Abstract

Summarize the problem of manual eligibility verification, the proposed Neuro-Symbolic pipeline solution, the experimental comparisons and the critical findings.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

The burden of manual bureaucracy in public administration and the potential of AI to automate legislative interpretation.

1.2 Problem Statement

Identify the challenge of bridging unstructured legal text with deterministic validation logic while mitigating LLM hallucinations.

1.3 Objectives

Define the specific goals of building a Text-to-SHACL pipeline and evaluating its semantic accuracy and operational feasibility.

1.4 Dissertation Structure

Outline the organization of the subsequent chapters and the logical flow of the research.

2 Systematic Literature Review

2.1 Introduction

This chapter details the Systematic Literature Review (SLR) conducted to establish the theoretical foundations of Neuro-Symbolic AI. We approach the current state of research in Neuro-Symbolic AI, specifically focusing on how Large Language Models (LLMs) and Knowledge Graphs (KGs) are combined. We aim to identify existing approaches for extracting rules from text and generating formal logic (SPARQL/SHACL), as well as methods of evaluating the results of such a process.

2.2 Methodology

To ensure scientific rigor and reproducibility, the review adheres to the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines. The process was structured into four distinct phases. First, we defined specific Research Questions (RQs). Second, we executed an automated search strategy on the Scopus database. Third, we applied a two-stage screening process: an initial practical screening of titles and abstracts, followed by a rigorous quality assessment of full texts. This phase utilized specific inclusion/exclusion criteria. Finally, data was extracted from the selected primary studies into a standardized matrix to synthesize key themes, directly related to the RQs.

2.2.1 Research Questions

To achieve our objective, we defined three specific Research Questions (RQs) that guide the data extraction and synthesis process:

- **RQ1:** How are Large Language Models (LLMs) currently utilized to extract structured knowledge and conditional rules from unstructured text?
- **RQ2:** What are the state-of-the-art approaches for translating natural language requirements into executable constraint languages (specifically SHACL and SPARQL)?
- **RQ3:** What methodologies exist for evaluating the functional correctness and operational stability of LLM-generated logic?

RQ1 explores the initial phase of the proposed pipeline (Text-to-Graph), while **RQ2** focuses on the core challenge of logic generation. **RQ3** allows us to critically analyze how existing studies ensure trust and correctness.

2.2.2 Search Strategy

To identify relevant records, we conducted an automated search on the **Scopus** database. Scopus was selected as the source due to its extensive coverage of academic literature. The search was executed on the 1st of **December 2025**. A search query was constructed to find the intersection of Generative AI and Semantic Web technologies. We employed Boolean logic to combine three conceptual blocks:

1. **Generative AI Terms:** ("Large Language Model" OR "LLM")
2. **Target Logic/Language:** ("SHACL" OR "SPARQL")
3. **Symbolic Context:** ("Semantic Web" OR "Knowledge Graph")

These blocks were combined using the `AND` operator. The final search string applied to the Title, Abstract, and Keywords fields was:

```
( "Large Language Model" OR "LLM" ) AND
( "SHACL" OR "SPARQL" ) AND
( "Semantic Web" OR "Knowledge Graph" )
```

We applied some metadata filters during this phase:

- **Language:** Only papers written in **English** were considered.
- **Document Type:** We focused on Articles and Conference Papers, excluding trade journals and errata.

Interestingly, despite the Date Range not being restricted, all results fell in the range of **2023–2026**. This could be explained by the fact that the application of Large Language Models to formal constraint languages (like SHACL) is a nascent field that emerged primarily after the widespread adoption of GPT-4 class models.

The described search strategy yielded an initial set of candidates which were then subjected to the screening process described in the following section.

2.2.3 Inclusion/Exclusion Criteria

Next, we established a set of inclusion and exclusion criteria that reflect the focus of this review. These were applied to Titles and Abstracts during the initial "Practical Screening" phase. Table 2.1 summarizes the criteria used.

Of the papers sought, some could not be retrieved due to access restrictions (paywall). The remaining ones were downloaded assessed for eligibility by reading the full text. In this "Quality Screening" phase, we applied a second set of quality exclusion criteria (QE):

- **QE1 (Name):** We ?
- idea: Specify "Schema-aligned extraction" or "Constraint/Rule extraction". This ensures we are looking for papers that deal with complexity (like eligibility rules), not just connectivity.
- idea: No OpenIE. (same as above?)

The next section summarizes the results following this quality assessment.

Table 2.1: Inclusion and Exclusion Criteria

Category	Inclusion Criteria	Exclusion Criteria
Task Focus	Text-to-Graph extraction, Text-to-SPARQL/SHACL generation, GraphRAG architectures.	Pure NLP (summarization), low-level graph mechanics (Entity Alignment, Link Prediction, Subgraph Extraction).
Methodology	Neuro-Symbolic architectures, Prompt Engineering for logic generation, Fine-tuning, Evaluation Frameworks for Semantic Accuracy.	Traditional Machine Learning (non-generative), Reinforcement Learning without LLMs.
Data Flow	**Forward:** Transforming unstructured text into formal logic or structured data (Text → Logic).	**Reverse:** Transforming structured data into natural language (Verbalization/Explanation).
Mode	Textual inputs with or without preprocessing.	Multimodal studies (Speech/Image), Computer Vision.
Type	Peer-reviewed Articles and Conference Papers.	Conference Proceedings (Meta-entries), Posters, Editorials, non-English papers.

2.3 Results

From an initial set of 125 records, 14 studies were identified as meeting all eligibility criteria.

2.3.1 PRISMA Flow Diagram

The search and screening process can be summarized in the PRISMA flow diagram (Figure 2.1).

Figure 2.1: PRISMA Flow Diagram of the selection process

2.3.2 Data Extraction

Table 2.2 presents the data extraction summary for the 14 included studies. The studies are categorized by their primary theme: (1) Domain-Specific Pipelines, (2) Automated Logic Generation, (3) Validation Frameworks, and (4) Retrieval (GraphRAG).

Table 2.2: Summary of Included Studies (Data Extraction)

Study	Domain / Input	Task	Target Logic	Validation Method
Category 1: Domain-Specific Neuro-Symbolic Pipelines				
Konstantinidis (2025) [1]	Public Service Regulations	Framework Proposal	RDF + SHACL	Conceptual Prototype (No regression testing)
Hanuragav (2025) **Hanuragav2025**	Clinical Study Reports	Compliance Check	SHACL + SPARQL	Deterministic Rule Execution
Oranekwu (2026) **Oranekwu2026**	IoT Security (NIST)	Compliance Check	Ontology + SWRL	Ontology-driven Reasoning
Spyropoulos (2025) **Spyropoulos2025**	Police Reports	Text-to-Graph	RDF Triples	Human-in-the-loop Verification
Category 2: Automated Logic Generation (Text-to-Logic)				
Walter (2026) **Walter2026271**	General (Wikidata)	Text-to-SPARQL	SPARQL	Execution Accuracy (Zero-shot)
Soularidis (2024) **Soularidis2024**	NL Rules	Text-to-SWRL	SWRL	LLM-assisted Generation
Jiang (2025) **Jiang202528**	Scholarly QA	Text-to-SPARQL	SPARQL	Ontology-Guided Prompting
Mashhaditafreshi (2025) **Mashhaditafreshi202536**	JSON Data	Modeling	SHACL Shapes	Human Evaluation of Models
Avila (2025) **Avila2025223**	General QA	Text-to-SPARQL	SPARQL	Benchmark Execution (Auto-KGQA)
Category 3: Validation & Hallucination Control				
Perevalov (2025) **Perevalov2025563**	Multilingual QA	Query Filtering	SPARQL	LLM-based Probabilistic Filtering
Gashkov (2025) **Gashkov2025177**	QA Systems	Query Judging	SPARQL	LLM-as-a-Judge
Tufek (2025) **Tufek202592**	Industrial Standards	Requirement Translation	SPARQL	F1 Score on Logic Translation
Category 4: Retrieval Frameworks (GraphRAG)				
Ongris (2025) **Ongris2025116**	General (Wikidata)	GraphRAG	SPARQL	Jaccard Similarity
Ahmed Khan (2026) **AhmedKhan2026**	Data Center Telemetry	Text-to-Query	SPARQL	Execution Accuracy vs NoSQL

2.4 Thematic Analysis

2.4.1 From Text to Structured Models

Current research demonstrates that LLMs are highly effective at the initial 'extraction' phase, successfully mapping unstructured text into RDF or SHACL skeletons.

2.4.2 Automated Logic Generation

Several studies focus on translating natural language directly into query languages. Walter et al. achieved state-of-the-art results in zero-shot SPARQL generation, while Soularidis et al. explored generating SWRL rules. However, these approaches often struggle with complex, nested logic without guidance.

2.4.3 Validation and Hallucination Control

A critical challenge is ensuring the generated logic is correct. Perevalov et al. propose using an LLM to 'judge' or filter the SPARQL queries. In contrast, Tufek et al. use F1 scores against a gold standard. Crucially, most existing validation methods are probabilistic (LLM-based) rather than deterministic.

2.5 Discussion and Research Gap

While the literature shows success in extraction (2.4.1) and generation (2.4.2), there is a gap in deterministic validation. Papers like Konstantinidis propose the theoretical framework for public services, and Hanuragav applies similar logic to clinical reports. However, no study has yet implemented a comprehensive Mutation Testing framework to rigorously test the structural stability of LLM-generated SHACL shapes for public service eligibility. This dissertation aims to fill that gap.

3 Pilot Study

3.1 Overview

This chapter details the design, implementation and experimental validation of a novel Neuro-Symbolic pipeline for automating public service eligibility checks. The proposed architecture addresses the limitations of "black-box" Large Language Models (LLMs) by enforcing a strict separation between neural interpretation (extracting meaning from text) and symbolic execution (validating logic against data).

The methodology is structured around a "Text-to-Graph-to-Logic" workflow. The system transforms unstructured administrative documents into formal Knowledge Graphs and executable SHACL shapes through a chain of intermediate structured representations. This design prioritizes explainability and determinism, ensuring that the final eligibility decision is derived from explicit, audit-able rules rather than probabilistic token generation.

The chapter is organized as follows: Section 3.2.2 defines the semantic schemas that ground the system. Section 3.2.3 details the four-stage extraction and generation pipeline. Section 3.2.4 describes the validation engine, and Section 3.3 outlines the experimental framework used to stress-test the system's logical capabilities through automated mutation testing.

3.2 Methodology and System Architecture

Test reference [2].

3.2.1 Setup Environment

The pipeline was implemented using Python 3.12.9, utilizing a modular architecture to separate core processing logic from experimental orchestration. The system relies local processing for semantic graph operations and cloud-based APIs for Large Language Model inference.

System Architecture

The codebase follows a functional separation of concerns, organized into three distinct layers:

1. **The Core Logic Layer:** A modular Python library encapsulating the functional logic of the system. Contains the reusable logic, such as API communication, graph

operations, parsing and testing utilities. It also contains the *pipeline core*, which encapsulates the end-to-end extraction-generation workflow.

2. **The Orchestration Layer (The "Cockpit"):** An interactive Jupyter Notebook serves as the control interface. This layer manages the experimental loop, injects configuration variables into the core modules and handles exceptions without interrupting batch processing.
3. **The Persistence Layer:** To ensure auditability and reproducibility, the system employs a strict "Artifact Preservation" strategy. Every experimental run generates a dedicated directory locally, containing all intermediate outputs of the core pipeline. Testing metrics and metadata are saved in a Master CSV file for post-hoc analysis.

Technologies and Libraries

The system integrates standard Semantic Web technologies with modern Data Science tools:

- **RDFLib:** Used for parsing, manipulating and serializing RDF graphs (Turtle format), as well as executing local SPARQL queries.
- **PySHACL:** The standard Python implementation of the SHACL validation engine, used to vaidate the LLM-generated shapes against the citizen data.
- **Pandas:** Used for the post-hoc aggregation and statistical analysis of the testing logs.

3.2.2 Semantic Data Modelling

This pipeline was designed specifically with public service documents in mind. To bridge the gap between unstructured administrative text and deterministic validation logic, two distinct semantic layers were defined. These RDFS schemas serve as the symbolic "grounding" for the Large Language Model.

The Public Service Meta-Model

To ensure semantic interoperability and standardization, the modeling of the public service itself adheres to European formal vocabularies, specifically the Core Public Service Vocabulary Application Profile (CPSV-AP) and the Core Criterion and Evidence Vocabulary (CCCEV). The schema follows a hierarchical structure:

- **cpsv:PublicService**: The root node representing the public service itself.
- **cccev:Constraint**: Connected to the root node via `cpsv:holdsRequirement`, these nodes represent individual preconditions extracted from the text.
- **cccev:InformationConcept**: These nodes are connected to Constraint nodes via `cccev:constrains` and represent the abstract information required to evaluate a constraint.

The adoption of established EU standards is a deliberate architectural choice, made to ensure cross-border interoperability and extensibility. By anchoring the pipeline's output in

the CPSV-AP and CCCEV ecosystems, the generated graphs are natively compatible with the broader European e-Government infrastructure (such as the Single Digital Gateway). Furthermore, this modular design allows for future expansion where the pipeline could automatically ingest the full breadth of these ontologies (complex Evidence mappings, Agent definitions, Output representations), without requiring a fundamental restructuring of the core logic.

Citizen Schema

While the Public Service Meta-Model describes the *rules*, the Citizen Schema describes the *applicant*. This work utilizes a domain-specific RDFS schema tailored to the requirements of each document and generated in a separate workflow (not presented here) by the same LLM used in the implementation of the rest of the pipeline. The model is instructed to use granular instead of aggregate data as nodes (e.g. prefer "Date of Birth" rather than "Age") and is encourgaed to use abstract and reusable classes.

It has been demonstrated that the generation of such schemas can be automated as part of the pipeline [1]. However, for the scope of this pilot study, the Citizen Schema is treated as fixed input context. This methodological choice serves two purposes:

1. **Experimental Control:** By fixing the target schema, we isolate the performance of the LLM in *logic generation* (SHACL/SPARQL) and *extraction*, without the confounding variable of schema generation errors.
2. **Prerequisite for Testing:** The automated testing framework relies on injecting specific faults into the citizen graph (e.g., modifying property values to trigger violations). This requires a deterministic, known-in-advance schema structure; had the schema been generated dynamically during each run, it would be impossible to define a static library of test scenarios targeting specific graph nodes.

3.2.3 The Extraction and Generation Pipeline

The core contribution of this work is the following multi-stage, neuro-symbolic pipeline. The process follows a sequential data flow, depicted in Figure 3.1, consisting of four primary stages.

Stage 1: Document Summarization and Precondition Extraction

The pipeline begins with the ingestion of the raw public service document (PDF). Using a Large Language Model (LLM), the unstructured text is processed to extract a summary of eligibility preconditions. The prompt is designed to filter out administrative noise and standardize the format of the rules. Summarization reduces the cognitive load required for the subsequent logic generation steps.

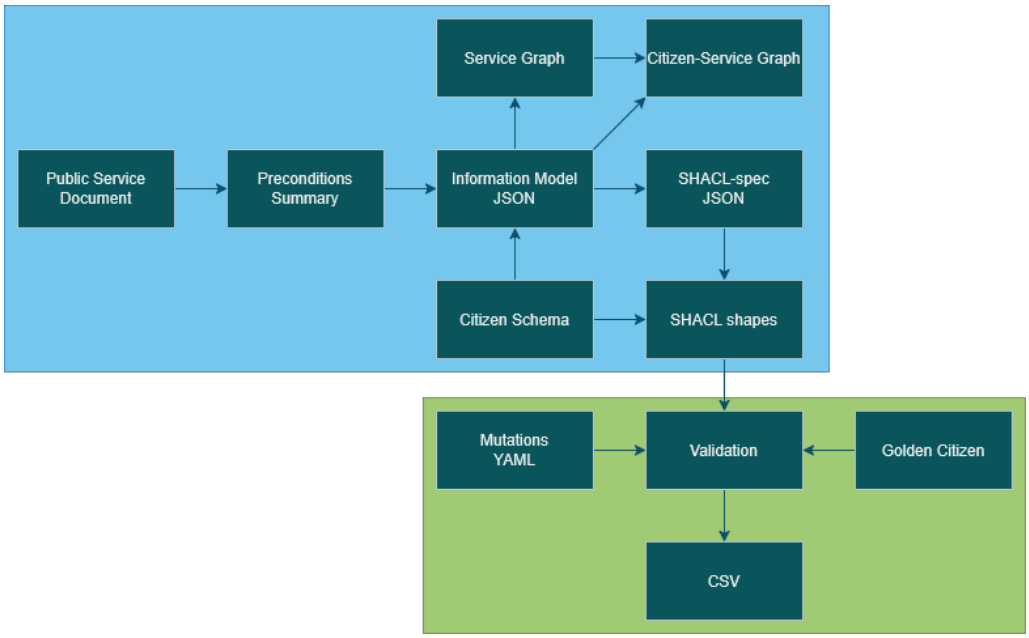

Figure 3.1: Flow Chart of the core pipeline

Stage 2: Information Model Generation

In this critical neuro-symbolic step, the extracted preconditions are transformed into a structured JSON "Information Model". The Information Model organizes the unstructured rules into a strict hierarchy that mirrors the Meta-Model structure:

- **Constraints:** Each eligibility rule is encapsulated as a Constraint object, containing the natural language description of the rule.
- **Information Concepts:** Nested within each Constraint are the abstract Information Concepts, representing the specific pieces of evidence or data required to evaluate that rule.

Inferring these concepts from the list of rules is the main reasoning task of the LLM at this stage. However, a second task it is prompted with is to act as a semantic mapper. The LLM is provided with the Citizen Schema (defined in Section 3.2.2) as a strict vocabulary constraint to prevent the hallucination of non-existent properties. With it, it is instructed to connect each Information Concept with a number of Citizen nodes, by constructing specific traversal paths through the ontology (e.g., mapping the concept of "Applicant Age" to the path `:Applicant/:birthDate`).

The output is strictly enforced using a Pydantic schema definition, ensuring valid JSON structure. The resulting artifact effectively creates a "blueprint" for downstream tasks. It contains all the necessary semantic links to be deterministically serialized into valid CPSV/CCCEV triples in the subsequent stage, while ensuring that all data references are grounded in the controlled vocabulary of the Citizen Schema.

Stage 3: Semantic Graph Construction

Once the Information Model is established, the system deterministically (via Python code) constructs two RDF artifacts without further LLM inference:

1. **The Service Graph:** A formal representation of the public service using the CPSV-AP and CCCEV vocabularies and following the Meta-Model schema defined in Section 3.2.2.
2. **The Citizen-Service Graph (Explainability Layer):** By loading an "Example Citizen" (a valid applicant instance), the system uses the Information Model to link the abstract Information Concepts from the Service Graph directly to the actual data nodes in the Citizen Graph via `ex:mapsTo` edges. This unified graph serves as a visual "audit trail," allowing human inspectors or automated agents to trace exactly which specific data points are being used to evaluate a specific legal requirement.

Both Graphs are serialized using `turtle` syntax and saved to file as artifacts. Interactive visualizations of them are generated using the `pyvis` library and also saved to file as `html` files.

Stage 4: SHACL Shapes Generation

The final stage of the pipeline is responsible for synthesizing the executable validation logic. This stage transforms the abstract requirements from the Information Model into a strictly valid Shapes Constraint Language (SHACL) document.

First, the system deterministically distills the rich Information Model into a simplified, noise-free JSON structure termed the "SHACL-Spec." This intermediate representation reorganizes the structure and retains only the logical primitives required for validation (e.g. rules, target paths and data types). This step acts as a "context cleaner", helping the LLM focus exclusively on code synthesis.

The LLM is then invoked to translate this specification into RDF triples (Turtle format). The system enforces a Dual-Strategy Protocol for logic synthesis. For atomic constraints involving single-hop properties and literal comparisons (e.g., `Citizenship = 'GR'`), the model generates standard `sh:property` shapes. For requirements involving arithmetic, aggregations, date comparisons, or cross-referenced data (e.g., `now() − birthDate > 18`), the model encapsulates the logic within `sh:sparql` constraints. This allows for the expression of complex conditional logic that exceeds the expressivity of the SHACL Core vocabulary. The model is once again restricted to using the fixed Citizen Schema, which is once again given as context to act as a failsafe, in case earlier path generation failed to include crucial nodes.

As a last addition, the LLM generates an error message for evry shape, which is intended to be displayed as part of the Validation Engine report in case of a violation (e.g., "Income exceeds threshold").

The output is a fully serialized `ttl` file containing the `sh:NodeShape` definitions. This

file serves as the executable input for the Validation Engine, the mechanics of which are detailed in the following section.

3.2.4 The Validation Engine

The final component of the architecture is the Validation Engine, which functions as the execution core of the system's symbolic layer. While previous stages focus on structuring and grounding the data, this engine is responsible for applying the generated constraints against specific citizen data to render a final, deterministic eligibility decision.
The engine operates on two distinct RDF graphs:

- **The Shapes Graph:** The `.ttl` file generated by Stage 4 of the pipeline, containing the `sh:NodeShape` definitions and SPARQL constraints within.
- **The Data Graph (Citizen Instance):** An RDF graph representing a specific applicant and a concrete instantiation of the Citizen Schema. It contains the factual assertions about an individual, structured strictly according to the domain ontology.

For the Execution and Reasoning step, the system utilizes PySHACL, a Python-based implementation of the W3C SHACL standard, to perform the validation. The execution follows a standard protocol:

- **Targeting:** The engine identifies the "Focus Node" in the Data Graph (defined as the instance of class `:Applicant`).
- **Constraint Evaluation:** For every Shape mapped to the Applicant, the engine evaluates the corresponding logic. Simple property shapes are validated via graph traversal, while complex conditions trigger the execution of the embedded SPARQL queries against the Data Graph.
- **Entailment:** The engine operates under the RDFS entailment regime, allowing it to infer class hierarchies (e.g., understanding that a `:Child` is also a `:Person`) during validation.

The output of the engine is a formal *Validation Report Graph* adhering to the SHACL standard. This report provides as output:

1. **Boolean Conformance:** A global `sh:conforms` value (True/False), which serves as the system's final decision on eligibility.
2. **Violation Details:** The report includes a set of `sh:ValidationResult` nodes in cases of non-conformance. Each result links to the specific Shape that failed and includes the generated error message, providing explanation for the rejection.

3.3 Experimental Design

To evaluate the reliability, functional correctness and operational stability of the proposed architecture, an experimental framework was developed. The design of this experiment moves beyond simple anecdotal testing, implementing a means to quantify the performance

of the Neuro-Symbolic pipeline under varying conditions.

The core unit of the experiment is defined as a "run". A run represents a single end-to-end execution of the pipeline governed by a specific Configuration Tuple:

`(Document, Model, Prompting Strategy)`

Given that Large Language Models are inherently non-deterministic when operating at non-zero temperature settings, a single successful generation is insufficient to prove any result. To address this, the framework executes a loop of multiple iterations for each unique configuration. This repetition allows for "drowning out" stochasticity and for the results metrics to converge to values that describe the actual stability of the pipeline with more fidelity.

The execution of these runs is done in The Orchestration Layer (see section 3.2.1), which oversees the following lifecycle for every iteration:

1. **Context Initialization:** At the start of a run, a dictionary is initialized. This volatile data structure acts as a "flight recorder," accumulating outputs and metadata.
2. **Pipeline Execution:** The extraction and generation pipeline is triggered. If the pipeline encounters a critical failure, the failure mode is logged and the run is marked as incomplete.
3. **Scenario Validation:** Upon successful generation of a valid SHACL graph, the system proceeds to the Mutation Testing phase (detailed in the following subsection), where the generated logic is stress-tested against a battery of specific scenarios.
4. **Persistence:** Finally, the accumulated metrics are "flushed" to a CSV file. Results are persisted immediately to prevent data loss during long-running batch experiments.

3.3.1 The Mutation Testing Framework

To evaluate the functional correctness of the generated SHACL shapes, the system implements a Mutation Testing Framework. Unlike traditional unit tests that might check for static string matches, this framework dynamically generates RDF graph instances to test whether the generated logic correctly distinguishes between eligible and ineligible applicants. The framework operates on a "Baseline and Perturbation" model, consisting of the components analyzed below.

The "Golden Citizen" Baseline

For each public service document, a single, syntactically perfect RDF graph termed the *Golden Citizen* is manually constructed. This data instance represents an applicant who satisfies *all* eligibility preconditions, albeit marginally. This baseline graph is constructed to adhere strictly to the Citizen Schema. The data values are calibrated to demonstrate marginal eligibility (e.g., if an income upper limit is €12,000, the Golden Citizen might have €11,999). This ensures that the testing framework evaluates the precision of the logic, not just its general functionality.

Scenarios

The test cases are defined in a declarative YAML configuration file. Each entry in this file represents a distinct Scenario, designed to isolate and test a specific logical constraint found in the document. A Scenario definition includes:

1. **Expected Violation Count:** The ground truth for the test. A compliant scenario expects 0 violations, a failure scenario typically expects 1.
2. **Mutation Actions:** A set of instructions to alter ("mutate") the Golden Citizen.

Crucially, mutations are designed to be atomic. Each scenario targets a single "fact" in the graph (e.g., changing a Literal value or a URI reference) to nudge the applicant from an "Eligible" state to a "Non-Eligible" state. This isolation allows the Validation Engine to pinpoint exactly which specific rule the LLM failed to generate correctly, if any.

The Mutation Engine

For every iteration ("run"):

1. The system loads the Golden Citizen graph into memory.
2. It creates a deep copy of the graph to ensure test isolation.
3. Once per scenario, it applies the Patch Logic. The engine parses the Turtle snippets defined in the YAML actions (e.g., `ex:Income :amount 12,000.1`) and updates the graph triples accordingly. This allows for complex graph transformations, such as replacing nodes or updating relationships, without manual RDF manipulation.

The resulting Mutated Citizen Graph and the Generated Shapes Graph (from Stage 4) are then passed to the afformentioned Validation Engine (section 3.2.4). The boolean outcome (`conforms`) and the number of violations are captured and logged to later be compared against the Expected Violation Count defined in the scenario.

3.3.2 Experimental Configurations

Recall the configuration tuple around which the experiment was designed:

`(Document, Model, Prompting Strategy)`

For the experimental part of this work we chose 2 documents, 2 models and 3 prompting strategies, for a total of 12 different experimental configurations. This combinatorial approach allows for the isolation of specific failure modes, distinguishing between errors caused by document complexity, model reasoning capacity, or prompting sufficiency. Below we analyze each component of the tuple and the configurations explored in the scope of this work.

Document Corpora (Use Cases)

This selection tests the pipeline's ability to generalize across different domains and logical structures. Two public service documents were selected to represent different levels of beurocratic complexity.

Student Housing Allowance (High Complexity)

Selected as the "Stress Test" for the system. This document is characterized by:

- **Deep Graph Traversal:** Verification requires traversing multiple hops (Applicant → Parents → Properties → Location).
- **Recursive Arithmetic:** It involves dynamic income thresholds, calculated based on the count of dependent children (e.g., $Limit = Base + (N \times Bonus)$).
- **Referential Integrity Constraints:** Verification requires comparing the identity of URI nodes rather than literal values (e.g., validating that the `:UniversityCity` node is distinct from the `:FamilyResidenceCity` node).

Special Parental Leave Allowance (Intermediate Complexity)

Selected to evaluate standard administrative processing. This document focuses on:

- **Categorical Classification:** Eligibility relies on specific enumerated values (e.g., Employment Sector must be "Private" or "Public").
- **Temporal Logic:** Involves duration calculations (e.g., "1 year of continuous employment") rather than complex arithmetic aggregations.

Large Language Models

The experiment utilizes the Google Gemini 2.5 family of models to evaluate the trade-off between reasoning capability and computational efficiency.

- **Gemini 2.5 Pro:** The high-parameter "reasoning" model. It is hypothesized to excel at complex SPARQL generation and abstracting vague requirements into formal logic, potentially at the cost of higher latency.
- **Gemini 2.5 Flash:** The lightweight, low-latency model. It serves to test the feasibility of a "high-throughput" pipeline. A key research question is whether this smaller model can adhere to the strict SPARQL syntax requirements without the deep reasoning capabilities of the Pro variant.

Prompting Strategies

Three distinct prompting strategies were implemented to evaluate the impact of "In-Context Learning" and "Self-Correction" on code quality.

Default Strategy (Few-Shot with Guardrails)

This strategy represents the baseline optimized approach. The system prompt instructs the model to act as an "Expert" and provides:

- **Proposed Strategy:** Explicit instructions to choose between, depending on the input.
- **Syntactic Guardrails:** A set of negative constraints derived from pilot testing errors.
- **Few-Shot Examples:** Concrete examples demonstrating correct and desired outputs.

Zero-Shot Strategy (Ablation Study)

To quantify the value of the engineering effort put into the Default prompt, the Zero-Shot strategy removes all Few-Shot Examples: the model is given the instructions but no reference implementations. This tests the model's innate reasoning prowess and knowledge of syntax versus its reliance on pattern matching from examples.

Reflexion Strategy (Iterative Self-Correction)

This strategy implements a *Prompt Chaining* loop to address the non-deterministic nature of LLM code generation.

1. The model generates a draft response using the Default strategy.
2. The output is passed back to the model with a new "persona": *"Senior Data Quality Assurance Auditor."* This agent is instructed to critique the quality of the draft with regards to criteria such as completeness, logical contradictions and syntactic validity.
3. If errors are found, the model rewrites the response based on its own critique.

This configuration evaluates the efficacy of self-correction mechanisms in code generation, specifically testing whether the computational overhead of iterative refinement yields a statistically significant reduction in syntactic and logical errors.

3.3.3 Evaluation Metrics

To move beyond qualitative observation, the experimental framework was designed in such a way to capture a granular dataset for every execution cycle. This data collection strategy was designed to decouple structural failures (code that does not compile) from logical failures (code that compiles but yields incorrect decisions), enabling a multi-dimensional analysis of pipeline performance.

Data Collection

For every experimental run, the system persists a dataset that captures the complete state of the pipeline at the moment of execution, categorized into four distinct dimensions:

- **Configuration Metadata:** Contextual fields regarding a unique Run ID, timestamp, the specific document input, the LLM employed and the active prompting strategy.
- **Artifact Fingerprinting:** To track the stability and uniqueness of the LLM's output, the system computes and logs the cryptographic hashes (MD5) of the generated graphs. This allows for the detection of potentially identical artifacts generated across different runs.
- **Syntactic Integrity Verification:** Before execution, the system first verifies if the generated text is a valid RDF/Turtle graph (parsable by RDFLib), and secondly, it performs a "deep compile" check on every embedded SPARQL constraint to ensure the query syntax adheres to the SPARQL standard. Both errors, if they occur, are flagged differently to be distinguishable.

- **Validation Outcome Metrics:** The raw output of the validation engine is captured in detail. This includes the Actual Violation Count, the Expected Violation Count (derived from the scenario definition) and a serialized list of the specific Violated Shapes. These fields facilitate the calculation of granular error metrics beyond simple binary accuracy.
- **Operational Diagnostics:** To monitor system health, metrics such as end-to-end Execution Time (latency) and specific Error Messages (e.g., Python exceptions) are logged. These fields are critical for quantifying the operational stability of the external API dependencies.

Performance Indicators

The analysis of this dataset focuses on two primary dimensions of success.

Syntactic Validity

The first hurdle for any code-generating system is the production of executable syntax. This metric quantifies the percentage of runs where the LLM produced a `.ttl` file that could be successfully parsed by the RDFLib graph library and whose embedded SPARQL queries could be compiled without error. A run that fails this check is distinguished from runs that simply produce incorrect logic.

Functional Logic Accuracy

For runs that pass the syntax check, the focus shifts to logical fidelity. This is measured by comparing the system's eligibility decision against the known ground truth of the mutation scenarios. By treating the validation outcome as a binary classification task—where a "Violation" is the Positive class and "Conformance" is the Negative class—standard machine learning metrics such as Precision, Recall, and F1 Score are calculated. Precision measures the system's trustworthiness (avoiding false alarms), while Recall measures its safety (successfully catching ineligible applicants).

3.4 Conclusion

This chapter has detailed the architectural and experimental foundations of the Neuro-Symbolic pipeline. By combining a schema-grounded generation process with a deterministic mutation testing framework, the system is designed to provide a quantifiable evaluation of LLM capabilities in the context of this task. The following chapter presents the results of these experiments, analyzing the pipeline's performance across the afformentioned dimensions.

4 Results

This chapter presents the quantitative findings obtained from the experimental evaluation of the neuro-symbolic pipeline. The analysis strictly follows the performance metrics defined in the methodology, assessing the system across three cascading thresholds of success: syntactic validity (code generation), functional logic accuracy (reasoning fidelity), and overall operational reliability (end-to-end feasibility). Broader interpretation of these patterns and their implications for public administration systems are discussed in Chapter 5.

4.1 Experimental Dataset

The experimental campaign consisted of a total of 170 end-to-end pipeline executions ("Runs"). Each run consisted of the steps that were analyzed on the previous chapter. The distribution of these runs across the varying configurations is detailed in Table 4.1. Due to the operational constraints discussed in Section 6.1, the dataset is unbalanced, with the "Flash" model variant accounting for a larger proportion of the total runs.

Table 4.1: Distribution of Experimental Runs per Configuration

Document	**Model**	**Prompt Strategy**	**Runs** (N)
Parental Leave	gemini-2.5-flash	Default	20
		Reflexion	20
		ZeroShot	20
	gemini-2.5-pro	Default	10
		ZeroShot	10
Student Housing	gemini-2.5-flash	Default	20
		Reflexion	20
		ZeroShot	20
	gemini-2.5-pro	Default	10
		Reflexion	10
		ZeroShot	10

4.2 Syntactic Validity

The first criterion for the pipeline's utility is the generation of syntactically valid code. A run is considered "Syntactically Valid" only if the LLM produces a Turtle (`.ttl`) file that can be parsed by RDFLib *and* contains SPARQL constraints that successfully compile without syntax errors.

4.2.1 Success Rate

Figure 4.1 illustrates the success rates across all configurations.

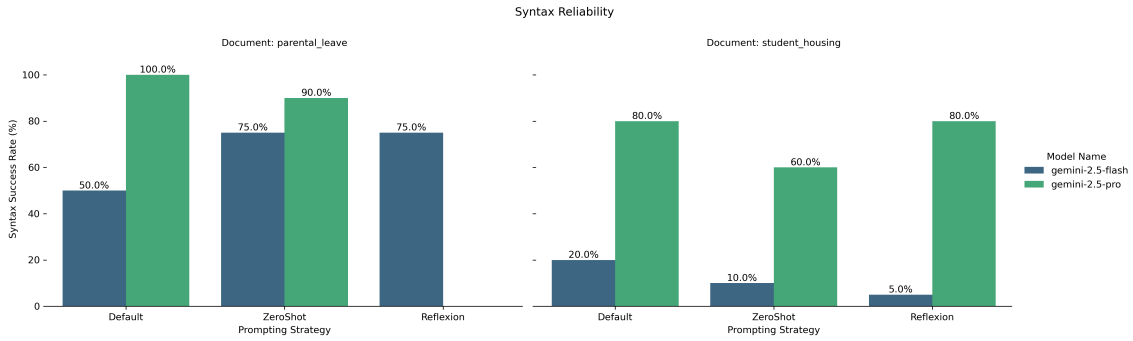

Figure 4.1: Syntactic Validity Rates by Configuration

Impact of Document Complexity

The complexity of the source document served as a strong predictor of failure.

In the case of the Parental Leave document (Intermediate complexity), both models performed adequately. Even the weaker Flash model achieved a 75% validity rate using the Reflexion and ZeroShot strategies.

On the contrary, the Student Housing document (High complexity) acted as a strict filter. Flash failed to produce valid code in the vast majority of attempts (36 out of 60 runs failed syntax checks), whereas Pro proved strong enough to handle the increased logical depth, though it still suffered a 20% degradation compared to the simpler use case.

Impact of Model Class

The data reveals a disparity in coding capability between the two model variants.

The Gemini 2.5 Pro model demonstrated high reliability, achieving a 100% success rate on the Parental Leave document and maintaining an 80% success rate on the complex Student Housing document (under Default prompting).

In contrast, the Gemini 2.5 Flash model struggled significantly with syntactic precision. While it achieved moderate success on the simpler Parental Leave document (ranging from 50% to 75%), its performance collapsed on the complex Student Housing document, with success rates dropping as low as 5% (Reflexion) to 20% (Default).

Impact of Prompting Strategy

The impact of prompting strategies varied by model architecture.

For Flash, the *Reflexion* strategy provided a significant boost on the simpler document (improving validity from 50% to 75%). However, this benefit vanished on the complex document, where Reflexion actually performed worse (5%) than the Default prompt (20%).

For Pro, the *Default* and *Reflexion* strategies performed identically (80% on Housing), while the *ZeroShot* strategy resulted in a notable drop in stability (falling to 60% on Housing).

4.2.2 Failure Mode Analysis

To better understand the mechanisms of failure, the invalid runs were categorized by error type: *RDF Syntax Errors* (invalid Turtle file structure) and *SPARQL Syntax Errors* (malformed queries within valid Turtle). Figure 4.2 presents the error rates normalized by the total number of runs for each model-document pair.

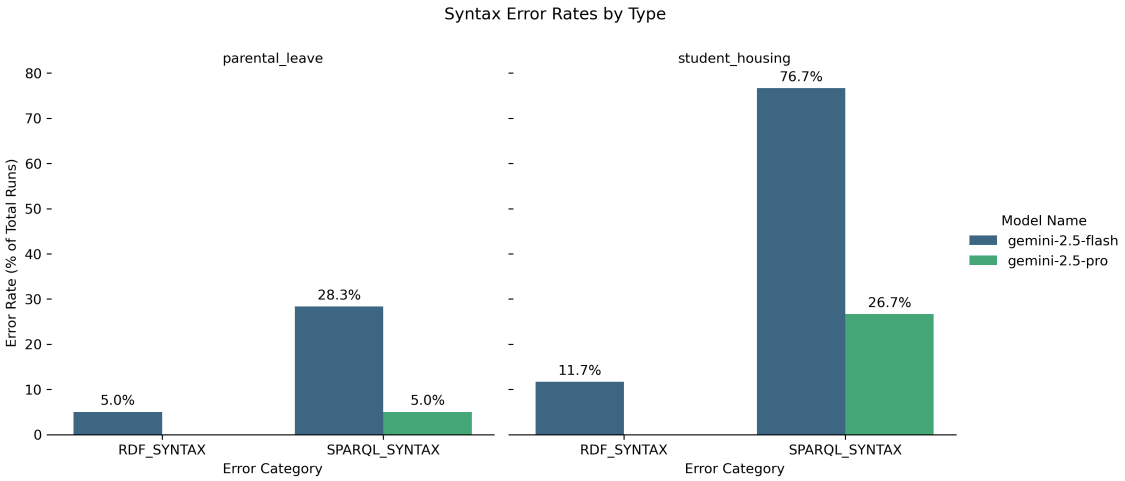

Figure 4.2: Distribution of Syntax Error Types by Model and Document

The data indicates that SPARQL Syntax Errors were the dominant failure mode across all configurations.

Gemini 2.5 Flash exhibited a high frequency of SPARQL errors, particularly on the complex Student Housing document, where 76.7% of all runs failed due to query syntax. Notably, Flash also produced a non-negligible rate of RDF Syntax errors (5.0% on Parental Leave, 11.7% on Student Housing), indicating occasional failures in generating even the fundamental file structure.

Gemini 2.5 Pro demonstrated significantly higher stability. It produced zero RDF syntax errors across all 50 experiments. Its failures were exclusively confined to SPARQL syntax, with error rates of 5.0% on the simpler document and 26.7% on the complex document.

4.3 Logic Validity

For the subset of runs that successfully produced syntactically valid code, the focus shifts to *Functional Logic Accuracy*. A run is classified as having "Perfect Logic" if and only if the generated SHACL shapes correctly identify the expected number of violations for *every single scenario* in the test suite (both the baseline Golden Citizen and all edge cases). Runs that crashed during execution or failed syntax checks were excluded from this analysis to isolate the reasoning capability of the models.

4.3.1 Success Rate

Figure 4.3 illustrates the rate of flawless logical execution across configurations.

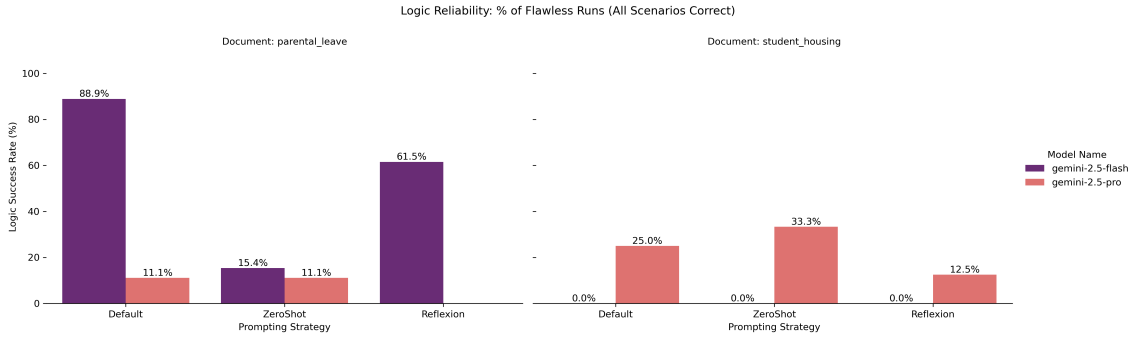

Figure 4.3: Logic Validity: Percentage of Flawless Runs (All Scenarios Correct)

Impact of Document Complexity

Consistent with the syntax results, the complexity of the document was the primary determinant of success.

Parental Leave, having simpler logic, allowed for high performance, with the best-performing configuration (Flash/Default) achieving an 88.9% perfect run rate.

Student Housing and its complex logic requirements caused a near-total collapse in functional accuracy. Across all models and prompts, the highest achieved success rate was only 33.3%, with many configurations failing to produce a single logically correct run.

Impact of Model Class

The performance relationship between models *inverted* depending on the task.

On the simple document, Flash significantly outperformed Pro, achieving 88.9% accuracy (Default) compared to Pro's 11.1%. However, on the complex document, Flash failed completely, with a 0.0% success rate across all 60 attempts.

While Pro underperformed on the simple task, it was the only model capable of solving the complex Student Housing logic, achieving success rates between 12.5% and 33.3%.

Impact of Prompting Strategy

Removing examples (ZeroShot) caused a sharp drop in accuracy from 88.9% to 15.4% for Flash on the simple document. Conversely, for Pro on the complex document, ZeroShot unexpectedly yielded the highest accuracy (33.3%).

The self-correction strategy (Reflexion) did not yield consistent improvements. For Flash, it reduced accuracy from 88.9% to 61.5% on the simple document. For Pro, it performed roughly equivalent to the Default strategy.

4.3.2 The Syntax-Logic Gap

A comparison between the syntax validity rates (Figure 4.1) and logic accuracy rates (Figure 4.3) reveals a distinct degradation in performance as the evaluation becomes stricter. This is apparent if we isolate the complex Student Housing use case. While the Pro model generated valid syntax in ≈80% of runs, only ≈25% of those valid runs contained correct logic. The Flash model struggled at both levels, with low syntax validity (20%) and zero functional correctness (0%).

4.4 Overall Pipeline Reliability

Beyond specific syntax and logic metrics, this section evaluates the system's viability as an end-to-end automated service. The analysis considers two perspectives: the operational stability of the pipeline and its reliability inside the broader context of this work, which is public service recommendations.

4.4.1 Pipeline Feasibility and Attrition

Figure 4.4 presents the distribution of final outcomes for all 170 experimental runs. This "Waterfall Analysis" categorizes every attempt into a single mutually exclusive outcome, revealing the attrition rate of the system.

The data indicates a high attrition rate:

- **Syntax Failures:** The majority of runs failed early. SPARQL Syntax Errors accounted for the largest share of failures (N=72), followed by RDF Syntax Errors (N=10).
- **Logic Failures:** Of the runs that compiled, a significant portion (N=55) produced code that executed but failed to correctly validate all test scenarios.
- **Success:** Only 25 runs (14.7% of the total) achieved the status of a "Perfect Run," generating both valid syntax and flawless logic across all edge cases.
- **System Stability:** Operational crashes (Python/API errors) were rare (N=8), indicating that the underlying infrastructure, retry mechanisms and exception handling were largely sufficient.

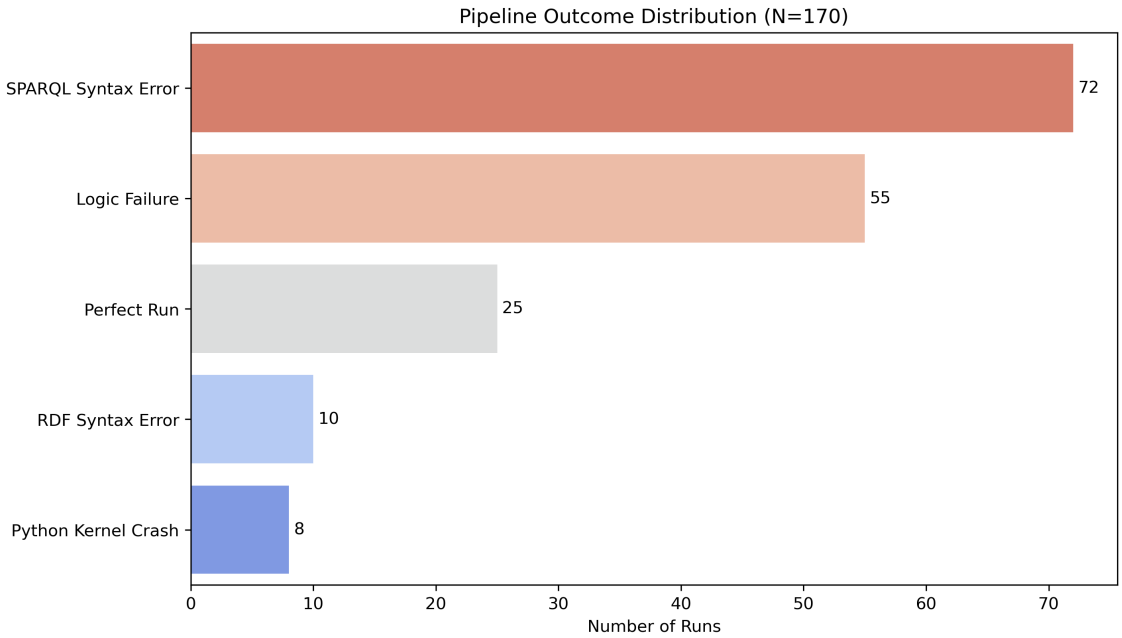

Figure 4.4: Distribution of Final Pipeline Outcomes.

4.4.2 Recommender System Accuracy

To evaluate the system's utility as a public service recommender, the validation outcomes were aggregated into a Confusion Matrix (Figure 4.5). In this context, the classes are defined based on the goal of recommending eligible services:

- **Positive Class (Recommendation):** The system validates the citizen as Eligible.
- **Negative Class (Rejection):** The system flags at least one Violation.

To interpret the confusion matrix in the specific context of public service recommendations, the standard machine learning classifications were mapped to domain-specific service outcomes, as defined in Table 4.2.

Table 4.2: Definition of Classification Outcomes in the Recommender Context

	System: "Violation" (Rejection) *Triggered Violations* > 0	**System: "Conforms"** (Recommendation) *Triggered Violations* $= 0$
Citizen is Ineligible *Expected Violations* > 0	**True Negative (TN)** *Correct Rejection* (System works)	**False Positive (FP)** *Bad Recommendation* (Trust Risk)
Citizen is Eligible *Expected Violations* $= 0$	**False Negative (FN)** *Missed Opportunity* (Service Failure)	**True Positive (TP)** *Correct Recommendation* (System works)

The confusion matrix is then created based on this terminology.

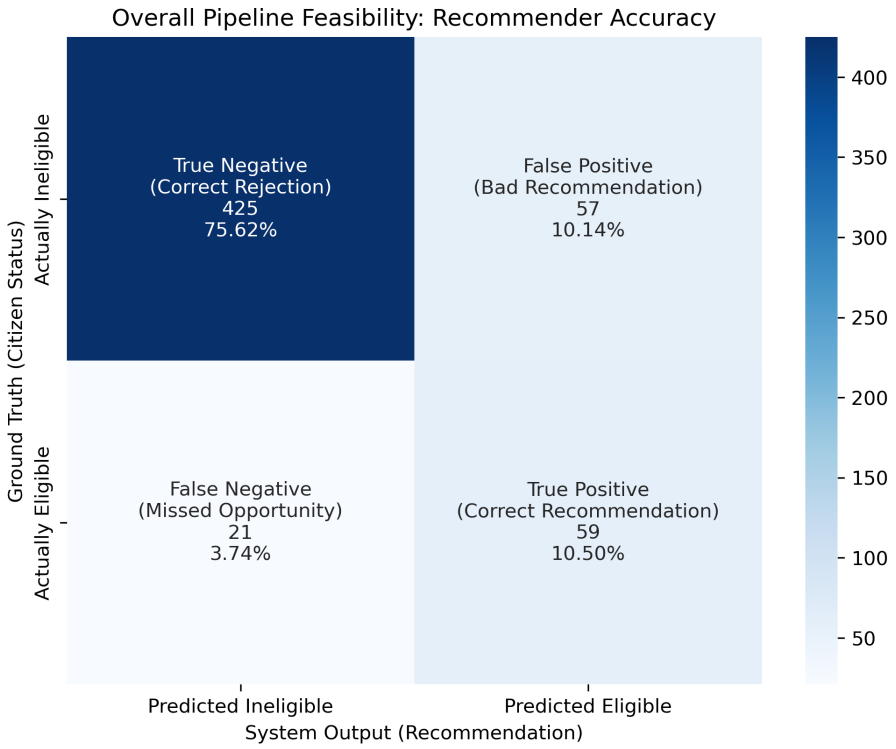

Figure 4.5: Confusion Matrix of Eligibility Recommendations

The matrix reveals the system's risk profile:

- **True Positives (10.5%):** In 59 cases, the system correctly identified and recommended the service to an eligible citizen ("Correct Recommendation"). This confirms the system's ability to successfully validate legitimate claims when the generated logic is sound.
- **False Positives (10.1%):** In 57 cases, the system erroneously recommended the service to an ineligible citizen ("Bad Recommendation"). This represents a "Trust Risk," where users might be guided to apply for benefits they cannot receive.
- **True Negatives (75.6%):** The system correctly rejected ineligible applicants in the majority of cases.
- **False Negatives (3.7%):** In 21 cases, the system incorrectly rejected an eligible applicant ("Missed Opportunity"). While this number is low, it represents a "Service Failure," denying access to entitled benefits.

what is important in this context?

4.5 Semantic Stability (nice-to-have)

If we have time to do it, this chapter will use data from semantic similarity metrics drawn from past run artifacts on file.

5 Discussion

Here we will interpret the results and give subjective opinions and other observations.

5.1 The Logic Traps

Interpret the recurring logical failures in logically sound SPARQL generation, discussing the LLM's struggle with aggregations, joins, recurrency and closed loops.

5.2 The Syntactic Failures

Interpret the models' usage of wrong syntax, using other languages (perhaps more training data) despite being explicitly told to avoid it.

5.3 Domain Influence

Discuss the impact of complex vs simple domains or docuents, semantic web wise.

5.4 Prompt Engineering

Discuss the differences the tested prompting techniques made (or didn't make).

5.5 Model Size Trade-offs

Discusses the counter-intuitive finding where the smaller model outperformed the larger model in literal constraint extraction and summarizarion (where else?)

6 Limitations & Future Work

6.1 Limitations

Acknowledge the limited document sample size, API restrictions and the reliance on a specific vendor's ecosystem, limited prompt engineering, limited models (free tier). Replication Crisis Critique the reliance on proprietary Model-as-a-Service infrastructure, arguing that operational instability renders them unsuitable for critical pipelines. Maybe this can be in the Limitations chapter.

6.2 Future Research Directions

Propose a roadmap for using local open-source models to ensure sovereignty, the implementation of iterative "Self-Correction" agents to fix syntax errors, ideas for more robust testing of this kind of pipeline, ideas not implemented by this work for scoping reasons.

The decision to base the schema on an existing vocabulary was with good reason. This design allows the graphs generated by the pipeline to include more classes of the used ontologies, for future integration with more sophisticated systems. The pipeline itself could also be expanded upon to include more classes.

Make an effort to find why the pipeline failed logically when it did. Many times it was the preconditions extraction and not the text-to-logic part. many times the models failed to correctly interpret what was a precondition and what was an administrative step.

Bibliography

[1] I. Konstantinidis, I. Magnisalis, and V. Peristeras, "A framework for a public service recommender system based on neuro-symbolic ai," *Applied Sciences (Switzerland)*, vol. 15, no. 20, 2025, Cited by: 0; All Open Access; Gold Open Access. DOI: 10.3390/app152011235 [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105020911060&doi=10.3390%2Fapp152011235&partnerID=40&md5=f4bdd4c183c3b9bf9f0468a5bc8d4651

[2] N. Laoutaris, *Exploring neuro-symbolic pipelines for structured knowledge extraction*, https://github.com/n-laoutaris/neuro-symbolic-dissertation, Accessed: 2025-12-14, 2025.

A Appendix placeholder

Extracts of the generated SHACL shapes (both valid and broken examples).

Samples of the YAML Mutation Scenarios.

Samples of RDFS Ontologies used.