INTERNATIONAL
HELLENIC
UNIVERSITY

Dissertation Title Goes Here

Student Name

SID: 12345678

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

OCTOBER 2012

THESSALONIKI – GREECE

(title page – delete this line)

Dissertation Title Goes Here

Student Name

SID: 12345678

Supervisor:	Prof. Name Surname
Supervising Committee Members:	Assoc. Prof. Name Surname
	Assist. Prof. Name Surname

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Information and Communication Systems

OCTOBER 2012

THESSALONIKI – GREECE

Abstract

This dissertation was written as a part of the MSc in ICT Systems at the International Hellenic University. Here goes a summary of the dissertation (1-2 paragraphs). Add one last paragraph acknowledging the supervisor and the people who contributed (collaborators, other scientists, etc.). Abstract length should not exceed one page.

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

Student Name Date

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.

Why is this work important?

1.2 Problem Statement

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

1.3 Objectives

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

1.4 Dissertation Structure

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

2 Systematic Literature Review

This chapter details the Systematic Literature Review (SLR) conducted to establish the theoretical foundations of Neuro-Symbolic AI.

2.1 Introduction

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 years 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 (Domain & Logic Mismatch):** Articles situated in descriptive scientific domains (e.g., bioinformatics, chemistry) where the knowledge structure is purely factual/relational rather than normative or rule-based, offering low transferability to eligibility logic.
- **QE2 (Complexity & Task Focus):** Sources focusing on simple factoid Question Answering (KGQA) that do not analyze the extraction or generation of complex

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), Dataset creation.
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, Temporal Data.
Type	Peer-reviewed Articles and Conference Papers.	Conference Proceedings (Meta-entries), Posters, Editorials, Preliminary Results.

conditional constraints (if-then-else logic) required for compliance or eligibility

- **QE3 (Methodological Maturity):** Studies limited to model-vs-model benchmarking or evaluation of existing datasets without proposing novel neuro-symbolic pipeline architectures or logic-validation frameworks.

The next section summarizes the results following this quality assessment.

2.3 Results

From an initial set of 125 records, 25 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).

2.3.2 Data Extraction

Table 2.2 presents the data extraction summary for the 25 included studies. The studies are categorized

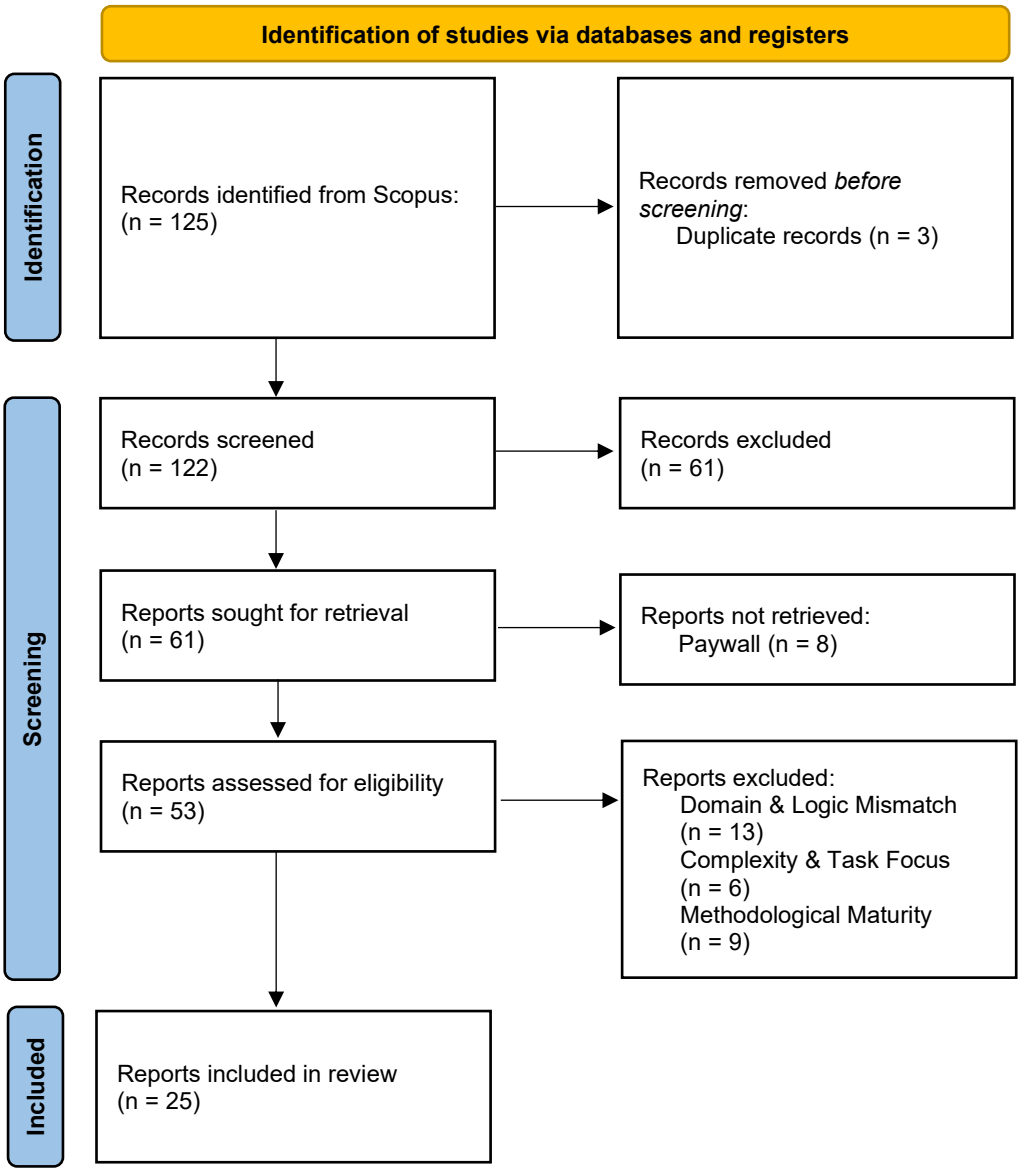

Figure 2.1: PRISMA Flow Diagram of the selection process

Table 2.2: Final Synthesis Matrix of Included Studies ($n = 25$)

Study	Core Task / Domain	Logic	Neuro-Symbolic Integration	Validation Method
Category 1: Public Administration & Normative Compliance				
Konstantinidis (2025)	Service Rec. (Public Admin)	SHACL	LLM extraction from PDF Laws	Human Expert Assessment
Oranekwu (2026)	Cyber Compliance (IoT)	SPARQL	Ontology-driven RAG pipeline	Deterministic Reasoning
Spyropoulos (2025)	Entity Mining (Police)	RDF/OWL	LLM-driven Narrative Extraction	Visual/SPARQL Verification
Hanuragav (2025)	CSR Validation (Medical)	SHACL	Deterministic ETL + YAML drafting	Deterministic Execution
Category 2: Automated Logic Synthesis & Semantic Parsing				
Agarwal (2024)	Complex QA (General)	KoPL	SymKGQA: Symbolic Program Gen.	Hits@1 and F1 Scores
Avila (2025)	Scientific QA (SciQA)	SPARQL	Hybrid RAG + Few-shot ICL	F1-score on Benchmarks
Jiang (2025)	Multi-KG QA (General)	SPARQL	Two-stage Sketch-based parsing	Hits@1 and F1 Scores
Shah (2024)	Multi-hop QA (General)	SPARQL	Planned Query Guidance	Execution Match Accuracy
Walter (2026)	Generic Reasoning (Multi)	SPARQL	Zero-shot Iterative Agent	Probabilistic (Benchmark F1)
Soularidis (2024)	Rule Gen. (Search/Rescue)	SWRL	Template-driven GPT-4o Prompting	Human vs. Expert (F1)
Lehmann (2023)	Semantic Parsing (Wiki)	CNL	Controlled Natural Lang. parsing	BLEU / ROUGE / Hits@1
Kovriguina (2023)	SPARQL Gen. (Fantasy)	SPARQL	SPARQLGEN: Subgraph Augment.	F1-macro on Benchmarks
Mountantonakis (2025)	Cultural Heritage (Art)	SPARQL	Two-stage Path Pattern prediction	Benchmark Accuracy
Ongris (2024)	Wikidata QA (General)	SPARQL	Sequential LLM Chaining (GraphRAG)	Jaccard Similarity
Vieira da Silva (2024)	Capability Model. (Manuf.)	OWL	TBox-contextualized prompting	OWL Reasoning/SHACL
Emonet (2025)	Federated QA (Bio)	SPARQL	ShEx/VoID-driven RAG retrieval	Execution Success Rate
Mashhaditafreshi (2025)	Digital Twins (IoT)	SHACL	SAMM Copilot: Iterative bootstrapping	Automated Jena Checks
Category 3: Evaluation, Stability & Trustworthiness				

Table 2.2: Final Synthesis Matrix of Included Studies ($n = 25$)

Study	Core Task / Domain	Logic	Neuro-Symbolic Integration	Validation Method
Sequeda (2025)	Enterprise Trust (SQL)	SPARQL	Position Paper on KG-based trust	Execution Benchmarks
Allemang (2024)	Query Correction (SQL)	SPARQL	Ontology-based Check + Repair	Iterative LLM Repair
Gashkov (2025)	Query Filtering (General)	SPARQL	Instruction-Tuned LLM-as-a-Judge	Answer Trustworthiness
Adam (2025)	Statement Verif. (Bio)	RDF	RAG using External Snippets	Precision / Recall
Meyer (2025)	KGE Benchmarking (Web)	SPARQL	LLM-KG-Bench 3.0 Framework	Automated Syntax/F1
Kosten (2024)	Prompt Eng. (General)	SPARQL	Multi-framework Evaluation	Execution Accuracy
Schmidt (2026)	Systematicity (Wiki)	SPARQL	CompoST: Compositional Testing	Compositionality F1
Tufek (2025)	Artifact Valid. (Industry)	SPARQL	Zero-shot Instruction Prompting	Domain-specific F1 Score

2.4 Thematic Analysis

2.4.1 Neuro-Symbolic Pipelines in Public Administration

In the domain of public administration and compliance, research is focused on translating high-stakes text (laws, regulations) into logic. However, these systems are still largely conceptual prototypes.

2.4.2 State-of-the-art in Logic Synthesis

The broader field of automated logic synthesis is dominated by descriptive question-answering tasks. While methods like 'Planned Query Guidance' improve accuracy, they are rarely tested against prescriptive legal constraints.

2.4.3 Methodologies for Logic Validation and Trust

Evaluation frameworks are shifting toward 'Knowledge Graphs as a Source of Trust'. Yet, current validation relies on probabilistic 'LLM-judges' or single benchmarks, leaving a gap for the deterministic testing required in rigorous governance environments.

2.5 Discussion and Research Gap

While the literature shows success in extraction and generation, there is a gap in deterministic validation. 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.

Other idea: most literature is about KGQA and information retrieval. Nobody (almost?) tries to make the sparql queries represent RULES and not just questions.

Other idea: The use of SHACL. Who uses it and how? Argument in favor of it being a good idea (unified graph, automation, explainability?)

3 Pilot Study

This chapter details the design, implementation and experimental validation of a novel Neuro-Symbolic pipeline for automating public service eligibility checks.

3.1 Overview

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 [1].

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 [2]. 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.

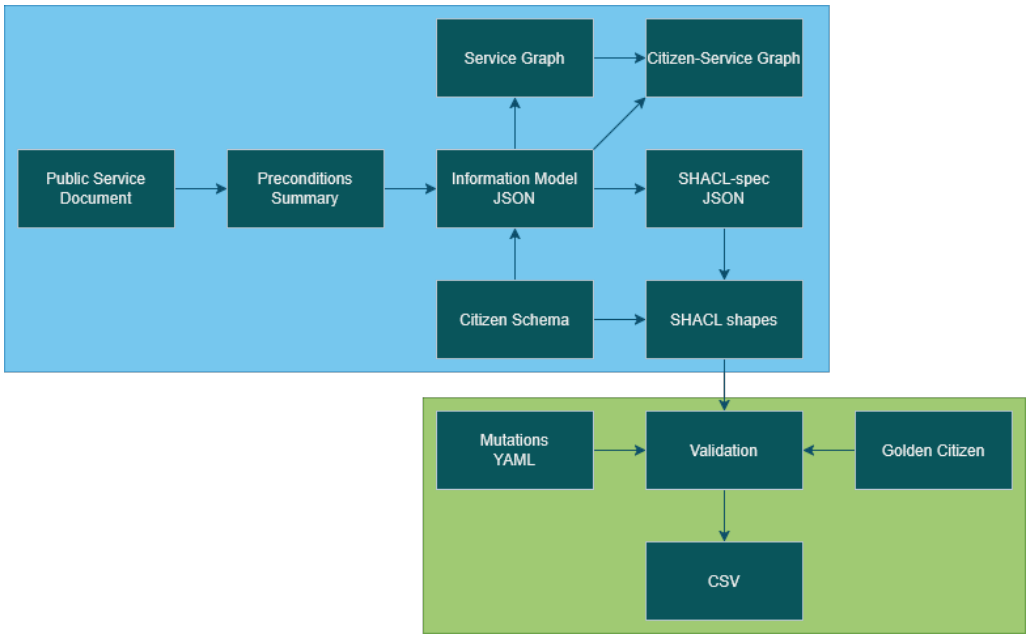

Figure 3.1: Flow Chart of the core pipeline

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 $\rightarrow$ Parents $\rightarrow$ Properties $\rightarrow$ 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 five 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 "Conformance" is the Positive class and "Violation" is the Negative class, standard machine learning metrics can be calculated.

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 aforementioned 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.

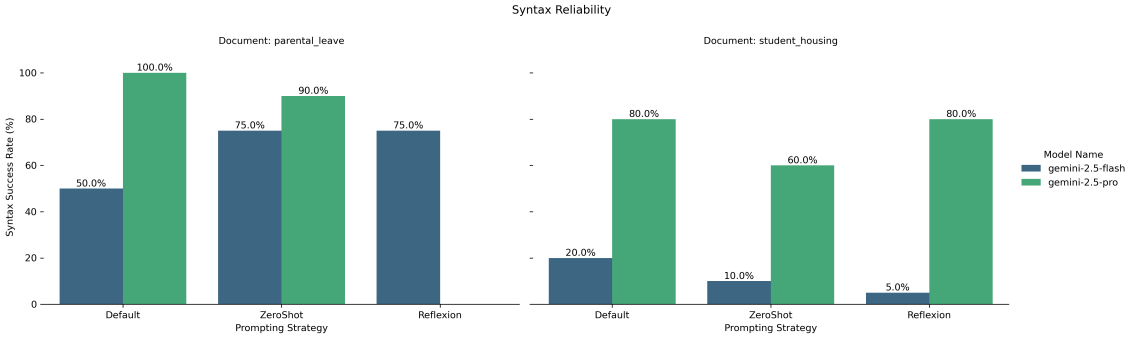

Figure 4.1: Syntactic Validity Rates by Configuration

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.

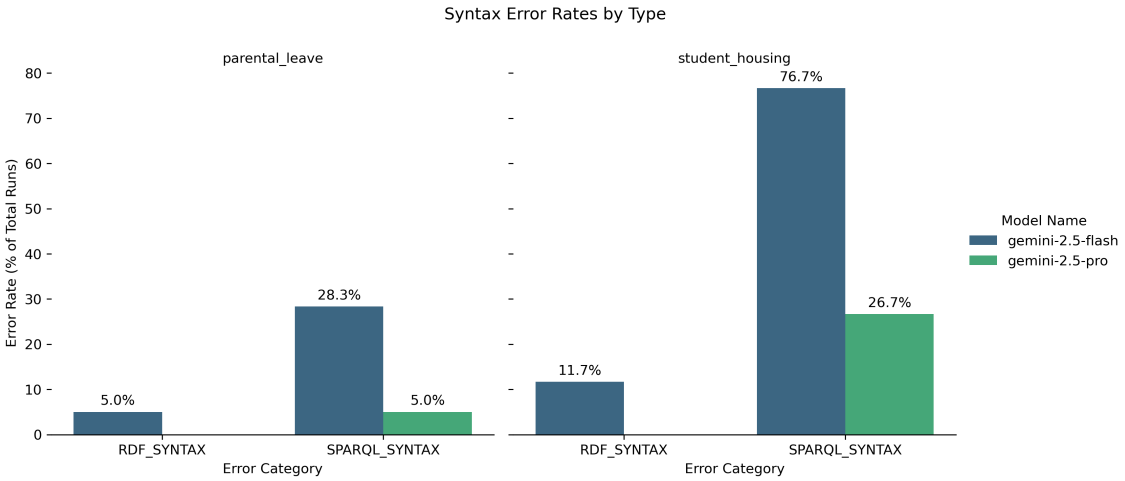

Figure 4.2: Distribution of Syntax Error Types by Model and Document

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.

It is critical to note that 'Logic Accuracy' is evaluated as a holistic, end-to-end performance metric. A failure to correctly validate a citizen scenario may stem from errors at any stage of the neuro-symbolic pipeline: a missed precondition during the initial summarization (Stage 1), a malformed mapping in the Information Model (Stage 2), or an incorrect SHACL generation (Stage 4). Consequently, a 'Logic Failure' indicates that the system, as a whole, failed to enforce the regulation, regardless of which specific component was the root cause.

4.3.1 Success Rate

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

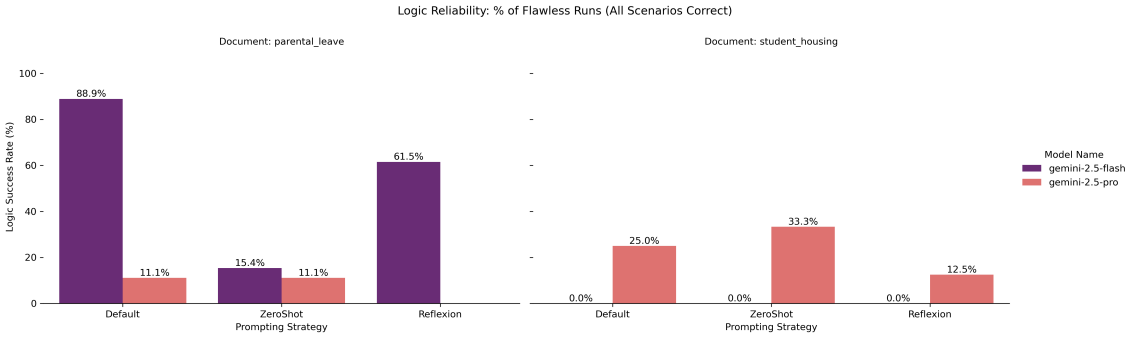

Figure 4.3: Logic Validity: Percentage of Flawless Runs (All Scenarios Correct)

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.

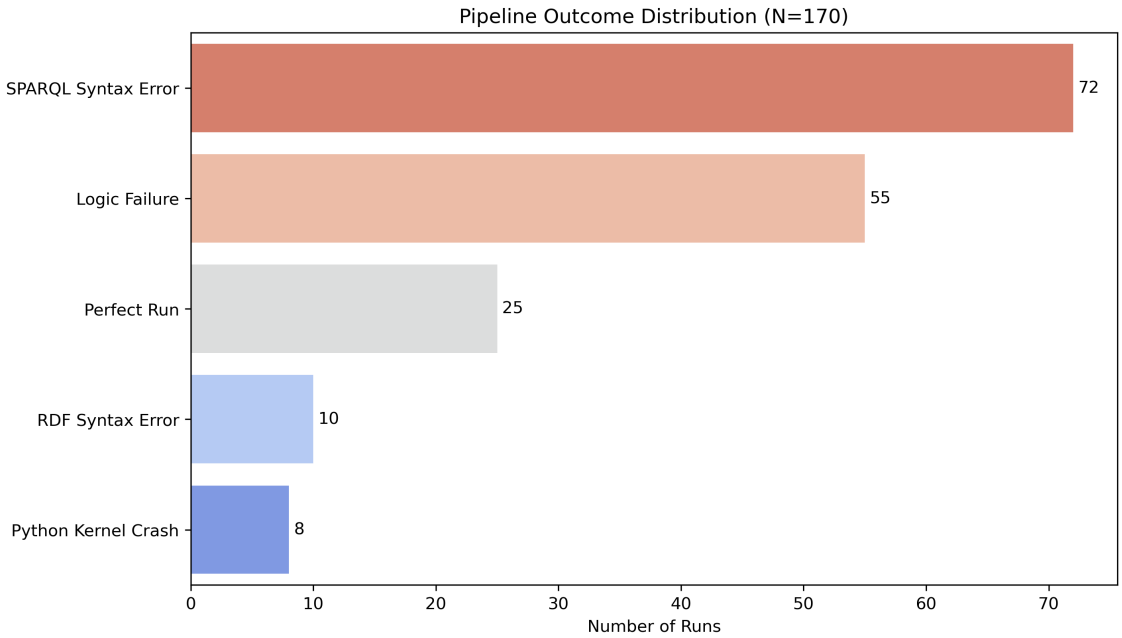

Figure 4.4: Distribution of Final Pipeline Outcomes.

- **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.

4.4.2 In-context (Recommender System) reliability

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.

The confusion matrix is then created based on this terminology.

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

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)

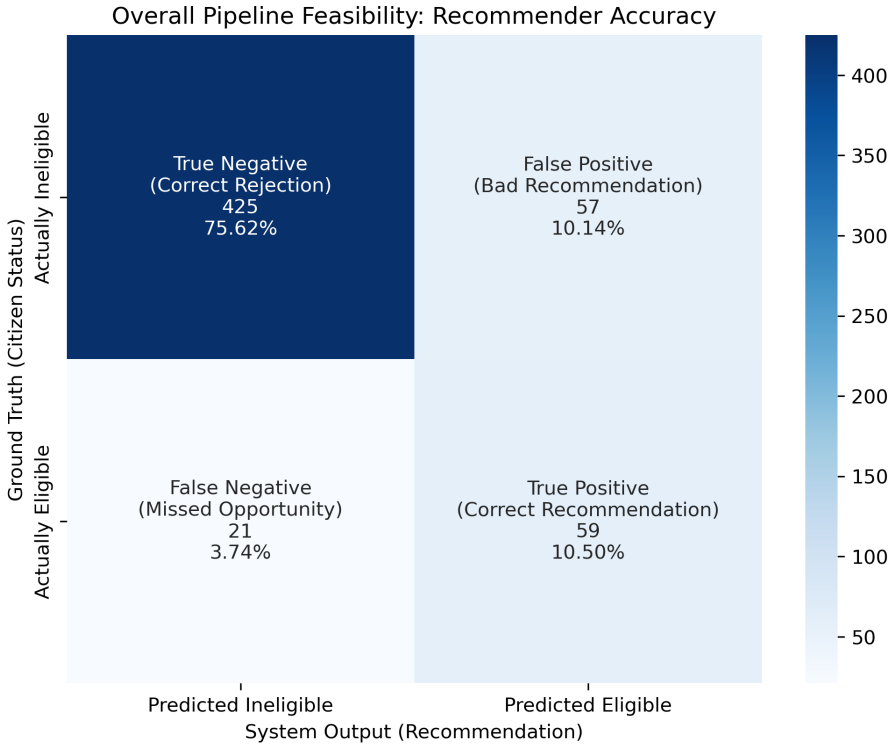

Figure 4.5: Confusion Matrix of Eligibility Recommendations

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.

5 Discussion

This chapter synthesizes the quantitative results presented in Chapter 4 to evaluate the broader implications of using Large Language Models for automated public service eligibility checks and recommendation. By analyzing the patterns of failure, ranging from syntactic hallucinations to logical paradoxes, this discussion aims to characterize the fundamental limitations of current neuro-symbolic architectures. The analysis moves beyond simple performance metrics to address the core challenges of semantic fidelity, algorithmic determinism and the operational sovereignty required for deployment in public administration.

5.1 Cognitive Dissonance in Code Generation

The most pervasive pattern observed throughout the experimental campaign was a fundamental disconnect between the Large Language Model's ability to generate valid *syntax* and its ability to construct valid *logic*. This phenomenon, termed here as "Cognitive Dissonance", underscores the architectural limitation of Transformer-based models when applied to formal reasoning tasks: they operate as approximate pattern matchers in a domain that requires exact symbolic execution.

5.1.1 The Illusion of Fluency

The results from the "Student Housing" use case serve as the primary evidence for this dissonance. The Gemini 2.5 Pro model achieved an 80% syntactic validity rate, successfully producing well-formed Turtle files with structurally correct SPARQL queries. To a human reviewer, this code appeared indistinguishable from expert-written logic. However, the functional accuracy of this "valid" code was only $\approx$25%.

This discrepancy reveals that the model has successfully memorized the *grammar* of SHACL (e.g., correct brackets, prefixes, keywords) but failed to grasp the *semantics* of the query it was constructing. The model "knows" how to write a query, but it does not "understand" well enough what the query actually calculates.

5.1.2 Structural Logic Failures

The model's inability to properly understand graph topology led to recurring critical failures in the generated SPARQL constraints. Below we analyze some of the most prominent mistakes discovered upon human inspection of the runs flagged by the system as not perfect.

The Double-Counting Trap

In scenarios involving family units (for example, two parents with two children), the model consistently failed to apply set-theoretic distinctness. By generating queries that traversed from `:Parent` to `:Child` without the `COUNT(DISTINCT ?child)` modifier, the model inadvertently created a multiplicative join. Since both parents are linked to the same children, the query counted each child twice (once per parent path), artificially inflating the "Child Count" variable. This subsequently distorted calculations, leading to false validations where ineligible families were approved due to miscalculated thresholds.

The Cartesian Product Trap

Similarly, when aggregating income, the model frequently joined Income patterns and Child patterns in a single WHERE clause without sub-query separation. This caused the SPARQL engine to generate a Cartesian product of the two datasets, effectively multiplying every income record by every child record. This resulted in erroneous rejections where families were flagged with violations due to massive over-estimations of total family incomes.

Recursive Loops and Infinite Regression

A more catastrophic failure mode was observed in the Flash model's handling of bidirectional relationships. The Student Housing ontology defines inverse relationships (e.g., a `:Parent` has a `:Child`, and that `:Child` has a `:Parent`). In several runs, the model generated SPARQL property paths that traversed these links cyclically (e.g., `:hasParent :hasChild :hasParent ...`) without a terminating condition. This created infinite recursion loops during execution, causing the PySHACL validation engine to crash entirely (logged as "Python Kernel Crash" in Section 4.4.1). This demonstrates that the model treats graph traversal as a linguistic association task ("Parents are related to Children") rather than a directed graph walk, failing to anticipate the computational consequences of such cycles.

5.2 Syntax Hallucination and Language Bleed

While the Pro model's failures were primarily logical, the Flash model struggled to maintain the boundaries of the language itself. The experimental data reveals a phenomenon of "Language Bleed," where the model, optimized for high-throughput generalized text generation, conflated the syntax of semantically similar languages.

5.2.1 SQL Contamination

The most frequent syntax error was the appearance of illegitimate keywords, such as `FILTER NOT (...)`. This construct is valid in SQL (WHERE NOT) but invalid in SPARQL (which requires FILTER (! ...) or FILTER NOT EXISTS). This suggests that the model's training

data contains significantly more SQL examples than SPARQL, leading it to default to the more dominant syntax when the probability distribution for the next token is ambiguous.

5.2.2 Token-Level Hallucinations

The model also exhibited errors that reveal its nature as a token predictor rather than a parser. The insertion of a dot (.) after `BIND` statements violates SPARQL grammar (where dots act as triple delimiters). This indicates the model might be trying to treat the code line as a natural language sentence that requires a period, ignoring the strict syntax tree of the query language.

Furthermore, the model frequently attempted to use dot notation (e.g., `?s.hasChild`) or complex property path slashes (e.g., `?s /:hasChild`) in contexts where explicit triple patterns were required. While property paths exist in SPARQL 1.1, the specific context in which such syntax was generated often mimicked Object-Oriented programming accessors rather than valid RDF graph traversal.

5.2.3 Namespace Invention

A distinct class of errors involved the hallucination of ontology definitions. Despite being provided with a fixed set of prefixes, the model occasionally invented new namespaces (e.g., using the deprecated 2007 SHACL draft URI or inventing an `ex:Citizen` ontology). This behavior was the primary cause of all recorded "RDF Syntax Errors" (as distinct from SPARQL errors), confirming that smaller models struggle to adhere to strict "Negative Constraints" (i.e., "Do not use any other prefix").

5.3 The Trade-off of Abstraction vs. Fidelity

A widely held assumption in Large Language Model research is that "Model Capability" (size, reasoning power) correlates linearly with performance across all tasks. However, the experimental results from the "Parental Leave" use case reveal a critical inversion of this principle.

5.3.1 The "Smart Model" Trap

The extraction of eligibility preconditions requires extreme fidelity to the source text. Legal constraints often rely on specific enumerations that define the scope of the law. Such a case was presented when models came accross the following precondition in the Parental Leave use case: *"The applicant must be employed under a dependent employment regime, in the Private or Public sector."*

In this task, the Gemini 2.5 Flash model (theoretically less capable model) significantly outperformed the Gemini 2.5 Pro model. Flash, lacking the capacity for deep abstraction, tended to "copy-paste" the precondition literally. When presented with the employment

requirement, it preserved the disjunction ("Private OR Public").

Pro, optimized for high-level reasoning and helpfulness, attempted to "summarize" the requirement. It interpreted "Private or Public" as a generic concept ("Employed"), effectively deleting the exclusion of other sectors (e.g., Freelancers). This finding suggests that for compliance tasks, "Smart" models may be fundamentally misaligned with the goal. Their training bias towards summarization and abstraction leads to Semantic Drift, where the gist of the rule is preserved but the legal boundary is lost.

5.3.2 The Deterministic Superiority

This trade-off extends to the architectural design of the pipeline itself. A stark contrast was observed between the error rates of the neural components and the symbolic components.

Notably, zero syntax errors were recorded in the generation of the "Citizen-Service Graph" (Stage 3). This stage relied on deterministic Python code to serialize the graph based on the LLM-derived Information Model, rather than asking the LLM to generate the Turtle syntax directly. The perfect stability of this stage, contrasted with the high failure rate of the LLM-generated SHACL shapes (Stage 4), empirically validates the architectural decision to offload structural tasks to deterministic code wherever possible.

This leads to a key design principle for Neuro-Symbolic systems in public administration: LLMs are necessary for interpretation (Extraction), but they are suboptimal for serialization (Code Generation). A robust pipeline must treat the LLM as a "Translator" of natural language, but never as an "Architect" of the final system structure.

5.4 The Complexity Ceiling

The divergence in performance between the "Student Housing" and "Parental Leave" use cases identifies a distinct Complexity Ceiling for current LLM-based logic generation. While the pipeline demonstrated high viability for administrative tasks involving categorical classification (Parental Leave), it experienced a near-total collapse when tasked with recursive arithmetic (Student Housing).

This failure mode correlates strongly with the Dependency Depth of the required logic:

- **Shallow Dependencies (Success):** Constraints that rely on single-node checks (e.g., *Nationality* depends only on *Applicant*) or flat Boolean logic (e.g., *isValid* is True OR False) were handled with high accuracy (88.9% logic success).
- **Deep Dependencies (Failure):** Constraints that require multi-hop traversal (e.g., *Applicant* $\rightarrow$ *Parent* $\rightarrow$ *Residence*) or recursive variable modification (e.g., *Income Limit* changes based on *Dependent Child Count*) consistently triggered the "Cartesian Product" bug or infinite recursion errors.

This suggests that while the tested LLMs can successfully act as "Semantic Parsers" for straightforward bureaucracy, they lack the internal "Working Memory" required to maintain

the state of multi-variable equations throughout the code generation process.

5.5 The Contribution of Prompt Engineering

The experimental results challenge the prevailing narrative that "better prompting" is a universal solution to model limitations. Instead, the data reveals complex trade-offs where techniques that improve syntactic stability may inadvertently degrade logical reasoning.

5.5.1 The "Copy-Paste" Bias

The Default (Few-Shot) strategy proved essential for stabilizing syntax in the Pro model, boosting syntactic validity from 60% to 80% on the complex document. However, this stability came at a cost to logical accuracy. Zero-Shot strategy, despite producing broken code more often, achieved the highest logical accuracy (33.3%) when it *did* compile.

This suggests a "Copy-Paste Bias": when provided with examples, the model seems to over-fit to the logic of the few-shot template, attempting to force the new problem into the old structure. Without examples (Zero-Shot), the model is forced to reason from first principles, leading to messier syntax but potentially more original (and correct) logical derivations.

5.5.2 The Failure of Self-Correction

The Reflexion strategy failed to deliver the expected performance gains. For the less capable Flash model on the complex document, Reflexion actually *degraded* performance, dropping syntax validity from 20% to 5%. This indicates that a model incapable of solving a problem in the first pass is equally incapable of critiquing its own solution. Asking a confused model to "double-check" its work merely introduces a second opportunity for hallucination, compounding errors rather than resolving them.

5.5.3 The Engineering Ceiling

These findings imply an "Engineering Ceiling": one cannot prompt their way out of a fundamental reasoning deficit. While Prompt Engineering can guide a capable model (Pro) to follow syntactic rules, it cannot bestow reasoning capabilities upon a smaller model (Flash) that physically lacks them. For high-stakes logic generation, architectural scale remains the dominant variable.

5.6 Feasibility and Sovereignty

The final dimensions of analysis concern the operational viability of deploying such a system within a public administration context. The experimental campaign revealed critical vulnerabilities in the reliance on proprietary Model-as-a-Service (MaaS) infrastructure.

5.6.1 The Semantic Drift of "Eligibility"

Feasibility is first challenged at the point of ingestion. The extraction and summarization phase (Stage 1) demonstrated a persistent ambiguity in defining "Eligibility." Despite explicit prompt instructions to ignore administrative steps, the model frequently conflated procedural requirements (e.g., "Log in to TaxisNet") with substantive facts (e.g., "Be employed"). This semantic drift creates a system that validates paperwork rather than reality. While acceptable for a pilot, a production system would require a stricter, legally-grounded ontology of "Evidence" vs. "Conditions" to prevent the digitization of bureaucracy from becoming merely the automation of red tape.

5.6.2 Replication Crisis

The most severe threat to feasibility, however, emerged from the infrastructure itself. During the experimental window, unannounced changes to the Google Gemini API rate limits and model availability caused a sudden, catastrophic degradation in pipeline throughput. This event serves as a potent case study for Digital Sovereignty.

A public administration pipeline that relies on opaque, third-party endpoints is fundamentally fragile. The inability to guarantee consistent latency, availability, or even model behavior (version drift) renders MaaS solutions unsuitable for critical government infrastructure. The findings of this study strongly advocate for a shift towards Sovereign AI: deploying open-weights models (e.g., Llama 3, Mistral) on government-controlled infrastructure. Only by owning the compute can the administration guarantee the reproducibility and stability required for legal automation.

5.7 Risk Asymmetry in Public Administration

The evaluation of the system's "Recommender Accuracy" (Section 4.4.2) reveals a critical insight into the deployment readiness of these neuro-symbolic agents. While standard machine learning models optimize for balanced F1 scores, the operational context of public administration imposes an asymmetric cost of error.

The experimental data showed a 10.1% False Positive Rate, which corresponds to instances where the system erroneously recommended a service to an ineligible citizen. In a commercial context (e.g., movie recommendations), such errors are trivial. However, in digital governance, a False Positive actively generates bureaucratic friction. Specifically, it compels a citizen to gather documents and submit an application that is destined to fail. This wastes public resources and also erodes trust in the automated system. Conversely, the 3.7% False Negative Rate (Missed Opportunities), while statistically undesirable, represents a "safer" failure mode that maintains the status quo.

Consequently, the current pipeline's bias towards 'over-recommending' presents a

significant barrier to unsupervised deployment in a real-world administrative setting.

6 Limitations & Future Work

6.1 Limitations

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.

more human interpretation of the results is needed. it's difficult to pinpoint logic errors within complex queries.

6.2 Future Research Directions

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.

Semantic Stability: data from semantic similarity metrics drawn from past run artifacts on file.

future iterations of the pipeline must prioritize Precision (Trustworthiness) over Recall (Coverage), potentially by calibrating the validation logic to be "conservative by default" or by implementing a "Human-in-the-Loop" review for all positive recommendations.

For public services involving means-testing, complex family unit aggregations, or temporal operations, the current neuro-symbolic approach requires either significantly more advanced prompting strategies or even a fundamental shift to deterministic calculation engines for the mathematical components.

Bibliography

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

[2] 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. DOI: 10.3390/app152011235

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.