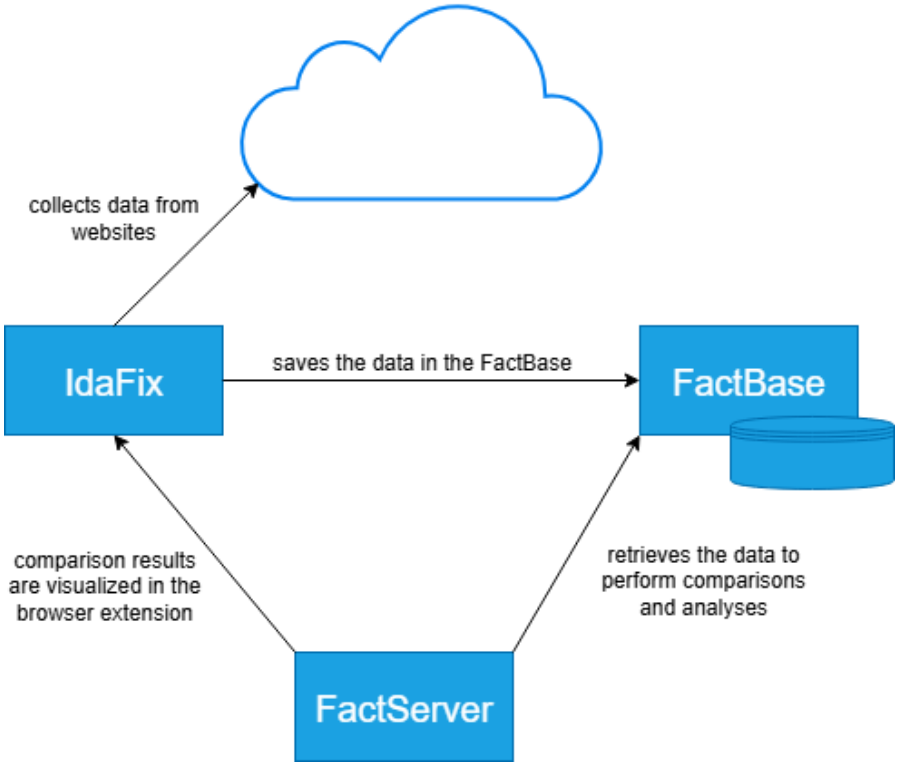


Figure 3.1.: High-level architecture of the FactCheck system

3.6. Summary and Research Gap

The preceding chapter has reviewed a broad spectrum of prior work relevant to the automated verification of structured data. Across domains such as automated fact-checking, string similarity computation, and conflict resolution in semi-structured datasets, significant progress has been made - both at the algorithmic level and in the development

of supporting toolkits and infrastructures. However, several limitations remain that motivate the work presented in this thesis.

First, while automated fact-checking systems have evolved to handle complex pipelines - including claim detection, evidence retrieval, and justification generation - most focus primarily on unstructured textual data. Structured data, particularly in web-scale environments using formats like JSON-LD and Microdata, has received comparatively less attention, despite its growing prevalence and importance in knowledge-based applications. When structured data is addressed, the focus is often on fusion into coherent views, not on identifying and correcting conflicting values at their source.

Second, string similarity measures have been thoroughly studied, with numerous syntactic and semantic metrics developed and evaluated. However, most tools for applying these measures in practice (e.g., SimMetrics, FuzzyWuzzy) offer limited flexibility in combining similarity components, reasoning over predicate logic, or adapting similarity thresholds to task-specific requirements. More advanced systems capable of composing and tuning similarity functions remain rare and are often tightly coupled to specific domains or datasets.

Third, existing conflict resolution frameworks often lack explainability and user-level configurability. Methods such as majority voting or probabilistic inference provide aggregate judgments about factual conflicts but offer limited transparency regarding the criteria or thresholds involved. Especially in open-world, multi-truth scenarios - where conflicting values may be valid in different contexts - such black-box reasoning is insufficient for applications that demand traceable, domain-sensitive decision-making.

In this landscape, the FactCheck framework introduces a novel architectural perspective: it formalizes conflict detection in terms of configurable similarity predicates and precision metrics, enabling explicit, user-controllable definitions of what constitutes a factual conflict. Its modular architecture separates the logical reasoning (FactServer) from user interaction (IdaFix) and feedback integration (Precision Metric Manager), creating a system that is not only adaptable and extensible but also conducive to human-in-the-loop refinement.

However, the original FactCheck prototype, as presented by Kalchgruber et al. [KKJM18], lacks several features that are crucial for its evolution into a robust and generalizable infrastructure:

- The predicate and metric definition logic in the original FactServer is limited in scope and flexibility. It provides a static set of predicates and hardcoded precision metrics, offering little support for dynamic composition, runtime modification, or formal introspection of similarity logic.
- The integration of domain-specific similarity knowledge (e.g., date fuzziness, taxonomic abstractions) is possible but cumbersome, lacking the abstractions needed for composable, interpretable predicate design.
- While the system anticipates user feedback for metric refinement, it provides no interface or formal mechanism for defining new predicate families or reconfiguring

existing metrics based on domain-specific requirements.

This thesis addresses these gaps by focusing specifically on the design and implementation of a modular predicate framework within the FactServer component of FactCheck. The proposed framework enables:

- The definition of reusable predicate families, categorized by similarity strategy (e.g., exact match, numeric tolerance, substring containment).
- The specification of precision metrics as logical compositions over predicate functions, using intuitive Boolean operators.
- The integration of user-defined predicates and runtime configuration interfaces to support domain adaptation and explainability.
- A clear interface for downstream components to interpret and trace similarity decisions, supporting the development of explainable fact-checking systems.

By extending and generalizing the FactServer in this way, this thesis lays the groundwork for a next-generation fact-checking infrastructure capable of resolving structured data conflicts in a transparent, flexible, and user-informed manner. The resulting system serves not only as a proof of concept but also as a foundation for broader research into explainable, predicate-driven similarity reasoning across web data ecosystems.

In the next section, we present the core contribution of this thesis: the design of an extensible predicate-based framework for structured data comparison, integrated into the FactServer component of the FactCheck system. This framework addresses the limitations identified in the previous section by introducing configurable predicate families, composable similarity functions, and dynamically adjustable precision metrics. We detail the architectural design choices, the modular predicate logic, and the runtime interface that enables users or developers to define, combine, and evaluate similarity criteria transparently. By focusing on the logical reasoning core of FactCheck, this framework lays the foundation for more accurate, explainable, and domain-adaptable conflict resolution in structured fact-checking pipelines.

4. Design

This chapter presents the conceptual and architectural design of the FactServer and the proposed framework for similarity functions and precision metrics that extends the existing FactCheck comparison framework. Building upon the limitations identified in the state of the art, as discussed in the previous chapter, this work aims to develop a modular, extensible, and domain-aware approach to structured data comparison.

The design focuses on enabling flexible and semantically rich similarity assessments by introducing new predicate functions, configurable precision metrics, and a system architecture that allows dynamic integration of these components into the existing FactCheck infrastructure in the form of a web-based management interface. The framework was developed with the goals of extensibility, domain-independence, and transparency in mind, ensuring that new functions and metrics can be added or adapted with minimal effort.

The chapter is structured as follows: we begin by describing the overall system architecture of the existing FactServer, detailing the role, structure, and logic of similarity predicates and explaining how they are combined into precision metrics. We also present concepts and tools that were used to implement similarity predicates in this master project for the extended framework, such as the use of local LLMs or SKOS, followed by design goals defined with regard to the extended framework. We also discuss the metadata configuration format and its role in managing similarity behavior. We will briefly discuss the existing API endpoints and the format that fact sets must have so that the FactServer can perform comparisons. Finally, the web-based predicate and precision metric editor, which is the heart of the framework, is presented.

4.1. System Architecture Overview

The proposed framework is built as an extension to the existing FactCheck architecture, which itself consists of several interrelated components including the FactServer, FactBase, and the frontend component IdaFix. The FactBase is the persistent database component within the FactCheck system. It stores structured factual data that is analyzed and compared by the FactServer. While the FactServer handles incoming comparison requests, the FactBase serves as the central repository for all known facts. During a comparison, the FactServer retrieves relevant entries from the FactBase and evaluates them using the configured precision metrics, and predicate logic. New facts can also be stored in the FactBase via the FactServer, enabling the system to build and maintain a continuously growing knowledge base for future comparisons. The core of the enhancement lies in the modular integration of similarity predicates and precision metrics into the FactServer.

At its foundation, the FactServer is implemented using Azure Functions in Python, following a modular controller-manager pattern. Each request received by the API gateway (exposed via the entry point `function_app.py`) is routed to an appropriate controller, which in turn delegates the logic to a system manager or a functional module. Azure Functions is a serverless computing service provided by Microsoft that allows developers to run small, event-driven pieces of code (called functions) without managing infrastructure.[1] In the FactCheck system, Azure Functions serve as the foundation of the FactServer. Each API request - such as a comparison call - is treated as an event and handled by a dedicated function. This architecture enables lightweight, scalable, and modular processing, making it well-suited for managing separate comparison modules and controllers.

To ensure loose coupling and extensibility, the functionality of the comparison logic was integrated into this architecture as follows:

- Predicate logic is implemented in a dedicated Python module (`predicate.py`), where reusable similarity functions are grouped into predicate families.
- Precision metrics are defined in a structured JSON file (`pm_data.json`), enabling composition of multiple predicate functions without changing any source code.
- The mapping between structured data properties and their assigned precision metrics is defined in a separate configuration file (`propertydata.json`).

These components interact at runtime to enable dynamic and domain-aware comparisons of structured data facts. New similarity logic can be added simply by writing a function in `predicate.py`, registering it in `pm_data.json`, and assigning it in `propertydata.json`. No redeployment or recompilation is required.

The overall system architecture of the FactServer, as it stood in May/June 2025, is illustrated in Figure 4.1, and the following subsections describe the key integration points in detail.

[1] https://azure.microsoft.com/de-de/products/functions

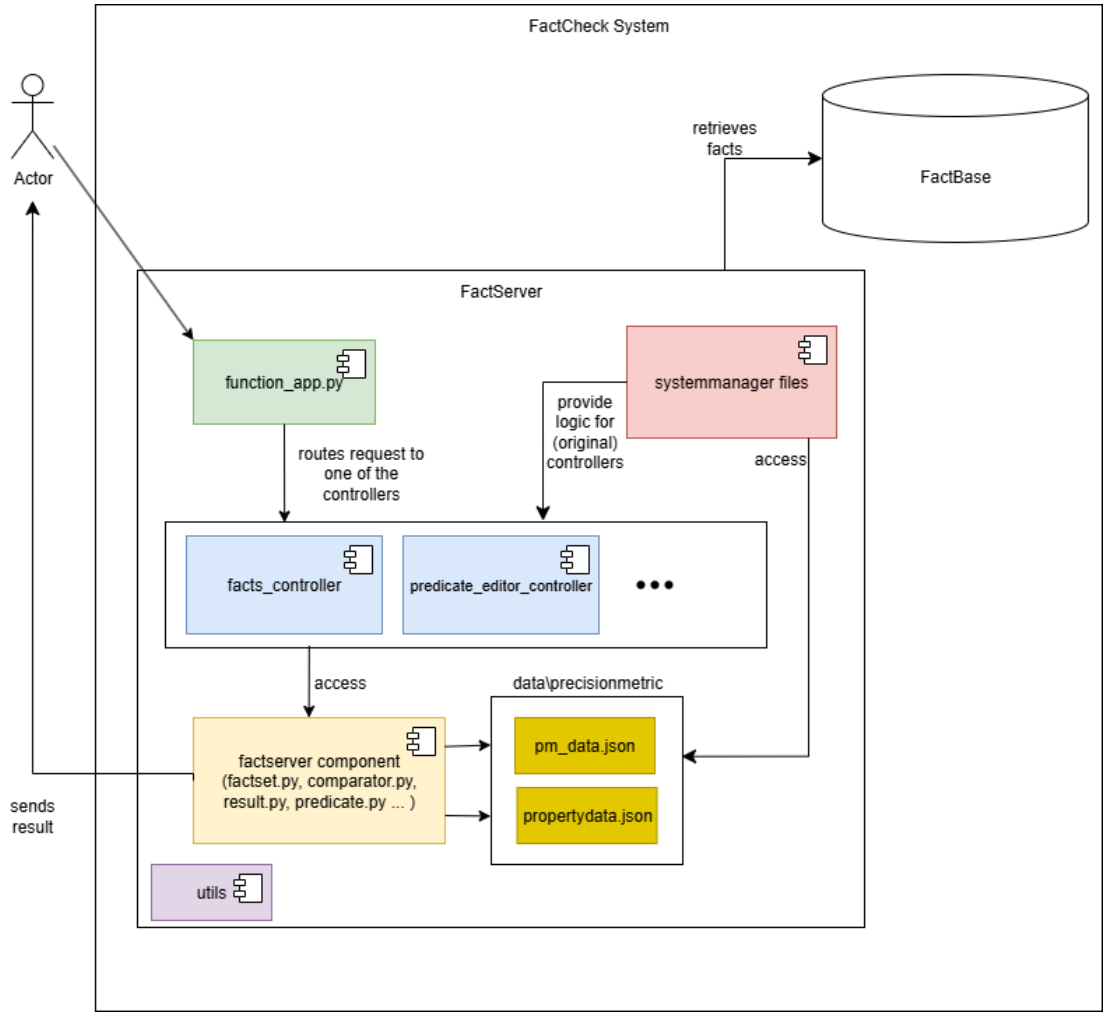


Figure 4.1.: System architecture of the FactCheck system showing interactions, with a focus on the FactServer.

Design Rationale

Since the existing FactServer was already implemented in Python and followed a modular Azure Functions architecture, it was a natural choice to continue using Python for the implementation of new predicate functions. Python offers several advantages in this context:

- **Ease of prototyping:** Pythons dynamic typing and expressive syntax make it ideal for rapidly implementing and testing new comparison logic.
- **Rich ecosystem:** Libraries such as `dateutil`, `re`, or `difflib` simplify the handling of common parsing and normalization tasks.

- **Interoperability:** Python allows seamless integration with tools like Ollama for LLM-based predicates and SPARQL endpoints for SKOS queries.
- **Consistency:** Keeping the entire logic in the same language avoids the overhead of cross-language integration or service orchestration.

Similarity Predicate Design

Similarity Predicates define the logic used to compare individual data values or structures. Each predicate belongs to a specific *Predicate Family* (e.g., 'StringFamily', 'ObjectFamily', 'DateFamily') that determines its expected input types and comparison logic. These families are defined in the `predicate.py` module and grouped based on the semantic nature of the values to be compared.

For instance, the 'StringFamily' includes functions like `titleContained`, which checks whether one string contains another, ignoring case sensitivity and subtitles, while the 'ObjectFamily' includes comparisons based on shared fields such as 'name'. To support complex real-world data, some predicates incorporate normalization strategies, such as unit conversion for temperature or wind speed in the 'ObjectFamily'. Additionally, advanced families like 'LLMComparisonFamily' and 'SKOSFamily' utilize semantic reasoning, either via local language models (e.g., Ollama) or external knowledge graphs (e.g., DBpedia), to handle fuzzy or multilingual matches.

Precision Metric Design

Precision Metrics define how the outputs of multiple predicate comparisons are aggregated into an overall similarity judgment. Each metric consists of an operator (e.g., logical AND, OR) and a list of predicate-function combinations that must be satisfied for a data pair to be considered a match. The metric configuration is stored in a JSON structure (see `pm_data.json`) and processed at runtime by the systems evaluation engine. Metrics can combine different predicate families and functions to reflect complex matching strategies for example, comparing names strictly while allowing minor deviations in numerical fields. The modular design enables flexible experimentation with alternative metrics and encourages fine-tuning based on data characteristics. In the frontend, metrics can be added or edited dynamically, ensuring a smooth integration of new comparison logic without requiring backend changes.

4.1.1. Predicate Integration via `predicate.py`

All comparison logic is implemented in the file `predicate.py`, which defines predicate functions grouped by families. Each function receives two input values (typically strings, numbers, or dates) and returns a boolean indicating whether the values can be considered equivalent.

An example of such a predicate is the `sameDateFormatted` function, which compares two dates independent of their original format (see Listing 4.1). This function is part of the `DateFamily` and demonstrates the framework's ability to normalize representations

prior to comparison. It is an example of one of the functions implemented in this master's thesis to improve the comparison logic of the fact-check system.

```
@staticmethod
def sameDateFormatted(v1, v2):
    """D4 - compares two date values, independent of the format"""

    import dateutil.parser

    def parse_date(date_str, dayfirst=None):
        try:
            return dateutil.parser.parse(date_str, dayfirst=dayfirst).
    date()
        except ValueError:
            return None

    if "-" in str(v1) and "-" in str(v2):
        try:
            return dateutil.parser.parse(str(v1)).date() == dateutil.
    parser.parse(str(v2)).date()
        except ValueError:
            return False

    formats_v1 = []
    formats_v2 = []

    if "." in str(v1):
        formats_v1.append(parse_date(str(v1), dayfirst=True))
    elif "/" in str(v1):
        formats_v1.append(parse_date(str(v1), dayfirst=True))
        formats_v1.append(parse_date(str(v1), dayfirst=False))
    else:
        formats_v1.append(parse_date(str(v1)))

    if "." in str(v2):
        formats_v2.append(parse_date(str(v2), dayfirst=True))
    elif "/" in str(v2):
        formats_v2.append(parse_date(str(v2), dayfirst=True))
        formats_v2.append(parse_date(str(v2), dayfirst=False))
    else:
        formats_v2.append(parse_date(str(v2)))

    for date1 in formats_v1:
        for date2 in formats_v2:
            if date1 and date2 and date1 == date2:
                return True

    return False
```

Listing 4.1: Predicate Function for Date Comparison

The `sameDateFormatted` function attempts to compare two input values as dates, regardless of their original formatting. It first checks whether both values use the ISO format (`YYYY-MM-DD`) and compares them directly. Otherwise, it attempts to parse

each value using common alternatives such as `DD.MM.YYYY`, `DD/MM/YYYY`, or `MM/DD/YYYY`, producing a list of possible date interpretations for each input. All resulting date combinations are compared, and the function returns `True` if any pair matches. If parsing fails or no match is found, it returns `False`. This approach ensures robustness against heterogeneous date representations in real-world data. For further details on the implementation of similarity predicates, see Chapter 5.

4.1.2. Precision Metric Configuration via `pm_data.json`

Precision metrics are declaratively defined in the file `pm_data.json`, where each metric has a unique identifier (e.g., "D4"), an optional logical operator, and a list of predicate functions that belong to a predicate family.

The following example (Listing 4.2) shows how the `sameDateFormatted` predicate is encapsulated as the metric "D4":

```
{
  "name": "D4",
  "operator": null,
  "predicate_list": [
    {
      "family": "DateFamily",
      "name": "sameDateFormatted"
    }
  ],
  "description": "compares two date values independent of the format"
}
```

Listing 4.2: Precision Metric Definition for Date Matching

Multiple predicates can be combined under a single metric using logical operators such as AND or OR, providing a mechanism for flexible evaluation strategies.

4.1.3. Property Annotation via `propertydata.json`

The file `propertydata.json` contains the final layer of configuration by assigning each schema.org property to an appropriate precision metric. This determines which metric should be used when comparing a specific property during fact comparison.

For instance, as seen in the following Listing 4.3 the metric "D4" is assigned to all properties that denote dates:

```
{
  "propname": "birthDate",
  "pm": "D4"
},
{
  "propname": "releaseDate",
  "pm": "D4"
},
{
  "propname": "datePublished",
  "pm": "D4"
```

```
},
{
  "propname": "startDate",
  "pm": "D4"
},
{
  "propname": "endDate",
  "pm": "D4"
}
```

Listing 4.3: Mapping Schema Properties to Precision Metrics

This separation of concerns enables flexible, transparent configuration and avoids hardcoding logic into the core comparison engine. In addition to assigning metrics to properties globally, precision metrics can also be scoped to specific *data types*. This allows a given metric to be applied only in the context of a particular type - for example, assigning metric `"S7"` to the property `"name"` only for facts of type `Movie`, but not for `Person` or `Book` (see Listing 4.4). This more granular form of annotation increases flexibility and enables context-aware comparisons across heterogeneous datasets.

```
{
  "propname": "name",
  "type": "Movie",
  "pm": "S7"
}
```

Listing 4.4: Assigning a Precision Metric to a Property-Type Combination

A range of new similarity predicates has been implemented as part of this framework. These functions were designed to address common inconsistencies and variations in structured data that the original FactCheck system could not resolve. Examples include comparisons tolerant to formatting differences (e.g., varying date formats or number separators), structural object comparisons (e.g., nested entities such as ratings or persons), and unit normalization for numerical values (e.g., Celsius versus Fahrenheit, or km/h versus m/s). These enhancements allow the system to recognize equivalence in cases that would previously result in false mismatches. In the next chapter, the implemented similarity predicates will be described in more detail.

4.1.4. Semantic Comparison via LLM and SKOS

In addition to manually defined predicates, the framework supports semantic similarity functions powered by two external knowledge-based methods:

- **LLM-based Semantic Comparison:** For domain-specific vocabulary or multilingual data, the framework can query a local instance of a Large Language Model (e.g., via the Ollama runtime) to evaluate whether two terms are semantically equivalent. This is used, for example, to determine if “Actor” and “Schauspieler” refer to the same concept in multilingual knowledge graphs. Ollama is a lightweight local runtime environment for Large Language Models (LLMs) that allows users to run

models like LLaMA, Mistral, or Phi directly on their own machines without relying on cloud-based APIs. It provides a simple command-line interface and supports model customization through configuration files. In the FactCheck framework, Ollama is used to enable semantic comparisons by running LLMs locally, allowing the system to evaluate whether two terms or phrases are semantically equivalent - even across different languages or contexts - without sending data to external services.[2]

- **SKOS-based Label Matching:** Using SKOS (Simple Knowledge Organization System) vocabularies, such as DBpedia or Wikidata taxonomies, concepts are mapped via broader/narrower/related labels. If two input values share the same SKOS identifier or an equivalent label, the predicate returns `True`. SKOS (Simple Knowledge Organization System) is a standardized model for representing structured vocabularies such as taxonomies, thesauri, and classification schemes. It is based on RDF (Resource Description Framework) and is commonly used to describe concepts and their relationships, including labels, synonyms, broader and narrower terms. In the context of the FactCheck framework, SKOS is used to support semantic comparisons by identifying when two terms refer to the same or related concepts, even if they differ in wording or language. This enables more flexible and context-aware matching of structured data. [3]

These methods are implemented as special predicate functions and are referenced like any other predicate in the JSON configuration. An example predicate may be called ‘llmSemanticEqual‘ or ‘skosEquivalentLabel‘.

These semantic methods significantly enhance the systems ability to perform deep contextual comparisons without requiring handcrafted rules for every possible variation.

Together, these components form a layered and configurable architecture for structured data comparison. Predicate functions encapsulate reusable comparison logic, precision metrics aggregate these functions into domain-specific evaluation strategies, and property mappings assign them to real-world data schemas. This separation of concerns ensures both flexibility and maintainability while supporting future extensions such as semantic reasoning via LLMs and SKOS vocabularies.

4.2. Design Requirements

The design of the proposed framework was guided by both functional requirements emerging from practical usage scenarios and non-functional requirements derived from broader system objectives such as maintainability, extensibility, and domain-independence.

[2]Visit *https://ollama.com/* for more information.
[3]Visit *https://www.w3.org/2004/02/skos/* for more information.

Functional Requirements

At its core, the framework aims to enhance the FactCheck system's ability to compare structured data in a context-aware and flexible manner. To that end, the following functional goals were identified:

- **Predicate Modularity:** The comparison logic should be decomposed into atomic, reusable predicate functions. Each predicate encapsulates a specific comparison strategy (e.g., numeric tolerance, semantic similarity) and can be reused across different domains.
- **Configurable Precision Metrics:** The system must support the definition of precision metrics as logical combinations of predicates. These metrics represent evaluation strategies tailored to specific entity types (e.g., books, movies, weather observations).
- **Dynamic Integration:** Newly developed predicate functions and precision metrics should be integrable at runtime, without the need for system redeployment. This enables users to quickly adapt the comparison behavior to new domains or data idiosyncrasies.
- **Metadata-Driven Behavior:** The comparison logic should be externally configurable via structured metadata files. This approach decouples the matching logic from hardcoded program logic, allowing updates without modifying the core source code.
- **Explainability:** The result of each comparison should be decomposable into individual predicate evaluations to support transparency and debugging.

Non-Functional Requirements

In addition to these core functionalities, the framework was designed with the following non-functional requirements in mind:

- **Extensibility:** New predicates and metric types should be easy to add, both in terms of implementation and integration into the existing FactCheck infrastructure.
- **Domain-Independence:** While examples from specific domains (e.g., movies, books, weather) are used during development and testing, the framework itself should remain agnostic to any particular domain and support general-purpose usage.
- **Minimal Intrusiveness:** The new components must integrate seamlessly into the existing FactServer architecture (based on Azure Functions and a modular controller/manager logic) without requiring invasive changes to other parts of the system.

- **Maintainability and Clarity:** The codebase should follow FactCheck's established conventions to ensure maintainability and to allow other contributors to easily understand and extend the functionality.

Taken together, these goals define the design space within which the framework was developed.

4.3. API Endpoints and Data Format Requirements

The FactServer exposes a set of RESTful API endpoints that enable integration with external systems. These endpoints can be grouped into different functional categories, such as comparison, fact storage, precision metric management, and statistics. This section summarizes the most relevant endpoints and explains the expected data format for submitting fact sets.

Comparison Endpoints

- `POST /compare` Compares a submitted data object against existing entries in the FactBase, but does not store anything.
- `POST /compare/add` Performs the same comparison as `/compare`, but also stores the submitted data in the FactBase for future access and statistics.
- `POST /get` An alias for `/compare/add`, kept for compatibility.

Each of these endpoints requires a structured data object in the request body under the key `data`. The data should be encoded in **JSON-LD** format and contain enough information to identify the named entity being compared. If available, additional fields like `source` (URL) and `userid` (UUID) can be provided for traceability and caching (see Listing 4.5.

```
{
  "data": {
    "@context": "https://schema.org",
    "@type": "Movie",
    "name": "RoboCop",
    "actor": [
      {
        "@type": "Person",
        "name": "Peter Weller"
      }
    ],
    "datePublished": "1987-07-17",
    "url": "https://www.example.org/RoboCop"
  },
  "source": "https://www.example.org/RoboCop",
  "userid": "5f2fb618-86ec-49e0-b905-92169e001457"
```

```
}
```

Listing 4.5: JSON-LD input example for /compare

The comparison requires that the data contains one identifiable top-level entity (e.g., a `Person`, `Movie`, or `Book`) with a valid `name` and preferably a `sameAs` or `url` link pointing to known knowledge bases such as Wikidata or DBpedia. Properties such as `datePublished`, `actor`, or `aggregateRating` are used as comparable facts if they match known schemas.

If the entity cannot be matched to an existing entry in the FactBase (e.g., missing `name` or `@type`), the request returns an HTTP 422 (Unprocessable Entity).

Additional Endpoint Categories

- **Facts API:** `GET /facts`, `POST /facts`, `GET /facts/{id}` Access and upload raw fact data.
- **Precision Metrics API:** `GET /pm`, `GET /predicates`, etc. Retrieve information about metrics and predicate functions.
- **Resolver API:** `POST /resolver` Attempts to resolve entities based on structured data input.
- **Statistics API:** `GET /stats`, `GET /historical`, etc. Retrieve fact-based statistics for entities and sources.

4.4. Web-Based Editor for Predicate and Metric Management

To promote usability, adaptability, and rapid development, the extended framework includes a web-based editor that allows users to define, manage, and test predicate functions and precision metrics through a graphical interface. This editor abstracts the technical complexity of the underlying configuration files and enables users to modify the comparison logic without interacting with the source code directly.

The tool is designed to integrate seamlessly into the existing FactServer infrastructure and follows the same modular principles as the rest of the system. All changes performed through the interface including the creation of new predicate functions, modification of precision metrics, and adjustment of property mappings are reflected in the relevant JSON configuration files and loaded dynamically at runtime. As a result, no system restart or redeployment is necessary after making updates.

The editor provides a structured form for each predicate function, including fields for name, description, predicate family, and function body. It supports syntax validation and prevents the accidental overwriting of existing functions unless explicitly confirmed by the user. Precision metrics can be created or modified by combining existing predicates with logical operators, and can then be assigned to schema.org properties either globally or scoped to specific data types.

A central feature of the editor is the integrated API playground, which allows users to submit JSON-LD fact sets and immediately observe the results of predicate and metric evaluations. This interactive environment is especially useful for testing the behavior of newly created predicates and verifying the correctness of metric definitions.

Together, these capabilities make the web-based editor a crucial component of the extended framework. It lowers the technical barrier for configuring the comparison logic, supports agile development workflows, and ensures that the FactCheck system can be quickly adapted to new domains and evolving data standards.

The editors three-tab layout -*Similarity Function Creator*, *Precision Metrics*, and *API Playground* - reflects the three mental models users adopt when working with the system: authoring, configuration, and testing. This separation maps one-to-one to the underlying artifacts of the framework: functions live in `predicate.py`, metrics in `pm_data.json`, and test payloads target the `/compare` API. By aligning the interface with these conceptual boundaries, the editor avoids cross-contamination of concerns (e.g., defining code while simultaneously editing property mappings) and reduces cognitive load. Users can focus on a single task at a time - writing a predicate, composing a metric, or validating the effect - without having to navigate competing controls on the same screen.

Within each tab, the combination of a compact *table view* and a focused *form* is deliberate. The table provides global visibility and quick scanning: it surfaces the current state (function names, families, attached metrics, and assigned properties or types) in a flat, sortable structure that supports discovery and comparison at a glance. The form, by contrast, is optimized for single-item CRUD operations with clear affordances and guardrails. For functions, a code editor with syntax highlighting and indentation enforces the uniform signature and helps prevent trivial errors; for metrics, structured inputs make logical composition explicit while mirroring the JSON that will be written. This overview + detail pattern preserves context: selecting a row immediately pre-fills the form with authoritative data, reducing duplication and accidental divergence.

The tabular overview also serves traceability. Because entries in the function table are enriched with their metric and property context (joined from `pm_data.json` and `propertydata.json`), users can see how an individual predicate propagates through the comparison pipeline without changing screens. Conversely, when composing a metric, the form references existing functions by family and name, ensuring consistency with what the table displays. This bidirectional alignment between read and write surfaces shortens the feedback loop and makes configuration intent explicit.

A final design choice is the placement of the *API Playground* as a first-class tab rather than a separate tool. Keeping test execution adjacent to authoring and configuration enables a tight editruninspect cycle: users define or adjust a function or metric, submit a representative JSON-LD payload, and immediately inspect the applied metric and per-predicate outcomes. The resulting workflow reduces context switching and encourages incremental, empirically grounded refinements instead of speculative edits.

Alternative layouts were considered - including a single-page wizard and a code-centric, file-browser interface - but were rejected for lack of flexibility or excess complexity. A wizard would over-constrain expert users and obscure the direct mapping to backend

artifacts, while a file-centric view would raise the barrier for non-programmers and blur the distinction between function authoring and metric composition. The chosen information architecture therefore balances learnability for newcomers with efficiency for power users, all while preserving the systems core separation of concerns.

4.5. Design Summary

The design presented in this chapter extends the FactCheck system with a robust and modular framework for configurable, semantically aware fact comparison. Building on the limitations of the original system - which relied on hardcoded logic and strict equality - the new design introduces a dynamic architecture that separates concerns into clearly defined layers: predicate functions, precision metrics, and metadata-driven property mappings.

At the core of this approach lies the ability to define comparison behavior flexibly through modular Python functions and structured configuration files. These components interact dynamically at runtime and can be extended or modified without altering the backend code. This design not only improves adaptability but also promotes transparency and traceability in comparison results.

The integration of semantic techniques - such as local large language models (LLMs) and SKOS-based concept matching - further enhances the frameworks capability to handle linguistic variation, multilingual data, and contextual similarity. These features are embedded as reusable predicate functions and can be configured in the same way as other comparison logic.

A key element of the framework is the newly developed web-based editor, which empowers users to manage predicates, metrics, and property assignments through an intuitive interface. The editor also includes a built-in testing environment, enabling real-time evaluation of comparison logic.

Taken together, the extended architecture enables domain-independent, explainable, and semantically enriched comparisons of structured data. The design decisions made in this chapter lay the foundation for the concrete implementation described in the next part of this thesis.

5. Implementation

Building upon the conceptual architecture and design goals outlined in the previous chapter, this section presents the technical implementation of the extended comparison framework developed in this thesis. While the design chapter focused on abstract structures, modular components, and system-level decisions, the following sections provide a detailed account of how these concepts were translated into functioning code, configuration files, and user-facing interfaces.

The implementation aims to realize the framework's core objectives: dynamic extensibility, domain-independence, and semantic richness in structured data comparison. To this end, the system was extended in several dimensions:

- A large set of new predicate functions was implemented and grouped into semantically meaningful predicate families within the `predicate.py` module.
- The systems configuration files `pm_data.json` and `propertydata.json` were expanded and dynamically modified to support runtime integration of predicates and precision metrics.
- A browser-based editor was developed that allows users to define, edit, and assign new similarity functions and precision metrics without modifying the backend source code.
- Advanced comparison methods were introduced using tools such as large language models (LLMs) and SKOS-based taxonomies, further enhancing the frameworks semantic capabilities.
- A set of new test cases and domain-specific fact sets (e.g., movies, books, weather observations) were created to validate the behavior of the extended comparison logic in realistic scenarios.

This chapter is structured as follows: We begin by outlining the code-level implementation of predicate functions and how they are grouped into predicate families. Next, we explain the logic for managing precision metrics and mapping them to schema.org properties using configuration files. Special focus is placed on the integration of semantic comparison methods using external tools such as Ollama and SKOS. We then describe the implementation of the web-based frontend editor, including its communication with the backend and its role in dynamically updating system components. We then detail how the editor applies race-safe, atomic updates to backend code and configuration. The final section summarizes and concludes the chapter.

By combining modular Python code, structured metadata, and intuitive user interfaces, the implementation realizes a powerful and adaptable system for similarity-based fact comparison within the FactCheck infrastructure.

5.1. Predicate Function Implementation

The predicate functions are implemented in the file `predicate.py`, where they are grouped into predicate families such as `StringFamily`, `ObjectFamily`, `DateFamily`, `LLMComparisonFamily`, and `SKOSFamily`. Each family is defined as a Python class. Functions are implemented as either `@staticmethods` or `@classmethods`, depending on whether they require access to class-level resources.

Each predicate function receives two input values, which represent the properties that need to be compared, and returns a boolean result indicating whether the comparison is successful. If parsing fails or inputs are invalid, the function typically returns `False` rather than raising an exception, ensuring robustness.

Examples of implemented predicates include:

- `sameDateFormatted` (`DateFamily`): Compares two date values independently of their original format, supporting variations such as ISO, dot-separated, or slash-separated formats.
- `titleContained` (`StringFamily`): Checks whether one string is contained in the other, ignoring case sensitivity, subtitles, and common punctuation, making it suitable for partial title matches.
- `llm_semantic_equivalent` (`LLMComparisonFamily`): Uses a locally running large language model (via Ollama) to determine whether two textual inputs can be considered semantically equivalent.
- `any_match` (`LLMComparisonFamily`): Applies the LLM-based comparison to elements in two lists and returns `True` if any pair of elements semantically match.
- `skos_altlabel_match` (`SKOSFamily`): Resolves both input values against a SKOS vocabulary and checks whether they share any alternative labels.

Functions are declared either as `@staticmethods` or `@classmethods`, depending on their internal dependencies. Functions that operate independently without referencing other methods within the same family class - such as `sameDateFormatted`, `titleContained`, `llm_semantic_equivalent`, or `skos_altlabel_match` - are implemented as `@staticmethods`. In contrast, `@classmethods` are used when a function calls another method within the same predicate family, such as `any_match` in the `LLMComparisonFamily`, which internally uses `llm_semantic_equivalent`. The LLM- and SKOS-based similarity predicates are discussed in more detail in Section 5.4.

Example: windSpeedMatch

The following function `windSpeedMatch`, part of the `ObjectFamily`, compares wind speed values by normalizing units such as `km/h`, `mph`, and `m/s` into a common scale (see Listing 5.1). This is essential for ensuring that fact comparisons remain meaningful even if different data sources use varying unit systems.

```
@staticmethod
def windSpeedMatch(v1, v2):
    """
    Ensures that "windSpeed" in v1 matches v2. Converts wind speed units (km/h, mph, m/s).
    """
    def extract_wind_speed(speed_data):
        if not isinstance(speed_data, dict) or "value" not in speed_data or "unit" not in speed_data:
            logger.warning(f"Invalid wind speed data: {speed_data}")
            return None
        value = speed_data["value"]
        unit = speed_data["unit"]
        # normalize to km/h for comparison
        if unit in ["km/h", "kilometers per hour", "kmph", "kmh"]:
            converted_value = round(value, 2)
            converted_unit = "km/h"
        elif unit in ["mph", "miles per hour"]:
            converted_value = round(value * 1.60934, 2)
            converted_unit = "km/h"
        elif unit in ["m/s", "meters per second"]:
            converted_value = round(value * 3.6, 2)
            converted_unit = "km/h"
        else:
            logger.warning(f"Unrecognized wind speed unit: {unit}, keeping original")
            converted_value = round(value, 2)
            converted_unit = unit
        logger.info(f"Converted wind speed: {value} {unit}  {converted_value} {converted_unit}")
        return {"windSpeed": (converted_value, converted_unit)}

    v1_filtered = extract_wind_speed(v1)
    v2_filtered = extract_wind_speed(v2)
    logger.info(f"Comparison after conversion: v1={v1_filtered} | v2={v2_filtered}")
    if v1_filtered is None or v2_filtered is None:
        return False
    v1_value, v1_unit = v1_filtered["windSpeed"]
    v2_value, v2_unit = v2_filtered["windSpeed"]
    return abs(v1_value - v2_value) < 0.01 and v1_unit == v2_unit
```

Listing 5.1: Predicate for Comparing Wind Speed

This predicate was implemented to address real-world inconsistencies in weather data, particularly in scenarios where multiple data sources express the same physical quantity (wind speed) using different units or naming conventions. In heterogeneous

web environments - such as those involving OpenWeatherMap, meteorological archives, or schema.org-based weather observation - values may be encoded in `km/h`, `mph`, or `m/s`, sometimes with additional formatting differences or alternative unit spellings (e.g., `"kmph"`, `"miles per hour"`).

Rather than enforcing a strict requirement that all compared facts use identical units, the function applies normalization to convert all supported units into a common canonical form, specifically `km/h`. This target unit was chosen for its broad international use, semantic clarity, and intuitive interpretation in most applications, including European and scientific contexts. Moreover, the conversion factors (e.g., `1 mph = 1.60934 km/h`) are well-established and introduce only minimal rounding errors, which are absorbed by the functions use of a small tolerance threshold (0.01).

From an implementation perspective, the decision to encapsulate the normalization logic within a nested helper function `extract_wind_speed` improves readability, reusability, and separation of concerns. It also allows the predicate to gracefully degrade when inputs are missing or malformed by returning `False` instead of raising exceptions. Logging statements further support debugging and transparency during test execution.

While the function was initially designed with `windSpeed` properties in weather fact sets in mind, the same pattern of unit normalization is applicable to a variety of other comparison scenarios involving physical quantities. These include:

- **Temperature comparisons** (e.g., `Celsius`, `Fahrenheit`, `Kelvin`)
- **Distance measures** (e.g., `miles`, `kilometers`, `meters`)
- **Time durations** (e.g., `seconds`, `minutes`, `hours`)
- **Speed in other domains** (e.g., `maxSpeed` in `VideoGame` or `Vehicle` types)

As a result, the predicate not only solves a specific problem in one domain but also serves as a blueprint for other semantically equivalent predicates that compare scalar values under variation in units and formatting. This reinforces the modular philosophy of the extended framework, where small, well-defined building blocks can be reused across domains with minimal adaptation.

Example: weatherObservationMatch

Another example of unit normalization is the `weatherObservationMatch` predicate, which compares temperature values across different unit systems (see Listing 5.2). Similar to the wind speed predicate, this function was designed to accommodate inconsistencies in data formatting, especially when integrating weather observations from heterogeneous sources. In practice, different APIs or datasets may encode temperature in `Celsius`, `Kelvin`, or `Fahrenheit`, often with varying abbreviations or inconsistent use of `unitCode` and `unit` fields. Moreover, the temperature value might be nested inside a field such as `temperature`, or presented directly at the top level - especially in loosely structured JSON.

To handle this, the predicate first ensures that both input values are valid dictionaries and then attempts to locate a temperature field either directly or nested. Once found, it extracts the numeric value and unit, normalizes the unit string (e.g., stripping whitespace and handling abbreviations like `"řC"`, `"cel"`, or `"kel"`), and converts the temperature to `Celsius` as the canonical unit. This decision mirrors common practice in scientific and public datasets, where Celsius provides an intuitive and widely used baseline, especially across European sources.

Internally, the conversion logic distinguishes between Kelvin and Fahrenheit and applies appropriate transformations:

- `Kelvin  Celsius: C = K - 273.15`
- `Fahrenheit  Celsius: C = (F - 32) / 1.8`

If an unknown unit is encountered, the function logs a warning and retains the original value, maintaining graceful degradation. Only when both values are successfully converted and units match (case-insensitively), does the predicate return `True` - otherwise, it returns `False`.

```
@staticmethod
def weatherObservationMatch(v1, v2):
    """
    Ensures that "temperature" in v1 matches v2.
    Converts temperature units (Kelvin, Celsius, Fahrenheit).
    """
    def extract_temperature(fact_data):
        if not isinstance(fact_data, dict):
            logger.warning(f"Expected dict, but got {type(fact_data)}: {fact_data}")
            return {}

        if "value" in fact_data and "unit" in fact_data:
            temp_data = fact_data
        else:
            temp_data = fact_data.get("temperature", None)

        if temp_data is None:
            logger.warning(f"'temperature' field is missing in fact_data: {fact_data}")
            return {}

        if not isinstance(temp_data, dict):
            logger.warning(f"Invalid 'temperature' structure: {temp_data}")
            return {}

        value = temp_data.get("value")
        unit = temp_data.get("unitCode") or temp_data.get("unit")

        if value is None or unit is None:
            logger.warning(f"Missing 'value' or 'unit' in temperature data: {temp_data}")
```

```
            return {}

        unit_normalized = unit.lower().strip()
        if unit_normalized in ["c", "řc", "cel", "celsius"]:
            converted_value = value
            converted_unit = "Celsius"
        elif unit_normalized in ["k", "řk", "kel", "kelvin"]:
            converted_value = round(value - 273.15, 2)
            converted_unit = "Celsius"
        elif unit_normalized in ["f", "řf", "fah", "fahrenheit"]:
            converted_value = round((value - 32) / 1.8, 2)
            converted_unit = "Celsius"
        else:
            logger.warning(f"Unrecognized temperature unit: {unit},
    keeping original")
            converted_value = value
            converted_unit = unit

        logger.info(f"Converted temperature: {value} {unit}  {
    converted_value} {converted_unit}")
        return {"temperature": (converted_value, converted_unit)}

    if v1 is None or not isinstance(v1, dict):
        logger.warning(f"v1 is None or not a dict: {v1}")
        return False

    if v2 is None or not isinstance(v2, dict):
        logger.warning(f"v2 is None or not a dict: {v2}")
        return False

    v1_filtered = extract_temperature(v1)
    v2_filtered = extract_temperature(v2)

    logger.info(f"Comparison after conversion: v1={v1_filtered} | v2={
    v2_filtered}")

    if "temperature" in v1_filtered and "temperature" in v2_filtered:
        v1_value, v1_unit = v1_filtered["temperature"]
        v2_value, v2_unit = v2_filtered["temperature"]
        return v1_unit.lower() == v2_unit.lower() and v1_value ==
    v2_value

    return False
```

Listing 5.2: Predicate for Comparing Temperature Values

The necessity for such a predicate arises from real-world use cases in which structured weather data is compared across multiple providers or across time, such as for climate models, forecast validation, or historical record alignment. Beyond temperatures, this predicate structure serves as a template for other quantity comparisons that may involve varied nesting and unit representation. It illustrates a robust, fault-tolerant approach to structured value comparison under semantic and syntactic variability.

Overall, `weatherObservationMatch` demonstrates the core design principles of the

extended framework: input flexibility, semantic normalization, structured error handling, and separation of concerns via nested helper functions. By transforming diverse raw data into a uniform representation before comparison, the system ensures more accurate and meaningful fact evaluation outcomes.

Beyond physical unit conversion, several other predicate functions required domain-specific normalization strategies to account for lexical and structural variability. One such example is `compareAggregateRating`, which compares complex rating objects (e.g., from schema.org) by extracting their numeric `ratingValue` and ensuring comparability despite possible differences in decimal representation or nesting. Another example is `genre_match`, which was designed to handle common genre variations in media datasets. To this end, a small internal normalization table was introduced within the predicate, mapping frequent aliases and abbreviations to canonical forms (e.g., `"sci-fi"` → `"Science Fiction"`, `"bio"` → `"Biography"`). This allowed the predicate to recognize equivalence between semantically identical but lexically distinct genres.

However, it was later observed that such normalization rules are necessarily incomplete and brittle in the face of broader linguistic variation. In a notable example, a fact set compared the values `"sci-fi"` and `"science fiction"` - which were still treated as conflicting due to the limitations of the hardcoded mapping. To resolve such cases more robustly, responsibility for these ambiguous comparisons was shifted to LLM-based semantic predicates such as `llm_semantic_equivalent`, which are discussed in detail in Section 5.4. This architectural decision reflects a broader strategy in the framework: use lightweight normalization and stateless predicates where possible, and defer to semantic models only when necessary. This ensures a balance between performance, interpretability, and coverage in practical fact comparison scenarios. This example also illustrates the practical value of the web-based editor, which allows users to flexibly assign or reassign properties to different predicate functions without modifying backend code. Initially, the property `genre` for type `Movie` was linked to the `genre_match` predicate, which successfully handled cases like `"sci-fi"` versus `"Science Fiction"` through an internal normalization table. However, when further discrepancies arose that exceeded the scope of rule-based normalization, the property could be easily remapped - via the editor - to use the `any_match` predicate from the `LLMComparisonFamily`. This dynamic configurability highlights a core advantage of the system's architecture: it separates predicate logic from its application to specific properties, allowing non-technical users to iteratively refine comparison behavior based on observed data issues.

Each predicate is registered through its family and function name in the configuration file `pm_data.json`. This ensures that functions can be dynamically composed into precision metrics without modifying the codebase. All implemented predicates follow a uniform interface and are designed to be composable, extensible, and resilient to noisy or inconsistent input.

5.2. Precision Metric Configuration

Precision metrics are configured via the file `pm_data.json`, which defines how predicate functions are logically grouped and evaluated during a comparison. Each precision metric entry includes a name, a description, a list of predicate functions (along with their corresponding predicate family), and an optional logical operator such as `AND` or `OR`. This operator determines whether all similarity predicates within the precision metric must return true so that the corresponding property is returned as equal (AND) or whether one is sufficient (OR).

This design allows for flexible, declarative composition of evaluation logic, enabling multiple comparison strategies to be defined and reused across properties and domains. The structure also facilitates runtime integration and modification without requiring changes to the backend source code.

A minimal metric definition consists of a single predicate and does not require an operator (see Listing 5.3):

```
{
  "name": "D4",
  "operator": null,
  "predicate_list": [
    {
      "family": "DateFamily",
      "name": "sameDateFormatted"
    }
  ],
  "description": "compares two date values independent of the format"
}
```

Listing 5.3: Minimal precision metric definition

More complex metrics combine multiple predicates using a logical operator. For example, the following metric considers two values equivalent if either one of two string-based predicates evaluates to `True` (see Listing 5.4):

```
{
  "name": "S7",
  "operator": "OR",
  "predicate_list": [
    {
      "family": "StringFamily",
      "name": "titleContained"
    },
    {
      "family": "StringFamily",
      "name": "substringContained"
    }
  ],
  "description": "true if one string is contained in the other"
}
```

Listing 5.4: Precision metric with logical OR operator

Each metric is uniquely identified by its `name` and can be referenced by multiple properties or data types in the `propertydata.json` configuration file. At runtime, the FactServer loads all defined metrics and uses them to guide the comparison of input values, ensuring a decoupled and transparent evaluation process.

The separation of metric logic from predicate implementation allows developers and domain experts to define, test, and refine comparison strategies independently of the core codebase. Changes to the metric configuration are effective immediately and require no redeployment, making it possible to adapt to new domains or data characteristics on the fly.

The extended framework introduced in this thesis makes extensive use of precision metrics to capture nuanced similarity requirements across different domains. In particular, the web-based editor provides a user-friendly interface for defining new metrics, assigning predicates, and specifying logical operators without direct interaction with the JSON files. Once submitted, all updates are immediately written to `pm_data.json` and reflected in the system at runtime.

Furthermore, some of the precision metrics leverage advanced predicates that incorporate semantic techniques such as language models or controlled vocabularies. These aspects are discussed in detail in Section 5.4.

5.3. Property Mapping via `propertydata.json`

The `propertydata.json` file defines the mapping between schema.org properties and the precision metrics to be used for comparing their values. This mapping layer enables targeted, property-specific comparison logic without hardcoding any evaluation rules into the core system.

Each entry in the file specifies a property name (e.g., `name`, `birthDate`, `datePublished`) and the corresponding precision metric to apply. A simple property-to-metric mapping looks as follows (see Listing 5.5):

```
{
  "propname": "birthDate",
  "pm": "D4"
},
{
  "propname": "datePublished",
  "pm": "D4"
}
```

Listing 5.5: Global mapping of properties to metrics

In this example, the precision metric `D4` is used to compare values of both `birthDate` and `datePublished`, which typically represent dates in structured data.

Beyond global mappings, the configuration also supports scoped mappings that apply only to certain data types. This allows more fine-grained control over comparison behavior, especially when the same property (e.g., `name`) may occur in different contexts. The following entry applies a custom metric only to the property `name` when used in the context of a `Movie` entity (see Listing 5.6):

```
{
  "propname": "name",
  "type": "Movie",
  "pm": "S7"
}
```

Listing 5.6: Type-specific property mapping

When evaluating facts, the FactServer prioritizes type-specific mappings over global ones. This hierarchical matching logic ensures that precision metrics can be tailored to the domain and semantics of the data being compared.

The separation of property mapping from predicate logic and metric composition promotes modularity and transparency. It allows new metrics to be rolled out by simply updating the configuration file, and supports reuse of the same metric across multiple properties or types. Combined with the dynamic loading mechanism of the FactServer, these mappings enable continuous refinement of comparison behavior without system downtime.

In practice, the extended framework frequently relies on type-specific mappings to fine-tune comparison behavior across different domains. For instance, the `name` property can be compared using different metrics depending on whether it refers to a `Movie`, `Book`, or `VideoGame`. This selective assignment improves precision and avoids false equivalences across semantically distinct contexts. All mappings can be adjusted dynamically through the web-based editor, enabling rapid iteration and customization without backend intervention or service downtime.

5.4. Integration of Semantic Methods

To enable deeper and more flexible comparisons beyond syntactic or structural matching, the extended framework includes support for semantic similarity evaluation. Two semantic methods were implemented: a local LLM-based comparison and a SKOS-based label matching system. These methods are exposed through dedicated predicate functions and can be used like any other predicate within a precision metric.

5.4.1. LLM-Based Semantic Comparison

To support deeper semantic equivalence beyond syntactic or structural similarity, the framework integrates local Large Language Models (LLMs) using the open-source runtime Ollama[1]. This integration enables the system to evaluate whether two input values refer to the same concept, even if they differ in surface form due to language, abbreviation, or rephrasing.

The primary predicate implemented for this purpose is `llm_semantic_equivalent`, which accepts two strings as input and queries a local language model to assess their semantic equivalence. Internally, the function constructs a binary prompt asking whether

[1] `https://ollama.com/`

the two inputs refer to the same real-world concept. A simplified version of the function is shown below (see Listing 5.7):

```
@staticmethod
def llm_semantic_equivalent(val1, val2):
    if not isinstance(val1, str) or not isinstance(val2, str):
        return False
    prompt = f"Is '{val1}' and '{val2}' the same (e.g. work, person,
    genre, entity etc.)? In other words, is one just an abbreviation,
    translation, or variation of the other? Answer with Yes or No only."
    try:
        client = ollama.Client()
        response = client.generate(model="llama3.2", prompt=prompt)
        answer = response.response.strip().lower()
        return answer.startswith("yes")
    except Exception:
        return False
```

Listing 5.7: Semantic comparison using a local LLM

At the time of writing this thesis, the model `llama3.2` is used, which provides relatively fast inference and a good balance between accuracy and system requirements. However, the implementation is model-agnostic and can be switched to other locally hosted models with minimal changes in the function code. The prompt is carefully designed to reduce ambiguity and elicit a binary response - either Yes or No - which is then parsed into a Boolean result. This makes the function usable in the same way as any traditional Boolean predicate.

In contrast to `llm_semantic_equivalent`, the predicate `any_match` is implemented as a `@classmethod` and applies semantic comparison iteratively across all pairs in two lists. It returns `True` if at least one pair is judged semantically equivalent. This is particularly useful for properties like genres, occupations, or categories, where multiple candidate values may need to be compared in batch.

The LLM-based approach has several key advantages:

- It is domain-independent and can handle comparisons across a wide range of fact types (e.g., people, locations, media titles, roles).
- It works well in multilingual contexts and recognizes synonyms, abbreviations, and naming variations without explicit rules.
- It avoids the need for large manually curated normalization tables or taxonomies.

However, this approach also has notable limitations:

- Inference time is significantly longer than for rule-based predicates, especially on resource-constrained machines. Even with 32GB of RAM, local models often require several seconds per comparison.
- The binary output is not always stable or explainable - identical prompts can yield different responses depending on context or system load.

- The quality of results varies strongly with the choice of LLM and the quality of the prompt. More lightweight models may yield false negatives or behave unpredictably on borderline cases.

Despite these trade-offs, the integration of LLMs offers a powerful fallback mechanism when rule-based methods fail or prove too brittle. The combination of `llm_semantic_equivalent` and `any_match` enables the framework to resolve subtle mismatches that would otherwise go undetected, thereby improving accuracy and robustness in noisy or heterogeneous data environments.

5.4.2. SKOS-Based Concept Matching

As a second semantic method, the framework supports comparisons using SKOS (Simple Knowledge Organization System) vocabularies. SKOS is a W3C standard designed for representing knowledge structures such as taxonomies and thesauri. It defines relationships like `skos:altLabel`, `skos:prefLabel`, `skos:broader`, or `skos:narrower`, enabling the identification of semantically related concepts across heterogeneous data.

The implemented predicate function `skos_altlabel_match` utilizes the `skos:altLabel` relation to check whether two input strings can be considered equivalent based on known synonyms in a SKOS-compliant vocabulary. A simplified version of the function is shown in Listing 5.8.

```
@staticmethod
def skos_altlabel_match(v1, v2):
    def get_altlabels(term):
        sparql = f"""
        SELECT ?alt WHERE {{
        ?s rdfs:label "{term}"@en ; skos:altLabel ?alt .
        FILTER (lang(?alt) = "en")
        }} LIMIT 10
        """
        url = "https://dbpedia.org/sparql"
        params = {"query": sparql, "format": "application/sparql-results+
    json"}
        try:
            r = requests.get(url, params=params, timeout=5)
            results = r.json()["results"]["bindings"]
            return [x["alt"]["value"].lower() for x in results]
        except:
            return []

    if isinstance(v1, str):
        v1 = v1.strip().lower()
    else:
        return False

    if isinstance(v2, list):
        candidates = [str(item).strip().lower() for item in v2]
    else:
        candidates = [str(v2).strip().lower()]
```

```
28
29      altlabels = get_altlabels(v1)
30      return any(c in altlabels for c in candidates)
```

Listing 5.8: SKOS-based synonym matching using altLabels

The function issues a SPARQL[2] query to the DBpedia endpoint to retrieve English-language `altLabels` for the first input. It then checks whether any of the normalized candidates in the second input match these labels. This allows terms like `USA` and `United States of America` to be recognized as equivalent based on curated synonym lists.

The design of the `SKOSFamily` allows for future expansion. Although only `skos_altlabel_match` is currently implemented, the framework has been tested with additional SKOS-based strategies, such as:

- matching across languages using `rdfs:label` in multiple languages (e.g., `Schauspieler` vs. `Actor`),
- resolving abbreviations via properties like `wdt:P1813` (short name) in Wikidata,
- extracting related concepts via `wdt:P138` (named after) or `wdt:P1449` (nickname).

The following Python-based SPARQL queries were evaluated during development to demonstrate the extensibility of the SKOS-based comparison strategy:

- Fetching **alternative labels in another language**:

```
get_wikidata_labels_in_other_language("Schauspieler", source_lang="
    de", target_lang="en")
```

- Retrieving **entity abbreviations and short forms**:

```
get_abbreviations("Science Fiction")  # returns e.g. "Sci-Fi"
```

- Querying **"named after"** relationships:

```
get_named_after_labels("Bernie")  # finds related person entities
```

These experimental queries confirm that the SKOS framework offers powerful capabilities for synonym, abbreviation, and multilingual resolution - each of which could be encapsulated into additional predicate functions within the `SKOSFamily`.

Like other semantic predicates, `skos_altlabel_match` can be dynamically referenced in precision metrics through the configuration file `pm_data.json`. This allows developers to combine rule-based, semantic, and SKOS-aware logic in a modular and data-driven fashion.

The approach offers a low-latency alternative to LLMs for many domain-specific synonym problems, especially in settings where structured vocabularies such as DBpedia or Wikidata are available and well-maintained.

[2]SPARQL (SPARQL Protocol and RDF Query Language) is a standardized query language for retrieving and manipulating data stored in Resource Description Framework (RDF) format. It is often used to access structured semantic data in knowledge graphs like DBpedia or Wikidata.

5.5. Frontend Editor Implementation

A central goal of this thesis is to make the configuration of similarity functions and precision metrics more accessible, dynamic, and transparent. To achieve this, a browser-based frontend editor was developed, which serves as the main interface for interacting with the comparison framework. This editor empowers users to define, test, and assign comparison logic at runtime - without requiring any manual modification of backend code or configuration files.

5.5.1. Overview and Architecture

The frontend is a lightweight, fully static web application built using standard web technologies (HTML, CSS, JavaScript) and Bootstrap[3] for styling. The editor is served as a static interface, either locally or via any simple file hosting solution. All communication with the system backend occurs via RESTful API endpoints exposed by the FactServer. These endpoints allow users to manage predicate functions, configure metrics, map properties, and test the comparison logic using live data.

The editor is structured into three distinct tabs, each serving a specific purpose:[4]

- **Similarity Function Creator** for defining, editing, and deleting predicate functions.
- **Precision Metrics** for composing and managing metric definitions.
- **API Playground** for live testing of comparison behavior with real fact sets.

These three different areas are now presented in detail in the following subsections. Full-size screenshots of these three tabs are provided in Appendix A.

5.5.2. Similarity Function Creator

The main entry point of the editor is the Similarity Function Creator, which offers a comprehensive interface to manage predicate functions. On the left side, users can fill out a detailed form to define new predicates. The form includes the following fields:

- **Function Name** A unique identifier for the function, used in backend logic.
- **Decorator** Specifies whether the function is a `@staticmethod` or `@classmethod`.
- **Family** The predicate family this function belongs to (e.g., `StringFamily`, `DateFamily`). Users can select from existing families via dropdown or define a new one.

[3] Bootstrap is a widely used open-source framework for building responsive web interfaces. It provides pre-designed CSS and JavaScript components that simplify the creation of visually consistent and mobile-friendly layouts.

[4] The fourth tab FactCheck leads to the official website of the FactCheck system, https://-factcheck.cs.univie.ac.at/login, which, just like the FactServer, is only accessible via a VPN connection with the official VPN of the University of Vienna.

- **Description** A short textual explanation of the functions purpose.
- **Function Code** The actual Python function body, written in a CodeMirror[5] editor with syntax highlighting and basic validation.
- **Metric Name** The name of the associated precision metric (to be added in `pm_data.json`).
- **Properties to Assign** One or more schema.org properties to which the function (via the metric) should be assigned (as listed in `propertydata.json`).
- **Data Type (Optional)** Used to restrict the function to a specific `@type` such as `Book`, `Movie`, or `Person`.

Upon submission, the predicate is immediately written to the backends `predicate.py` file. If a function with the same name already exists, the user is prompted for confirmation before overwriting it. The system ensures that the formatting, indentation, and decorators remain consistent with existing functions. Implementation details are described in Section 5.6.

On the right-hand side of the interface, a live-updated table displays all existing predicate functions along with their key attributes. Each row in the table shows the function name, its description, the predicate family it belongs to, the precision metric it is assigned to, the list of associated properties, and, if applicable, the expected data type. Users can select any function from the table to load its details into the form on the left, enabling them to edit or delete the function directly. All modifications are applied instantly, both to the underlying backend configuration files and to the interface itself, ensuring a seamless and responsive editing experience.

5.5.3. Precision Metrics Tab

The second tab focuses on the composition and management of precision metrics. Each metric acts as a container that groups one or more predicate functions under a named logic. The interface provides a form to define new metrics, which includes:

- **Metric Name** A unique identifier.
- **Logical Operator** Either `AND`, `OR`, or empty (if only one predicate is used).
- **Predicate Functions** One or more functions to be included in the metric.
- **Description** A short explanation of what the metric checks.

[5] CodeMirror is a versatile in-browser code editor implemented in JavaScript. It provides syntax highlighting, auto-indentation, and other editor features for a wide range of programming languages.

Similar to the predicate creator, all metric definitions are written directly to `pm_data.json`. If a new predicate is defined in the Similarity Function Creator and linked to a new metric name, that metric will automatically appear in the Precision Metrics tab as well.

A table on this page displays all currently defined precision metrics, showing their name, the logical operator used (if any), the list of associated predicates, and a brief description. Each metric can be edited or deleted directly through the interface. Any changes made are immediately written to the backend configuration files and loaded into the system without requiring a server restart, allowing for dynamic and uninterrupted updates.

5.5.4. API Playground

The third and arguably most powerful component of the frontend is the API Playground. It serves as an interactive testing environment for predicates and precision metrics. Users can send real-time requests to the FactServers endpoints and view responses directly in the browser.

The playground supports:

- Selecting from predefined sample requests (e.g., Book: Animal Farm, Person: Bernie Sanders, Movie: Inception, etc.).
- Creating and saving custom fact set examples (e.g., VideoGame objects).
- Choosing the HTTP method (typically `POST`).
- Editing the request body (JSON-LD format).
- Customizing the endpoint URL (default is `/compare`).

A typical request body (e.g., for a Book object) looks like this (see Listing 5.9):

```
{
  "data": {
    "@context": "https://schema.org",
    "@type": "Book",
    "name": "Animal Farm",
    "author": {
      "@type": "Person",
      "name": "George Orwell"
    },
    "datePublished": "2008-10-01",
    "isbn": "9780141036137",
    "genre": "Fantasy"
  }
}
```

Listing 5.9: Example Fact set Request Body

When the request is submitted, the response displays the output from the FactCheck comparison pipeline, including matched properties, the applied precision metric, and detailed equivalence results. These include:

- **equals** All matching property values.
- **local_missing / remote_missing** Properties missing in either fact set.
- **conflicting** Values that differ significantly.
- **pm** The precision metric used per property.

For example, when the above Book example is sent to `http://localhost:7071/compare`, the system compares it with existing entries from the FactBase and returns an array of comparison results.

Optionally, users can filter comparisons to a specific property using the query parameter `?property=author`. This is particularly useful to isolate and debug specific comparison behavior during development.

This architecture not only enables flexible configuration but also makes comparison results highly interpretable. In the response of a typical test request - such as the sample fact set for `Animal Farm` - users can directly observe which values were considered equal, missing, or conflicting. For example, if two different representations of the property `datePublished` appear - such as `2008-10-01` and `01 Oct 2008` - and are marked as `equal` in the response, the system also indicates that the precision metric used was `D4`. By navigating to the **Precision Metrics** tab, users can inspect which predicates are associated with this metric - in this case, `DateFamily.sameDateFormatted`. If needed, the corresponding function can then be reviewed, edited, or reassigned within the **Similarity Function Creator** tab, enabling a fully traceable and adaptable comparison workflow.

5.5.5. Runtime Integration and Deployment

All frontend actions translate directly into backend operations. New predicate functions are appended to `predicate.py`, precision metrics are inserted into `pm_data.json`, and property mappings are recorded in `propertydata.json`. These files are reloaded automatically without restarting the server, ensuring a smooth workflow.

The entire frontend is self-contained and requires no additional runtime infrastructure. It can be hosted locally via `file://` protocol or integrated into a broader administrative interface.

5.5.6. Conclusion

The frontend editor represents the practical core of the proposed framework. It abstracts away the complexity of backend integration and enables users - even those without programming experience - to define sophisticated comparison logic in a visual, intuitive manner. This dynamic approach to predicate and metric configuration directly addresses the goals laid out in the thesis: enhancing flexibility, transparency, and adaptability in similarity-based systems.

5.6. Similarity Editor and Safe, Concurrent Code Updates

This section details how edits made in the browser-based editor become safe, consistent updates to the backend code and configuration, without requiring a redeploy. Concretely, the editor talks to a small set of Azure Functions endpoints in `predicate_editor_controller.py`. Reads and writes are implemented in a way that is both *structure aware* (via Pythons AST) and *race-safe* (via family-scoped patches plus atomic file writes guarded by a lock).

On every page load, the editor retrieves the current state through `GET /api/predicate/list` and `GET /api/predicate/families`. The controller reads `predicate.py` as source text and parses it with `ast.parse`. Rather than returning raw function blocks, it walks each class that descends from `PredicateFamily` and extracts, for each `FunctionDef`, the explicit decorator (`@staticmethod` or `@classmethod`), the docstring, and - crucially - *only the function body*. Returning bodies rather than full definitions avoids duplication of headers and docstrings when users round-trip edits through the UI. In the same response, the controller also merges in the metric and property context from `pm_data.json` and `propertydata.json`, so the table in the editor can present a coherent, joined view of code and configuration.

When a user submits a change, the editor calls `POST /api/predicate/add` with the family name, function name, short description (to be used as the docstring), the chosen decorator, and the function body. Before any file is touched, the server validates the request: function and family names must be valid Python identifiers; if the decorator is omitted, it is inferred (`@classmethod` when `cls` appears in the body, otherwise `@staticmethod`); and the body is syntax-checked by wrapping it in a synthetic definition with the correct signature - `def __x__(cls, v1, v2):` for class methods or `def __x__(v1, v2):` for static methods - and parsing it with `ast.parse`. Any syntax error results in an HTTP `400` with the compilers message; in this case nothing is written and the user can correct the code directly in the editor.

If validation succeeds, the controller applies a *family-scoped* patch to `predicate.py`. The file is split into logical class blocks by locating the `class <Family>(PredicateFamily):` header and taking all lines up to (but excluding) the next `class` declaration or end of file. If the family does not yet exist, the controller creates the class with a minimal docstring and appends the new method. If the family exists, it searches only within that class block for a method with the same name; if found, the entire method (including any decorator lines and the added via UI marker) is replaced; otherwise, the method is appended to the end of the class. Newly written methods follow a uniform structure to preserve signature and documentation conventions. See as example the following Listing 5.10:

```
# <function_name>  added via UI
@<decorator>
def <function_name>(<args>):
    """
    <description>
    """
    <body from the editor, correctly indented>
```

Listing 5.10: Structure of a UI-generated predicate method

To prevent race conditions under concurrent edits, every mutation is performed under a short-lived, file-based lock and written atomically. The controller attempts to create a sibling lock file `<path>.lock` using `os.O_CREAT|os.O_EXCL`. If the lock already exists, the request waits (with a timeout) and surfaces an HTTP `423 Locked` if the lock cannot be acquired in time. With exclusive access granted, the controller writes the new content to a temporary file in the same directory, flushes it to disk, and then swaps it into place with `os.replace`. This *lock + atomic replace* combination ensures that (i) only one writer computes and applies a patch at a time, and (ii) readers never observe a partially written file; the file appears to jump from the old to the new version in a single, atomic step.

After `predicate.py` has been safely updated, the controller maintains consistency of the configuration layer by updating `pm_data.json` and `propertydata.json` using the same locking and atomic write scheme. If the referenced precision metric already exists, the pair `(family, function_name)` is added to its `predicate_list` when absent; missing descriptions can be filled with the submissions text. If the metric does not exist, it is created with a minimal structure and the new predicate as its first entry. For property mappings, the controller ensures that each `(propname, pm)` pair is present at most once and, if provided, records the type restriction (e.g., `Book`, `Movie`). Deletions work in reverse: the specified method is removed from its family block, all references to it are pruned from `pm_data.json`, and any now-empty metrics are deleted along with their property mappings.

The frontend itself is conservative about writes. When editing an existing predicate, it normalizes the current and submitted bodies (trimming whitespace and empty lines) and compares them before sending a request. If the body and metadata have not changed, the editor informs the user that the function is identical and skips the network call. If differences are detected for an existing name, the user is asked to confirm the overwrite. This reduces lock contention and avoids no-op filesystem churn while keeping the UI responsive.

Error handling is explicit at every stage: malformed identifiers, syntax errors, invalid decorators, and lock timeouts are returned with descriptive HTTP statuses and messages, and no partial state is left behind. Because the editor is an administrative tool in a trusted environment, submitted code is intentionally treated as first-class: once validated and written, it becomes part of the live `predicate.py` and is available to subsequent requests immediately. The narrow, uniform predicate signature (`(v1, v2)` or `(cls, v1, v2)`) keeps these dynamic updates compatible with the rest of the comparison pipeline

while leaving room for future families and predicates to evolve.

5.7. Summary and Conclusion

The implementation of the extended comparison framework presented in this chapter demonstrates how the conceptual goals of flexibility, modularity, and semantic expressiveness have been effectively translated into practice. Through a combination of code-level innovations, structured configuration, and intuitive user interfaces, the system provides a powerful, extensible platform for fact comparison in heterogeneous and semantically rich domains.

At the heart of the system lie the modular predicate functions, organized into coherent predicate families. These functions encapsulate a variety of comparison strategies - ranging from syntactic checks and structural normalization to semantic reasoning via local language models and external knowledge graphs. The chosen interface design, based on uniform input and Boolean output, ensures that all predicates can be composed interchangeably and integrated dynamically via configuration.

The use of configuration files, particularly `pm_data.json` for precision metrics and `propertydata.json` for property mappings, enables a clear separation between logic definition and application. This decoupling allows the system to remain flexible and extensible, as changes to comparison behavior no longer require code modifications or redeployments. Instead, metrics and property-to-metric assignments can be adjusted declaratively at runtime.

Notably, the integration of semantic comparison methods marks a significant advancement over purely rule-based approaches. By incorporating locally hosted LLMs and leveraging SKOS taxonomies, the system can resolve subtle conceptual equivalences that exceed the capabilities of traditional normalization techniques. These semantic methods are seamlessly embedded into the existing predicate framework, ensuring interoperability and preserving the modular design.

The browser-based frontend editor plays a central role in operationalizing these concepts. It enables users to define, edit, and test predicate functions, configure precision metrics, and map properties without needing direct access to the codebase. The editor bridges the gap between technical implementation and user accessibility, offering a visual and intuitive interface for managing complex comparison logic.

Altogether, the implementation fulfills the thesis' core objectives: it supports runtime extensibility, domain-independent configuration, and semantically informed evaluation - all within a coherent and usable system architecture. The resulting framework empowers users to iteratively refine similarity logic in response to real-world data challenges, laying a foundation for scalable and adaptable fact comparison in diverse domains.

6. Experimental Evaluation

To assess the practical effectiveness of the extended comparison framework developed in this thesis, this chapter presents a structured evaluation based on realistic test data. While the previous chapter focused on implementation details, the current chapter turns to empirical validation, examining whether the system behaves as expected when confronted with heterogeneous, web-derived data. The evaluation not only verifies the correctness of individual components - such as predicate functions and precision metrics - but also investigates the systems overall robustness, flexibility, and semantic capabilities in practice. The chapter is structured around evaluation objectives, test data, evaluation procedure, coverage, and error analysis, and concludes with a summary of findings.

6.1. Evaluation Objectives

The primary objective of this evaluation is to determine whether the extended comparison framework achieves its design goals when applied to realistic, heterogeneous data. Rather than relying on synthetic examples or isolated unit tests, the evaluation is grounded in representative fact sets that reflect the structural complexity, lexical variation, and semantic ambiguity commonly found in real-world web data.

A central focus lies in assessing the **correctness and reliability of the implemented predicate functions** across various domains, including books, movies, weather observations, and chemical substances. These domains were deliberately chosen to test different types of variation - such as date formats, measurement units, nested structures, and naming inconsistencies.

Equally important is the systems **ability to handle structural and semantic variation in fact sets**, including multilingual inputs and flexible object representations. This aspect is particularly relevant for web-derived data, which often contains inconsistencies in formatting, abbreviations, and vocabulary.

Another key aspect of the evaluation is the **dynamic configurability of predicates and precision metrics via the frontend editor**. The ability to modify comparison logic at runtime - without requiring code changes or server restarts - serves as a critical test for the frameworks usability and extensibility.

Finally, the evaluation investigates the **practical utility of integrating semantic methods** such as local language models (LLMs) and SKOS-based vocabulary matching. These components are tested for their ability to resolve cases where traditional rule-based predicates fail, particularly in scenarios involving abbreviations, synonyms, or cross-lingual concept equivalence.

Taken together, these objectives provide a comprehensive foundation for evaluating the robustness, adaptability, and semantic depth of the extended comparison system.

6.2. Test Data

To assess the practical behavior of the extended comparison framework under realistic conditions, a set of domain-specific fact sets was constructed and evaluated. These test data were designed to reflect the heterogeneity typically found in web-derived information, including lexical variation, structural inconsistency, unit differences, and multilingual content. Some fact sets - such as those for the movie *Inception*, the book *Animal Farm*, or the person *Bernie Sanders* - were already present in the systems FactBase and served as initial reference points. Others, including structured data for weather observations, chemical substances, and video games, were created specifically to test newly implemented predicate functions and semantic evaluation strategies.

Domain-Specific Fact Sets

Five primary domains were selected for evaluation:

Movie fact sets included properties such as `name`, `actor`, and `aggregateRating`, with a focus on nested object comparison and name normalization. In the case of *Inception*, the comparison successfully handled nested actors and rating values, while name matching worked as expected for surface-identical strings.

Book fact sets tested predicates over properties like `author` and `name`. A notable case involved the book *Animal Farm*, where the English title conflicted with its Polish translation *Folwark zwierzcy*. This served as a testbed for comparing semantic strategies: while SKOS-based translation labels yielded partial matches, LLM-based predicates were also evaluated to capture conceptual equivalence. These semantic methods performed inconsistently, depending on the model prompt, language detection, and surface-form similarity, highlighting both their potential and their limitations.

Weather observation fact sets were created to test predicates for `temperature`, `windSpeed`, and `observationDate`. These examples successfully validated the normalization of physical units such as `km/h`, `mph`, and `m/s` for wind speeds, as well as `Celsius`, `Fahrenheit`, and `Kelvin` for temperature values. The corresponding predicates (`windSpeedMatch`, `weatherObservationMatch`) correctly abstracted over unit differences and handled value rounding within tolerable margins.

Chemical substance fact sets focused on structured measurements such as `boilingPoint` and `freezingPoint`. As with the weather domain, the comparison logic was tested against different unit formats and nested structures. For instance, boiling points expressed in Celsius and Fahrenheit were successfully aligned after conversion, confirming the robustness of unit normalization logic across domains.

Video game fact sets served as test cases for property mappings that apply only to specific types. In particular, comparisons of `name` and `releaseDate` properties were tested in combination with LLM-based and SKOS-based predicates. These fact sets also

explored the use of type-restricted property mappings in `propertydata.json`, validating that the correct predicate logic was applied only when the fact sets `@type` matched the configured scope.

Evaluation Procedure

All test cases were submitted to the system via the `/compare` API endpoint, triggering the comparison pipeline of the FactServer. The resulting output included a detailed record of matched, conflicting, or missing properties, the predicates applied to each property, and the precision metric responsible for the final decision. This enabled a transparent inspection of predicate behavior and metric composition for each case.

In addition, the same tests were replicated using the browser-based API playground integrated into the frontend editor. This allowed real-time observation of changes to predicate definitions, metric configurations, and property mappings. It also verified that runtime updates - such as remapping a property to a different metric or modifying a predicates logic - were reflected immediately, without restarting the system.

Through this procedure, the test data provided a comprehensive empirical foundation to evaluate the frameworks correctness, flexibility, and adaptability under realistic and domain-specific conditions.

Testing Workflow via the Web Interface

A key goal of the developed framework was to make the configuration and evaluation of comparison logic accessible even to users without programming experience. Consequently, the browser-based web interface played a central role during testing. All predicate functions and precision metrics were not only defined but also evaluated through this interface, which proved essential for ensuring correctness, usability, and traceability of the results.

The testing process followed an iterative workflow. First, a new predicate function was created or edited using the Similarity Function Creator tab of the frontend. After selecting a suitable predicate family and defining the function body in Python, the user submitted the form, triggering an immediate update of the backends predicate module. The system then automatically registered the function and assigned it to a precision metric, which could subsequently be edited or extended in the Precision Metrics tab. Once the function-metric combination was complete, the relevant schema.org property (e.g., `name`, `releaseDate`, or `temperature`) was mapped to the metric using the same interface.

To test the behavior of the newly defined logic, the API Playground tab provided a live environment for submitting example fact sets. These were typically based on realistic structured data (e.g., JSON-LD representations of books, movies, weather observations). The comparison request was sent to the FactServers `/compare` endpoint and the response was displayed directly in the interface, including detailed output for each compared property: whether values were classified as equal, conflicting, missing locally or remotely, and which metric was applied.

The results of each comparison were manually inspected for correctness. This included verifying that formatting variations (such as date formats), unit differences (e.g., `km/h` vs. `m/s`), or structural divergences (e.g., nested rating objects) were correctly handled by the intended predicate. Furthermore, logging output from the FactServer was monitored during testing to capture edge cases, invalid inputs, or unexpected behavior. Extensive logging statements - particularly in predicates handling numeric conversion or semantic queries - helped trace errors and validate intermediate processing steps. This logging was essential in diagnosing predicate failures and improving robustness.

6.3. Coverage Analysis

The experimental setup aimed not only to validate individual predicates and metrics but also to achieve broad coverage across the conceptual dimensions of the extended comparison framework. To this end, test cases were systematically constructed to reflect a wide range of comparison challenges and predicate types.

From a functional perspective, all major predicate families introduced in the implementation were exercised during testing. These include the `StringFamily`, `ObjectFamily`, `DateFamily`, `LLMComparisonFamily`, and `SKOSFamily`. Each family was represented through at least one active predicate in the test data, ensuring that both syntactic and semantic comparison strategies were empirically evaluated.

Within the `StringFamily`, predicates such as `titleContained` and `genre_match` were tested across domains like books and movies, especially in multilingual name comparisons and partial matches. The `DateFamily` was evaluated through date properties like `birthDate` and `datePublished`, where variation in formatting (e.g., ISO, textual, localized) was successfully resolved using the `sameDateFormatted` predicate.

The `ObjectFamily` received extensive coverage through weather and chemical fact sets. Functions like `windSpeedMatch` and `weatherObservationMatch` were validated on structured numerical data involving unit normalization, rounding tolerance, and nested object access. These examples confirm that the framework supports deeply structured inputs and robust physical unit handling.

In addition, the `AggregateRatingFamily` was tested through predicates such as `compareAggregateRating` and `equalIgnoringFormat`, particularly in the context of movie fact sets where rating values may vary in precision or representation. These cases demonstrate the systems ability to normalize numerical input even in loosely structured schema.org fields.

Semantic methods were also systematically included in the evaluation. The `LLMComparisonFamily` was tested through predicates like `llm_semantic_equivalent` and `any_match`, particularly in cases involving cross-lingual labels and ambiguous genre classifications (e.g., `"sci-fi"` vs. `"Science Fiction"`). The `SKOSFamily` was evaluated using DBpedia queries to retrieve alternative labels, and was applied in controlled tests involving name translations and abbreviations.

In addition to functional coverage, the evaluation also confirmed that dynamic configuration was effectively supported. All predicate functions used in the tests were

either created, edited, or reassigned using the web-based frontend. Precision metrics were composed and modified using the editor interface, and property mappings were adapted during the evaluation process to test behavior under different assignments. These interactions verified that the systems runtime flexibility and modular architecture are not only conceptually sound but also practically operable.

Overall, the test coverage demonstrates that the evaluation encompasses the full design space of the extended framework - across predicate types, data domains, structural variation, semantic depth, and user-driven configuration.

6.4. Error Analysis and Limitations

While the evaluation confirms the overall robustness and extensibility of the comparison framework, several limitations and sources of potential error were observed during testing. These issues highlight the trade-offs between rule-based, data-driven, and semantic methods, and provide important guidance for future refinements.

One of the most prominent challenges arises from the integration of large language models (LLMs). While LLM-based predicates such as `llm_semantic_equivalent` or `any_match` offer powerful semantic reasoning capabilities, their behavior is inherently difficult to interpret and debug. Unlike rule-based functions with transparent logic and predictable output, the results of LLM-based comparisons depend on the internal state of the model, the exact prompt wording, and the underlying inference engine. For example, the fact pair `"Animal Farm"` and `"Folwark zwierzcy"` (the Polish translation of the same book) was sometimes marked as `equal` and sometimes as `conflicting`, despite using the same prompt:

```
Are 'Animal Farm' and 'Folwark zwierzcy' the same (e.g.  work, person,
genre)?  In other words, is one just an abbreviation, translation,
or variation of the other?  Answer with Yes or No only.
```

Such inconsistencies make the results difficult to validate, especially when the model's decision process is not explainable. Furthermore, inference quality depends heavily on hardware constraints. Although testing was conducted on a system with 32 GB of RAM, which allowed local models like `llama3.2` to run reliably, semantic comparisons using LLMs were noticeably slower than other predicate types. Depending on system load, a single comparison could take several seconds, making real-time processing of larger fact sets impractical.

SKOS-based predicates, while more transparent and explainable, also revealed limitations in multilingual and translation-based comparisons. The predicate `skos_altlabel_match` retrieves alternative labels from DBpedia using SPARQL queries, but these queries are inherently language-specific. For instance, resolving whether `"Animal Farm"` and `"Folwark zwierzcy"` refer to the same concept requires specifying both the source and target language manually - e.g., via:

```
1 get_wikidata_labels_in_other_language("Animal Farm", source_lang="en",
    target_lang="pl")
```

While useful for targeted lookups, this approach does not generalize well to arbitrary language pairs or to fact sets without known language annotations. Moreover, the reliance on external SPARQL endpoints introduces latency and potential availability issues.

Another noteworthy limitation lies in the use of static normalization tables, such as those used in the `genre_match` predicate. Here, a predefined mapping translates common genre variations (e.g., `"sci-fi"`, `"scifi"`, `"science fiction"`) into canonical forms (see Listing 6.1):

```
GENRE_NORMALIZATION_MAP = {
    "sci-fi": "Science Fiction",
    "scifi": "Science Fiction",
    "sci fi": "Science Fiction",
    "science-fiction": "Science Fiction",
    "science fiction": "Science Fiction",
    ...
}
```

Listing 6.1: Normalization map for common genre variants

While effective for common patterns, this approach is inherently brittle and requires manual upkeep. Any unseen variant - such as minor typos, less frequent synonyms, or translations - will be missed unless explicitly encoded. As the diversity of input data grows, the feasibility of maintaining exhaustive normalization tables becomes increasingly questionable. In practice, several edge cases were encountered where genres were misclassified due to missing mappings. These shortcomings motivated the fallback to LLM-based predicates for semantically richer comparisons, despite their performance and explainability drawbacks.

Beyond technical aspects, another limitation concerns the usability of the system. Although the browser-based frontend editor provides a user-friendly interface for defining predicates, composing metrics, and assigning property mappings, effective use of the framework still requires a certain degree of familiarity with the FactCheck system and its internal structure. For example, understanding how precision metrics relate to schema.org properties, how property scopes are resolved, or how predicate functions must be structured often presupposes background knowledge that casual users may not possess. While the interface abstracts away many technical details, the underlying logic remains complex and domain-specific.

Lastly, the current framework assumes that ground truth is binary and unambiguous - i.e., that two values either match or do not. In practice, especially in multilingual or semantically ambiguous scenarios, such binary decisions may be too coarse. Some value pairs may be close enough depending on context, user expectations, or downstream applications. Future versions of the system could explore graded similarity scores or confidence measures to better reflect real-world complexity.

In summary, the evaluation revealed several meaningful limitations:

- LLM-based comparisons are semantically powerful but opaque, inconsistent, and slow.

- SKOS-based matching is explainable but limited in multilingual coverage and generalizability.
- Rule-based normalization (e.g., for genres) is brittle and difficult to scale.
- Effective use of the system requires contextual knowledge of FactChecks internal mechanics.
- The current binary matching logic may oversimplify nuanced equivalences.

Despite these challenges, each method contributes valuable capabilities to the system. Their combined use - under clear assumptions and with fallback mechanisms - enables the framework to perform meaningful comparisons across a wide variety of domains and input types.

Common Failure Cases and Semantic Limitations

In addition to technical errors, several conceptual and structural challenges emerged during evaluation, particularly in domains characterized by lexical ambiguity, heterogeneous structures, or implicit semantics.

One frequent source of false negatives involved semantically equivalent values that differed in language, abbreviation, or phrasing. For example, the comparison between the book titles Animal Farm and Folwark zwierzcy failed under traditional string matching, despite both referring to the same entity. Similarly, genre labels such as Sci-Fi versus Science Fiction or Bio versus Biography were often mismatched unless specifically normalized.

Rule-based predicates proved effective for well-defined formats - e.g., date parsing or numeric unit conversion - but struggled with unstructured or abbreviated inputs. A predicate designed to compare aggregateRating objects failed when one source provided only a numeric value, while another included a full nested structure (value, bestRating, worstRating). In such cases, minor structural differences masked the fact that both values denoted the same rating.

Hardcoded normalization tables, such as those used in genre comparison, revealed inherent brittleness. While effective for known variations (e.g., sci-fi $\rightarrow$ Science Fiction), they could not account for typos, edge-case spellings, or rare synonyms. Moreover, maintaining such mappings at scale across domains proved impractical. These limitations motivated the use of semantic fallback mechanisms.

LLM-based predicates, such as `llm_semantic_equivalent`, addressed many of these shortcomings by reasoning over linguistic and conceptual similarity. For example, comparisons between actor and Schauspieler, or between semantically related genres, were often resolved correctly by local LLM models. However, their performance was inconsistent and difficult to interpret. Identical prompts yielded different results under identical conditions, highlighting the stochastic nature of language model inference. Furthermore, latency remained a bottleneck: even with 32 GB of RAM, response times for individual comparisons were typically in the range of 25 seconds - acceptable for testing but problematic at scale.

SKOS-based predicates offered a more transparent alternative for certain multilingual or taxonomical comparisons. Queries to DBpedia using `skos:altLabel` or `rdfs:label` were able to resolve some abbreviation and translation mismatches. However, coverage was incomplete, and SPARQL endpoint availability introduced unpredictable delays. Moreover, many domain-specific concepts (e.g., video game genres or weather codes) lacked consistent SKOS representation, limiting applicability.

A final class of challenges arose from ambiguities in structured data itself. For instance, the same movie might be released with slightly varying release dates (e.g., theatrical vs. home release), and both values could be considered correct. The current systems binary match logic, while effective for technical comparisons, does not yet support nuanced or probabilistic equivalence scoring. Incorporating confidence intervals or fuzzy match scores could significantly improve flexibility in such borderline cases.

Overall, while the enhanced comparison logic addressed a wide range of structural and semantic inconsistencies, the evaluation revealed that no single strategy suffices across all domains and input types. The strength of the framework lies in its configurability: users can combine rule-based and semantic predicates as needed, define precision metrics tailored to domain requirements, and iteratively refine logic through a transparent interface.

6.5. Performance Considerations

Beyond correctness and configurability, performance plays a critical role in determining the practical viability of the extended comparison framework - particularly in scenarios involving large-scale data integration or real-time fact-checking. During testing and development, several performance-relevant observations were made regarding runtime behavior, system responsiveness, and resource usage.

The most significant performance bottleneck arises from the use of local large language models (LLMs) for semantic comparison. While these models offer valuable interpretative capabilities in cases of abbreviation, translation, or conceptual similarity, their computational demands are substantially higher than those of rule-based predicates. On a machine with 32 GB of RAM, models such as `llama3.2` could be run reliably via the Ollama runtime. Nevertheless, individual comparisons involving LLM predicates often required multiple seconds to complete, particularly for list-based inputs evaluated via functions like `any_match`. This latency makes LLM-based evaluation unsuitable for bulk processing or high-frequency real-time scenarios, unless backed by significantly more powerful hardware or pre-processing optimizations.

By contrast, rule-based and normalization-based predicates - such as those handling date formats, unit conversions, or genre strings - executed with negligible latency. Even when applied to nested structures or complex objects, these functions returned results within milliseconds, making them ideal for baseline evaluation across large datasets. Their low computational cost also enabled efficient batch testing of fact sets via the `/compare` API or the API playground in the frontend.

The semantic comparison method based on SKOS taxonomies falls between these

two extremes. While the matching logic itself is lightweight, performance is constrained by the response time of external SPARQL endpoints (e.g., DBpedia). Typical queries completed within one to three seconds, but occasional delays or failures due to endpoint availability were observed. Moreover, repeated queries involving the same term were not cached, meaning performance could degrade when multiple comparisons relied on identical external lookups. Future improvements could include query result caching or the use of local triple stores to eliminate this external dependency.

On the configuration side, the systems runtime behavior remained consistently responsive. Updates to predicates, precision metrics, or property mappings submitted through the frontend editor were reflected immediately in the backend without requiring a restart. This hot-reload capability significantly enhanced development and testing speed, allowing rapid iteration and interactive debugging. Even when large configuration files (such as `pm_data.json`) were modified, the reload mechanism performed reliably and within sub-second latency.

From a deployment perspective, the system architecture remains lightweight. The backend can be run locally as an Azure Functions application, requiring minimal setup and no container orchestration. The static frontend can be served via file-based hosting or integrated into existing administrative interfaces. However, when LLMs are enabled, the memory footprint increases substantially. Running a single local model instance comfortably requires 1632 GB of RAM; simultaneous inference or parallel comparison pipelines would require even more.

In summary, the performance evaluation reveals that the extended framework remains responsive and lightweight for typical rule-based comparison tasks, while offering semantically enhanced matching at the cost of increased latency and resource usage. The design allows for selective use of expensive methods - such as LLMs or SKOS lookups - only when needed, ensuring that performance remains acceptable in most real-world usage scenarios.

6.6. Summary of Findings

The experimental evaluation demonstrates that the extended comparison framework fulfills its design goals across a diverse set of test scenarios and domains. Through realistic fact sets and systematic testing, the framework proved capable of handling structural variation, multilingual data, unit normalization, and semantically ambiguous input. The introduced predicate functions operated correctly across different property types and schema.org domains, confirming the correctness of both rule-based and semantic comparison strategies.

Dynamic configurability via the web-based editor was successfully validated: predicates and precision metrics could be added, edited, and reassigned at runtime without server restarts. This interactive setup enabled rapid iteration and emphasized the practical usability of the system even in non-developer contexts.

At the same time, several limitations were identified. LLM-based predicates, while powerful, introduced issues of explainability, latency, and consistency. SKOS-based meth-

ods were more transparent but less flexible in multilingual use cases. Static normalization approaches, although efficient, were difficult to scale and maintain. Finally, the systems reliance on internal FactCheck concepts (e.g., precision metrics, predicate scoping) may pose challenges for users unfamiliar with its underlying structure.

From a performance standpoint, the system remained fast and lightweight in rule-based scenarios, with semantic extensions introducing acceptable trade-offs when used selectively. The architecture supports low-overhead deployment and real-time configuration updates, contributing to an efficient development and testing workflow.

Overall, the evaluation confirms the robustness, flexibility, and semantic depth of the extended framework while also outlining areas for improvement. It provides a solid empirical foundation for the final conclusions and opens several directions for future refinement.

7. Conclusion

This thesis set out to enhance the FactCheck system developed by the MIS research group at the University of Vienna by addressing its limitations in handling heterogeneous, multilingual, and semantically ambiguous structured data. While the existing system already relied on predicate-based comparison logic to detect conflicting information, its scope was limited by a relatively small set of basic predicates and a lack of runtime configurability. Moreover, the system remained difficult to use or adapt without direct access to the backend code. The central contribution of this thesis was therefore the development of a fully web-based frontend editor that enables users to define, configure, and test both predicate functions and precision metrics in a user-friendly, real-time environment. This editor decouples the internal logic from the system interface, allowing advanced comparison strategies to be employed without requiring technical expertise. It enables the dynamic creation of predicate families and functions, supports the mapping of precision metrics to specific data properties, and integrates a live API playground to test comparisons on-the-fly. By doing so, it opens the FactCheck system to wider use and experimentation while preserving its internal modular structure.

Chapters 1-3 established the context and motivation for this work. Chapter 1 outlined the increasing relevance of automated fact-checking in the digital age, with a focus on structured data such as schema.org annotations. It motivated the need for advanced comparison techniques to resolve inconsistencies that arise across web sources, particularly when such inconsistencies are due to differences in formatting, measurement units, language, or structure rather than actual factual disagreement. Chapter 2 provided the necessary background, introducing the reader to the foundations of fact-checking, structured data, and similarity functions. It emphasized the potential of structured data while also highlighting its inherent ambiguity when authored by independent sources. Chapter 3 reviewed related work in the areas of fact-checking, similarity computation, and conflict resolution. It became evident that although a broad range of syntactic and semantic similarity functions exists, few systems allow for their dynamic combination in configurable pipelines. The original FactCheck system stood out for its modular design and focus on structured data conflicts, but lacked the extensibility and runtime configurability needed for broader applicability. This led to the identification of a clear research gap: the need for an extensible, explainable, and semantically rich comparison framework that supports dynamic predicate configuration, integrates domain-specific knowledge, and empowers users to refine similarity logic interactively.

Chapter 4 detailed the design of the enhanced comparison framework. A key design decision was to keep the strict separation between predicate functions, precision metrics, and property-to-metric mappings, each defined externally in modular configuration

files. This architecture enabled the introduction of semantically aware comparison logic, including methods based on local Large Language Models (LLMs) and SKOS concept hierarchies. These allow the system to recognize equivalence even when values differ linguistically, structurally, or contextually. The chapter also established why Python remained the implementation language of choice and how the Azure Functions architecture was extended to integrate the new framework without disrupting existing components. Overall, the design ensures flexibility, transparency, and domain independence.

Chapter 5 translated the conceptual framework into a concrete implementation. A wide range of new predicate functions was developed, covering date normalization, nested object comparison, numerical unit conversion, and semantic string matching. All predicates conform to a unified interface, allowing them to be dynamically referenced by metrics. These metrics, in turn, were defined in JSON and configured to combine predicates logically. A type-aware mapping file determines which precision metric should be used for each schema.org property, allowing comparisons to vary based on the domain. A major contribution of the implementation is the development of a browser-based editor that allows users to configure the comparison logic without modifying the backend. The editor enables users to define predicate functions, compose precision metrics, assign metrics to properties, and test comparison results live through an API playground. This frontend lowers the barrier to entry and supports interactive refinement of the system's behavior.

Chapter 6 provided a thorough experimental evaluation of the framework. Realistic fact sets were constructed across five domains - books, movies, weather observations, chemical substances, and video games - to test structural variation, multilinguality, and semantic ambiguity. Predicate behavior was evaluated both in isolation and as part of composite metrics. The system demonstrated its ability to resolve inconsistencies in unit formats, detect cross-lingual equivalence, and correctly match structured objects. The runtime configurability of predicates and metrics via the frontend was successfully validated. At the same time, the evaluation revealed limitations. LLM-based predicates were powerful but slow and difficult to interpret. SKOS-based methods required careful query design and were limited in multilingual coverage. Static normalization tables, while fast, proved brittle and incomplete. Furthermore, while the editor simplifies usage, a basic understanding of the FactCheck architecture is still required to use it effectively. Finally, the system currently applies binary matching logic, which may be insufficient for use cases that demand nuanced similarity scoring or probabilistic reasoning.

Despite these challenges, the framework developed in this thesis significantly advances the state of the art in configurable similarity-based fact comparison. It introduces a general-purpose architecture that combines explainability, semantic depth, and practical usability. It allows structured data conflicts to be resolved using tailored metrics that reflect domain-specific requirements. More importantly, it empowers users to influence the logic of fact comparison dynamically and transparently.

Several directions for future work arise from the results of this thesis. First, replacing binary comparison outcomes with graded similarity scores could allow for probabilistic or confidence-based decision-making. Second, similarity functions could be composed

hierarchically or weighted dynamically, allowing for more flexible fusion strategies. Third, semantic comparison could be improved through more consistent and performant methods, such as embedding-based models or hybrid strategies combining symbolic and neural approaches. Finally, the frontend could be extended with intelligent support features such as suggestions, error highlighting, or training materials, enabling a smoother onboarding experience for new users.

Assessment of Goals and Requirements

Taken together, the results presented in this thesis meet the goals stated in the introduction and satisfy the design requirements from Chapter 4 to a substantial degree. The first goal - developing domain-specific similarity functions that move beyond basic string and numeric checks - has been realized through a broad catalogue of predicates that normalize formats and units, handle nested structures, and incorporate semantic reasoning. In particular, the introduction of *LLM*- and *SKOS*-based families allows the system to detect conceptual equivalence where surface forms differ due to language, abbreviation, or style, thereby extending the FactCheck pipeline with capabilities that were previously out of reach.

The second goal - fine-tuning similarity behavior via precision metrics - has likewise been achieved in practice. Metrics are defined declaratively as logical compositions of predicates and can be tailored to the semantics of a given property or type. Thresholds and tolerances are currently expressed at the predicate level, which proved sufficient for conflict detection across domains. At the same time, the evaluation in Chapter 6 indicates a promising direction for future work: moving from binary outcomes toward graded similarity with globally tunable weights, which would enable probabilistic or confidence-based decisions without changing the overall architecture.

The third goal - building a modular, extensible framework that admits new functions and metrics at runtime - has been realized through the browser-based editor and a carefully engineered backend path for safe, concurrent updates. Family-scoped, AST-aware patches and lock-guarded, atomic writes ensure that edits made in the frontend are promoted to `predicate.py`, `pm_data.json`, and `propertydata.json` without redeployment and without risking inconsistent intermediate states. This mechanism preserves the integrity of the codebase while enabling rapid iteration by non-expert users.

The fourth goal - evaluation on realistic, multi-domain datasets - has been satisfied through experiments spanning books, movies, weather observations, chemical substances, and video games. These tests demonstrate that the framework scales across heterogeneous structures and languages, and that semantic predicates resolve cases where purely syntactic methods fail. The evaluation also surfaces honest limitations - latency and explainability of LLM-based checks, uneven multilingual coverage of SKOS endpoints, and the brittleness of small hand-crafted normalization tables - thereby motivating the concrete extensions discussed as future work.

Viewed through the lens of the functional requirements, the system exhibits predicate modularity via a uniform $(v1, v2)$ (and, where appropriate, $(cls, v1, v2)$) interface; config-

urable metrics through JSON-based composition with optional logical operators; dynamic integration by hot-updating code and configuration at runtime; metadata-driven behavior by externalizing decisions in structured files; and explainability through per-property traces that expose the selected metric and its constituent predicates. Regarding the non-functional requirements, the implementation remains extensible and domain-independent, integrates minimally and cleanly with the Azure Functions architecture, and improves maintainability by constraining edits to family-scoped patches guarded by file locks and atomic replaces. In sum, the work delivers on its stated objectives while delineating a clear and feasible path for further refinement toward graded similarity and even stronger semantic coverage.

In conclusion, this thesis contributes a modular, explainable, and semantically enriched framework for comparing structured data. While developed in the context of the FactCheck system, the underlying design and tooling have broader relevance for knowledge graph alignment, data cleaning, and explainable AI. The combination of predicate modularity, semantic enrichment, and runtime configurability marks an important step toward trustworthy, adaptable, and user-controllable fact-checking systems.

Bibliography

[AAB23] Esma Aïmeur, Sabrine Amri, and Gilles Brassard. Fake news, disinformation and misinformation in social media: a review. *Social Network Analysis and Mining*, 13(1):30, February 2023.

[Aya25] Fatih Ayaz. Semantic Similarity Algorithms: Advances, Techniques, and Emerging Trends Summary Article, February 2025.

[BBHG] Stephen H Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-Loss Markov Random Fields and Probabilistic Soft Logic.

[BN09] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1:1–1:41, January 2009.

[CH13] Michelle Cheatham and Pascal Hitzler. String Similarity Metrics for Ontology Alignment. In *The Semantic Web ISWC 2013*, pages 294–309. Springer, Berlin, Heidelberg, 2013. ISSN: 1611-3349.

[CRF] William W Cohen, Pradeep Ravikumar, and Stephen E Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks.

[DBES13] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Data Fusion: Resolving Conflicts from Multiple Sources. In *Handbook of Data Quality*, pages 293–318. Springer, Berlin, Heidelberg, 2013.

[DS13] Xin Luna Dong and Divesh Srivastava. Big data integration. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1245–1248, April 2013. ISSN: 1063-6382.

[GBM16] R. V. Guha, Dan Brickley, and Steve Macbeth. Schema.org: evolution of structured data on the web. *Commun. ACM*, 59(2):44–51, January 2016.

[GMIHF19] Najlah Gali, Radu Mariescu-Istodor, Damien Hostettler, and Pasi Fränti. Framework for syntactic string similarity measures. *Expert Systems with Applications*, 129:169–185, September 2019.

[GSV22] Zhijiang Guo, Michael Schlichtkrull, and Andreas Vlachos. A Survey on Automated Fact-Checking. *Transactions of the Association for Computational Linguistics*, 10:178–206, February 2022.

[HRJM16] Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi, and Jacky Montmain. Semantic Measures for the Comparison of Units of Language, Concepts or Instances from Text and Knowledge Base Analysis, October 2016. arXiv:1310.1285 [cs].

[KKJM18] Peter Kalchgruber, Wolfgang Klas, Nour Jnoub, and Elaheh Momeni. FactCheck - Identify and Fix Conflicting Data on the Web. pages 312–320. May 2018.

[LGM+16] Yaliang Li, Jing Gao, Chuishi Meng, Qi Li, Lu Su, Bo Zhao, Wei Fan, and Jiawei Han. A Survey on Truth Discovery. *SIGKDD Explor. Newsl.*, 17(2):1–16, February 2016.

[NCH+21] Preslav Nakov, David Corney, Maram Hasanain, Firoj Alam, Tamer Elsayed, Alberto Barrón-Cedeño, Paolo Papotti, Shaden Shaar, and Giovanni Da San Martino. Automated Fact-Checking for Assisting Human Fact-Checkers, May 2021. arXiv:2103.07769 [cs].

[Ont20] Santiago Ontañón. An overview of distance and similarity functions for structured data. *Artificial Intelligence Review*, 53(7):5309–5351, October 2020.

[PPM04] Ted Pedersen, Siddharth Patwardhan, and Jason Michelizzi. WordNet::Similarity: measuring the relatedness of concepts. In *Demonstration Papers at HLT-NAACL 2004 on XX - HLT-NAACL '04*, pages 38–41, Boston, Massachusetts, 2004. Association for Computational Linguistics.

[TV18] James Thorne and Andreas Vlachos. Automated Fact Checking: Task formulations, methods and future directions, September 2018. arXiv:1806.07687 [cs].

[VRA18] Soroush Vosoughi, Deb Roy, and Sinan Aral. The spread of true and false news online. *Science*, 359(6380):1146–1151, March 2018. Publisher: American Association for the Advancement of Science.

[Wen] Fan Wenfei. Dependencies revisited for improving data quality. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 159–170, Vancouver, BC, Canada. ACM.

[WZS+25] Shuang Wang, He Zhang, Quan Z. Sheng, Xiaoping Li, Zhu Sun, Taotao Cai, Wei Emma Zhang, Jian Yang, and Qing Gao. A Survey on Truth Discovery: Concepts, Methods, Applications, and Opportunities. *IEEE Transactions on Big Data*, 11(2):314–332, April 2025.

[YHY07] Xiaoxin Yin, Jiawei Han, and Philip S. Yu. Truth discovery with multiple conflicting information providers on the web. In *Proceedings of the 13th*

ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '07, pages 1048–1052, New York, NY, USA, August 2007. Association for Computing Machinery.

[ZIF15] Ganggao Zhu and Carlos Ángel Iglesias Fernández. Sematch: semantic entity search from knowledge graph. In *Joint Proceedings of the 1st International Workshop on Summarizing and Presenting Entities and Ontologies and the 3rd International Workshop on Human Semantic Web Interfaces (SumPre 2015, HSWI 2015) co-located with the 12th Extended Semantic Web Conferen | SumPre 2015 - 1st International Workshop on Summarizing and Presenting Entities and Ontologies | 1/06/2015 | Portoroz, Slovenia*, volume 1556, pages 1–12, Portoroz, Slovenia, 2015. E.T.S.I. Telecomunicación (UPM).

[ZRGH12] Bo Zhao, Benjamin I. P. Rubinstein, Jim Gemmell, and Jiawei Han. A Bayesian Approach to Discovering Truth from Conflicting Sources for Data Integration, March 2012. arXiv:1203.0058 [cs].

A. Appendix: Frontend editor screenshots

This appendix collects full-size screenshots of the web-based editor, including the Similarity Function Creator, Precision Metrics, and the API Playground, as described in detail in Chapter 5. The images are shown at page width for readability.

Figures A.1, A.2, and A.3 show the editors main view with the complete list of similarity functions that were present in the codebase at the time the screenshot was taken. Figures A.4 and A.5 present the full set of precision metrics defined at that time, including their assigned similarity functions and other properties (e.g., operator, description, and scope). Finally, Figure A.6 displays the API Playground with an illustrative request/response pair: a JSON-LD fact set of type `VideoGame` for the example *Elden Ring*.

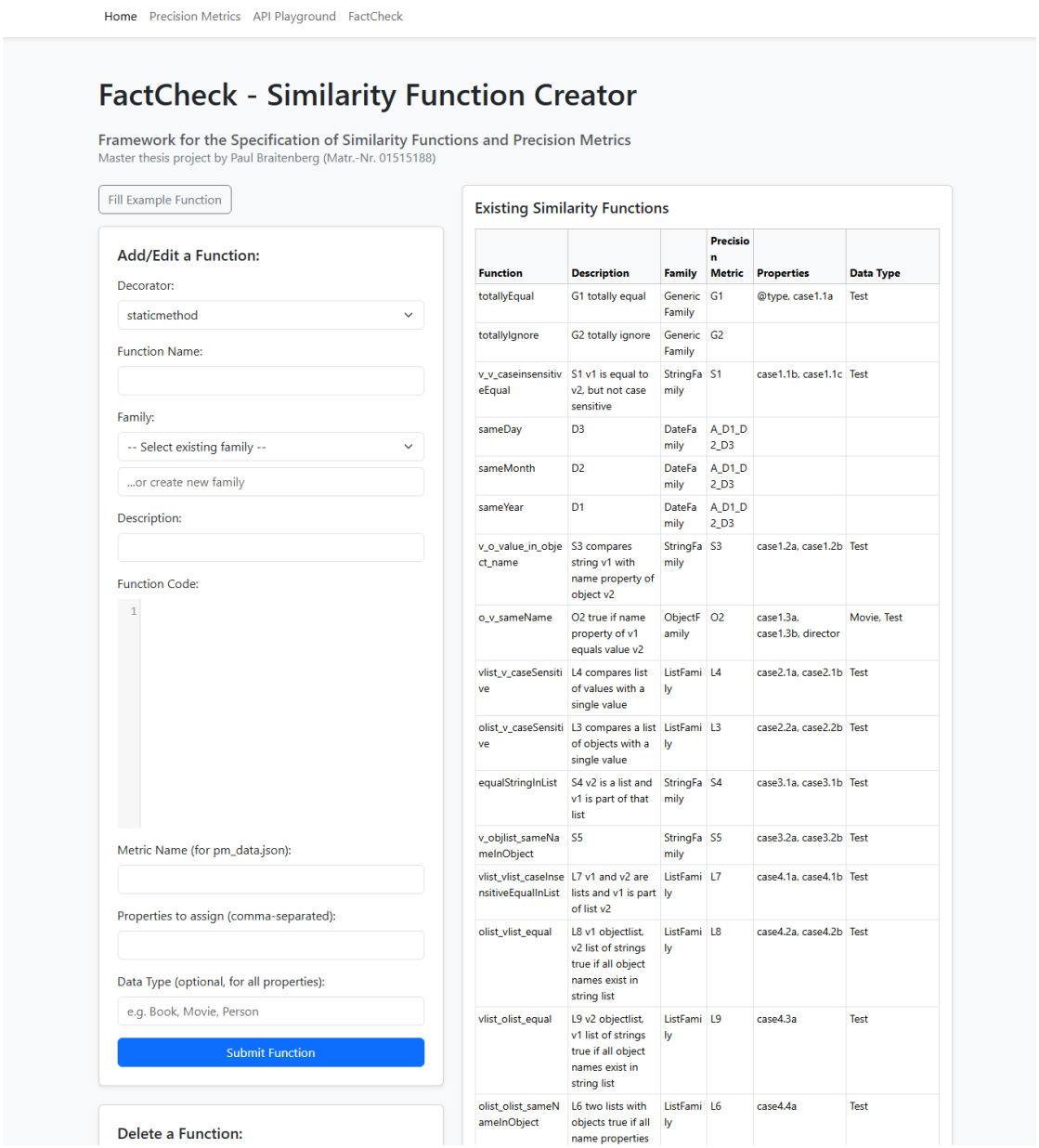
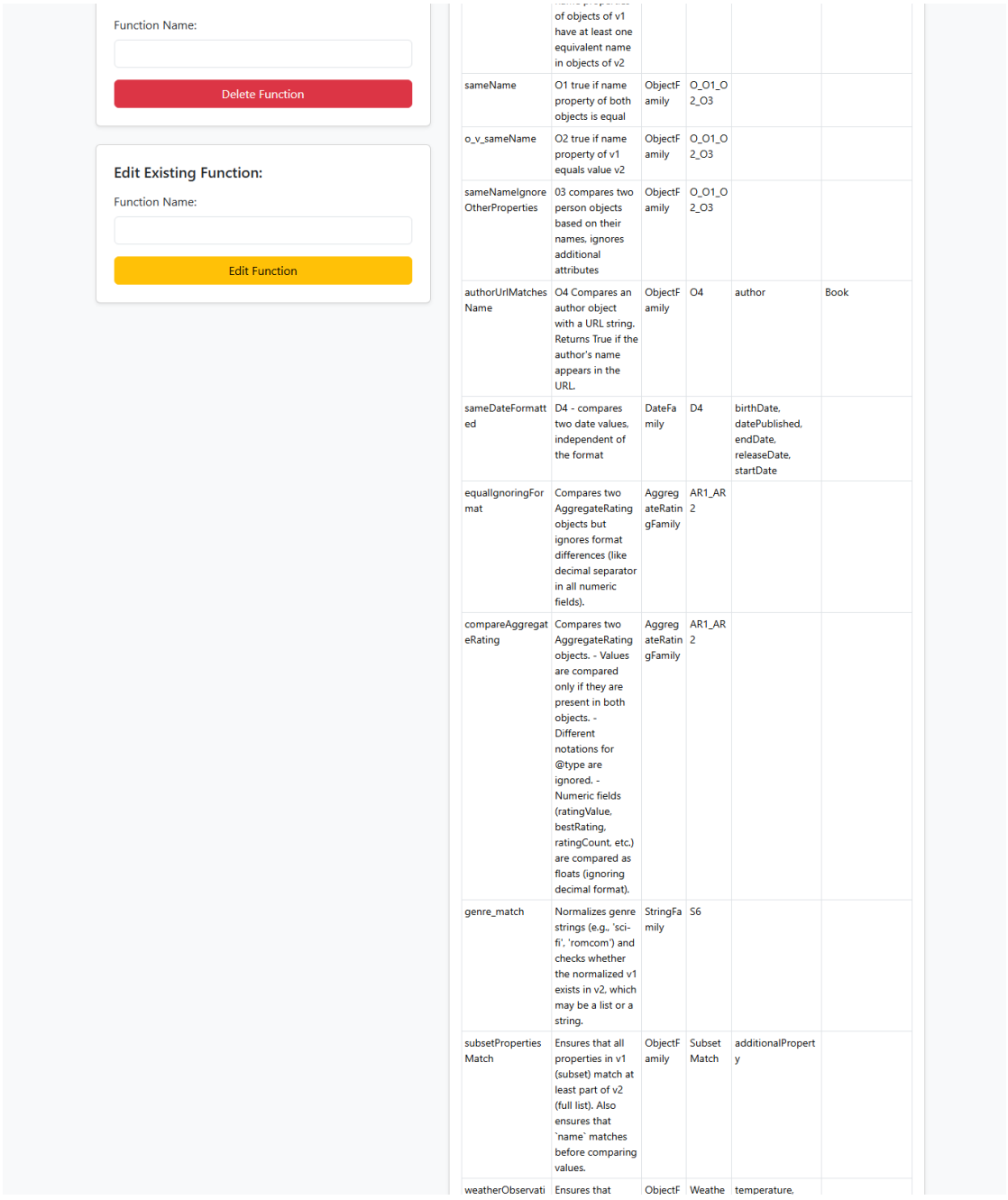


Figure A.1.: Similarity Function Creator Overview (Part 1/3).



	of objects of v1 have at least one equivalent name in objects of v2				
sameName	O1 true if name property of both objects is equal	ObjectFamily	O_O1_O2_O3		
o_v_sameName	O2 true if name property of v1 equals value v2	ObjectFamily	O_O1_O2_O3		
sameNameIgnoreOtherProperties	03 compares two person objects based on their names, ignores additional attributes	ObjectFamily	O_O1_O2_O3		
authorUrlMatchesName	O4 Compares an author object with a URL string. Returns True if the author's name appears in the URL.	ObjectFamily	O4	author	Book
sameDateFormatted	D4 - compares two date values, independent of the format	DateFamily	D4	birthDate, datePublished, endDate, releaseDate, startDate	
equalIgnoringFormat	Compares two AggregateRating objects but ignores format differences (like decimal separator in all numeric fields).	AggregateRatingFamily	AR1_AR2		
compareAggregateRating	Compares two AggregateRating objects. - Values are compared only if they are present in both objects. - Different notations for @type are ignored. - Numeric fields (ratingValue, bestRating, ratingCount, etc.) are compared as floats (ignoring decimal format).	AggregateRatingFamily	AR1_AR2		
genre_match	Normalizes genre strings (e.g., 'sci-fi', 'romcom') and checks whether the normalized v1 exists in v2, which may be a list or a string.	StringFamily	S6		
subsetPropertiesMatch	Ensures that all properties in v1 (subset) match at least part of v2 (full list). Also ensures that `name` matches before comparing values.	ObjectFamily	SubsetMatch	additionalProperty	
weatherObservati	Ensures that	ObjectF	Weathe	temperature,	

Figure A.2.: Similarity Function Creator Overview (Part 2/3).

onMatch	"temperature" in v1 matches v2. Converts temperature units (Kelvin, Celsius, Fahrenheit).	amily	rMatch	windSpeed	
windSpeedMatch	Ensures that "windSpeed" in v1 matches v2. Converts wind speed units (km/h, mph, m/s).	ObjectF amily	Weathe rMatch	temperature, windSpeed	
titleContained	Returns True if either title is contained within the other (case-insensitive), useful for comparing titles with subtitles.	StringFa mily	S7	name	Book, Movie
any_match	Handles comparison where one or both values may be lists. Returns True if any comparison between elements returns True.	LLMCo mpariso nFamily	S7	name	Book, Movie
llm_semantic_equivalent	Uses Ollama to check if val1 and val2 refer to the same concept semantically. Returns True if LLM answers "Yes", otherwise False.	LLMCo mpariso nFamily	G3	jobTitle	
any_match	Handles comparison where one or both values may be lists. Returns True if any comparison between elements returns True.	LLMCo mpariso nFamily	G3	jobTitle	
valueInList	Checks if a single value exists in a list (case-insensitive)	Generic Family	G3	jobTitle	
genre_match	Normalizes genre strings (e.g., 'sci-fi', 'romcom') and checks whether the normalized v1 exists in v2, which may be a list or a string.	StringFa mily	SKOS1	genre	
skos_altlabel_match	Compares if v1 is semantically equal to any v2 entry via DBpedia SKOS altLabels.	SKOSFa mily	SKOS1	genre	

Figure A.3.: Similarity Function Creator Overview (Part 3/3).

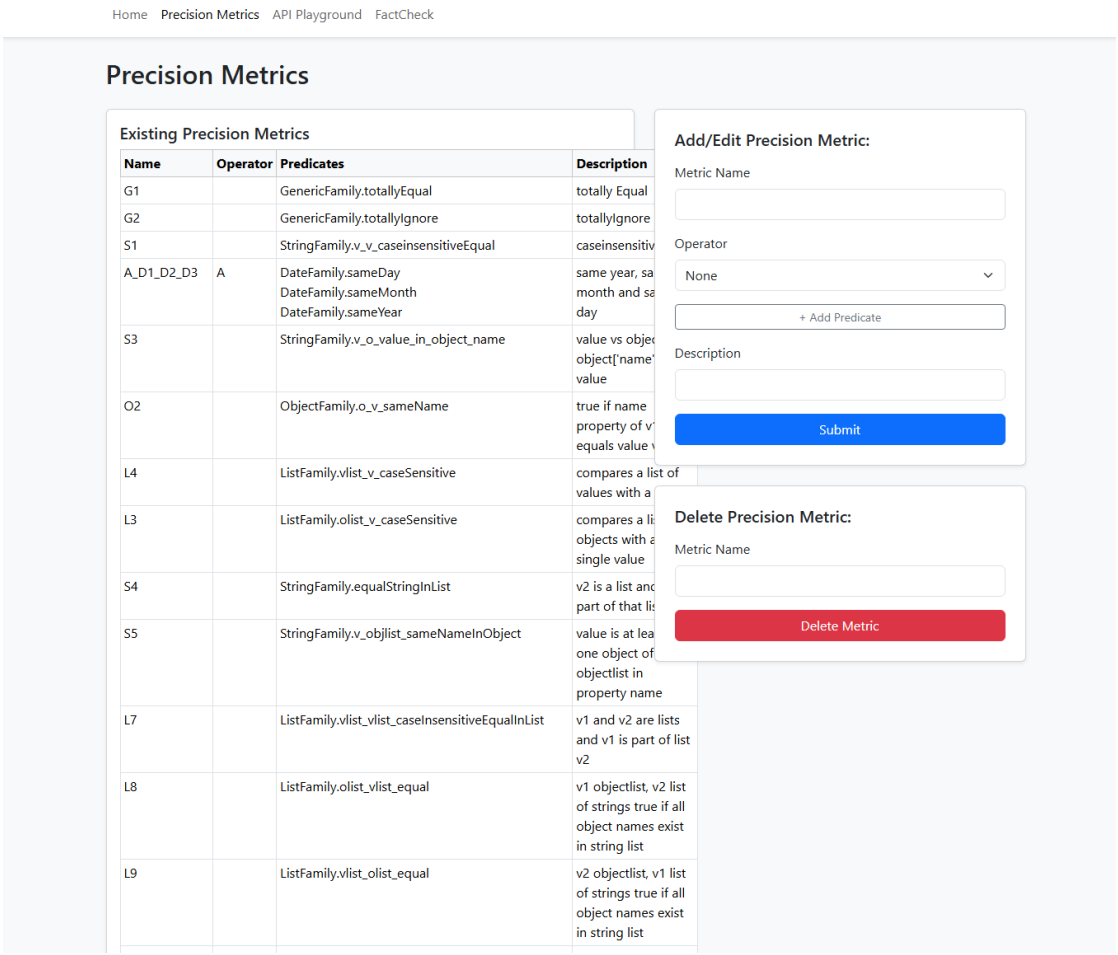


Figure A.4.: Precision Metrics Overview (Part 1/2).

			objects true if all name properties of objects of v1 have at least one equivalent name in objects of v2
O_O1_O2_O3	O	ObjectFamily.sameName ObjectFamily.o_v_sameName ObjectFamily.sameNameIgnoreOtherProperties	same name in object or value
O4		ObjectFamily.authorUrlMatchesName	name is inside the URL
D4		DateFamily.sameDateFormatted	compares two date values independent of the format
AR1_AR2	A	AggregateRatingFamily.equalIgnoringFormat AggregateRatingFamily.compareAggregateRating	Ensures AggregateRating objects are compared with format normalization and common property matching
S6		StringFamily.genre_match	Checks if a genre exists in a list of genres after normalization
SubsetMatch		ObjectFamily.subsetPropertiesMatch	Allows subset matches for additionalProperty, preventing false conflicts.
WeatherMatch	O	ObjectFamily.weatherObservationMatch ObjectFamily.windSpeedMatch	Ensures that "windSpeed" in v1 matches v2. Converts wind speed units (km/h, mph, m/s).
S7	O	StringFamily.titleContained LLMComparisonFamily.any_match	Title is contained within the other (case-insensitive)
G3	O	LLMComparisonFamily.llm_semantic_equivalent LLMComparisonFamily.any_match GenericFamily.valueInList	Checks if a single value exists in a list (case-insensitive)
SKOS1	O	StringFamily.genre_match SKOSFamily.skos_altlabel_match	Compares two values using DBpedia SKOS altLabels

Figure A.5.: Precision Metrics Overview (Part 2/2).

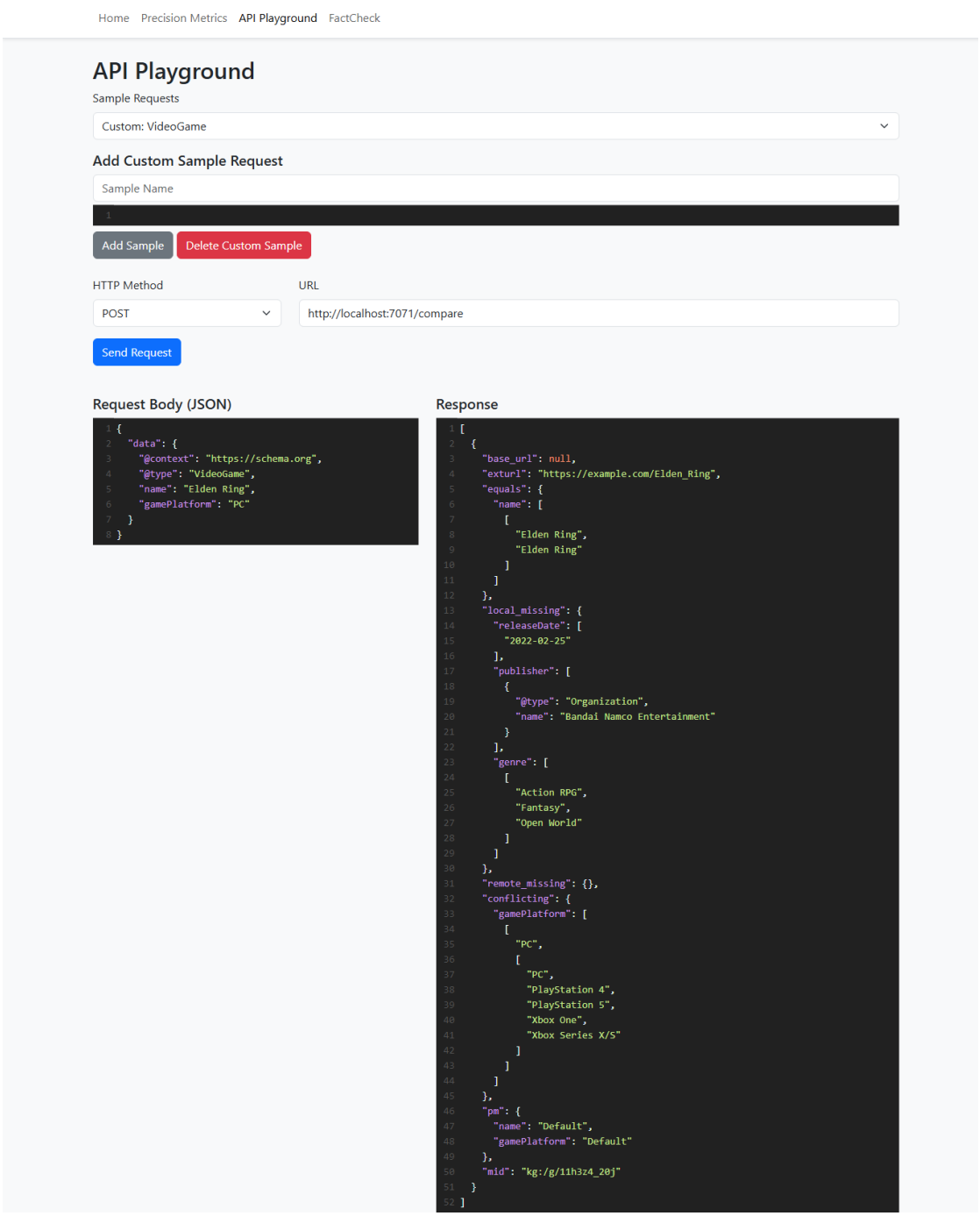


Figure A.6.: API Playground Request/Response