Friedrich-Alexander-Universität

Guided Semantic Model Generation: A Comparative Study of Symbolic and Learned Guidance for Tableau-Based Interpretation

Master's Thesis in Computer Science

submitted
by

Ahmed Mamdouh

Written at

Lehrstuhl für Wissensrepräsentation und -verarbeitung
Department Informatik

Supervised by: Prof. Dr. Michael Kohlhase, M.Sc. Jan Frederik Schaefer

Declaration

Abstract

This thesis develops a guided search system for interpreting short narratives and evaluates two ways to control that search. The first design (Java) relies on symbolic guidance: an ontology and hand-written policies decide which partial interpretations to expand. The second design (Python) keeps the same calculus but adds data-driven guidance through simple neural cost functions. Guided model generation is useful because it keeps inference interpretable and auditable: each decision can be traced to a policy or a score.

The study tests whether learned guidance can replace or complement symbolic control. Both designs run on the same story sets. We report wall-clock time, explored models, and success rate for every run, plus branch-ordering agreement (Spearman ρ) for the Python scorer and its training losses/accuracies. Symbolic policies were stable: the custom policy remained fast as thread counts increased, while plain DFS and BFS often explored more states than necessary. The learned guidance fit the training traces almost perfectly but was brittle on new material; the external heuristic was sensitive to prompts and seeds.

Taken together, the results support a hybrid strategy. Keep the symbolic scaffold to ensure basic correctness, and use learned scores as lightweight tie-breakers rather than as the main controller. The contributions are: (1) A layered system design separating tableau calculus, model generation, and model guidance; (2) a reproducible, multi-threaded symbolic system with policy comparators; (3) a Python port that exposes the same interface to learned scorers; and (4) an empirical comparison that documents symbolic robustness, learned-policy instability, and their practical consequences. Reproducibility instructions appear in the appendix.

Contents

Chapter 1

Introduction

1.1 Overview

Natural-language understanding in short narratives depends on recovering a coherent discourse model, not merely analysing sentences in isolation. Human readers continuously integrate local linguistic cues with background knowledge to determine what entities are being referred to, what events are introduced, and how those events relate across the dialogue. When references are underspecified—most visibly in pronouns—correct interpretation often requires discourse-level reasoning and commonsense, rather than surface syntax alone [Mit22; Keh02; Gro+95].

This dependence on context is central in anaphora resolution, where selecting an antecedent amounts to choosing among competing global readings of the narrative. Standard statistical models have improved substantially, but they remain brittle on cases where the intended antecedent is determined by implicit world knowledge or situation structure. The Winograd Schema Challenge was introduced precisely to probe this limitation: it isolates short pronoun ambiguities that flip under minimal lexical edits and therefore cannot be solved robustly by correlation-based methods alone [Lev11].

The aim of this thesis is to study how structured semantic reasoning can be paired with explicit guidance to improve context-dependent interpretation while preserving interpretability. Concretely, we develop a guided model-generation system for short narratives and evaluate two paradigms for controlling its search over candidate interpretations. Design A follows a symbolic route in which an ontology and hand-written policies steer tableau expansion. Design B keeps the same calculus but projects the representation to event-level literals and introduces lightweight learned scoring functions. By comparing these guidance families under shared constraints, the study clarifies where symbolic control guarantees stability and where learned preferences can complement it without degrading robustness.

1.2 Background and Context

To address context-dependent interpretation, recent research increasingly turns to hybrid reasoning systems that combine explicit semantic structure with flexible guidance signals. Neural encoders dominate many NLP tasks, but anaphora resolution still benefits from explicit structure and commonsense signals [Mit14; Vas+23]. Tableau calculi and neo-Davidsonian event semantics provide precise, inference-friendly representations [Smu95; Par90]. Knowledge graphs add compact stores of factual relations [Ji+22]. These ingredients motivate hybrid systems that combine symbolic reasoning with learned guidance.

1.3 Problem Statement

Building reliable hybrids is difficult. Symbolic policies must be tuned to stay efficient, while neural scorers often lack the inductive bias needed for consistent semantics [Ben+20]. Knowledge graphs are frequently discussed but rarely integrated end-to-end. This thesis addresses these gaps by specifying a theroetical semantic model generation framework, a Java design that steers tableau expansion with a knowledge graph of situations, frames, roles, and semantic types, and by deriving a Python implementation that projects to event-level literals for data-driven guidance.

Contributions. This thesis contributes: (1) the system design as reference for implementations with tableau, guidance, general language constructs and search algorithm; (2) an implementation of a reproducible multithreaded tableau framework with salience-aware rule application and symbolic policy comparators; (3) the derivation of a Python implementation that collapses the ontology onto subject/object/verb/adjective predicates for neural cost functions and an external heuristic; and (4) an empirical comparison that documents stability gaps in learned guidance and outlines ablations and error patterns.

1.4 Research Objectives and Questions

The study evaluates guidance strategies for tableau-based semantic search while holding the logic and ontology fixed. It asks:

- How well do breadth-first, depth-first, and salience-aware policies use the knowledge graph, and what happens under parallelism?
- Can neural cost functions, the human-fine-tuned variant, and an external heuristic deliver guidance that remains steady on new stories and lexical edits?

1.5 Research Hypothesis

We test the following hypothesis under a shared calculus and ontology:

> **H1.** Symbolic guidance (in particular, the Custom policy) yields more stable branch orderings and lower runtime than learned or external guidance under equivalent constraints (single-thread score-at-leaf for Design B; fixed timeouts/queue budgets for Design A). Rule-level salience improves witness selection consistency across thread configurations.

This hypothesis links the Results (Chapter 6) and Discussion (Chapter 7) back to the problem statement by making stability and efficiency under fixed constraints the primary evaluation criterion.

1.6 Methodological Approach

The thesis follows a two-stage design. Chapter 3 introduces the shared theoretical system architecture based on a sorted neo-Davidsonian logic and a tableau engine recursive structure with salience-driven reasoning [Par90; Ji^{+}22]. A model generator operates on top of this tableau engine and accepts guidance policies for controlling model search.

The subsequent chapters implement and evaluate two distinct designs built on this shared foundation. Design A retains the full knowledge-graph ontology and logic signature of the reference design. It evaluates three symbolic guidance strategies (breadth-first, depth-first, and a salience-aware custom policy) within a multithreaded Java system.

Design B reuses the same tableau calculus but applies it to a projected representation that omits explicit situations and frames. This projection simplifies the logic signature to align with sentence structure, grouping literals by event into the predicates *subject*, *object*, and *verb*, and introducing an ad-hoc *adjective* predicate used only in the Python experiments. The projection enables neural cost functions trained on policy traces, a contrastive fine-tuning stage with human preferences, and an external heuristic that scores branches from textual descriptors [Lev11; Dev+19; Bro+20].

Both designs are evaluated on synthetic narratives using comparable metrics, including the number of explored models, runtime performance, and agreement with the custom policy.

1.7 Scope and Limitations

In scope. Static ontology, controlled event-semantics fragment (with adjectival modifiers in Design B), short synthetic narratives, symbolic policies and learned guidance for branch prioritisation.
Out of scope. Temporal knowledge graphs (conceptual context only), dynamic ontology updates, large open-domain corpora, and full transformer-based guidance.

Both designs use curated fragments that align with tableau calculus. Design A relies on the leaner Java grammar and the full ontology, while Design B covers adjectival cues after projecting each situation/frame bundle into event descriptors, so the stories stay comparable even though the representations differ. The knowledge graph focuses on celebration and investigation scenes, which keeps variables controlled but limits coverage. The neural and external heuristic guidance stages start from policy traces, move to a small human-annotated set, and depend on contextual embeddings that add runtime cost [Dev+19].

1.8 Thesis Outline

The chapters are organized as follows:

- **Chapter 2** reviews related work on formal logic, knowledge graphs, neural networks, and the Winograd Schema, establishing the background for the hybrid reasoning approach.

- **Chapter 3** introduces the overall system design and architecture, describing the shared theoretical framework, logic signature, ontology, tableau calculus, and guidance paradigms that underpin all experiments.
- **Chapter 4** presents the two designs and their implementations: Design A (symbolic policies in Java) and Design B (data-driven guidance in Python).
- **Chapter 5** specifies datasets and evaluation protocols for both designs.
- **Chapter 6** reports numerical results without interpretation.
- **Chapter 7** provides discussion and the overall conclusion, including limitations and directions for future work.
- **Appendix A** provides complete instructions to reproduce all experiments in Chapters 5–6, including environment setup, repository layout, build/run commands, neural training, and seed/log management.

Chapter 2

Background

2.1 Introduction

This chapter reviews the theoretical and methodological background for studying context-dependent interpretation in NLP. Many standard approaches struggle when meaning depends on discourse rather than surface syntax alone, as in anaphora resolution [Mit14]. We therefore focus on four strands of prior work: formal logic, knowledge graphs, neural networks, and benchmarks such as the Winograd Schema.

Formal logic and tableau methods provide the core mechanism for representing structured semantic relations, especially within neo-Davidsonian event semantics [Par90]. Knowledge graphs and their temporal variants capture relations that evolve over time, which is central to modeling events in language [Ji+22]. Neural networks with self-attention have transformed NLP, but they still fall short when tasks require deep, context-aware interpretation [Vas+23].

The Winograd Schema Challenge illustrates the need for richer models of reference [Lev11]. Its hand-crafted sentences require background knowledge and pragmatic reasoning to resolve ambiguous pronouns, and they motivate the hybrid approach pursued in this thesis.

2.2 Synthesis and Positioning

Hybrid reasoning systems typically fall into two families. One line integrates symbolic search with lightweight, hand-crafted control (e.g., policy comparators over a frontier); another injects learned signals into otherwise symbolic solvers (e.g., SAT-guided or neural-theorem-proving approaches) [Sel+19]. Both encounter limits: the former needs careful design to remain efficient; the latter can drift when scores correlate with surface cues rather than the structure of the search state.

This thesis positions itself between these strands. It preserves a symbolic calculus and ontology while exposing guidance through an interface that can host either comparators or learned scorers. Unlike neural theorem proving that often supervises truth judgments over clauses end to end, Design B reads structured event bundles and scores frontier states, which aligns with the decision points that drive tableau expansion. Agreement with a reference policy is therefore a meaningful target, and the approach retains the interpretability of symbolic methods.

2.3 Formal Logic

2.3.1 First-Order Logic

Formal logic, especially first-order logic, has been central in the representation of structured, rule-based reasoning in natural language processing. First-order logic allows for the exact description of sentence structure and the relations between entities, hence providing a base for semantic analysis and logical inference in natural language contexts. Early works by [Fre79] and later developments by [Tar56] established the grounding for FOL, which allows the formal expression of propositions concerning entities and their properties and logical manipulation thereof. Thus, FOL is important in NLP for constructing structures that represent relations among the different parts of sentences, and this becomes very relevant in tasks requiring logical inference. Based on FOL, tableau calculus adds a systematic, rule-based method for checking logical consistency of a model. Tableau calculus was introduced by [Smu95], which allows one to search systematically through the possible interpretations of the logical statements, thus allowing checks for satisfiability and reasoning. The tableau approach has been particularly useful in capturing possible meanings in ambiguous sentences, providing a flexible but rigorous framework for modeling different possible semantic interpretations.

2.3.2 Event Semantics

Event semantics, particularly neo-Davidsonian event semantics, extends the standard First-Order Logic (FOL) setup by focusing on the representation of events as core semantic units [Dav67; Par90; Kra96; Dow91].

Neo-Davidsonian semantics introduces an event variable by which a sentence can be decomposed into specific event-related roles, such as subjects, objects, and actions, while also accommodating modifiers. Such a decomposition into event roles underpins a more fine-grained representation of meaning, wholly consonant with the structure of natural language itself, in which verbs often encapsulate complex relations between agents, actions, and contexts.

Starting from the neo-Davidsonian framework, event semantics makes verb phrases easier to analyse, especially when a clause involves several arguments or complex actions. Instead of tying an agent directly to an object, neo-Davidsonian semantics inserts an event entity and links participants through separate roles. This structure allows additional semantic roles, such as instruments or locations, to appear as event modifiers, which helps capture how sentence meaning changes as the discourse unfolds. Event semantics has therefore proved useful in NLP tasks such as information extraction and event detection, where relations between participants and actions are central [Hac06].

2.3.3 Extensional v.s. Intensional Semantics

A fundamental distinction in formal semantics is that between extensional and intensional understandings of meaning. Extensional semantics, in its truth-conditional logic, is limited to real, observable relations, insofar as it indexes directly to objects in a given domain [Mon73]. While extensional logic suffices for atomic propositions, it becomes inadequate when reasoning about context-dependent linguistic components, such as hypothetical or possible facts, ubiquitous in natural languages. On the other hand, intensional semantics respond to this gap by accounting for context-sensitive interpretations and meanings that exceed observable truth conditions. First theorized by [Car11] and further developed in the work of Montague, intensional semantics allows the representation of possible worlds or states in such a way as to provide a structure supporting meaning grounded in potential or hypothetical contexts [Mon73].

This adaptability is important in tasks such as anaphora resolution, where the interpretation of pronouns often depends on tacit presuppositions about the situation or the mental states of the entities. Intensional semantics is then a general framework for the interpretation of language, including those nuances of meaning that are not expressed but inferred from the context. Knowledge graphs are structured models that represent entities and the interrelations between them in an integral form. As discussed in [Ji+22]

2.4 Knowledge Graphs

2.4.1 Knowledge Graphs Data Structure

Knowledge Graphs are a graph based data structure with an emphasis on the relations and dependencies involved in a certain domain [Ji+22]. Every node in this structure denotes an entity; edges are used for representing relations, building up a network of information that may enhance various tasks within NLP, such as information retrieval and semantic search.

2.4.2 Temporal Knowledge Graphs Extension

Temporal Knowledge Graphs (TKGs) extend static knowledge graphs with time-aware relations that track how events and roles evolve [Ji+22]. This perspective mirrors neo-Davidsonian event semantics, which already treats situations, frames, and roles as structured representations unfolding over time. Although the ontology in this thesis remains static and does not implement temporal predicates, TKGs illustrate how richer, time-indexed constraints could further align knowledge-graph guidance with the inherently dynamic nature of discourse.

2.5 Neural Networks

Neural networks for natural language processing have progressed from simple feedforward models to architectures that capture intricate linguistic structure. Key milestones include recurrent networks for sequential data [Elm90], long short-term memory networks that mitigate long-range dependency issues [Hoc+97], and transformers whose self-attention layers process entire sentences at once [Vas+23]. Pretrained transformers such as BERT [Dev+19] achieve strong results on tasks ranging from question answering to text generation.

Even so, contemporary networks still lack human-level understanding on context-sensitive tasks such as anaphora resolution. They learn from large static datasets and correlations, which limits their ability to represent meaning directly or adapt to subtle contextual shifts [Ben+20]. As a consequence they often mis-handle pronouns that rely on implicit world knowledge, including examples from the Winograd Schema Challenge.

Recent work addresses this gap by combining the strengths of formal logic and neural models so that intensional, context-aware reasoning can inform neural representations. Hybrid systems often incorporate heuristic guidance inspired by tableau reasoning [Sel+19], which supplies a structured view of discourse while leaving room for data-driven scoring. The overall aim is to ground neural predictions in logic-based constraints and make interpretation more transparent.

2.6 The Winograd Schema Challenge

The Winograd Schema Challenge [Lev11] benchmarks human-like reasoning in NLP. Each schema is a short sentence with an ambiguous pronoun that can only be resolved by applying context, background knowledge, or commonsense reasoning. Unlike broader benchmarks that reward pattern matching, the Winograd tasks focus squarely on interpreting meaning in context, particularly for anaphora resolution. Every item permits two readings, and choosing the intended one demands a careful account of how entities and actions relate.

The Winograd Schema Challenge in anaphora resolution displays special characteristics that go beyond the usual language tasks, since it heavily depends on context-dependent interpretations and common sense. Anaphora resolution often requires much more than superficial syntactic knowledge, since it demands the capability of inferring subtle, context-driven meanings. For example, the pronoun “it“ in the sentence “The trophy does not fit in the suitcase because it is too small“ refers to “the suitcase“, a relation that can only be worked out if one knows the typical size relations between trophies and suitcases. This type of resolution actually falls under intensional semantics, where meaning is recovered from implicit knowledge or common sense rather than explicit linguistic cues [Hir81]. This reliance on contextual elements and intrinsic semantics reveals the inadequacy of models relying strictly on syntactic methods or statistics. Syntactic parsing can be at most useful in the identification of possible antecedents, but only a model that can synthesize global information with intensional semantics can output resolutions correctly and consistently, especially in cases that are as intricate as the Winograd Schema Challenge.

Most contemporary approaches to anaphora resolution within natural language processing fall short of the requirements of the Winograd Schema Challenge. Traditional rule-based approaches are very accurate but have low generalizability; they fail to handle huge amounts of real-world knowledge and context-dependent inferences [Mit14]. On the other hand, neural network-based models, including transformers, are heavily reliant on large datasets and statistical correlations, which become insufficient for capturing context-dependent and inherent meanings [Ben+20]. Such models, however, cannot deal with anaphora where either contextual or real-world knowledge would be involved, because they don't have the ability to comprehend meaning beyond mere data correlations.

Recent studies address this gap by pairing neural networks with structured reasoning components, for instance the hybrid systems described by [Bos+19]. However, these models are still in their early stages and mostly lack the strength needed for more advanced semantic understanding in complex linguistic tasks. The Winograd Schema Challenge remains one of the important benchmarks due to its ability to bring out failure on the part of NLP models in understanding inherent semantics and intensional clues crucial for human-like comprehension of natural language.

2.7 Gaps in Research

Existing approaches across formal logic, knowledge graphs, and neural networks each fall short of human-like semantic understanding. Formal logic and tableau calculus offer rigorous reasoning frameworks [Smu95], but they struggle with context-driven nuance.

Knowledge graphs and their temporal extensions represent entity relations efficiently, yet they only capture commonsense dynamics when supported by extensive manual modelling [Ji+22]. Transformer-based neural networks excel at learning patterns from large corpora [Vas+23], but their reliance on correlation constrains performance on intensional tasks such as the Winograd Schema [Ben+20]. These limitations point toward models that blend symbolic structure with flexible representations. Recent work explores neural systems equipped with logic-based heuristics or temporal knowledge graphs to improve contextual reasoning [Sel+19]. TKGs in particular align with event semantics by depicting relations that change over time, a property that helps disambiguate references in discourse. This thesis contributes to that trajectory by analysing hybrids that combine symbolic control with learned guidance.

Chapter 3

System Design

3.1 Design Motivation and Objectives

This chapter studies how tableau-style semantic search handles short narratives with ambiguous references. The reference overall system design defines the base logic, the knowledge graph ontology, the tableau engine, and the model generation engine. This system design cleanly separates formal representation expressivity from the overall system, allowing changes to be applied easily to expand on the system in future work.

Motivating example.

To illustrate the kind of discourse ambiguity we target, consider the short dialogue:

> *"Sara saw the dog. She bit her."*

The difficulty is not in parsing either sentence in isolation, but in deciding how the pronouns in the second sentence attach to the entities introduced in the first. The dialogue admits (at least) two plausible interpretations:

(1) **Dog-as-biter:** Sara saw the dog, and then the dog bit Sara.

(2) **Sara-as-biter:** Sara saw the dog, and then Sara bit the dog.

Crucially, nothing in the surface form of the second sentence forces one reading over the other. Human readers rely on background knowledge about typical situations to recover the intended antecedents. For instance, that dogs are more likely to bite humans than the reverse. The task for an interpretation system is therefore to represent both readings explicitly and to prefer the one that is most coherent with commonsense and the discourse context.

In this thesis, we treat such dialogues as underspecified descriptions of events. Our system generates candidate semantic models corresponding to different reference resolutions, and guidance mechanisms (symbolic policies in Design A or learned scores in Design B) are used to rank these candidates. The remainder of this chapter makes this process precise.

3.2 Common Theoretical Framework

3.2.1 Logic Signature

The system operates over a sorted first-order event logic with the structure of frame semantics. The signature comprises the sorts

- $situation$ and $situation\_type$;
- $frame$ and $frame\_type$;
- $role\_type$;
- sem_type; and
- $actor$

Terms are generated from the grammar

- U for unique constants,
- C for non-unique constants, and
- V for variables,

where each term may take any sort. Unique constants satisfy a uniqueness axiom that prevents aliasing, whereas non-unique constants behave as existentially bound variables that can later unify with other terms. Formulas are obtained from the constructors

- $\top$,
- $\bot$,
- $\neg\phi$,
- $\phi \wedge \psi$, and
- $\exists_e : \phi[e].\psi[e]$,
- $situation(s, S)$,
- $frame(s, f, F)$,
- $role(f, A, R)$, or $role(f, a, R)$, and
- $semType(A, T)$,

with variables s, f, r taken from the *situation*, $frame$, and *actor* sorts respectively, and constants S, F, A, R, and T taken from the *situation_type*, $frame\_type$, *actor*, *role_type*, and *sem_type* sorts respectively. The modified existential quantifier allows a precondition formula ϕ for the constant or witness used. It roughly translates to "There exists an individual with the property ϕ, where ψ.". This enables picking terms that already satisfy a precondition in the branch before adding more information about them in the tableau. The logic allows existential conditional instantiation and equality reasoning, providing sufficient structure for modelling event semantics, frame roles, and salience-sensitive reasoning.

3.2.2 Natural Language Fragment

The framework restricts utterances to sentences with a single verb, which is can be used to match a frame, multiple actors, such as subject, object, and prepositions which are introduced as roles. Additional extensions to the fragment such as adjectives or adverbs remain outside the ontology and are left as future work. Table 3.1 demonstrates the deterministic mapping from English sentences to formulas.

3.2.3 Ontology of Situations, Frames, and Roles

The framework grounds the logic in a typed ontology that captures prototypical situations and their roles. Each *situation* such as *celebration* or *murder* lists one or more *frames*. Frames use the predicate *frame*(s, f, F) with situation s, frame f, and frame type F. Roles use from predicates *role*(f, A, R), where f relates the role to the specific frame invokation, A supplies the individual acting the role, and R represents the type of the role. In addition, situations can have axioms that can enforce ontological well-formedness, such as: no frame my assign the same role twice, and actors must be located in the same situation-specific context. This ontology therefore constrains tableau expansion by restricting which formulas can be added for each sentence and by pruning branches that violate narrative coherence.

3.2.4 Tableau Calculus and Salience

Both designs rely on the same tableau calculus summarised in Table 3.2. Rules break down conjunctions, instantiate quantifiers, and detect contradictions. Search nodes hold open branches paired with salience-weighted discourse entities. Salience is applied strictly at the rule level: every time new individuals appear, they receive full salience, while existing individuals are decayed by a constant factor. Salience therefore prioritises which constants

Sentence	**Logical Representation**
Alice attended the birthday celebration.	`semType(birthday, stype_occasion)` `semType(alice, stype_human)` $\exists s\, \exists f\, \exists l\, \big(situation(situation\_celebration, s) \wedge frame(s, frame\_attend, f) \wedge role(f, role\_attendee, alice) \wedge role(f, role\_occasion, birthday) \wedge semType(l, stype\_location) \wedge role(f, role\_location, l)\big)$
Alice was at home.	$\exists s\, \exists f\, \big(situation(situation\_celebration, s) \wedge frame(s, frame\_be\_at, f) \wedge role(f, role\_location, home) \wedge role(f, role\_attendee, alice)\big)$ `semType(alice, stype_human)` `semType(home, stype_location)`
Bob attended the birthday celebration.	`semType(birthday, stype_occasion)` `semType(bob, stype_human)` $\exists s\, \exists f\, \exists l\, \big(situation(situation\_celebration, s) \wedge frame(s, frame\_attend, f) \wedge role(f, role\_occasion, birthday) \wedge semType(l, stype\_location) \wedge role(f, role\_location, l) \wedge role(f, role\_attendee, bob)\big)$
Charlie brought a cake to the birthday celebration.	`semType(birthday, stype_occasion)` `semType(charlie, stype_human)` `semType(cake, stype_object)` $\exists s\, \exists f\, \exists l\, .\big(situation(situation\_celebration, s) \wedge frame(s, frame\_bring, f) \wedge role(f, role\_occasion, birthday) \wedge role(f, role\_brought, cake) \wedge role(f, role\_location, l) \wedge role(f, role\_bringer, charlie)\big)$

Table 3.1: Translation from English to logic.

are used when instantiating existential rules, but it does not dictate the order in which branches are dequeued; that distinction is reserved for policies and cost functions introduced later.

Rule	Name	Description
$\dfrac{\phi, \neg\phi}{\bot}$	$\bot(Contra)$	Contradiction is unsatisfiable
$\dfrac{\neg T}{\bot}$	$\bot(\neg T)$	“Not true” is unsatisfiable
$\dfrac{F}{\bot}$	$\bot(F)$	False closes the branch
$\dfrac{\phi \wedge \psi}{\phi, \psi}$	$\wedge$	Break down conjunctions
$\dfrac{\neg\neg\phi}{\phi}$	*Dneg*	Remove double negation
$\dfrac{e_1, \ldots, e_n \in \mathcal{H} \quad \neg\exists_e.\phi[e]}{\neg\phi[e_1], \ldots, \neg\phi[e_n]}$	$\forall$	Instantiate universal claims using known individuals
$\dfrac{\neg(\neg\phi \wedge \neg\psi)}{\phi \mid \psi}$	$\vee$	Branch on disjunctions
$\dfrac{e_1, \ldots, e_n \in \mathcal{H} \quad \exists_e.\phi[e],\ e_{\text{new}} \notin \mathcal{M}}{\phi[e_1] \mid \ldots \mid \phi[e_n] \mid \phi[e_{\text{new}}]}$	$\exists$	Instantiate existential quantifiers

Table 3.2: Tableau inference rules shared across designs.

3.3 Abstract System Architecture

The system follows a modular pipeline that transforms controlled natural-language narratives into formal semantic interpretations. Each module operates on a well-defined representation and passes structured data to the next stage, ensuring reproducibility and traceability of the reasoning process. Figure 3.1 illustrates the overall workflow from parsing to guided model generation.

Parsing and Logical Translation. The input to the system is a list of parsed sentences, each represented as a structured record containing a verb, a list of semantic-type-annotated actors, and a Boolean flag indicating whether the sentence is negated. These sentence objects serve as the intermediate form between natural language and the logical layer. Rather than translating directly into formulas at this stage, the parser preserves surface information for reporting and passes the structured entries to the model generator for integration into the ongoing dialogue context.

Ontology Layer. When a new sentence is presented to the model generator, its verb and annotated actors are matched against the ontology of situations, frames, and roles. The ontology provides typed templates describing which roles are expected for each frame and how they relate to semantic types such as *human*, *object*, *location*, or *occasion*. A successful match instantiates the corresponding frame, links its roles to the sentence's actors, and activates the situation associated with that frame. These instantiated predicates become part of the logical workspace handled by the tableau engine, ensuring that every linguistic input is grounded in a coherent semantic structure before reasoning proceeds.

Tableau Engine. The tableau engine is implemented as a recursive data structure that maintains a (possibly partial) model under construction. Each tableau instance contains a branch of logical formulas representing one candidate interpretation of the discourse. Applying an inference rule extends this structure: some rules expand the current model directly, modifying the existing branch, while others introduce disjunctions that generate branching extensions. In the latter case, new leaves are created, each corresponding to a distinct continuation of the interpretation. These leaves form the boundary between completed and open branches and serve as nodes for the model generator.

Model Generator. The model generator orchestrates the search over tableau leaves, treating each open branch as an expandable node. It maintains the fringe of the search and controls how new tableau instances are explored. Expansion follows a modular interface that accepts either a comparison function or a cost function to prioritise nodes. This abstraction allows the system to instantiate different guidance mechanisms while preserving the same reasoning core. In practice, the generator can implement a priority queue ordered by the comparator or a cost-based ranking strategy, both of which integrate seamlessly with the concurrent expansion loop. This design isolates inference from search control and enables flexible, policy-driven model generation.

Guidance Interface. The guidance interface defines the contract between the model generator and the policy or scoring component that orders the search frontier. It abstracts the notion of "guidance" as a pluggable evaluation module that assigns a priority to each open tableau node. The interface can be realised either as a comparison function, supporting

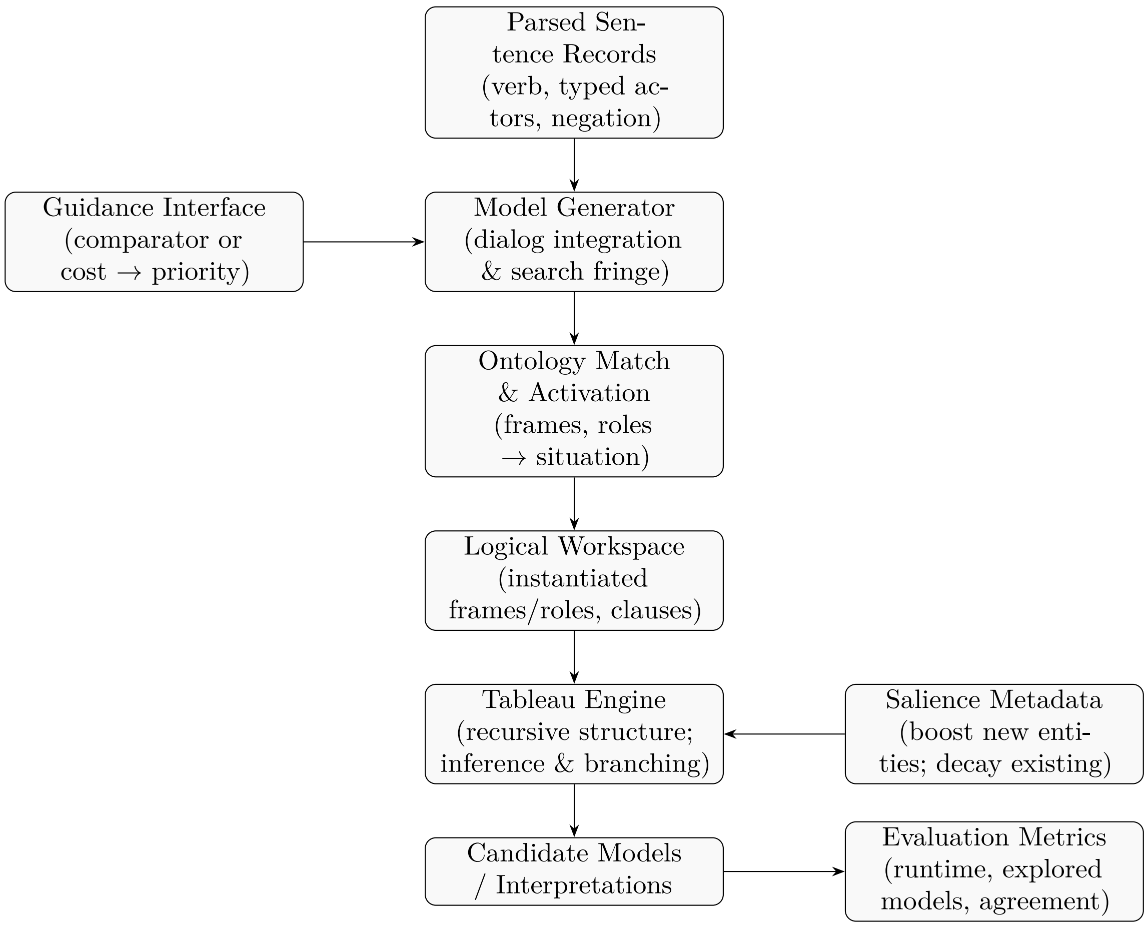
Parsed Sentence Records (verb, typed actors, negation)
Guidance Interface (comparator or cost → priority)
Model Generator (dialog integration & search fringe)
Ontology Match & Activation (frames, roles → situation)
Logical Workspace (instantiated frames/roles, clauses)
Tableau Engine (recursive structure; inference & branching)
Salience Metadata (boost new entities; decay existing)
Candidate Models / Interpretations
Evaluation Metrics (runtime, explored models, agreement)

Figure 3.1: System overview. Parsed sentence records are integrated into the dialogue by the model generator, which invokes ontology matching to instantiate frames and activate the situation. The tableau is a recursive structure whose leaves (open branches) feed the generator's search fringe; a pluggable guidance interface prioritises which leaf to expand next.

a priority-queue ordering, or as a cost function that maps nodes to scalar scores. It receives metadata such as branch depth, salience statistics, and the number of accumulated formulas, and returns a priority value used by the model generator to decide which branch to expand next. By isolating this control layer from the inference logic, the system remains agnostic to the specific guidance mechanism and can accommodate symbolic heuristics or learned scoring functions without altering the underlying tableau or search structures.

3.4 Guidance Paradigms

Guidance refers to the control strategy that determines how the model generator explores the fringe of the search space to construct a consistent model. It defines the order in which open tableau nodes are expanded, independent of the logical calculus itself. The guidance mechanism can follow a simple comparison policy, such as depth-first or breadth-first traversal, or a more elaborate priority scheme that considers properties of the candidate model, including its size, depth, or the current salience of discourse entities. By steering exploration through these structural cues, guidance balances completeness and efficiency while keeping the reasoning process reproducible.

At the implementation level, guidance is realised through a modular interface that receives the tableau node, as well as metadata about the node, and returns an ordering signal to the model generator. This interface can take the form of a comparison function used by a priority queue, or a cost function that assigns scalar scores to nodes, allowing statistical or neural models to influence search without altering the underlying tableau or ontology. Each time a new node is generated, the model generator inserts it into the ordered fringe according to the guidance output. This design isolates search control from inference, ensuring that both symbolic and data-driven strategies can operate transparently within the same reasoning framework.

3.5 Summary of Design Principles

The overall architecture rests on a small set of principles that ensure modularity, reproducibility, and extensibility. Each component of the system (logic, ontology, tableau, and guidance) operates through a clearly defined interface so that extensions or replacements can be introduced without altering the underlying reasoning model. The result is a unified framework capable of hosting both symbolic and data-driven control strategies within the same tableau infrastructure.

- **Separation of concerns.** Logical inference, ontological grounding, and search control are implemented as independent modules. This separation preserves the clarity of each component and prevents side effects between reasoning and guidance.

- **Reproducibility.** A fixed set of tableau rules and deterministic data structures ensures that model generation and evaluation remain stable across runs.

- **Extensibility.** The abstract guidance interface supports multiple control paradigms, from rule-based comparators to cost-based scoring, without modifying the tableau or ontology code paths.
- **Interpretability.** Salience-aware reasoning provides an explicit and traceable account of how discourse entities influence search decisions, making the inference process transparent.

Together, these principles define a reproducible and interpretable reasoning framework that separates semantic representation from search control. The following chapters build on this foundation by instantiating two implementations that adopt different guidance strategies while preserving the same underlying design.

Chapter 4

Implementations

4.1 Shared Runtime Architecture

This section summarizes the runtime shared by both designs and how parsing, ontology grounding, tableau reasoning with salience, and a pluggable guidance interface form a single pipeline. The aim is to keep inference (the tableau calculus and ontology grounding) decoupled from search control (policies or cost functions), so that symbolic comparators (Design A) and learned scorers (Design B) can be swapped without touching the reasoning core. The terminology (*model generator*, *guidance interface*, *salience*) follows Chapter 3 (cf. §3.4 and §3.2.4).

Overview and dataflow. Inputs are parsed sentence records that retain surface verb, typed actors, and a negation flag. These records are not fully compiled to logic at parse time; instead, the model generator integrates each record into the discourse context by matching the ontology and instantiating frames/roles before tableau expansion proceeds. This preserves a clean interface between linguistic preprocessing and logical reasoning.

- **Parsing $\rightarrow$ ontology activation.** For each sentence, the ontology module matches situation/frame templates and fills role slots for actors with semantic types (e.g., `human`, `object`, `location`, `occasion`); unknown referents are introduced existentially for later unification. The instantiated predicates are deposited into the logical workspace.
- **Tableau + salience (rule-level).** The tableau engine expands a branch to saturation with non-branching rules and creates successors on disjunctive steps; salience metadata is updated after rule applications by boosting newly introduced individuals and

decaying existing ones. This is rule-level salience applied during rule expansion; it biases *witness selection* for existential instantiation but does not determine frontier ordering.

- **Guidance interface.** The model generator maintains an ordered fringe of open branches. Ordering is delegated to a *guidance interface* that can be realized as (i) a comparator (symbolic policies) or (ii) a scalar cost function (learned guidance), enabling both designs to use the same expansion loop.

Tableau calculus and salience scope. Both designs share the same calculus (conjunction/decomposition, double-negation elimination, instantiation of ∀/∃, and disjunction branching). Salience is confined to rule application: every introduction of new individuals receives full salience while previously active discourse entities decay by a fixed factor. This mechanism biases which constants are chosen as witnesses when instantiating existential claims but remains orthogonal to the ordering of frontier nodes. This separation is emphasized because later sections vary only the ordering signal while holding the calculus and salience policy fixed.

Model generator and queues. The model generator treats each *open leaf* as a candidate model and orchestrates its expansion under a priority discipline. In practice, it uses a priority queue keyed by the guidance interface:

(1) **Symbolic policies (Design A).** Policies implement a comparator and thus directly define the queue ordering (e.g., BFS, DFS, Custom comparator). Details of the queue implementation, workers, and tunables appear in §4.2.

(2) **Learned/external scoring (Design B).** The same interface accepts scalar scores from neural cost functions or an external heuristic. Because encoders consume grouped event literals, Design B computes scores only when a branch has reached an *open leaf* with the necessary bundle (subject/verb/object and, in this design only, adjective). This constraint leaves the calculus unchanged but postpones scoring relative to Design A.

Representations by design (storage view). Design A stores ontology-grounded structures: situations, frames, roles, and semantic types, mirroring the event-semantics/frames ontology and enabling constraint checks (e.g., unique role assignments, domain restrictions). Design B projects the same discourse state to event-level literals grouped per event:

`subject(e, _)`, `verb(e, _)`, `object(e, _)`, and an ad-hoc `adjective` predicate used only here; explicit frame identifiers are not retained in stored state. The projection is purely representational and leaves rule application unchanged.

Decoupling inference from guidance. The shared runtime enforces the boundary between inference and guidance:

- *Inference path*: ontology grounding → tableau rule application → salience update (rule-level).
- *Guidance path*: metadata extraction (depth, new-individual count, accumulated formulas, salience summaries as needed) → comparator/score → queue priority.

This design enables interchangeable control strategies and supports reproducible comparisons since the reasoning substrate is identical across policies/scorers.

Concurrency and scope of details. Parallel expansion and queue-drain/timeout controls belong to the Java implementation and are described in §4.2. The present section only fixes the *locus* of guidance (frontier ordering) and the *scope* of salience (rule-level witness selection) so that later sections can vary policy/scorer and thread model independently.

Design B scoring constraint (operational note). Because Design B's cost functions and external heuristic operate on bundled event literals, branches are scored when they are fully available as open leaves. Until then, expansion proceeds deterministically under the calculus and salience. This "score-at-leaf" constraint is operational only and does not alter the calculus itself.

Cross-references and labels. This runtime assumes the theoretical framework in Chapter 3: logic signature (§3.2.1), ontology (§3.2.3), tableau rules and salience (§3.2.4), and guidance paradigms (§3.4). Implementation-specific details follow in §4.2 (Design A) and §4.3 (Design B).

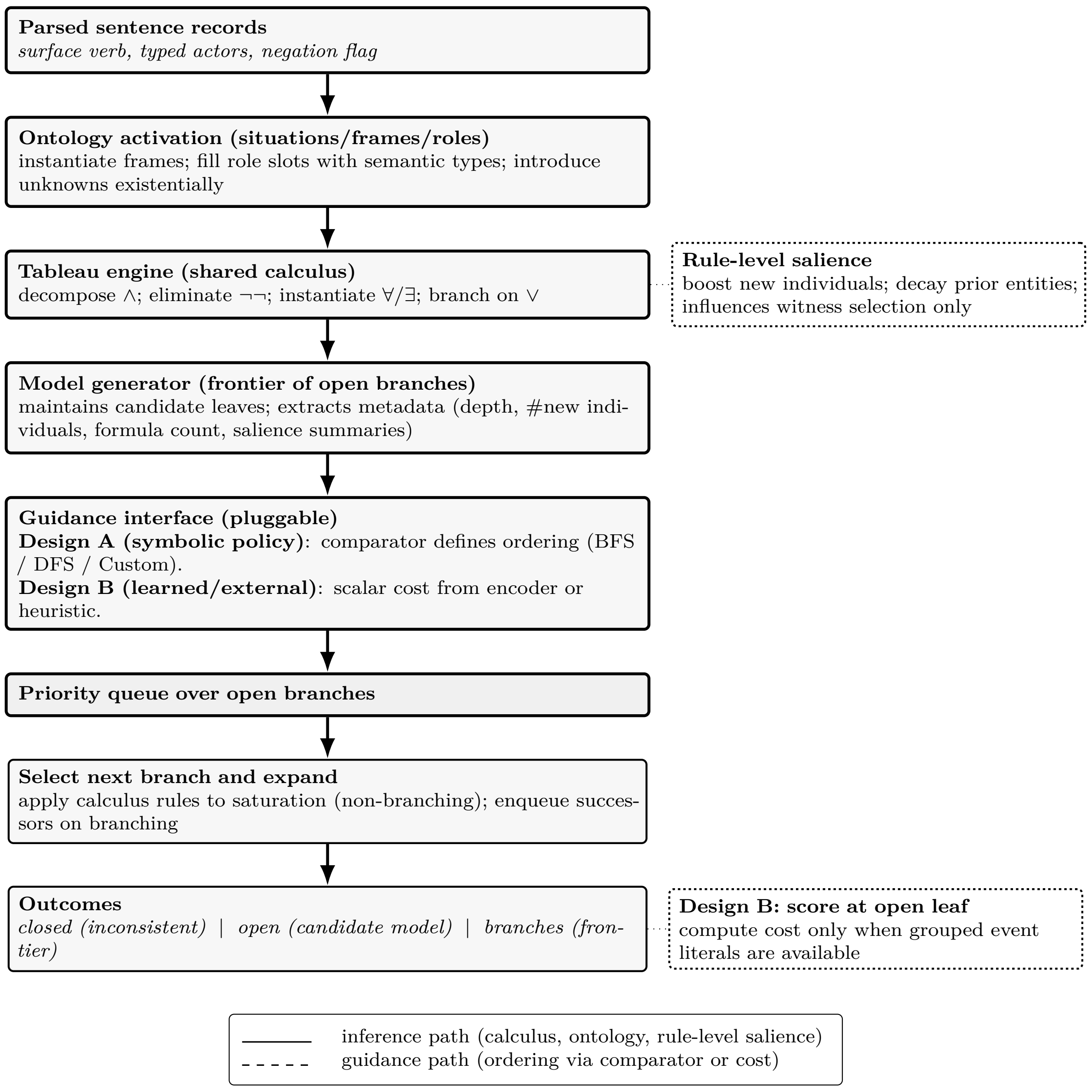
Parsed sentence records
surface verb, typed actors, negation flag
Ontology activation (situations/frames/roles)
instantiate frames; fill role slots with semantic types; introduce unknowns existentially
Tableau engine (shared calculus)
decompose ∧; eliminate ¬¬; instantiate ∀/∃; branch on ∨
Rule-level salience
boost new individuals; decay prior entities; influences witness selection only
Model generator (frontier of open branches)
maintains candidate leaves; extracts metadata (depth, #new individuals, formula count, salience summaries)
Guidance interface (pluggable)
Design A (symbolic policy): comparator defines ordering (BFS / DFS / Custom).
Design B (learned/external): scalar cost from encoder or heuristic.
Priority queue over open branches
Select next branch and expand
apply calculus rules to saturation (non-branching); enqueue successors on branching
Outcomes
closed (inconsistent) | open (candidate model) | branches (frontier)
Design B: score at open leaf
compute cost only when grouped event literals are available
inference path (calculus, ontology, rule-level salience)
guidance path (ordering via comparator or cost)

Figure 4.1: Shared runtime architecture (top to bottom). Inference path: parsing → ontology activation → tableau calculus with rule-level salience. Guidance path: the model generator delegates frontier ordering to a pluggable interface (symbolic comparator in Design A or learned/external cost in Design B). The priority queue orders open branches; Design B scores at open leaves only.

4.2 Design A: Java Policy Engine

Design A instantiates the shared system architecture 3.4 in a multithreaded Java stack that steers tableau expansion with symbolic guidance policies. The engine maintains a knowledge-graph ontology of situations, frames, roles, and semantic types; integrates parsed sentences into this ontology; and explores candidate interpretations by expanding open tableau leaves under a policy-driven priority queue. Salience is updated after rule applications and informs witness selection, while guidance remains a separate control layer that orders which branches are expanded next.

4.2.1 Modules and Data Flow

Ontology and parsing. Input narratives arrive as lists of parsed sentence objects that preserve surface forms (verb, typed actors, negation). The ontology module matches these entries to situation/frame templates, instantiates frames, fills role slots, and activates the associated situation; the resulting predicates populate the logical workspace consumed by the tableau engine. Individuals are typed via *semType* relations and unknown referents are introduced existentially, enabling later unification. This grounding step ensures that each linguistic input is embedded in a coherent structure before inference proceeds.

Tableau core and salience. The tableau is a recursive structure whose open leaves represent candidate interpretations; non-branching rules saturate the current branch, while disjunctive rules create new leaves. Salience metadata is boosted for newly introduced individuals and decays across expansions, providing a signal for witness selection that remains independent from the queue ordering policy. The Java stack consumes the ontology, updates salience after each rule instantiation, and prepares metadata for guidance.

Model generator and guidance interface. A model generator maintains the search fringe as a priority queue of open branches. Guidance is realised as a modular interface: in Design A, policies implement a *comparator* that orders nodes; the same abstract contract can later host cost functions in Design B without altering the tableau or ontology code paths. Each time a new node is created, it is inserted according to the comparator, and workers dequeue according to current priorities. This isolates inference from search control and keeps runs reproducible.

4.2.2 Priority-Queue Policies

Design A implements three symbolic policies via comparator modules. These policies are used consistently throughout the Java evaluation pipeline; experiments record elapsed time, explored models, and whether a consistent model was found for each story, while interpretation of those outcomes is deferred to later chapters.

- **Breadth-first (BFS).** Orders nodes by shallowest depth first, approximating level-order traversal over the constructed models.
- **Depth-first (DFS).** Prefers deeper nodes, exploring along one branch before back-tracking.
- **Custom (salience-aware).** Prioritises (i) greater sentence depth, (ii) fewer newly introduced individuals, and (iii) more accumulated formulas. This comparator exploits ontology- and salience-derived structure while staying lightweight.

Motivation for custom policy. The Custom comparator encodes search preferences that are natural for tableau-style model generation in short narratives. Preferring greater sentence depth biases expansion toward branches that have already incorporated more of the dialogue, so constraints introduced later (where pronoun ambiguities typically appear) become available earlier and the search avoids spending effort on still-underspecified partial models. Preferring fewer newly introduced individuals mitigates existential blow-up: when multiple witnesses are possible, branches that re-use salient discourse entities tend to stay smaller and align better with the salience assumptions used elsewhere in the system. Finally, preferring more accumulated formulas approximates a most-constrained-first discipline: branches closer to saturation expose inconsistencies earlier under ontology checks and can be pruned sooner. Together, these priorities provide a lightweight way to steer exploration toward coherent, compact candidates without introducing domain-heavy scoring or learned guidance.

4.2.3 Concurrency Model and Runtime Controls

The implementation supports both sequential and parallel execution via a worker pool. In the sequential configuration, a single worker repeatedly pops from the priority queue, applies tableau rules (saturating non-branching rules), pushes any branched successors, and continues until a stopping condition. In the parallel configuration, multiple workers (e.g., twelve) share the same concurrent priority queue and coordinate expansion under per-iteration time limits. Two queue-related knobs stabilise behaviour:

- **Drain size.** The number of items popped from the priority queue per iteration before priorities are recomputed; this limits priority churn under high concurrency.
- **Per-iteration timeout.** A wall-clock budget (milliseconds) for an expansion cycle; used to bound latency and keep runs comparable across policies.

A *worker count* parameter switches between 1-thread and multi-threaded operation. The same code path is used for both modes, enabling direct comparison of single-thread and 12-thread runs later in the Results chapter.

4.2.4 Consistency Checks and Pruning

Consistency checks enforce ontological well-formedness during expansion. The engine prunes branches that violate constraints such as unique frame-role assignments and domain restrictions (e.g., murderer $\neq$ victim) or that disrupt situation-specific coherence. These checks interact with salience by preventing growth of implausible branches before they consume queue budget, but they do not alter the guidance interface; pruning happens inside inference, whereas ordering remains the policy's responsibility.

4.2.5 I/O, Logging, and Reproducibility Hooks

Runs emit human-readable logs that capture policy, thread count, timeout, and summary metrics (time in milliseconds, explored-model count). Output filenames encode worker and timeout settings; naming conventions and examples are listed in Appendix A. Build and run instructions, along with directory layout (ontology location and story files), are specified in the reproducibility appendix and referenced by the experiments chapter. *No interpretation* is attached to these files here; they serve solely as artefacts for later reporting.

4.2.6 Capabilities and Limitations (Design A, Implementation View)

From an implementation standpoint, Design A provides (i) a reproducible tableau engine with salience-aware rule application, (ii) pluggable comparator policies, (iii) a concurrent priority queue with drain-size and per-iteration timeout controls, and (iv) deterministic

Table 4.1: Java engine tunables and controls used in Design A. Values are set per run; commands and filenames are listed in Appendix A.

Parameter	**Description**
Priority comparator	One of *BFS*, *DFS*, or *Custom*. The custom policy prefers (i) greater depth (↑), (ii) fewer newly introduced individuals (↓), and (iii) more accumulated formulas (↑).
Drain size	Number of items popped from the priority queue per iteration before recomputing priorities; reduces priority churn and stabilises ordering under concurrency.
Per-iteration timeout	Wall-clock budget (in milliseconds) for an expansion cycle; applied in both single-thread and multi-thread runs to bound latency and keep configurations comparable.
Worker count	Runtime parallelism: sequential (1 worker) or multi-threaded (e.g., 12 workers) sharing a concurrent priority queue.
Consistency checks	Ontology- and frame-level constraints enforced during expansion (e.g., unique frame-role assignments; domain restrictions such as murderer $\neq$ victim); violating branches are pruned inside inference.
Log outputs	Text files with configuration-encoded names summarising policy, thread count, timeout, elapsed time, and explored-model counts; used later for tabular reporting. See Appendix A for file-naming conventions.

logging suitable for tabular reporting. The stack deliberately avoids learned components; it relies on the explicit ontology and salience calculus and defers any scoring-based guidance to Design B. Any observations about stability or comparative efficiency are reserved for Chapters 5–6.

4.2.7 Notes on Terminology and Cross-References

This section adheres to the new-draft terminology: *guidance* as the decoupled search-control layer; *policy* as a symbolic comparator over search states; *model generator* as the queue-backed orchestrator; and *salience* as rule-level metadata for witness selection. Conceptual background resides in Chapter 3; evaluation protocols and numeric outcomes appear in Chapters 5–6.

4.3 Design B: Python Data-Driven Prototypes

Design B reuses the shared tableau calculus and salience mechanism but replaces symbolic policies with data-driven guidance that operates on a projected state. The Python stack mirrors the search loop of Design A while storing event-level literals and deferring scoring to *open leaves* so that encoders can consume complete bundles. Guidance modules include (i) static policies for parity checks with Java ordering, (ii) neural cost functions (GRU baseline and a variant with contextual embeddings), and (iii) an external heuristic that ranks branches from short textual descriptions.

Projected state and scoring locus. After ontology-grounded expansion (situations/frames/roles) is conceptually available, the Python tooling *projects* each open branch to grouped event literals: `subject(e, _)`, `verb(e, _)`, `object(e, _)`, plus an ad-hoc `adjective(adj, a)` predicate used only in Design B. The projection omits explicit frame identifiers; encoders therefore consume event-indexed bundles rather than ontology nodes. Because encoders require the full bundle, branches are scored only once they are open leaves. The tableau rules and rule-level salience are otherwise unchanged.

Search core. The prototype keeps the model generator and priority-queue discipline from the shared runtime. It applies the same focus strategy and salience updates as Design A but stores the projected bundles instead of ontology frames/roles. Node expansion proceeds deterministically under the calculus; when an open leaf forms, the guidance interface requests a score and pushes the branch back into the priority queue with that priority.

Static policies (parity checks). For validation, the Python implementation re-creates simple static ordering signals (average-salience and minimum-events) to verify that, when identical guidance is used, the Python stack yields the same ordering as the Java engine. These baselines serve as a sanity check on the search implementation rather than an alternative to learned scoring.

Neural cost functions. The main learned module encodes the sequence of event bundles with a gated recurrent unit (GRU) and maps the final state to a scalar priority (Figure 4.2). A variant augments inputs with contextual embeddings (BERT) where applicable. Training follows two stages: (1) *trace regression* to reproduce the average-salience ordering learned from synthetic policy traces, and (2) *contrastive fine-tuning* on human-annotated preferences

Table 4.2: Stored state representation by design (for later cross-reference).

Design	**Stored state (search nodes)**
Design A (Java)	Ontology-grounded situations, frames, roles, semantic types; salience metadata used for witness selection during rule application.
Design B (Python)	Projected event-level bundles `subject/verb/object` plus ad-hoc `adjective` (Design B only); no frame identifiers retained; scoring invoked at open leaves.

over projected interpretations. The intent is to match symbolic policy orderings and then adapt toward human judgements while keeping the calculus fixed.

External heuristic. As an additional scorer, a prompted heuristic ranks successor branches from short textual descriptions and returns a scalar priority to the same queue. This module does not replace the tableau or the projection; it only supplies priorities that the search loop consumes.

Runtime interface and reproducibility hooks. Design B uses the same priority-queue API as Design A but calls the guidance interface at open leaves. Example commands and paths are provided in Appendix A: experiment demos, data perparation, and training scripts. These references define the environment (Python 3.10+, PyTorch) and seeds.

Representation contrast (storage view). Table 4.2 summarises the stored-state difference between designs: Design A persists ontology structures and constraints directly; Design B stores event bundles and relies on the shared calculus/salience during expansion while leaving ontological checking implicit in the logic.

Operational notes.

- **Score-at-leaf constraint.** Scoring is postponed until a branch is an open leaf with a complete event bundle; until then, expansion is calculus-driven with rule-level salience.
- **Interface parity.** Static policies confirm queue-ordering parity with Java when the same guidance signal is used.
- **Encoders.** The GRU cost function is the primary learned scorer; an attention-based encoder was prototyped but is optional to include as a figure in this chapter.

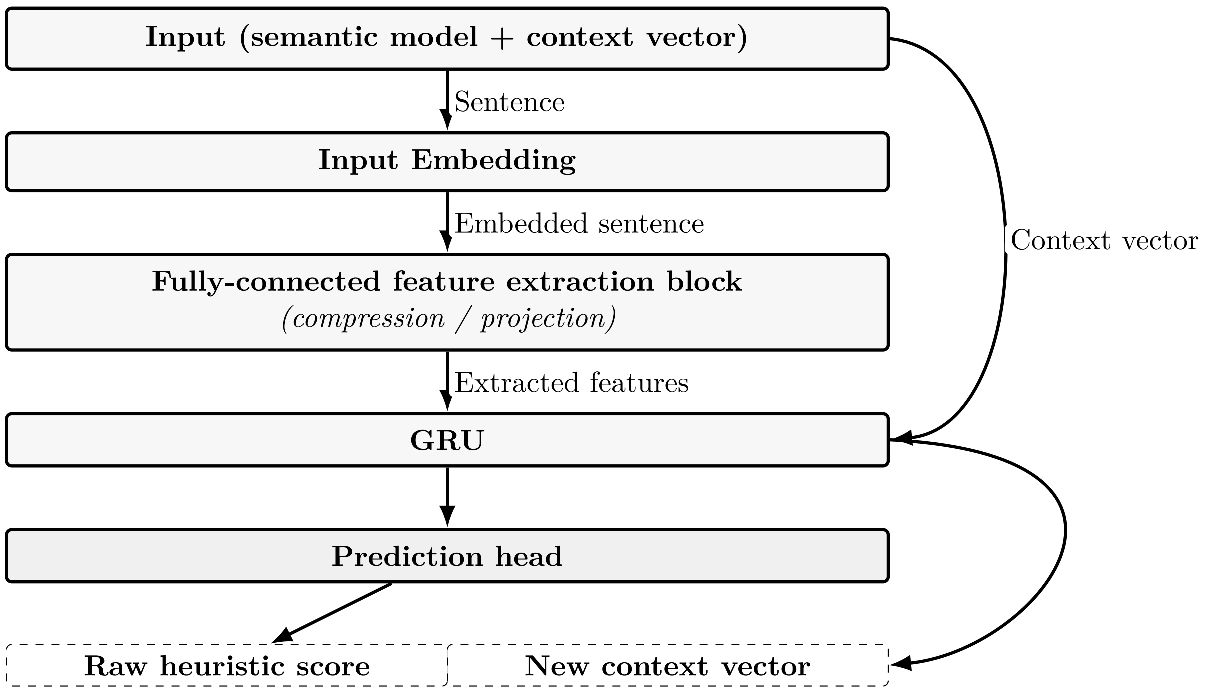

Figure 4.2: Design B GRU cost function.

Cross-references. This section adopts the new-draft terminology for *guidance*, *policy* (symbolic comparator), and *cost function* (learned scalar scorer). For the theoretical background on the projection and encoder motivations, see Chapter 3; for concrete training/evaluation commands, see Appendix A. Numeric outcomes and stability observations appear in Chapters 5–6.

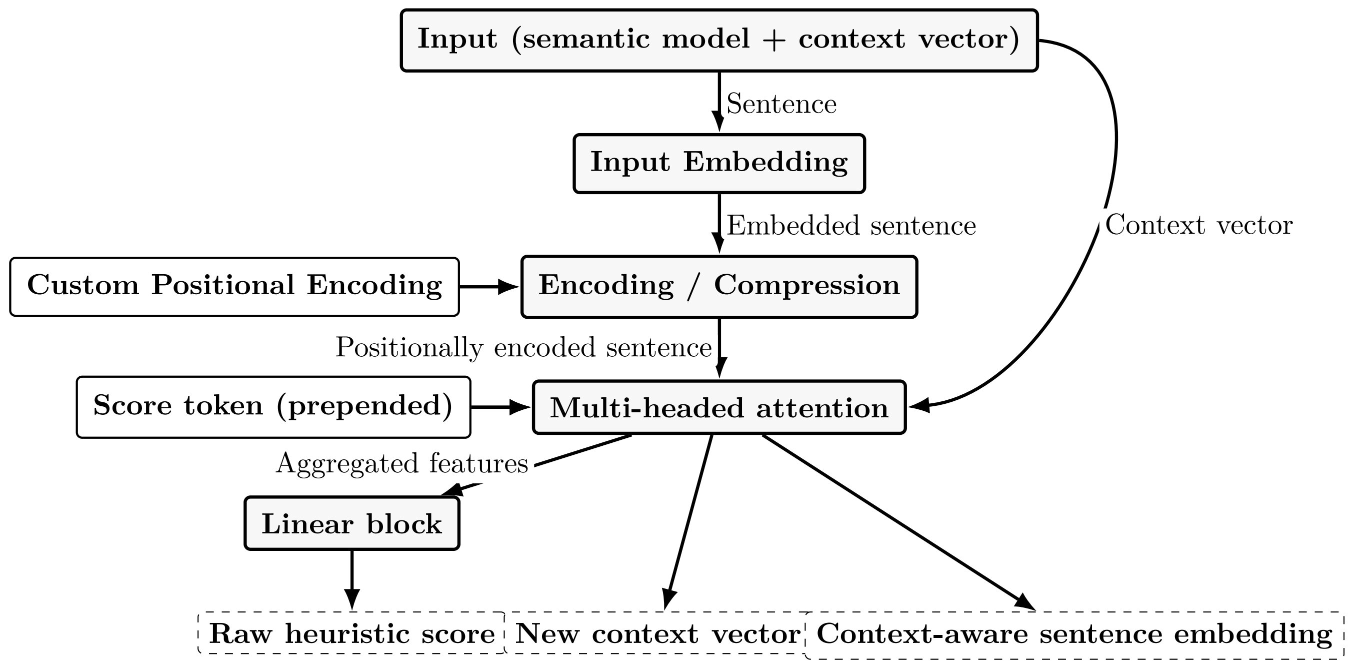

Figure 4.3: Design B attention-based encoder (input at top, outputs at bottom) with arrow annotations.

4.4 Implementation Summary

Scope. This section consolidates the implementation facts established in §4.1, §4.2, and §4.3. The shared runtime separates inference (tableau calculus and ontology grounding with rule-level salience) from search control (guidance), so that symbolic comparators (Design A) and learned/external scorers (Design B) can be swapped without modifying the reasoning core. Salience affects witness selection during rule application only; it does not order the frontier.

Capabilities (Design A: Java).

- *Policy-driven priority queue.* Pluggable comparator policies—Breadth-First (BFS), Depth-First (DFS), and a Custom comparator that prefers greater sentence depth, fewer newly introduced individuals, and more accumulated formulas—determine frontier order.

- *Concurrent expansion.* A worker pool shares a concurrent priority queue; the same code path supports single-thread and multi-thread runs. Queue *drain size* and a per-iteration *timeout* bound churn and latency.

- *Ontology checks and pruning.* Frame-role uniqueness and domain constraints (e.g., *murderer* $\neq$ *victim*) are enforced during expansion to prune infeasible branches, independent of policy ordering.
- *Reproducible I/O.* Runs emit human-readable logs with policy, worker count, timeout, elapsed time, and explored-model counts; filenames encode settings and are referenced from the reproducibility appendix.

Limitations (Design A: Java, implementation view).

- Search control is purely symbolic (no learned scoring); salience remains a rule-level signal rather than a frontier order.
- Coverage is constrained by the ontology and the controlled language fragment used in the pipeline.

Capabilities (Design B: Python prototypes).

- *Projection + parity baselines.* The search core mirrors Design A's calculus but stores projected event-level bundles (`subject/verb/object` and an ad-hoc `adjective` predicate used only here). Static policies reproduce the ordering of their Java counterparts for parity checks.
- *Learned/external guidance.* A GRU cost function (and a variant with contextual embeddings) assigns scalar priorities; an external heuristic can also rank successor branches from short textual descriptors. All scorers plug into the same queue interface.
- *Score-at-leaf discipline.* Because encoders consume complete bundles, branches are scored only at open leaves; the calculus and rule-level salience remain unchanged.
- *Reproducibility hooks.* Example commands, paths, seeds, and environment notes are provided alongside the static-policy demo and two-stage training scripts.

Limitations (Design B: Python, implementation view).

- Stored state omits explicit frame identifiers (projection); guidance depends on encoder features and, when used, an external heuristic.
- Operationally, score computation is deferred until open leaves; invoking an external heuristic introduces additional latency by design.

Forward links. The experimental matrix and artefacts referenced here are summarised later (§5.1–§5.2), and numeric outcomes are reported in the Results chapter, with logs produced via the reproducibility steps in the appendix. *(Labels for §5 and appendix entries follow the final numbering of Chapters 5–A.)* The following chapter (§5) describes how these implementations were evaluated under comparable runtime constraints.

Chapter 5

Experiments

This chapter specifies the datasets, runtime settings, and evaluation protocols. All runs share the tableau calculus and rule-level salience described earlier; only the guidance layer and thread configuration vary (see Chapters 4 and 6). Numerical results appear in Chapter 6.

5.1 Datasets and Experimental Setup

Story banks. Two sets of stories were used for evaluation:

- **Story-bank baseline:** four short stories.
- **Annotated regression set:** nine mixed stories (office, heist, and Java-derived templates) annotated with preferred interpretations for salience and anaphora.

Each story was processed into event-level predicates following the ontology in Chapter 3.2.3. For every sentence, salience updates and ontology role mappings were recorded. Ontology coverage by role type is summarized in Table 5.1. Role counts were not recorded in the available logs for the compiled runs; we therefore report dataset sizes and descriptions.

Common runtime. Both designs share a queue-driven tableau generator with configurable per-iteration timeouts and drain-sizes. Guidance policies are the sole variable across conditions.

Metrics. The evaluation metrics collected for all runs are defined in §5.1.1.

Hardware and process control. All experiments were executed on the same workstation. Worker count, per-iteration timeout, and queue drain-size are fixed per configuration and encoded in the log filenames. Exact command lines and environment variables are documented in Appendix A.

Table 5.1: Benchmark story sets and ontology-role coverage.

Dataset	**Description / Role Coverage**
Story-bank baseline	4 short stories (role-type counts not recorded in logs).
Annotated regression set	9 stories mixing office, heist, and Java templates (role-type counts not recorded).

5.1.1 Evaluation Metrics

We distinguish between *search-time* metrics (reported for both designs) and *training-time* metrics (reported only for Design B).

Search-time metrics. All evaluation runs record:

- **Elapsed time (ms).** Total wall-clock runtime per story.
- **Explored models.** Number of tableau leaves expanded until the first consistent interpretation is found (or the run exhausts its budget).
- **Success.** Binary indicator: True if a consistent model is found within the timeout/queue budget, false otherwise.
- **Ranking agreement (Design B).** Agreement between the Python branch ordering induced by a scorer and the reference ordering produced by the Java Custom policy. We compute Spearman's ρ over the ranked open leaves of each story and report the mean across stories; $\rho = 1$ indicates identical rankings, while $\rho = 0$ reflects chance-level agreement.

Training-time metrics (Design B). Neural scorer training happens outside of the search loop. For each phase we log:

- **Trace-regression loss.** Mean-squared error between predicted priorities and target priorities derived from policy traces (recorded per epoch in the training logs).

- **Trace-regression accuracy.** Fraction of training records whose sigmoid outputs fall on the same side of 0.5 as the trace-derived target (treated as a proxy for ordering alignment).

Unless stated otherwise, all reported training values are computed on a held-out validation split with fixed seeds (train seed = 40, evaluation seed = 42).

5.2 Evaluation Protocols

5.2.1 Design A: Java Policy Engine

Design A evaluates three symbolic comparator policies (*Breadth-First Search*, *Depth-First Search*, and a salience-aware *Custom* policy) under two thread settings:

- **1 Worker:** single-thread search; per-iteration timeout = 200 ms.
- **12 Workers:** shared concurrent priority queue; per-iteration timeout = 200 ms.

Queue drain-size and logging follow the implementation defaults. See Appendix A for file-naming conventions and commands. No numeric values are reported in this chapter; see Chapter 6 for aggregated results.

5.2.2 Design B: Python Data-Driven Prototypes

Design B tests both static parity baselines and data-driven guidance mechanisms built on the same tableau interface.

Static parity baselines. These baselines (*Average-salience*, *Minimum-events*) verify ordering parity with the Java implementation. Runs use a single-thread, score-at-leaf search; per-iteration timeout = 600 ms. Metrics follow §5.1.1 (search-time metrics).

Learned cost functions. Learning proceeds in two phases: Phase 1 reproduces average-salience orderings on synthetic traces, Phase 2 fine-tunes on human-annotated preferences. Evaluation uses single-thread search with score-at-leaf, 600 ms per-iteration timeout, and fixed seeds (training seed = 40; evaluation seed = 42). Evaluation uses the search-time metrics in §5.1.1, and training reports the training-time metrics defined there.

External heuristic. A prompted scoring function (external LLM heuristic) ranks open leaves. Evaluation uses single-thread search, per-iteration timeout = 800 ms. Prompts and templates remain constant across runs; variance will be reported factually in Chapter 6.

Common logging

All protocols record: policy name (or scorer), worker count, per-iteration timeout, explored models, and elapsed time in milliseconds. Filenames encode configuration parameters; see Appendix A for naming conventions and exact commands.

Table 5.2: T5.B: Run configurations for all experiments.

Design	**Policy type**	**Threads**	**Timeout (ms)**	**Notes / Seeds**
A (Java)	Custom / DFS / BFS	1	200	Baseline single-thread setup.
A (Java)	Custom / DFS / BFS	12	200	Parallel shared queue.
B (Python)	Static parity	1	600	Evaluation seed = 42.
B (Python)	Learned policy	1	600	Two-phase; train seed = 40; eval seed = 42.
B (Python)	External heuristic	1	800	Prompted scoring; seed = 42.

5.3 Reproducibility Notes

All environments, path settings, and exact command lines are listed in Appendix A. This chapter defines only datasets and protocols; the following chapter reports the numerical outcomes of these runs (Chapter 6).

Chapter 6

Results

This chapter reports empirical outcomes for the experiments without interpretation. Design A (Java) is evaluated under three comparator policies (BFS, DFS, Custom) and two thread configurations (1 worker, 12 workers). Design B (Python) reports static/learned guidance on the current story sets. All runs share the same tableau calculus and rule-level salience; only the guidance layer and thread configuration vary (per Chapters 4–5).

Common logging

All results are derived from the same structured logs (policy/scorer, worker count, per-iteration timeout, explored models, elapsed time). See Appendix A for file locations and naming conventions. Results are reported in order of implementation: Design A (symbolic) followed by Design B (data-driven).

6.1 Results A: Policy Variants (Design A, Java)

Table 6.1 aggregates elapsed time (milliseconds) and explored models for the Java engine. Single-thread averages are taken directly from the 1 worker logs (200 ms per-iteration budget). Multi-thread averages are computed over the five benchmark variants from the 12 worker logs (same per-iteration budget).

Table 6.1: T6.A: Policy runtimes and explored models for Design A (Java). Each entry reports the mean across the five benchmark variants under 1-thread and 12-thread configurations (per-iteration timeout: 200 ms).

Configuration	**Metric**	**Custom**	**DFS**	**BFS**
1 worker	Avg. time (ms)	1,590	230,470	2,499
	Avg. models explored	1,312	102,606	2,307
12 workers	Avg. time (ms)	1,314	124,191	113,220
	Avg. models explored	4,080	70,431	257,536

Notes. 12-worker means are computed over the five story variants in the log. Some uninformed-policy variants exhibit runaway expansions; the averages above include those cases.

6.2 Results B: Static, Learned, and External Guidance (Design B, Python)

Design B operates on projected event-level bundles with score-at-leaf evaluation. The current results summarize two test sets:

- **Story-bank baseline** (4 short stories): policies evaluated as controls.
- **Annotated regression set** (9 mixed stories): static and learned/external guidance.

Table 6.2 reports the annotated-set averages.

Table 6.2: T6.C: Average performance of Design B (Python) guidance variants over the 9-story annotated regression set.

Policy	**Avg. time (ms)**	**Avg. models**	**Success rate**	**Notes**
CustomJava	881.4	90.9	1.00	Mirrors Java Custom ordering on this set
DFS	76,879.9	8,433.6	0.89	Complete on most runs; slow
BFS	9,122.0	747.3	0.44	Incomplete on this set
NeuralPolicy	9,083.4	794.9	0.33	Lower solve rate on longer variants

Ranking agreement. The branch-ordering agreement metric from §5.1.1 was computed for the CustomJava scorer against the Java reference ordering. On the annotated regression set the mean Spearman ρ is 1.0 with top-5 accuracy of 1.0 (per `experiments/policies/summary.json`).

Training metrics (Design B). The trace-regression phase reports mean-squared error declining from 0.1466 in epoch 1 to 7.7×10^{-4} by epoch 10, while binary trace-alignment accuracy reaches 1.0 from epoch 2 onward (`experiments/policies/CustomJava_logs/metrics.json`). Preference fine-tuning was not run for the compiled results.

For completeness, the baseline sweep over the 4-story set is not tabulated here; aggregate values are available in the accompanying logs (see Appendix A).

6.3 Cross-Family Comparison

Table 6.3 contrasts representative aggregates from both implementations using their respective controls or learned counterparts.

Table 6.3: T6.B: Cross-family comparison between Design A (Java) and Design B (Python). Values shown are representative averages drawn from Tables 6.1 and 6.2 and the baseline summary.

Aspect	**Design A (Java)**	**Design B (Python)**
Guidance mechanism	Symbolic comparator: BFS / DFS / Custom	Static parity, learned cost, external heuristic
Representative runtime	$\sim 1.3 \times 10^3$ ms (Custom, 1 worker)	$\sim 0.9 \times 10^3$ ms (CustomJava, annotated set)
Representative explored models	$\sim 1.3 \times 10^3$ (Custom, 1 worker)	$\sim 9 \times 10^1$ (CustomJava, annotated set)
Success on current sets	Model found in all benchmark stories (policies as configured)	CustomJava: 1.00; NeuralPolicy: lower success on longer variants
Variance (operational)	Low under fixed drain-size / timeout	External heuristic adds latency; learned policy shows variability

6.4 Ablations and Qualitative Error Patterns

Ablation knobs and error categories follow the setup in Chapter 5 (§5.2). Where applicable in the current logs: (i) varying thread count and queue budget affects uninformed policies disproportionately; (ii) longer templates increase variance for learned/external guidance. The concrete examples below illustrate the two dominant failure modes observed in the annotated and Java-variant runs.

Failure type (story)	**Surface cues**	**Custom**	**Neural**
Lexical drift (annotated story "David and Emma infiltrate a bank...")	`she brought the cake to vault.`	1.05 s	15.4 s
	`he infiltrated there.`	87 models	1,049 models
	`she accused him over it.`	success	failure
Role swap (Java design variant 2)	`someone murdered her there.`	0.48 s	9.85 s
	`he did not murder her.`	60 models	1,021 models
	`he murdered her.`	success	failure

Figure 6.1: Representative failure cases from the annotated experiments. Times are converted from milliseconds. Lexical drift introduces unseen vocabulary and moves the learned scores away from the reference; the symbolic policy remains stable. In the role-swap case, alternating mentions of "he" (john vs. unknown human) cause oscillation, so the correct branch is not prioritised. The last two columns report time/models and whether the run succeeded.

Chapter 7

Discussion and Conclusion

This chapter interprets the empirical findings of Chapter 6, ties them back to the guidance paradigms and design principles in Chapter 3, and states the main conclusions. We separate *Discussion* (Section 7.1) from *Conclusion* (Section 7.2) to keep interpretation distinct from summary and outlook. The terminology remains consistent: a model generator over a tableau, salience at the rule level, and guidance from either symbolic policies (Design A, Java) or learned scorers (Design B, Python).

7.1 Discussion

7.1.1 Symbolic Guidance (Design A)

The Java engine paired with the ontology of situations, frames, roles, and semantic types provided a robust baseline for guided model generation. Policies were implemented as comparators over frontier states in a concurrent priority queue, orthogonal to the salience mechanism that governs witness selection at the rule level. This separation is methodologically important: salience influences *which* constants instantiate quantifiers when rules fire, while the policy determines *which* partially expanded interpretations reach the head of the queue.

Empirically, the custom policy (which prefers greater sentence depth, fewer new individuals, and more accumulated formulas) was the most dependable. On the benchmark stories it kept runtimes and explored models stable in both the single-thread and the twelve-worker settings (Table 6.1). Parallelism increased speculative work, as expected, but did not prevent the policy from finding a satisfying interpretation. The uninformed baselines behaved differently: BFS explored many more states with twelve threads, while DFS sometimes benefited from concurrency yet remained brittle because it commits early to deep paths that are later pruned by ontology checks.

Two observations follow. First, lightweight symbolic comparators, when coupled with ontology-level consistency constraints and salience, are sufficient to steer tableau search effectively over the curated fragment used in this thesis. Second, the multithreaded execution and queue-drain/timeout tunables exposed by the Java implementation serve as meaningful levers for resource/control trade-offs without altering the logical calculus or the guidance abstraction (cf. Chapter 4 and Table 4.1).

7.1.2 Data-Driven Guidance (Design B)

The Python prototypes retain the same calculus and salience logic but operate on a projected event-semantics footprint where situation/frame structures are collapsed into grouped literals at the event level (subject, verb, object; with an ad-hoc adjective predicate confined to the Python side). This projection is by design: it reduces the stored state to a form that is more amenable to neural encoders while keeping rule application identical. The guidance layer in Design B thus swaps symbolic comparators for learned cost functions (GRU encoders with optional contextual embeddings) and, in a separate control path, an external heuristic that ranks successor branches from textual descriptors.

On held-out stories, neither the trace-regression models nor the human-refined variants delivered stable guidance. Small lexical substitutions re-ordered branches unpredictably, and the limited human-annotated set could not adequately correct biases learned from policy traces. The external heuristic sometimes recovered the intended branch but introduced additional latency and run-to-run variance that complicate controlled evaluation (cf. Section 6.2; see also Table 6.3).

These outcomes highlight a central challenge for learned guidance here: scorers need an inductive bias that reflects the process features of tableau growth (for example, salience dynamics and rule saturation) rather than surface lexical cues alone. Guidance must track the search state as a structured object, not just the surrounding sentence. Without that, learned scorers remain sensitive to lexical drift and to limited supervision.

Overfitting signal. Training logs for the CustomJava mimic show mean-squared error dropping to 7.7×10^{-4} with trace-alignment accuracy at 1.0 by epoch 2 on a dataset of roughly 185 traces (`experiments/policies/CustomJava_logs/metrics.json`). The same run reports Spearman $\rho = 1.0$ and top-5 accuracy of 1.0 on both train and validation splits (`experiments/policies/summary.json`). Such perfect scores on a small dataset, together with the generalisation failures noted above, indicate that the scorer has effectively memorised the traces rather than learned a robust ordering signal.

7.1.3 Cross-Family Comparison

Contrasting the two designs clarifies their complementary strengths. Symbolic guidance (Design A) derives its robustness from explicit constraints: frames and roles enforce ontological coherence, salience keeps witness selection aligned with discourse prominence, and the policy comparator chooses among already-constrained partial models. Data-driven guidance (Design B) offers a pathway to learn plausibility signals and preferences beyond the engineered policy features, but, as implemented here, remains fragile when distributions shift or when supervision is scarce.

From the perspective of the shared runtime architecture, both designs validate that: (i) tableau + salience as the inner loop is effective across guidance families; and (ii) queue-based frontier ordering is a clean interface for swapping guidance mechanisms. The divergence emerges predominantly in how guidance reacts to lexical or structural perturbations, with Design A absorbing them via constraints and Design B amplifying them via embeddings.

7.1.4 Observed Failures and Stability Gaps

The error patterns documented in Chapter 6 coalesce around five themes:

(1) **Lexical drift**: Small word substitutions (e.g., near-synonyms or domain-adjacent nouns) shifted embedding space sufficiently to reorder branches, even when the underlying logical structure was unchanged.

(2) **Template sensitivity**: The external heuristic exhibited sensitivity to prompt phrasing and context length, leading to inconsistent rankings across near-identical inputs.

(3) **Salience conflicts**: Learned scores occasionally counteracted salience-driven witness selection, producing expansions that ignored discourse-prominence cues critical for anaphora-like choices.

(4) **Role swaps**: Ambiguities involving pronouns (e.g., subject/object swaps) remained a frequent failure mode, with learned scorers oscillating between readings across paraphrases.

(5) **Sparse supervision**: The human-annotated set improved alignment on a narrow slice of the data but lacked coverage to regularise the scorer against the full space of stories/lexical variations used in evaluation.

Together, these point to instability outside the training traces. The scoring modules need either (i) richer supervision that encodes discourse constraints directly, or (ii) architectural biases that make those constraints native to the scoring computation.

7.1.5 Implications for Guided Model Generation

The most immediate implication is pragmatic: for the language fragment and ontology covered here, symbolic policies remain the operational solution. They yield dependable search control with transparent failure modes, while still exposing tunables (drain size, timeouts, threads) for engineering trade-offs.

A second implication concerns hybridisation. Rather than replacing symbolic policies outright, data-driven guidance is better positioned as a constrained re-ranker *after* symbolic pruning has reduced the candidate set. This respects runtime budgets, curtails variance introduced by external heuristics, and limits the exposure of learned scorers to distribution shift. Under this view, stronger salience-aware features and compact, structure-preserving encoders (that read the *tableau state* rather than only text) are likely to be more effective than generic contextual embeddings alone.

Finally, the results reaffirm the value of a clear separation of concerns in the runtime: *(a)* a calculus that enforces sound local expansions; *(b)* salience that steers instantiation; and *(c)* a guidance interface that orders frontier states. Each layer can evolve independently without reinterpreting previous chapters' formal commitments (cf. Chapter 3).

7.1.6 Threats to Validity

We summarise internal and external threats, along with mitigations already applied or recommended.

Internal validity. Synthetic traces dominated supervision for learned scorers; the human-annotated set was small. The external heuristic introduced latency and run-to-run variance. Parallel Java runs (twelve workers) were reported as single trials in the current tables. *Mitigations*: fix seeds and prompt templates; report variance across repeated trials; constrain external heuristic usage to bounded re-ranking; expand human annotations.

External validity. The ontology is static and the language fragment is restricted to short narratives with a controlled vocabulary. Hardware was limited to a single workstation profile. *Mitigations*: broaden ontology coverage and the story set; include more open-domain lexical variation; evaluate on additional hardware profiles; incorporate ablations that vary decay, drain size, and timeout (cf. planned ablations in Chapter 6, Section 6.4).

7.2 Conclusion

7.2.1 Summary of Findings

This thesis presented a guided tableau system for natural-language semantic model generation and evaluated two guidance families over a shared runtime:

- **Design A (Java, symbolic policies)**: A knowledge-graph-grounded engine (situations, frames, roles, semantic types) with salience at the rule level and frontier ordering via policy comparators. The custom policy provided a stable operational baseline across sequential and parallel settings (cf. Table 6.1).
- **Design B (Python, data-driven)**: A projected event-semantics representation with neural cost functions and an external heuristic. Despite promising training loss and occasional correct branch selections, the guidance was not robust under lexical variation and added latency/variance, preventing consistent reproducible gains (cf. Section 7.1.2).

Across both designs, the shared calculus and salience mechanism proved effective; the difference lay in how guidance interacted with lexical and structural variation. Symbolic guidance remained reliable; learned guidance exposed stability gaps.

7.2.2 Hypothesis Revisited

The empirical results support **H1** from Chapter 5: the symbolic Custom policy produced more stable orderings and lower runtime under fixed budgets than learned or external guidance, and rule-level salience contributed to consistent witness selection across thread configurations. The learned/external variants showed promise as re-rankers but did not, in their current form, match the robustness of symbolic control.

7.2.3 Contributions

The work contributes:

(1) **A layered system design** that cleanly separates tableau calculus, salience-driven instantiation, and guidance via a priority-queue interface, enabling interchangeable policy/scoring modules without altering the core reasoning loop.

(2) **A multithreaded Java policy engine** with ontology integration, salience metadata, and configurable tunables (drain size, timeouts, threads), delivering a reproducible symbolic baseline over the benchmark narratives.

(3) **Python prototypes for data-driven guidance** that operate on projected event literals and host neural cost functions and an external heuristic under the same runtime assumptions, thereby documenting the gap between symbolic stability and learned plausibility scoring in this setting.

(4) **Empirical comparison** that documents symbolic robustness, learned-policy instability, and their practical consequences. Reproducibility instructions appear in the appendix.

7.2.4 Limitations

Limitations stem from scope and data: (i) a restricted language fragment and static ontology; (ii) limited human annotations compared to the diversity of lexical forms used in evaluation; (iii) single-machine runtime profiles; and (iv) external-heuristic variability that complicates strict runtime accounting. These choices were deliberate to keep the calculus, ontology, and engineering surface area tractable, but they constrain generality.

7.2.5 Future Work

Future extensions follow four lines:

Structured inductive bias for learned guidance. Augment cost functions with features that directly encode tableau state and salience evolution, rather than relying primarily on surface lexical context. Compact encoders that read *events-as-structures* (and their attachment to discourse entities) are a promising direction.

Constrained hybrid re-ranking. Deploy neural/external scorers as re-rankers over small candidate sets pruned by symbolic policies. This limits variance and bounds latency while allowing learned preferences to influence tie-breaks among near-saturated interpretations.

Broader coverage and supervision. Expand ontology coverage, story templates, and human annotations. Where feasible, incorporate contrastive supervision that addresses role swaps and lexical drift directly, and report variance across repeated trials.

Runtime ablations and reproducibility. Systematically vary decay factors, queue drain size, timeouts, and worker counts. Keep prompt templates and seeds fixed for any external heuristic component and report both average and dispersion statistics. Pointers to environment and commands remain in Appendix A.

Overall, symbolic guidance with salience-aware tableau expansion is the most reliable path for guided model generation in this domain. Learned guidance remains a useful complement, especially as a bounded re-ranker, once its inductive biases and supervision better capture the structure of the search state.

Appendix A

Reproducibility

This appendix explains how to rebuild all experiments reported in Chapter 5 using the two reference implementations in this repository: the Java engine under `design_java/` (Design A) and the Python port under `python_port/` (Design B). The instructions assume the project root `/home/ahmed/projects/masters`; adjust the paths if you cloned the repository elsewhere. This thesis is fully versioned at github.com/A-Mamdouh/masters-thesis.

A.1 Environment

- **Operating system.** Linux or macOS. The thesis results were collected on 6.12.48-1-MANJARO.
- **CPU.** Multi-core CPU recommended. Java runs use up to 12 worker threads.
- **RAM.** 16 GB or more recommended (8 GB is sufficient for Java plus moderate Python experiments).
- **Java toolchain.** OpenJDK 17 or later; `javac` and `java` must be on `PATH`.
- **Python toolchain.** Python 3.11+ with `pip` or `uv`. The port depends on PyTorch, NumPy, and tqdm. Use the supplied `python_port/uv.lock` for deterministic installs via `uv sync`.
- **GPU (optional).** Needed only if you want to accelerate neural policy training.

- **Random seeds.** Java relies on deterministic traversal; Python scripts fix seeds internally (training seed 40, evaluation seed 42, external heuristic seed 42). Set `PYTHONHASHSEED=0` for complete determinism.

A.2 Repository layout

- **Design A:** `design_java/`
  - `Main.java`: entry point for all Java experiments.
  - `modelGeneration/`: priority queue, tableau rules, heuristics.
  - `ontology/` and `stories/`: knowledge graph fragment and benchmark narratives.
  - Logs follow the filename pattern `test_results_<W>_workers_<T>ms.txt` (for example, `test_results_12_workers_200ms.txt`).
- **Design B:** `python_port/`
  - `src/nls_python/model_generation/`: tableau port, policies, neural helpers.
  - `data/`: annotated story sets (train.json, val.json, test.json, java_design_test.json).
  - `experiments/policies/`: saved logs (story_bank_results.txt, annotated_test_results.txt), summaries and checkpoints.
  - `scripts/`: bash helpers for batch experiments.

A.3 Design A (`design_java/`)

Build

```
cd design_java
javac $(find . -name "*.java")
```

Recompile after every code change; incremental compilation is fast (<1 s on the reference machine).

Sequential baseline (1 worker)

```
cd design_java
java Main 1 200 > test_results_1_workers_200ms.txt
```

- Arguments are `<workers>` and `<per-iteration-timeout-ms>`.
- Each log block lists the story sentences and heuristic statistics (time in ms, explored models, success flag) with separators for readability.
- Aggregated numbers in Table 6.1 are the arithmetic mean across the five Java story variants included in the file.

Parallel configuration (12 workers)

```
cd design_java
java Main 12 200 > test_results_12_workers_200ms.txt
```

The output format matches the single-worker run. To replicate the thesis tables, compute per-policy means over the five story variants and report both time and explored-model counts. The Java policy modules are deterministic, so re-running with the same story order yields identical numbers.

A.4 Design B (`python_port/`)

All Python commands below assume you are inside `python_port/`. Install dependencies via `uv sync` or `pip install -e .`.

A.4.1 Story-bank and annotated-set sweeps

The experiment harness mirrors the Java `Main`. It replays narratives, runs the requested policies, and writes aggregated logs.

```
# Story-bank baseline (4 stories, Custom/DFS/BFS)
uv run python -m nls_python.model_generation.story_experiments \
  --datasets data/test.json \
  --base-policy CustomJava \
  --policies DFS BFS \
  --timeout-ms 600 \
  --output experiments/policies/story_bank_results.txt

# Annotated regression set (9 stories)
uv run python -m nls_python.model_generation.story_experiments \
  --datasets data/java_design_test.json data/test.json \
  --base-policy CustomJava \
  --policies DFS BFS NeuralPolicy \
  --timeout-ms 600 \
  --output experiments/policies/annotated_test_results.txt
```

- Each output file contains one block per story with the same `HeuristicStatistics` summary used in Chapter 6.
- The averages quoted in Tables 6.2–6.3 are computed from these files (mean across stories per policy).

A.4.2 Neural policy training and evaluation

The port provides a repeatable pipeline: dataset generation, model training, then experiments. Commands use `uv run` to guarantee the locked dependency set.

Generate supervision traces

```
uv run python -m nls_python.model_generation.training_data \
  --datasets data/train.json data/val.json data/test.json \
  --policies CustomJava \
  --output data/all.jsonl \
  --train-output data/train_traces.jsonl \
  --val-output data/val_traces.jsonl
```

Train the neural cost function

```
uv run python -m nls_python.model_generation.neural_training \
  --dataset data/train_traces.jsonl \
  --val-dataset data/val_traces.jsonl \
  --epochs 10 --lr 3e-4 --device cpu \
  --output checkpoints/neural.pt \
  --log-dir logs/neural_train
```

The script fixes `torch.manual_seed(40)` internally. Use `-device cuda` if you have a GPU.

Evaluate on annotated stories

```
uv run python -m nls_python.model_generation.story_experiments \
  --datasets data/java_design_test.json data/test.json \
  --base-policy CustomJava \
  --policies DFS BFS NeuralPolicy \
  --neural-checkpoint checkpoints/neural.pt \
  --timeout-ms 600 \
  --output experiments/policies/annotated_test_results.txt
```

This overwrites the earlier log with fresh neural numbers. To keep both versions, supply a different `-output`. The agreement metric used in Chapter 6 is recomputed automatically when a neural policy is present.

A.5 Logging and seed management

- **Logging.** Both implementations log individual story blocks followed by aggregate statistics. Keep the raw files (`test_results_*.txt` and `experiments/policies/*.txt`); the Results chapter cites these exact paths.
- **Seeds.** Java relies on deterministic ordering; no additional flags are required. Python scripts set seeds internally, but you can override them via environment variables (`PYTHONHASHSEED`, `TORCH_SEED`) or CLI options (see `python_port/scripts/` for examples).
- **Hardware notes.** All timings in the thesis were collected on a 6-core/12-thread CPU with 32 GB RAM. When comparing against the reported numbers, ensure that the timeout, drain-size, and worker-count flags match those shown above and in Table 5.2.

Following the steps above reproduces every quantitative result in Chapters 5–6, including the neural-policy ablations and the cross-family comparison.

Bibliography

[Ben$^+$20] E. M. Bender and A. Koller. Climbing towards NLU: On meaning, form, and understanding in the age of data. In D. Jurafsky, J. Chai, N. Schluter, and J. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5185–5198, Online. Association for Computational Linguistics, July 2020. DOI: 10.18653/v1/2020.acl-main.463. URL: https://aclanthology.org/2020.acl-main.463 (cited on pp. 2, 10–12).

[Bos$^+$19] A. Bosselut, H. Rashkin, M. Sap, C. Malaviya, A. Celikyilmaz, and Y. Choi. Comet: commonsense transformers for automatic knowledge graph construction. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. DOI: 10.18653/v1/p19-1470 (cited on p. 12).

[Bro$^+$20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165 (cited on p. 4).

[Car11] R. Carnap. *Meaning and necessity: A study in semantics and modal logic*. The University of Chicago Press, 2011 (cited on p. 9).

[Dav67] D. Davidson. The logical form of action sentences. *The Logic of Decision and Action*:81–120, November 1967. DOI: 10.2307/jj.13027259.6 (cited on p. 8).

[Dev$^+$19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: pre-training of deep bidirectional transformers for language understanding, 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (cited on pp. 4, 10).

[Dow91] D. Dowty. Thematic proto-roles and argument selection. *Language*, 67(3):547–619, 1991 (cited on p. 8).

[Elm90] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990 (cited on p. 10).

[Fre79] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete formelsprache des reinen Denkens*. 1879 (cited on p. 8).

[Gro+95] B. J. Grosz, A. K. Joshi, and S. Weinstein. Centering: a framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203–225, 1995. URL: https://aclanthology.org/J95-2003/ (cited on p. 1).

[Hac06] V. Hacquard. PhD thesis, 2006 (cited on p. 9).

[Hir81] G. Hirst. *Anaphora in natural language understanding: A survey*. Springer-Verlag, 1981 (cited on p. 11).

[Hoc+97] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997 (cited on p. 10).

[Ji+22] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu. A survey on knowledge graphs: representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 33(2):494–514, February 2022. DOI: 10.1109/tnnls.2021.3070843 (cited on pp. 2, 3, 7, 9, 10, 12).

[Keh02] A. Kehler. *Coherence, Reference, and the Theory of Grammar*. CSLI Publications, Stanford, CA, 2002. URL: https://web.stanford.edu/group/cslipublications/cslipublications/pdf/1575862166.pdf (cited on p. 1).

[Kra96] A. Kratzer. Severing the external argument from its verb. In J. Rooryck and L. Zaring, editors, *Phrase Structure and the Lexicon*, pages 109–137. Kluwer Academic Publishers, Dordrecht, 1996 (cited on p. 8).

[Lev11] H. Levesque. The winograd schema challenge. In January 2011 (cited on pp. 1, 4, 7, 11).

[Mit14] R. Mitkov. *Anaphora resolution*, February 2014. DOI: 10.4324/9781315840086 (cited on pp. 2, 7, 11).

[Mit22] R. Mitkov. Anaphora resolution. In *The Oxford Handbook of Computational Linguistics*. Oxford University Press, June 2022. ISBN: 9780199573691. DOI: 10.1093/oxfordhb/9780199573691.013.36. eprint: https://academic.oup.com/book/0/chapter/358152193/chapter-pdf/45719936/oxfordhb-9780199573691-e-36.pdf. URL: https://doi.org/10.1093/oxfordhb/9780199573691.013.36 (cited on p. 1).

[Mon73] R. Montague. The proper treatment of quantification in ordinary english. *Approaches to Natural Language*:221–242, 1973. DOI: 10.1007/978-94-010-2506-5_10 (cited on p. 9).

[Par90] T. Parsons. *Events in the Semantics of English: A Study in Subatomic Semantics*. MIT Press, 1990 (cited on pp. 2, 3, 7, 8).

[Sel$^+$19] D. Selsam and N. Bjørner. Guiding high-performance sat solvers with unsat-core predictions. *Lecture Notes in Computer Science*:336–353, 2019. DOI: 10.1007/978-3-030-24258-9_24 (cited on pp. 7, 10, 12).

[Smu95] R. M. Smullyan. *First-order logic*. Dover, 1995 (cited on pp. 2, 8, 12).

[Tar56] A. Tarski. *Logic, Semantics and Metamathematics*. Oxford University Press, London, 1956 (cited on p. 8).

[Vas$^+$23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cited on pp. 2, 7, 10, 12).