JOHANNES KEPLER
UNIVERSITY LINZ

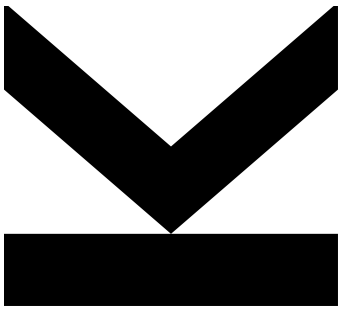
Submitted by
Michael Reinegger

Submitted at
Institute of Networks and Security

Supervisor
Univ.-Prof. Priv.-Doz. DDI Dr.
Stefan Rass

August 2026

Abstracting complex Permission Systems into a Prolog-based Metamodel



Master's Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare under oath, that the submitted thesis has been written solely by me without any third-party assistance. Only the declared sources and/or resources have been used. Sources for all literal, paraphrased and cited quotes have been accurately credited. In particular, I have clearly and unambiguously explained or credited all usage of generative AI.

I am aware that the violation of this regulation will lead to failure of the thesis.

Michael Reinegger
Name


Signature

Kronstorf, August 25, 2026
Place, Date

Acknowledgments

I thank Univ.-Prof. Priv.-Doz. DDI Dr. Stefan Rass, my academic supervisor at JKU, for his detailed feedback across multiple review rounds and for guiding the thesis.

I thank Dynatrace and my internal supervisor, Paul Klingelhuber, for the opportunity to carry out this thesis in a production setting, for access to the systems and data this work builds on, and for his support throughout.

AI tools (Claude by Anthropic, Codex by OpenAI, and GitHub Copilot) assisted with code generation, debugging, data analysis, and text editing throughout this thesis. All AI-assisted output was reviewed, adapted, and verified by me. Appendix E gives the full declaration of AI tool usage.

Abstract

Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) implement access control using distinct identity and access management (IAM) concepts, terminology, and policy formats. Answering cross-provider access questions therefore requires manually evaluating separate configuration-as-code sources and vendor-specific representations.

To address this heterogeneity, this thesis introduces a shared 18-predicate schema and common query interface that represents identity, permission, and boundary concepts across all three cloud providers. An automated Go gathering pipeline populates this schema from configuration-as-code files and provider catalog metadata. SWI-Prolog provides the reference implementation used for semantic comparison, while an in-memory Go engine (GoStore) serves as the production query layer after query-by-query validation against the Prolog baseline. Performance evaluation shows that GoStore achieves median speedups of 3.20× cold and 3.07× warm relative to the matched Prolog serving path (in-process Go execution versus an external Prolog subprocess reached over IPC) in the catalog-wide evaluation. Beyond the Web Explorer and web API, the metamodel is exposed through a Model Context Protocol (MCP) server that provides language-model agents with typed, read-only access to structured queries. In an exploratory 18-prompt system evaluation run 5 times per condition (90 scored outputs per condition), where we performed correctness scoring without independent review, the Model Context Protocol (MCP) condition received full credit on 71/90 outputs compared with 69/90 for direct access (total cost $16.03 versus $22.68), reflecting an aggregate system outcome rather than an isolated effect of the protocol or model reasoning.

The metamodel covers the shared permission and organisational concepts represented in configuration-as-code. Provider-specific enforcement mechanisms such as ABAC conditions and Service Control Policies, along with configuration missing from the analysed repositories, lie outside the investigation scope. Within these boundaries, effective-permission query results reflect each provider's negative-permission model as gathered. Amazon Web Services (AWS) uses explicit deny facts at both the role and assignment level. Azure resolves `notActions` subtraction at gather time and stores only allow-effect facts. Google Cloud Platform (GCP) provides allow-only evaluation because hierarchy-scoped deny policies are not collected by the gathering pipeline. The thesis demonstrates that multi-cloud Identity and Access Management (IAM) data can be normalised into a single, automatically populated, and queryable model that covers the gathered permission state for all three providers.

Zusammenfassung

Amazon Web Services (AWS), Microsoft Azure und die Google Cloud Platform (GCP) verwenden für ihr Identity and Access Management (IAM) jeweils eigene Konzepte, Begriffe und Richtlinienformate. Die Beantwortung anbieterübergreifender Zugriffsfragen erfordert daher die manuelle Analyse separater Configuration-as-Code-Quellen und anbieterspezifischer Darstellungen.

Um dieser Heterogenität zu begegnen, führt diese Arbeit ein einheitliches Metamodell mit 18 Prädikaten ein, das Identitäten, Berechtigungen und Ressourcengrenzen über alle drei Cloud-Anbieter hinweg abbildet. Eine automatisierte Erfassungspipeline in Go befüllt dieses Schema aus Configuration-as-Code-Dateien und Metadaten der Anbieterkataloge. SWI-Prolog dient dabei als Referenzimplementierung für den semantischen Vergleich, während eine In-Memory-Engine in Go (GoStore) nach der Validierung jeder Abfrage gegen die Prolog-Baseline als Abfrageschicht für den Produktivbetrieb eingesetzt wird. Die katalogweite Leistungsevaluierung ergab für GoStore gegenüber dem abgeglichenen Prolog-Abfragepfad mediane Beschleunigungen von 3,20× im kalten und 3,07× im warmen Durchlauf. Neben dem Web-Explorer und einer Web-API wird das Metamodell über einen Server für das Model Context Protocol (MCP) bereitgestellt, der Agenten auf Basis von Sprachmodellen über typisierte, schreibgeschützte Werkzeuge Zugriff auf strukturierte Abfragen gibt. In einer vergleichenden Systemevaluierung anhand von 18 Prompts, die je Bedingung 5 Mal wiederholt wurde (90 bewertete Ausgaben je Bedingung), bei der die Korrektheit ohne unabhängige Überprüfung durch denselben Autor bewertet wurde, erhielt die MCP-Variante bei 71/90 Ausgaben die volle Punktzahl gegenüber 69/90 beim direkten Zugriff (16,03 $ gegenüber 22,68 $), was ein Gesamtsystemergebnis widerspiegelt und keine isolierte Aussage über den Einfluss des Protokolls oder das Verhalten des Modells erlaubt.

Das Metamodell deckt die in Configuration-as-Code abgebildeten gemeinsamen Berechtigungs- und Organisationskonzepte ab. Anbieterspezifische Durchsetzungsmechanismen wie ABAC-Bedingungen und Service Control Policies sowie in den analysierten Repositories fehlende Konfigurationen liegen außerhalb des Untersuchungsumfangs. Innerhalb dieser Grenzen spiegeln Berechtigungsabfragen das Deny-Modell des jeweiligen Anbieters wider. AWS verwendet explizite Deny-Fakten auf Rollen- und Zuweisungsebene. Azure löst `notActions`-Einschränkungen zum Importzeitpunkt auf und speichert ausschließlich Allow-Fakten. Bei GCP erfolgt eine reine Allow-Auswertung, da hierarchiebezogene Deny-Policies von der Erfassungspipeline nicht exportiert werden. Die Arbeit zeigt, dass sich Multi-Cloud-IAM-Daten in ein gemeinsames, automatisch befülltes und abfragbares Modell normalisieren lassen, das den erfassten Berechtigungszustand aller drei Anbieter abdeckt.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Proposed Solution . . . 4
1.3 Evolution of the Implementation . . . 5
1.4 Why Prolog for the Research Baseline? . . . 6
1.5 Scope and Limitations . . . 6

2 Fundamentals **9**
2.1 Cloud Permission Systems . . . 9
2.2 Configuration as Code . . . 11
2.3 Prolog . . . 11
2.4 Data Exchange and Serialisation . . . 13
2.5 Related Work . . . 14
2.6 Fundamentals Summary . . . 18

3 Design of a Generic Metamodel **19**
3.1 Configuration as Code (CasC) Analysis . . . 19
3.2 Mapping to Prolog Predicates . . . 21
3.3 Extended Metamodel Predicates . . . 27
3.4 Query Design and Implementation . . . 32

4 Automated Metamodel Generation **39**
4.1 Two-Path Gathering . . . 39
4.2 Permission Catalog Aggregation . . . 44
4.3 Loading and Caching . . . 45
4.4 Data Anonymisation . . . 47
4.5 Edge Cases and Special Handling . . . 48
4.6 Go–Prolog Communication: The Machine Query Interface Protocol . . . 49
4.7 Performance Optimisations . . . 50
4.8 From Prolog Engine to GoStore: A Full Reimplementation . . . 50

5 Metamodel Application and Querying **61**
5.1 Advanced Query Patterns . . . 61
5.2 Go Web Application . . . 64
5.3 Natural Language Interface . . . 66
5.4 Explorer UI . . . 71
5.5 Agentic Integration (Model Context Protocol, MCP) . . . 74

6 Evaluation **84**
6.1 Evaluation Questions . . . 84
6.2 Dataset and Test Fixtures . . . 84
6.3 Correctness Evaluation . . . 85
6.4 Performance Evaluation . . . 87
6.5 Interface Evaluation . . . 92
6.6 Exploratory LLM Workflow Observation: Direct CasC Access vs. MCP . . . 94
6.7 Threats to Validity . . . 96

7 Conclusion and Future Work **98**
7.1 Answers to the Evaluation Questions . . . 98

7.2 Summary of Contributions 99
7.3 Engineering Mistakes and Lessons Learned 100
7.4 Limitations 102
7.5 Extensions of the Current System 103
7.6 Broader Applications of the Metamodel Approach 103

References **105**

A Acronyms **109**

B Code Listings **111**

C Artefact and Reproducibility Record **115**

D LLM Workflow Evaluation Prompt Set **117**

E Declaration of AI Tool Usage **119**

1 Introduction

Hybrid and multi-cloud strategies are widespread in modern large-scale software systems [1]. The tools and literature reviewed for this thesis did not reveal an open system that answers *who can do what?* across Amazon Web Services (AWS), Azure, and Google Cloud Platform (GCP) at the level of individual permissions. Per-provider analysers and commercial entitlement dashboards cover parts of this problem. No cited system combines one queryable model across all three providers, Configuration as Code (CasC) population with file-level provenance, and query surfaces for both human operators and Large Language Model (LLM) agents (Section 2.5). Asset-graph tools such as Cartography [2] and commercial Cloud Infrastructure Entitlement Management (CIEM) dashboards go broader across live infrastructure but do not expose this combination.

This thesis introduces a 18-predicate common schema and query interface (named relations over ground facts, 7 core and 11 extended) that represents the core permission primitives of all three providers. A Go service stack operationalises the model so that a user, a script, or an agent backed by an LLM can submit the same structured catalog queries through different interfaces. Provider-specific enforcement mechanisms remain out of scope (Section 2.1), and the engine comparison documents the remaining result-shape differences rather than assuming complete item-level equivalence.

The main obstacle is that each cloud vendor implements its own Identity and Access Management (IAM) model with different terminology and structures, while sharing the same core primitives: identities, roles, permissions, and assignment scopes. The bullets below show how those shared primitives surface differently on each platform:

- **Azure** grants subscription-scoped roles assembled from `actions`, `notActions`, `dataActions`, and `notDataActions` [3].
- **GCP** hands out roles bound to projects, folders, or organisations, with permissions written as `service.resource.verb` [4].
- **AWS** attaches policy documents to identities, scoped by `service:Action`, with deny statements that take precedence over allows [5].

A widely used concept to configure permissions across cloud platforms is CasC. It puts these definitions in a Git repo, but it does not unify them: the conceptual fragmentation survives the move into version control [6], [7].

1.1 Problem Statement

To evaluate a multi-cloud permission system at scale, this thesis analyses a production-derived anonymised corpus containing 14 505 platform-specific identity records, 1 066 044 expanded permissions, and 857 resource boundaries across three major cloud providers (see Section 4.3). Consider a concrete security audit over this corpus: identifying every identity capable of deleting production virtual machines across all recorded environments. Determining this requires inspecting each provider-specific data source independently, expanding wildcards according to vendor-specific logic, resolving distinct permission inheritance structures, reconciling divergent permission models, and aggregating the resulting identity sets. Within these totals, identity records represent individual platform principals, such as IAM users, groups, or service accounts, counted separately per platform rather than deduplicated across providers.

This thesis is a build-and-validate contribution rather than a hypothesis test. The central research question asks: *can the IAM permission state of AWS, Azure, and GCP, as declared in CasC repositories and*

provider action catalogs, be represented through a shared schema and common query interface that supports wildcard expansion, deny-precedence resolution, provenance tracing, and cross-provider enumeration at production scale? The evaluation answers this question affirmatively for the modelled scope and production-derived corpus, which excludes AWS Service Control Policies (SCPs) (Section 2.1.3), Azure Deny Assignments, and GCP hierarchy-scoped deny policies (Section 1.5), so permission answers reflect role-based grants without these constraints.

To address this question, the system implements an 18-predicate metamodel, first as a Prolog baseline and subsequently as GoStore, the native Go production query engine (Section 4.8). This metamodel is populated automatically from production-scale CasC configurations and provider catalogs (Section 4). Current access is provided through a visual Explorer, a Representational State Transfer (REST) Application Programming Interface (API), and a Model Context Protocol (MCP) server for LLM agents, while an earlier deterministic natural-language processor is retained and tested only within the frozen Prolog baseline. Section 6 evaluates the engines and interfaces against the anonymised corpus. Section 6.6 compares the same LLM using the MCP server with the same model reading the CasC files directly.

Figure 1.1 shows the high-level overview of the system and provides a roadmap for Sections 3 to 5. The CasC repositories and provider catalogs are processed by a Go gathering pipeline, loading the unified data into the query engines that expose metamodel queries across the interfaces.

We identify the following challenges for implementing a unified multi-cloud IAM metamodel:

- **Different terminology and structures.** Providers use distinct names and structures for resource boundaries (such as AWS Accounts, Azure Subscriptions, and GCP Projects), role definitions (such as AWS IAM Policies, Azure Role Definitions, and GCP IAM Roles), and assignments (such as AWS Policy Attachments, Azure Role Assignments, and GCP IAM Bindings). Although these objects appear equivalent, their underlying semantics differ.
- **No provider-native cross-platform queries.** Without a unified query layer, evaluating a cross-cloud question using provider-native tools requires running separate platform-specific scripts and manually merging the results.
- **Wildcard expansion.** A wildcard such as `s3:*` is not a single permission, but rather expands against AWS's full action catalog of approximately 26 000 actions, while Azure's `Microsoft.Compute/*/read` expands against approximately 24 000 actions. The action counts change over time, with the specific catalog sizes evaluated in this thesis listed in Table 4.2 (Section 4.3). Determining what a wildcard permits requires expanding the pattern and accounting for explicit-deny statements, making the effective permission set distinct from a simple union of allowed actions.
- **Group and role inheritance.** Real assignments may not target a user directly, instead flowing through Single Sign-On (SSO) groups, directory groups, and nested role bindings depending on the platform. Determining effective permissions therefore requires traversing these group and role relationships rather than inspecting an isolated assignment record.
- **Navigating the CasC side itself.** Even when the CasC repository serves as the source of truth, resolving permissions requires following directory structures, module boundaries, and templated role definitions. Comparing the repository definition with the live infrastructure state is a separate problem and is outside the scope of this thesis.

This thesis does not attempt to detect runtime drift between CasC declarations and live cloud state (Section 2.2).

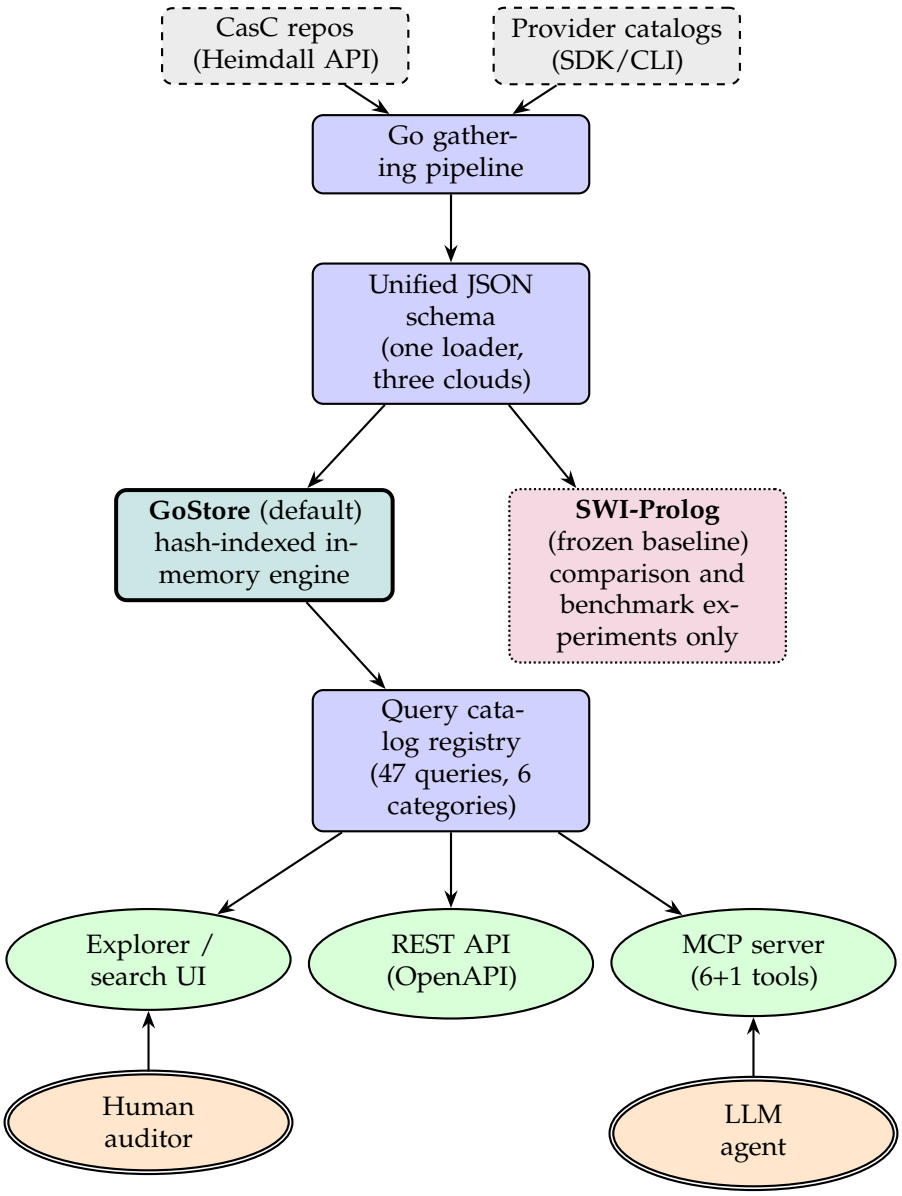


Figure 1.1: High-level structured-query architecture. CasC repositories (accessed via Heimdall, an internal CasC-parsing service) and provider action catalogs feed into a unified JavaScript Object Notation (JSON) schema. GoStore serves as the production query engine for the structured Explorer, REST, and MCP interfaces, while the Prolog baseline is loaded separately for parity and performance experiments.

Three concrete scenarios illustrate the manual cost this thesis helps to reduce. Each is named after the corresponding query in the catalog (Section 5):

- **`can_do`:** *Can identity `X` perform action `Y`?* Answering this manually requires resolving direct and group-derived role assignments, expanding action wildcards against vendor catalogs, and verifying that no explicit-deny rules supersede the permission. The system answers this in a single structured query by evaluating the identity's normalised assignments, matching the requested action against the pre-expanded permission catalog, and applying the model's explicit-deny precedence.
- **`explain_access`:** *Why does identity `X` have access to boundary `B`?* Manually, this requires tracing every applicable assignment and locating its corresponding CasC source record. For a given identity and boundary, the query instead returns all matching role assignments alongside pre-indexed provenance details for direct, inherited, or group-derived access and exposes the relevant CasC source-file metadata.
- **`trust_chain` and `production_identities`:** *Which identities can reach production through an assumed-role hop?* Investigating this scenario requires combining one cross-platform production-assignment query with one AWS trust-path query, as the system does not automatically join them into a single result. `production_identities` scans assignments to production boundaries across selected platforms, whereas `trust_chain` evaluates AWS role trust relationships. Azure and GCP do not expose equivalent cross-account delegation relationships through the organisation's CasC files, so this query is AWS-only within the evaluated corpus. For role traversal, GoStore calculates bounded, cycle-safe inbound and outbound paths, whereas the base Prolog predicate `trust_chain/3` returns only direct trusted principals without a hop count argument.

1.2 Proposed Solution

The primary contribution of this work is a 18-predicate IAM schema and common query interface. An automated gathering pipeline, a Prolog reference implementation, the native GoStore production engine, a query catalog, and supporting interfaces operationalise and evaluate this model. Concretely, the architecture comprises:

- A **18-predicate metamodel** consisting of 7 core and 11 extended conceptual predicates spanning actions, role definitions, permissions, assignments, identities, boundaries, and provenance (Section 3).
- A **Go data-gathering pipeline** and **unified JSON schema** that populate GoStore, SQLStore, and the retained Prolog implementation from CasC-derived Heimdall [8] (an internal service that parses the organisation's CasC repositories and exposes a unified JSON export) data and provider Software Development Kit (SDK)/Command Line Interface (CLI) catalogs.
- A **structured query catalog**, **Explorer web UI**, and **MCP server** exposing the same query surface to human operators and LLM agents.
- A **five-stage Natural Language Processing (NLP) pipeline**, retained as part of the frozen Prolog baseline.

Component-level detail is given in Section 1.5.

The Explorer web UI, REST API, and MCP server expose structured catalog queries from a single shared catalog registry, dispatching those calls directly to the same in-process GoStore instance. Each interface applies its own validation, pagination defaults, and output formatting. SQLStore

raw queries use a separate execution path. The shared GoStore dispatch keeps structured query execution consistent, but does not make the rendered responses identical.

1.3 Evolution of the Implementation

The implementation developed in four stages, each of which changed how data was loaded, queried, and served. Each transition addressed a limitation of the previous stage and helps explain the design decisions behind the current architecture.

1.3.1 Stage 1: Hand-Written Prolog Facts

Initial development validated the six-entity schema on a single AWS test account using manually written `.pl` files containing facts such as `action/2`, `role_definition/4`, and `assignment/4`. While this early prototype proved that the underlying model could resolve core audit questions, the lack of an external loader and the reliance on manual fact maintenance prevented the approach from scaling beyond a small test environment.

1.3.2 Stage 2: Python Loader

To replace hand-maintained facts, a Python script parsed CasC JSON exports, constructed Prolog term strings, and piped them into a SWI-Prolog process using standard input. While this introduced automated external loading, the loader architecture remained provider-specific. Supporting an additional provider required constructing a dedicated loader path alongside structural schema changes because the initial schema relied on AWS-specific predicates. As the dataset grew, cold-start time also increased, exposing performance bottlenecks in early prototypes.

1.3.3 Stage 3: Go Gatherer with Machine Query Interface (MQI) Pool

We replaced the Python loader with a concurrent Go gathering pipeline that runs each provider in its own goroutine to normalise data into a unified JSON representation. The engine loaded these facts over SWI-Prolog's MQI interface [9]. A `fastrw` binary cache avoided repeated JSON parsing and normalisation on restart, although fact assertion into Prolog's dynamic database still occurred on every load.

1.3.4 Stage 4: GoStore

The overhead of MQI and the operational complexity of managing a separate SWI-Prolog process motivated the development of GoStore, a native Go in-memory store. While retaining the existing loader schema, catalog metadata, and structured interfaces, production structured queries moved entirely to GoStore. The natural-language route remained frozen alongside the Prolog baseline, and sandboxed raw queries moved to an in-process, SQLite-backed SQLStore. SWI-Prolog now runs only for comparison and benchmark experiments. The empirical comparison between the two engines forms a thesis contribution (Section 4.8). During early GoStore development, an unoptimised permission-count index reached 8.1 GB Resident Set Size (RSS) on Elastic Container Service (ECS) Fargate before a bitset implementation reduced the footprint to approximately 1 GB. This historical intra-Go defect is detailed in Section 7.3 and is distinct from the controlled Prolog-versus-GoStore comparison.

1.4 Why Prolog for the Research Baseline?

Prolog was selected for the original metamodel and research baseline because facts, rules, unification, and recursion mapped naturally to IAM relationships, including group membership and trust chains, and custom predicates handled wildcard matching and explicit-deny precedence. However, the declarative baseline remains central to verifying metamodel behaviour through parity testing. The alternatives below provide a qualitative design comparison across candidate query technologies rather than a controlled benchmark of every language or framework.

- **General-purpose imperative languages (Go):** Imperative languages require hand-built indexes, explicit traversal loops, and manual matching logic in place of declarative rules and native backtracking. GoStore demonstrates this trade-off directly as the production engine evaluated in Section 4.8.
- **Relational databases and Structured Query Language (SQL):** SQL [10] handles joins and set filtering well, but recursive trust traversal and nested group membership require verbose recursive Common Table Expressions (CTEs) or secondary application code. SQLStore serves as the raw-query backend for cross-engine comparison (Section 4.8).
- **Answer Set Programming (ASP):** ASP [11] suits finite, closed-world policy verification, but its grounding phase introduces preprocessing overhead that is poorly suited to interactive relational enumeration over large action catalogs.
- **Constraint solvers (Satisfiability Modulo Theories (SMT)):** SMT engines [12] evaluate boolean satisfiability, but returning complete identity or provenance lists requires repeated model enumeration rather than direct unification and backtracking.
- **Graph databases (Neo4j):** Graph stores align naturally with the node-and-edge structure of IAM relationships [13], but evaluating explicit-deny precedence across providers requires additional schema design and query logic, and deploying a graph database introduces a separate service to operate.
- **Datalog:** Datalog [14] offers guaranteed query termination and stratified negation [15], but pure Datalog lacks the string operations needed for wildcard expansion, has no Definite Clause Grammar (DCG) equivalent for the NLP interface, and the research phase used SWI-Prolog's MQI protocol for subprocess integration.

Prolog provided a suitable research baseline because its facts-and-rules model supported an executable, declarative representation of access relationships. The central engineering challenge lay not in transcribing these core relations, but in defining consistent cross-provider semantics for normalisation, wildcard expansion, explicit-deny precedence, group hierarchy, and trust paths. The subsequent GoStore and SQLStore engines demonstrate that the underlying metamodel remains independent of any single query technology.

1.5 Scope and Limitations

1.5.1 In Scope (Implemented)

The implemented system consists of eight main components:

- **18-predicate metamodel:** A declarative schema covering AWS, Azure, and GCP IAM. The system supports derived query capabilities across its engines, including role enumeration, effective permission resolution, redundant assignment detection, and privilege-escalation analysis, with bounded trust-path expansion provided specifically by GoStore.

- **Go gathering pipeline:** A collection pipeline combining CasC-derived organisational state from Heimdall with provider action catalogs and role definitions gathered through cloud SDKs and CLI fallbacks.
- **Binary term caching:** An implementation using SWI-Prolog's `fastrw` library [16] to accelerate Prolog baseline loading. Internal documentation estimates an 8–10× speedup over re-parsing raw JSON files (Section 2.4.2).
- **Five-stage NLP pipeline:** A baseline query interface retained as part of the frozen Prolog system, featuring 40 built-in query clauses, a plugin dispatcher, a four-tier matcher, and a full-text search fallback for free-form keyword queries.
- **Go Hypertext Transfer Protocol (HTTP) backend service:** A web server delivering REST API endpoints, streaming search results via Server-Sent Events (SSE), and hosting static frontend assets.
- **Explorer web UI:** A vanilla-JavaScript web interface driven by the structured query catalog (47 queries across six categories), providing interactive search forms and displaying CasC source-file provenance.
- **MCP server interface:** An integration exposing 46 of the 47 catalog queries to LLM agents through six category-grouped tools, supplemented by a seventh sandboxed SQLStore raw-query tool using its own schema. The remaining query, `list_actions`, streams a full provider action catalog and is available in the Explorer only. Structured query results use pipe-separated output formatting and may include per-query `NEXT:` hint footers.
- **Relationships and provenance:** Modelling of trust relationships (e.g. AWS assume-role chains), team memberships, boundary hierarchies, and assignment provenance back to CasC source files.

1.5.2 Out of Scope and Known Limitations

The current system does not implement the following operational features and policy semantics:

- **Data ingestion and live drift:** The deployed Go server reloads Heimdall export files every 60 minutes rather than responding to real-time change events (Section 2.2).
- **AWS policy scope:** Organisation-level SCPs are not modelled (Section 2.1.3), and resource-based policies are not evaluated as effective policy sources.
- **GCP deny semantics:** GCP hierarchy-level deny policies are not gathered by the ingestion pipeline.
- **Conditional access and Attribute-Based Access Control (ABAC):** Dynamic ABAC [17] evaluation and request-context matching are not implemented (Section 2.1).
- **Azure hierarchy flattening:** Azure management-group hierarchies are flattened to the subscription level rather than evaluated recursively.
- **Input validation:** While the ingestion pipeline performs required syntactic parsing, it does not verify the semantic correctness or internal consistency of otherwise readable CasC/Heimdall data or provider catalogs.
- **NLP evaluation boundaries:** The frozen NLP baseline interface has not been systematically evaluated against adversarial phrasing or out-of-vocabulary queries.

All dataset entries, source paths, and identity records evaluated and presented in this thesis were anonymised to maintain organisational confidentiality (Section 4.4). Section 7.5 discusses the main extension directions for addressing these boundaries in future work.

1.5.3 Structure of This Thesis

Chapter 2 introduces the four technical foundations assumed by later chapters and presents the related-work comparison. Chapter 3 defines the unified metamodel schema and its query semantics. Chapter 4 details the data gathering pipeline, loading process, and the migration from Prolog to GoStore, along with the SQLStore comparison and raw-query backend. Chapter 5 describes the application interfaces built on top of the metamodel, covering the retained NLP baseline alongside the production Explorer web UI, REST API, and MCP server. Chapter 6 evaluates system correctness, query performance, and the controlled LLM comparison. Finally, Chapter 7 summarises the findings, reviews limitations, and outlines future extension directions.

2 Fundamentals

This chapter introduces the concepts required to model and evaluate cross-provider access control: cloud IAM structures across AWS, Azure, and GCP, and CasC as the source of intended configuration. It also outlines the Prolog concepts used in the research baseline, alongside the data serialisation and caching mechanisms that support that baseline and inform the GoStore migration.

2.1 Cloud Permission Systems

AWS, Azure, and GCP use distinct terminology and permission structures across their IAM systems. While all three incorporate recurring role-based access control patterns connecting principals, roles or policies, and permissions within specified resource boundaries [18], [19], they differ in resource scope, permission representation, the use of wildcard patterns, and the handling of role exclusions and explicit deny rules.

2.1.1 Microsoft Azure

Azure structures resources across a hierarchy of Azure AD Tenant (Directory) → Management Group → Subscription → Resource Group → Resource. The metamodel adopts the Subscription as Azure's canonical resource boundary and does not evaluate management-group inheritance, leaving management-group hierarchy support as future work (Section 7).

Azure security principals include users, groups, service principals, and managed identities. A role assignment connects a principal to a role definition at a specific scope [3].

Azure action names use a provider/resource/operation-style path, such as `Microsoft.Compute/virtualMachines/start/action`. A role definition separates control-plane permissions into `actions` and `notActions` and data-plane permissions into `dataActions` and `notDataActions` [3]. During gathering, wildcards in the control-plane arrays are expanded only against control-plane operations, while wildcards in the data-plane arrays are expanded only against data-plane operations. The gatherer then subtracts `notActions` and `notDataActions` from their corresponding positive sets as exclusions within the role definition rather than explicit deny rules. Azure also supports Deny assignments as a separate mechanism that blocks access to resources independently of role assignments. Deny assignments are not collected by the gathering pipeline and are therefore excluded from the evaluated effective-permission model, in the same way that GCP hierarchy-scoped deny policies and AWS Service Control Policies are excluded (Section 1.5).

2.1.2 Google Cloud Platform

GCP structures resources across a hierarchy of Organization → Folder → Project → Resource. The implemented model represents Google accounts (users), service accounts, and groups as security principals. Permissions follow a `service.resource.verb` pattern, such as `storage.buckets.create` [4].

Unlike Azure role definitions and AWS policies, GCP IAM roles do not use wildcard action patterns and instead enumerate permissions explicitly. GCP supports deny policies as separate resources attached at the organisation, folder, or project level, rather than as fields within role bindings [4]. Because the gatherer in this work ingests role definitions and allow bindings but excludes these deny-policy resources, GCP deny evaluation remains outside the implemented effective-permission model. GCP provides basic, predefined, and custom roles [4]. Because permissions are enumerated explicitly, identifying a predefined role that grants a required permission set becomes a catalog-search problem.

2.1.3 Amazon Web Services

AWS Organizations can group AWS accounts, with the individual account serving as the canonical AWS resource boundary in this work. AWS IAM identities include users, groups, and assumable roles, although groups organise users rather than acting as principals [5]. Managed policies may be attached to multiple users, groups, or roles, while an inline policy is embedded in one such identity. The metamodel maps both policy forms to its generic role-definition concept.

Actions follow a `service:Action` pattern, such as `s3:AbortMultipartUpload`, and policy statements may include wildcard patterns such as `s3:*`. Explicit deny statements take precedence over matching allows [5]. Within the implemented action-level model, a matching explicit deny for `s3:DeleteBucket` therefore blocks that action even when another loaded policy allows `s3:*`.

The core cross-provider model reasons over resource boundaries, identities, role definitions, and actions. Provider-specific resource constraints and condition statements can be retained as metadata annotations, but they are not evaluated as part of effective-permission decisions. Conditional grants are therefore treated as unconditional. Organisation-level SCPs and resource-based policies remain outside the current effective-permission computation and are identified as future work (Section 7).

2.1.4 Comparison and Common Patterns

Despite the surface differences, the same six core concepts recur across all three providers. Table 3.1 in Chapter 3 maps each concept to its provider-native term and to the corresponding metamodel predicate, and Chapter 3 defines the unified metamodel built on top of them.

The three platforms diverge across resource boundaries, role terminology, and evaluation rules. The metamodel treats Azure Subscriptions, GCP Projects, and AWS Accounts as canonical resource boundaries and unifies Azure role definitions, GCP IAM roles, and AWS policies under a shared role abstraction (Figure 2.1). During gathering, the pipeline expands AWS and Azure wildcard patterns against provider action catalogs. GCP roles instead list permissions explicitly. The providers also differ in how they represent negative permissions: AWS explicit denies override matching allows, Azure `notActions` and `notDataActions` serve as exclusions local to a role definition, and GCP deny policies exist as separate hierarchy-scoped resources. The implemented model therefore resolves Azure exclusions during gathering, applies a deny-override rule to explicit AWS denies, and leaves GCP deny policies that the gatherer does not ingest outside the effective-permission computation.

Figure 2.1 aligns the three resource hierarchies visually. The highlighted level in each column is the boundary the metamodel treats as canonical.

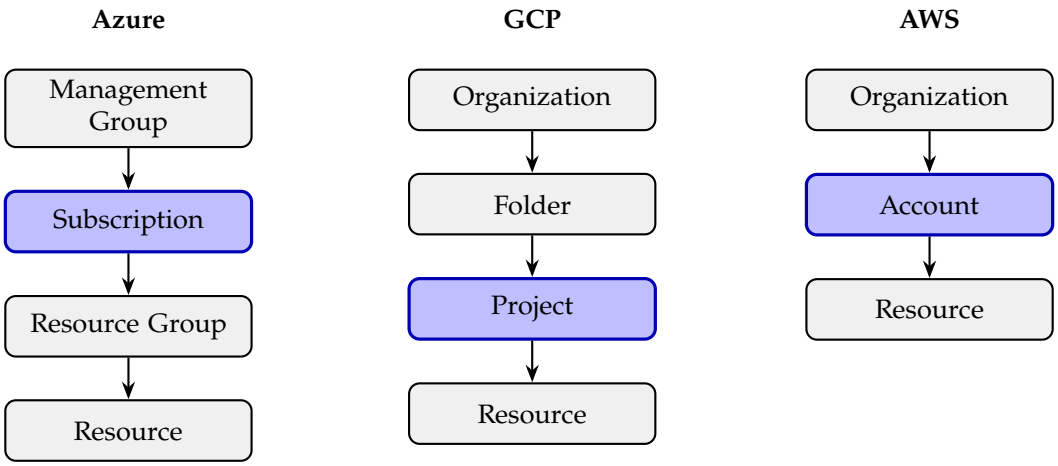


Figure 2.1: The three providers' resource hierarchies and the canonical resource boundaries used by the metamodel. The highlighted levels correspond to `resource_boundary/4`, where `/4` denotes the number of predicate arguments (Section 2.3 explains the `name/arity` notation). Other hierarchy levels provide context but do not affect the current effective-permission computation.

2.2 Configuration as Code

CasC is the practice of storing intended infrastructure and access-control configuration in version-controlled files, such as YAML Ain't Markup Language (YAML), JSON, Terraform HCL, or Bicep, rather than configuring environment state manually through provider consoles. Expressing setup declaratively provides reviewable changes, version history, automated validation, and reproducible deployments [6], [7]. Listing 4.1 (Section 4) illustrates a normalised Heimdall-exported assignment from such a configuration, which contains a `source` object recording its originating file.

In this work, CasC provides the intended organisational IAM state rather than the live enforced cloud state, with the Heimdall API parsing the organisation's Bitbucket-hosted YAML and JSON files across AWS, Azure, and GCP [8]. Heimdall exposes the resulting identities, assignments, boundaries, teams, and trust relationships through a unified JSON feed that includes provenance links from assignments and entity definitions to their originating files. Although these provenance records preserve file-level traceability, the evaluated dataset does not include source line numbers or a usable commit identifier.

Provider SDKs and CLIs supplement the CasC-derived organisational data with provider action and role catalogs, but they do not gather live identities, assignments, or enforced permission state from the organisation's cloud environments. The system therefore cannot compare intended CasC state with deployed state or detect drift, establishing the scope boundary described in Section 1.1.

2.3 Prolog

Prolog is a declarative logic-programming language: instead of writing *how* to compute an answer, the programmer writes *what is true* as facts and *what follows* as rules, and the runtime infers the rest by unification and backtracking [20]. Predicates are identified by name and arity (number of arguments), written as `name/N`. For example, `permission/4` names the four-argument predicate `permission`. This thesis uses the same `name/arity` notation whenever a predicate is referenced outside a code listing.

Listing 2.1 provides a minimal example of the metamodel representation, defining the role definition `reader`, identity `alice`, resource boundary `prod_env`, and provider tag `aws`. Written with initial

lower-case letters, these four constants are Prolog atoms rather than variables.

The `can_do/3` rule combines role assignments with permissions to enumerate actions associated with an identity's assigned roles. Capitalised identifiers are Prolog variables. The leading underscore on `_Boundary` marks it as intentionally unused, and parameterising `Platform` lets the same rule operate across AWS, Azure, and GCP.

Listing 2.1: Example Prolog facts and an illustrative rule joining role assignments to allow permissions.

```
role_definition(reader, 'Reader Role', builtin, aws).
permission(reader, 's3:GetObject', allow, aws).
assignment(alice, prod_env, reader, aws).

can_do(User, Action, Platform) :-
    assignment(User, _Boundary, Role, Platform),
    permission(Role, Action, allow, Platform).
```

In the query `?- can_do(alice, A, aws).`, `A` is an unbound query variable that receives each matching action connected to `alice` through at least one AWS role assignment and an allow permission. Because `_Boundary` is unconstrained in the rule, matches can originate from any AWS boundary on which `alice` has a role assignment. Prolog enumerates every matching combination of role assignment and permission by backtracking through the fact base.

Listing 2.2 presents the example query in an interactive session and shows how the engine uses backtracking to produce further solutions.

Listing 2.2: An interactive SWI-Prolog session over the example facts. Each press of `;` asks the engine to backtrack and produce the next solution. `false` signals that no further solutions exist. Captured 2026-07-06 from a minimal `swipl` session with the three facts defined above.

```
?- can_do(alice, Action, aws).
Action = 's3:GetObject' ;
false.

?- assignment(alice, Boundary, Role, Platform).
Boundary = prod_env, Role = reader, Platform = aws.

?- can_do(alice, 'iam:CreateUser', aws).
false.
```

Capitalised names entered at a `?-` prompt are fresh variables local to that query, remaining distinct from identically named variables in the earlier rule.

Figure 2.2 traces how the Prolog baseline answers `can_do(alice, A, aws)` through unification and backtracking.

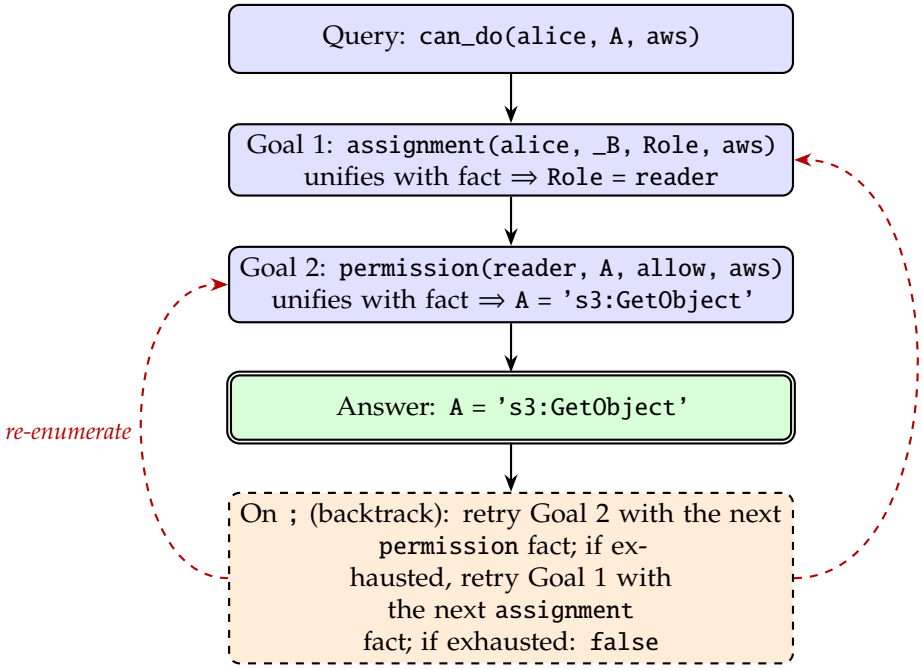


Figure 2.2: Unification and backtracking for `can_do(alice, A, aws)`. The rule body is a conjunction of two goals. The Prolog engine unifies each goal with matching facts, binding variables as it goes, and backtracking (dashed) re-enumerates alternatives until every combination is exhausted.

Two SWI-Prolog-specific features and one core Prolog mechanism are used in later chapters of this work:

- **Dynamic and multifile predicates.** Provider facts are loaded at runtime into a single logical model. The `:- dynamic` directive permits runtime changes to stored facts, while `:- multifile` allows predicate definitions to span multiple source files.
- **DCGs.** Prolog's built-in grammar formalism [21] is used in the retained Prolog NLP pipeline to map structured input phrases to logical query terms. For example, a rule combining `who_can_phrase`, `action_verb(A)`, `on_phrase`, and `resource_name(R)` parses input such as "who can read the production bucket" into a structured query term.
- **Unification and backtracking.** Unification matches goals with facts, while backtracking enumerates alternative matches. Wildcard expansion backtracks over catalog action facts and applies recursive character-level matching to their names.

These three mechanisms are the Prolog-specific foundation on which the metamodel's fact base, query rules, and NLP pipeline are built.

2.4 Data Exchange and Serialisation

The gathering pipeline produces a unified JSON representation consumed by GoStore, the production execution engine, and by the Prolog baseline used for comparison, benchmarking, and the retained Prolog NLP pipeline. In the Prolog baseline architecture, Go handled service concerns such as HTTP, authentication, provider calls, retries, and process supervision, while Prolog loaded facts and evaluated rules. This shared JSON contract separates data acquisition from the execution-engine implementation.

2.4.1 JavaScript Object Notation as the Intermediate Format

For each provider, the gathering pipeline writes two JSON inputs:

1. **Heimdall JSON.** This input contains organisational IAM state derived from CasC, including identities, boundaries, assignments, trust relationships, teams, and provenance.
2. **Catalog JSON.** This input contains provider action catalogs and role definitions used for wildcard expansion and role discovery.

Both JSON inputs follow the unified schema described in Section 4, where a platform tag distinguishes AWS, Azure, and GCP records. This common structure allows the shared loading logic in GoStore and Prolog to ingest data across all three providers.

2.4.2 Binary Caching with fastrw

In the Prolog baseline, an uncached load parses the JSON documents, maps their records to metamodel facts, and asserts those facts into the dynamic database. After that load, the loader collects the constructed facts and writes them to a `.fastrw` file using SWI-Prolog's binary term format [16]. On a later cache hit, `fast_read/2` reconstructs the cached fact list, which the loader then reasserts into the dynamic database. While this bypasses JSON tokenisation, parsing, and schema-to-fact conversion, it does not bypass binary decoding or fact assertion, and no controlled benchmark measured the exact speedup.

2.5 Related Work

Work related to this thesis falls into five categories: formal single-provider policy analysis, cross-provider and multi-cloud models, general policy-as-code engines, logic programming for access control, and practitioner audit tools. Candidate literature was identified through targeted keyword searches across ACM Digital Library, IEEE Xplore, and Google Scholar using terms combining cloud IAM, multi-cloud access control, policy analysis, policy-as-code, and metamodel, supplemented by citation tracing. Descriptions of selected practitioner tools rely on vendor and open-source project documentation, and the overall selection process is intended to be representative rather than systematic or exhaustive.

Table 2.1: Comparison of related work by provider scope, primary query focus, and underlying model persistence.

System	Provider Scope	Query Focus	Persistent Model?
Margrave [22]	Generic XACML	Policy properties and change impact	No
Turkmen et al. [23]	Generic XACML	SMT-based rule verification	No
Zelkova [24]	AWS	Policy satisfiability and permissiveness	No
IAM Access Analyzer [25]	AWS	Resource-accessibility summaries	No
Gouglidis et al. [26]	GCP	Model-checked property verification	No
Parisel [27]	Azure	Wildcard semantic overreach	No
Zahoor et al. [28]	AWS, Azure, GCP	Multi-cloud policy-conflict detection	Yes (conflict graph)
Quacky [29]	AWS, Azure, GCP	Permissiveness quantification	No
Sukmana et al. [30]	AWS, GCP	Storage-broker access enforcement	Yes (broker scope)
Open Policy Agent [31]	Provider-agnostic	Real-time policy evaluation	No (stateless)
AWS Cedar [32]	Provider-agnostic	Real-time request-response authorization	No
PMapper [33]	AWS	Privilege escalation and access paths	Yes (single-provider)
Cloudsplaining [34]	AWS	Least-privilege violation reporting	No
ScoutSuite [35]	AWS, Azure, GCP	Multi-cloud configuration security checks	No
Prowler [36]	AWS, Azure, GCP	Compliance posture and security findings	No
Cartography [2]	Multi-provider	Cross-provider infrastructure graph	Yes
Steampipe / CloudQuery [37], [38]	AWS, Azure, GCP	Ad-hoc SQL over live state	Yes (un-normalised)
Defender for Cloud [39]	AWS, Azure, GCP	Identity-to-resource effective permissions over live state	Yes (closed, commer-cial)
This work	AWS, Azure, GCP	Unified enumeration, provenance, and audit	Yes (18-predicate model)

2.5.1 Formal Per-Provider Policy Analysis

Formal access-control policy analysis predates cloud-native security tooling. Margrave compiles eXtensible Access Control Markup Language (XACML) policies into decision diagrams to answer policy property and change-impact queries [22], while Turkmen et al. apply SMT solvers to analyse XACML rules [23]. Zelkova brings SMT-based reasoning to AWS policies at cloud scale, translating access control rules into logical formulas to evaluate policy satisfiability and relative permissiveness [24]. The published evaluation reports that Zelkova executes millions of times daily within AWS and answers most queries in well under one second. Backes et al. later introduced

stratified abstraction to produce compact statements describing who can access a resource, a technique deployed in AWS IAM Access Analyzer [25].

The distinction between these formal reasoning approaches and this thesis lies in question shape and structural scope. Zelkova evaluates policy satisfiability and relative permissiveness, whereas AWS IAM Access Analyzer summarises resource accessibility. Both systems remain strictly AWS-specific. The system developed and evaluated in this thesis uses structured catalog queries to enumerate matching identities, role assignments, and permissions.

Gouglidis et al. use model checking to verify whether GCP IAM policies satisfy specified access-control properties [26]. Parisel examines Azure Role-Based Access Control (RBAC) wildcard patterns by expanding them and quantifying their semantic overreach [27]. The system in this thesis performs a similar expansion through `expand_action_pattern/3` (Section 3.4), but uses it as a query primitive rather than the primary subject of evaluation. Both studies address provider-specific analysis, but neither provides a shared, cross-provider model and query layer.

2.5.2 Cross-Provider and Multi-Cloud Models

Zahoor et al. aggregate AWS, Azure, and GCP authorisation policies in an Event Calculus model to identify conflicts between policies [28]. Their work includes automated tooling and a performance evaluation. Its primary goal is policy-consistency verification, whereas this thesis provides an estate-level entity model with provenance and reusable enumeration queries.

Quacky translates authorisation policies across AWS, Azure, and GCP into SMT formulas, applying model counting to quantify absolute and relative permissiveness [29]. Quacky's cross-provider support addresses multi-cloud policy analysis, but remains centred on policy-document permissiveness. This thesis instead constructs an estate-level model of identities, role assignments, permissions, and provenance to answer structured enumeration queries.

Sukmana et al. propose a unified access-control model for a cloud-storage broker spanning AWS and GCP [30]. Their model centralises accounts, storage resources, privileges, provisioning, and authorisation across both providers, making it structurally close to the cross-provider abstraction developed in this thesis. However, its operational scope remains limited to storage-broker provisioning and enforcement rather than analysing an existing IAM estate or evaluating general audit queries.

2.5.3 Policy-as-Code Engines

Open Policy Agent evaluates policies written in Rego, a Datalog-derived language, against arbitrary JSON inputs across Kubernetes admission control, API gateways, and Continuous Integration (CI)/Continuous Delivery (CD) pipelines [31]. AWS Cedar addresses authorisation through a purpose-built language with formally specified semantics and automated-reasoning tooling, designed primarily for real-time request-response evaluation within application services [32]. While both engines provide policy evaluation frameworks, their operational models differ, and neither provides a pre-built domain model or automated ingestion pipeline for AWS, Azure, and GCP IAM state. Using either engine to answer the audit questions in Section 1.1 would require defining a representation of the provider-specific IAM data and building an ingestion layer that converts the CasC state into suitable engine input. The contribution of this thesis lies in the cross-provider model, the automated CasC gathering pipeline, and the structured domain queries, rather than in general-purpose policy evaluation capabilities.

2.5.4 Logic Programming for Access Control

Datalog and related logic languages have long served to express access-control and trust-management semantics. Li and Mitchell use Datalog with constraints as a foundation for trust-management languages [40], SecPAL defines a logic-based language for decentralised authorisation across administrative domains [41], and Pasarella and Lobo present a Datalog framework for modelling and querying relationship-based access-control policies [14]. The Prolog baseline follows this logic-based approach by representing normalised IAM state as facts and evaluating permission queries through derived rules. However, this thesis does not propose a new logic language or formal access-control calculus. It applies the rule-based approach to a shared model populated from production-derived anonymised CasC declarations and provider action catalogs. The evaluation compares the Prolog baseline with GoStore, which reimplements the same structured query catalog over the unified schema and production-derived corpus.

2.5.5 Practitioner Audit Tooling

The following seven practitioner tools analyse deployed IAM configurations or policy definitions and provide operational points of comparison.

- **PMapper** models AWS identities and roles as a directed graph to discover privilege-escalation and alternative access paths within one AWS account or across accounts in an AWS Organization [33]. Its graph analysis overlaps with the trust and privilege-escalation queries evaluated in this thesis, but PMapper operates exclusively within AWS, while this work evaluates a shared, cross-provider model.
- **Cloudsplaining** assesses AWS IAM policies for least-privilege violations and produces risk-prioritised reports from an individual policy file, all policies in one AWS account, or policies across multiple AWS accounts [34]. Its reporting goals align with the Security Report functionality in Section 5.4.5, but Cloudsplaining specialises in AWS-native policy analysis rather than executing unified queries across multiple cloud platforms.
- **ScoutSuite** performs multi-cloud security configuration auditing across AWS, Azure, GCP, and additional providers by collecting live cloud state through provider APIs and generating categorised HTML reports of misconfiguration findings [35]. **Prowler** assesses AWS, Azure, and GCP configurations against CIS benchmarks, National Institute of Standards and Technology (NIST), SOC 2, and similar compliance frameworks, producing pass/fail findings grouped by control and provider [36]. Both tools target a fixed compliance-check catalogue applied to live provider state rather than answering arbitrary enumeration queries against a CasC-derived metamodel.
- **Cartography** imports infrastructure assets, identities, and permissions from AWS, Azure, GCP, and other platforms into a Neo4j graph database to evaluate cross-tenant and cross-provider access relationships [2]. Cartography provides a broad, extensible infrastructure graph populated from live provider APIs, while this thesis evaluates a permission-focused model derived from CasC declarations and provider action catalogs, including file-level provenance for assignments and core entity definitions.
- **Steampipe** and **CloudQuery** expose live AWS, Azure, and GCP resource and IAM state as queryable SQL tables through provider-specific plugins, letting operators run ad-hoc cross-provider queries against a common relational interface [37], [38]. Both are the closest practitioner analogues to the query surface evaluated in this thesis: each combines multiple providers behind one query interface rather than covering one provider at a time. However,

their tables mirror each provider's native API shape rather than a normalised conceptual schema, so cross-provider operations such as wildcard-action expansion or deny-precedence resolution require hand-written, provider-specific SQL rather than a single query against a unified predicate, and neither tool derives its state from CasC declarations with file-level provenance.

- **Microsoft Defender for Cloud** provides commercial CIEM capabilities across Azure, AWS, and GCP by analysing effective permissions and exposing identity-to-resource queries through Cloud Security Explorer [39]. Defender operates as a closed commercial service monitoring live cloud environments, while this thesis documents its schema, query logic, and evaluation method for a fact base derived from CasC declarations and provider action catalogs. Two differences go beyond the deployment model. Because Defender analyses live cloud state rather than configuration files, it cannot trace an observed permission back to the CasC file that declares it, which is the provenance link this thesis provides through `assignment_source/5` and `definition_path/3` (Section 3.2.1). Defender also exposes no typed tool surface for language-model agents, whereas the query catalog here serves both human operators and LLM agents through one shared interface (Section 5.5).

The reviewed documentation does not describe the specific combination of features evaluated in this work: a single model covering selected AWS, Azure, and GCP permission concepts, automated CasC population with file-level provenance for assignments and entity definitions, and query interfaces tailored for both human operators and LLM agents. Per-provider tools focus on individual providers or policy languages, broader platforms track live infrastructure state, and policy engines allow operators to define custom schemas. The contribution claimed here is the implementation and empirical evaluation of this specific combination, rather than proof that no comparable commercial or unpublished system exists.

2.6 Fundamentals Summary

This chapter establishes the background required for the shared query model, enabling the query layer to operate on a unified, platform-tagged representation while provider-specific acquisition and normalisation remain isolated within the gathering and loading pipeline. CasC declarations supply intended organisational state, provider catalogs supply action and role definitions, and JSON serves as the interchange format connecting these inputs to the execution engines. Prolog serves as the research baseline for logical evaluation and benchmarking, while GoStore handles production query execution (Section 4.8). Section 3 defines the resulting 18-predicate schema.

3 Design of a Generic Metamodel

The metamodel provides a common query vocabulary and structure across cloud providers. Provider data and semantics differ, so the same conceptual query may legitimately produce different results depending on the platform. What remains unified is the way those queries are formulated and the representation through which results are returned. Four design constraints emerge from the central research question (Section 1.1):

- **Platform-agnostic.** The same predicates work for AWS, Azure, and GCP. The platform tag is data, not code.
- **Minimal and sufficient for the evaluated concepts.** Every predicate captures an operational concept used by the evaluated queries. Documented provider-specific enforcement features remain outside the schema.
- **Queryable.** The schema is shaped primarily for Prolog's execution model rather than relational normal forms.
- **Extensible.** The conceptual schema can represent an additional provider without modification, provided that provider's concepts map to the existing generic predicates. However, the implemented system requires more than a data feed. Onboarding a new provider means registering it in the loader (`loader.go`), implementing its gatherer with SDK/CLI calls and normalisation logic, and configuring file paths (Section 4).

SWI-Prolog was selected over Datalog for this implementation. Datalog provides termination guarantees and supports stratified negation [14], both useful properties for a security analysis tool. SWI-Prolog [20], however, allows arbitrary procedural code in rule bodies, which is required here for two purposes: wildcard expansion looks up concrete actions from the provider catalog at query time, and the DCG natural-language interface is built on SWI-Prolog's grammar formalism. The MQI protocol provided the Go-to-Prolog interop mechanism for the initial serving implementation. A pure Datalog engine would require these capabilities to be re-implemented outside the logic layer.

This section derives the metamodel's core and extended predicates from the CasC analysis that motivated the schema. We begin with the seven core predicates that capture the shared RBAC structure plus the audit-specific approver relation, add the eleven extended predicates that handle the parts of real IAM that a minimal model omits, and conclude with the derived query predicates. Wildcard expansion and deny precedence convert stored facts into query results.

3.1 Configuration as Code (CasC) Analysis

The schema derives from an analysis of real CasC repositories. Heimdall continuously parses all three providers' CasC repositories and normalises the results into a unified JSON export. The analysis examined that full export corpus across the assignment files, role-definition catalogs, and boundary records for AWS, Azure, and GCP (Section 4.3 quantifies the resulting dataset scale). To identify which concepts the corpus shares across providers, the exported records were inspected with the assistance of LLM-aided structural analysis in an iterative process: each provider's CasC repository schema was examined in turn, the LLM grouped fields by semantic role and surfaced recurring patterns, and we reviewed and consolidated the resulting groupings into a unified schema with platform-specific fields mapped to shared concepts. Each configuration, regardless of provider, encodes the same structure: a *who* (an identity, user, or group), a *what* (a named role or policy bundling permissions), and a *where* (a resource boundary). The link between those three is the *assignment*.

Six generic concepts cover what those files express:

Table 3.1: The six generic metamodel concepts and their corresponding terminology in Azure, GCP, and AWS.

Generic concept	Azure	GCP	AWS
Resource boundary	Subscription	Project	Account
Identity	User / SP / Group	User / SA / Group	User / Role / Group
Role definition	Role Definition	IAM Role	IAM Policy
Assignment	Role Assignment	IAM Binding	Policy Attachment
Permission	Action	Permission	Action

Action name formats follow provider conventions: `Provider/Resource/Operation` in Azure, `service.resource.verb` in GCP, and `service:Action` in AWS.

These six concepts form the shared RBAC structure in the Prolog schema shown in Listing 3.2. The core model adds the audit-specific approver relation, while the extended predicates supply the enterprise context: teams, trust chains, provenance, hierarchy, and conditions.

Provider source schema shape. We deliberately did not mirror the provider source files one-to-one, since those include many details that are not relevant to the unified view and differ considerably per platform. Instead, the metamodel accepts two broad source shapes: a catalog file that describes actions, role definitions, and role-to-action permissions, and a Heimdall file that describes assignments, identities, boundaries, teams, trust edges, and provenance. The loader maps both shapes into the same predicates. The query layer can therefore process AWS `AdministratorAccess` and Azure `Contributor` as normalized facts without knowing their original JSON representation.

Listing 3.1 compares one normalized AWS role and one normalized Azure role, both represented as flat action-effect entries that map directly to `role_definition/4` and `permission/4`.

Listing 3.1: Trimmed records from `aws_roles_actions.json` and `azure_roles_actions.json`, showing one AWS managed policy and one Azure built-in role. Both use flat `permissions[].action/.effect` entries. The gatherers transform AWS policy statements and Azure's four role-definition permission arrays (`actions`, `notActions`, `dataActions`, `notDataActions`) into this common representation during data collection, so both records map to `role_definition/4` and `permission/4` the same way.

```
{
  "aws": {
    "roles": [
      {
        "id": "arn:aws:iam::aws:policy/GreengrassOTAUpdateArtifactAccess",
        "name": "GreengrassOTAUpdateArtifactAccess",
        "type": "aws_managed",
        "permissions": [
          { "action": "s3:GetObject", "effect": "allow" }
        ]
      }
    ]
  },
  "azure": {
    "roles": [
      {
        "id": "azure:storage-blob-data-reader",
        "name": "Storage Blob Data Reader",
        "type": "builtin",
        "permissions": [
          { "action": "Microsoft.Storage/storageAccounts/blobServices/containers/read",
            "effect": "allow" },
          { "action": "Microsoft.Storage/storageAccounts/blobServices/containers/blobs/
              read",
            "effect": "allow" },
          { "action": "Microsoft.Storage/storageAccounts/blobServices/
              generateUserDelegationKey/action",
            "effect": "allow" }
        ]
      }
    ]
  }
}
```

3.2 Mapping to Prolog Predicates

The core model contains seven predicates: six represent the shared provider-neutral concepts introduced above, and `access_approver/3` adds a separate governance relation. Being authorised to approve access is distinct from already holding access through a role assignment.

Two design decisions apply to all seven predicates:

Platform argument Every predicate places `Platform` in its final argument. A query scoped to one provider binds that argument. A cross-platform query leaves it unbound. The same predicate definitions and query rules therefore work in both modes.

Unified identity type The predicate `identity/4` retains the provider type and adds a unified classification. Provider labels are normalised during gathering or export. Compatibility aliases such as `service` and `automation` are grouped under the broader `machine` filter.

The seven core predicates are:

- `resource_boundary(BoundaryID, Name, Category, Platform)`: the scope where permissions apply.
- `identity(Name, Type, UnifiedType, Platform)`: the principal (user, service, group), with the normalised `UnifiedType`.
- `role_definition(ID, Name, Type, Platform)`: a named bundle of permissions. `Type` normalises provider-specific origin terminology into three unified values: `builtin` (platform-managed, read-only), `predefined` (platform-provided but potentially customisable), and `custom` (user-defined). Providers use different internal names for equivalent concepts. AWS, for example, calls platform-managed roles `aws_managed`, which the metamodel maps to `builtin`.
- `assignment(Principal, Boundary, Role, Platform)`: the central junction joining identity, boundary, and role.
- `permission(RoleID, Action, Effect, Platform)`: per-role allow/deny for a specific action.
- `action(Name, Platform)`: a known valid operation in the provider's action catalog.
- `access_approver(BoundaryID, Identity, Platform)`: the identity authorised to grant access to a boundary.

Most arguments in Listing 3.2 follow defined conventions. `Platform` is always `aws`, `azure`, or `gcp`. `Name`, `ID`, `RoleID`, and `BoundaryID` are human-readable labels or provider-native identifiers.

The argument `UnifiedType` is the normalised identity classification. The corpus contains `human`, `machine`, `assumable`, and `group`. The query layer includes `service`, `automation`, and `assumable` in the broader `machine` filter. The `group` type covers provider directory groups (such as Azure AD groups) represented as identities. Teams (`team_member/3`, `team_owner/3`) are a separate concept from Heimdall: they model organisational groupings such as AWS SSO permission-set groups and are distinct from directory groups. Within the evaluated corpus, team records appear only for AWS (Section 3.3.1).

Two arguments carry structured compound terms. `Source` is a `source(Type, Detail, File)` compound, and `Condition` is a `condition(Title, Expression)` term. In `trust_relationship/5`, `TrustingRole/TrustingBoundary` and `TrustedPrincipal/TrustedBoundary` name the two sides of one trust edge. Section 3.3.3 covers the `Source` compound in detail.

Listing 3.2: Predicate overview for the complete metamodel. The first seven predicates form the core model: six shared IAM concepts and the access-approval relation. The remaining eleven add enterprise context, including teams, hierarchy, trust, provenance, conditions, and identity attributes.

```
% Core permission model
action(Name, Platform).
role_definition(ID, Name, Type, Platform).
permission(RoleID, Action, Effect, Platform).
identity(Name, Type, UnifiedType, Platform).
assignment(Principal, Boundary, Role, Platform).
resource_boundary(ID, Name, Category, Platform).
access_approver(BoundaryID, Identity, Platform).

% Team structure
team_member(TeamName, Identity, Platform).
team_owner(TeamName, OwnerIdentity, Platform).

% Provenance and traceability
assignment_source(Principal, Boundary, Role, Source, Platform).
definition_path(EntityID, Path, Platform).

% Boundary hierarchy and scope
boundary_scope(BoundaryID, ScopeType, Platform).
boundary_parent(ChildID, ParentID, Platform).
boundary_installation(BoundaryID, InstallationID, Stage, Platform).

% Administration, conditions, attributes
boundary_admin(BoundaryID, AdminName, AdminType, Platform).
assignment_condition(Principal, Boundary, Role, Condition, Platform).
identity_attribute(IdentityName, Key, Value, Platform).

% Cross-account trust
trust_relationship(TrustingRole, TrustingBoundary,
                   TrustedPrincipal, TrustedBoundary, Platform).
```

These 18 predicates constitute the conceptual schema addressed throughout this thesis. The implementation in `metamodel.pl` declares six additional predicates that support queries and loading without introducing new conceptual entity or relationship types: `compact_permission/4` preserves unexpanded policy patterns, `team_deny/3` and `assignment_effect/5` represent deny semantics, and `role_description/3`, `permission_resource/4`, and `permission_condition/4` provide descriptive or annotation metadata. These six predicates are excluded from the figures and predicate counts because they extend the implementation, not the conceptual model.

One of these six, `assignment_effect/5`, is nonetheless load-bearing: `can_perform/4`'s assignment-level deny check (Listing 3.5) queries it directly, so the 18 counted predicates alone cannot reconstruct the query's full deny-precedence semantics. It is excluded from the conceptual count as an *effect* annotation on the already-modelled `assignment/4` relationship rather than a new entity or relationship type, the same distinction that excludes `permission_condition/4` despite also being read inside `can_perform/4`.

3.2.1 Predicate-Level Coverage

Table 3.2 and Table 3.3 map each of the 18 predicates to its source JSON fields and compare how the Prolog loader (`loader.pl`) and the Go loader implement those mappings. Both loaders read the same two source shapes (Section 3.1). Go represents the extended facts as typed records rather than

Prolog predicates, but retains the same values.

Focused loader and store tests cover all eleven extended mappings, including conditional assignments, hierarchy cycles, team-member overrides, global roles, and metadata on otherwise unassigned entities. The structured Go queries consume this data through assignment results, entity details, CasC provenance, graph edges, and platform-scoped boundary traversal.

Table 3.2: Coverage for the seven core model predicates: the source JSON fields each is populated from, and how the Prolog and Go loaders preserve those mappings. All seven are present in both loaders with equivalent values.

Predicate	Source JSON field(s)	Coverage note
`action/2`	catalog `actions[]` (flat string list, per platform)	Identical in both loaders, lower-cased on load.
`role_definition/4`	catalog `roles[].id`, `.name`, `.type`	Identical. `.description` is also read by both, but stored outside the 18 predicates (Prolog's `role_description/3`, Go's `Role.Description` field, not exposed as a predicate).
`permission/4`	catalog `roles[].permissions[].action`, `.effect`	Identical for action/effect. Both loaders also retain resource and condition annotations outside the conceptual predicate. GoStore indexes them by role, action, and platform for permission-result cards.
`identity/4`	Heimdall `identities[].name`, `.type`, `.unified_type`	Identical.
`assignment/4`	Heimdall `roleAssignments[]` (or snake_case `role_assignments`) `.principal`, `.boundary`, `.role`	Identical, role references normalised per-platform (`normalize_role_ref`).
`resource_boundary/4`	Heimdall `boundaries[].id`, `.name`, `.category`	Identical.
`access_approver/3`	Heimdall `boundaries[].approvers[]`	Identical.

Table 3.3: Coverage for the eleven extended model predicates: the source JSON fields each is populated from, and how the Prolog and Go loaders preserve those mappings. Both loaders retain equivalent values. Prolog stores dynamic facts while GoStore stores typed records and indexes.

Predicate	Source JSON field(s)	Coverage note
`team_member/3`	Heimdall `teams[].members[].email` (falls back to `.name` for legacy files)	Identical in both loaders.
`team_owner/3`	Heimdall `teams[].owner`	Identical, consumed by team lists and details and by ownership edges in the graph view.
`assignment_source/5`	Heimdall `roleAssignments[].source.type`, `.detail`, `.file`	Identical.
`definition_path/3`	Heimdall `identities[].definition_path`, `boundaries[].definition_path`, `roles[].definition_path`	Identical, used directly by `get_casc_path`, entity details, and unused-role results, including entities with no assignment source.
`boundary_scope/3`	Heimdall `boundaries[].scope_type`	Identical, exposed in boundary lists and entity details.
`boundary_parent/3`	Heimdall `boundaries[].parent`	Identical, real loader facts seed GoStore's platform-scoped, cycle-safe hierarchy traversal and boundary graph edges.
`boundary_installation/4`	Heimdall `boundaries[].installation.installation_id`, `.stage`	Identical storage. Go exposes installation and stage in boundary entity details. Prolog currently exposes this fact only through raw predicate access.
`boundary_admin/4`	Heimdall `boundaries[].administrators[].name`, `.admin_type`	Identical.
`assignment_condition/5`	Heimdall `roleAssignments[].condition.title`, `.expression`	Identical annotation storage. Assignment and access-explanation cards show title and expression, but neither engine evaluates them.
`identity_attribute/4`	Heimdall `identities[].attributes.*` (arbitrary key/value pairs) plus `identities[].owners[]` / `.owning_teams[]`	Identical typed values. Go entity details expose all retained keys, while Prolog uses common mail aliases for identity enrichment and leaves other keys available to raw queries.
`trust_relationship/5`	Heimdall `trustRelationships[]` (or `trust_relationships`) `.trusting_role`, `.trusting_boundary`, `.trusted_principal`, `.trusted_boundary`	Identical.

In the anonymised corpus, several extended predicates are populated for only a single platform. Trust relationships appear exclusively in AWS (390 facts), whereas boundary installations mapping projects to deployment stages (191 facts) and assignment conditions (48 facts) occur only in GCP. Team records are similarly restricted to AWS within the evaluated exports (Section 3.3.1). Identity attributes are likewise present only for AWS identities (19968 facts). Because the anonymisation pipeline retains only a fixed set of non-PII structural attribute keys, this tally reflects the anonymised dataset rather than the raw export. The data loaders parse these fields generically for every provider, confirming that these asymmetries originate from the source exports rather than platform-specific ingestion logic.

3.2.2 Implementation Annotations Outside the Conceptual Count

Three source JSON structures fall outside the 18 core conceptual predicates, corresponding to three of the six auxiliary stores detailed in Section 3.2. Table 3.4 maps each annotation to its associated Prolog predicate, Go representation, and source JSON field.

Table 3.4: Implementation annotations maintained outside the 18 core conceptual predicates. Each entry annotates existing facts without defining new entity types.

Annotation	Prolog Predicate	Go Representation	Source JSON Field
Resource constraint	`permission_resource/4`	`PermissionResource`	Policy statement `Resource` (AWS ARN patterns)
Condition operator summary	`permission_condition/4`	`PermissionCondition`	Policy statement `Condition` operator keys
Non-allow assignment effect	`assignment_effect/5`	`Assignment.Effect`	Heimdall `roleAssignments[].effect`

GoStore indexes the permission annotations and includes them in verbose `role_permissions`, compact-role, and identity-permission results. The production-derived anonymised corpus contains 56433 permission-resource and 15895 permission-condition annotations. Synthetic loader and query tests exercise non-empty values. Resource and condition annotations are displayed, not evaluated against request context. However, non-`allow` assignment effects are considered during access evaluation.

Two further Heimdall array shapes are normalised by both loaders rather than represented as new predicates. The first transformation maps `boundaries[].team_member_access_control[]` to ordinary assignments with `team_member_override` provenance. The second maps AWS's `global_user_roles[]` to assignments on a synthetic `global_{platform}` boundary. Both therefore flow through the existing access queries.

Listing 3.3 demonstrates the provider-neutral `assignment/4` predicate with one representative fact per platform. Each fact is ground and therefore contains no variables. Its four arguments are the principal, a boundary identifier (account, subscription, or project), a provider-native role identifier, and a platform tag (`aws`, `azure`, or `gcp`). The referenced identities, boundaries, and roles are defined separately through the predicates listed above.

Listing 3.3: One representative ground `assignment/4` fact per platform (AWS, Azure, and GCP), expressed as SWI-Prolog [20] dynamic facts. All three use the same provider-neutral predicate. The final argument identifies the platform.

```
1 assignment('alice@example.com', 'acc-123456789-prod',
2            'arn:aws:iam::aws:policy/AdministratorAccess', aws).
3 assignment('alice@example.com', 'sub-a1b2c3d4',
4            'azure:contributor', azure).
5 assignment('alice@example.com', 'proj-analytics-prod',
6            'roles/storage.admin', gcp).
```

These three Prolog-native examples illustrate how the same predicate schema unifies facts from different providers without requiring platform-specific predicate variants. The following figures visualise the relationships between the core predicates. A single Unified Modeling Language (UML) class diagram for all 18 predicates was considered, but representing every n-ary association and cross-cutting reference on one canvas produced a result too dense to read clearly. The seven core predicates are therefore introduced using the flow notation below. The eleven extensions are presented through five focused UML diagrams in Section 3.3, one alongside each extension group.

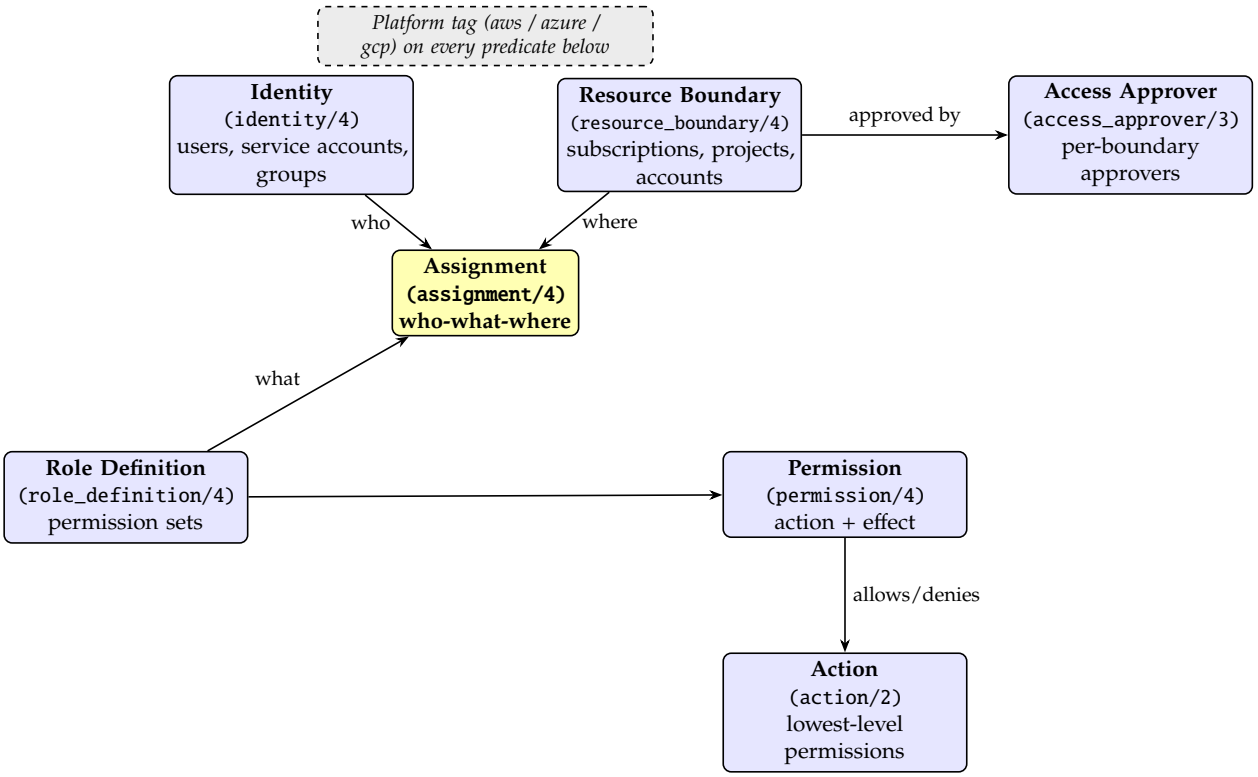


Figure 3.1: Core metamodel: six provider-neutral IAM concepts and the access-approval relation. Assignments connect identities to roles within resource boundaries. The role–permission–action chain expresses which actions each role allows or denies. The predicate `access_approver/3` records who may approve access to a boundary. The platform argument on every predicate scopes facts by provider and supports uniform cross-platform querying.

Figure 3.1 shows the seven core predicates, which capture the shared RBAC structure together with the access-approval relation. Section 3.3 groups the eleven extended predicates, which add organisational and governance context beyond this core, into five concerns, each presented in a focused UML diagram.

3.3 Extended Metamodel Predicates

Production IAM data carries organisational and governance context that textbook RBAC does not represent: team membership, cross-account trust, boundary hierarchy, conditional bindings, identity attributes, and provenance. The eleven extended predicates add this context to the compact core model without replacing its concepts. Queries may draw on the additional relationships and metadata, but the core assignment, role, and permission structure remains the foundation.

3.3.1 Team Membership

AWS SSO permission-set groups appear in Heimdall as *teams*, which form a layer of indirection between identities and role assignments. Two predicates cover this: `team_member(TeamName, Identity, Platform)` and `team_owner(TeamName, OwnerIdentity, Platform)`. Representing team structure separately from the core assignment relation supports queries about team composition, ownership, and the access currently held by team members.

In the production-derived anonymised corpus (Section 4.3), AWS contributes 1 167 team-bearing source records representing 30 distinct anonymised team names and 6 299 membership records. The

examined Heimdall exports contain no equivalent records for Azure or GCP. The platform argument on both predicates means that equivalent records from another provider would be represented without schema changes.

Figure 3.2 shows how the membership and ownership predicates both reference the core identity while remaining independent facts.

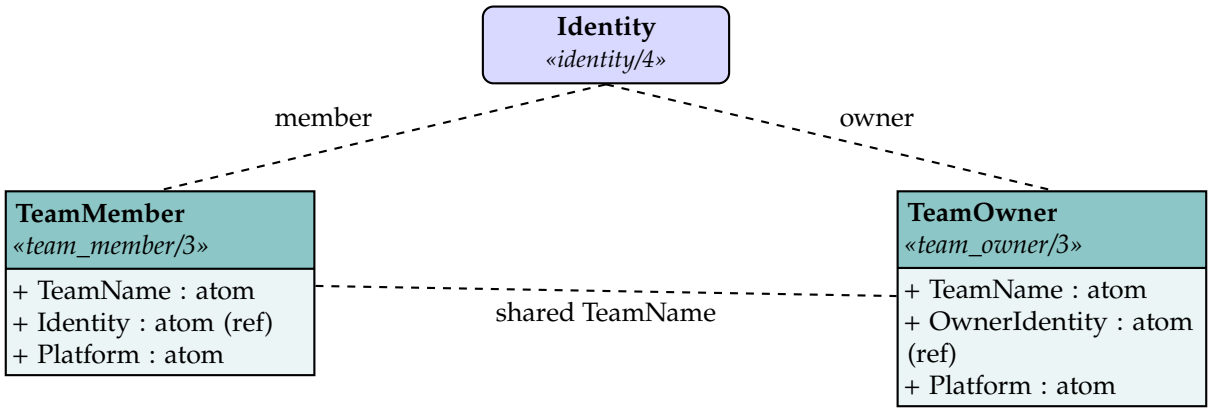


Figure 3.2: Team-membership extension: `team_member/3` and `team_owner/3` as a UML class diagram. The identity fields of both predicates reference the core **Identity** concept (stub box, full attributes in Figure 3.1). Membership and ownership facts with the same team-name value describe the same team.

3.3.2 Cross-Account Trust Relationships

An AWS role trust policy can permit a principal in one account to assume a role in another, creating a potential cross-account privilege path [5]. Ordinary role assignments do not represent this trust edge, so `trust_relationship/5` records each directed trust edge explicitly: the trusting role, its boundary, the trusted principal, the trusted boundary, and the platform. A single fact represents one directed edge. Multi-hop reachability is derived by recursively traversing multiple facts. A trust relationship is not, by itself, proof that every condition required for access is satisfied.

In Listing 3.4, `Role` and `Platform` are input arguments and `TrustedPrincipals` is the output, the deduplicated principals collected by `findall/3` and sorted. This helper returns direct relationships only.

Figure 3.3 shows the role and boundary references carried by one trust-relationship fact.

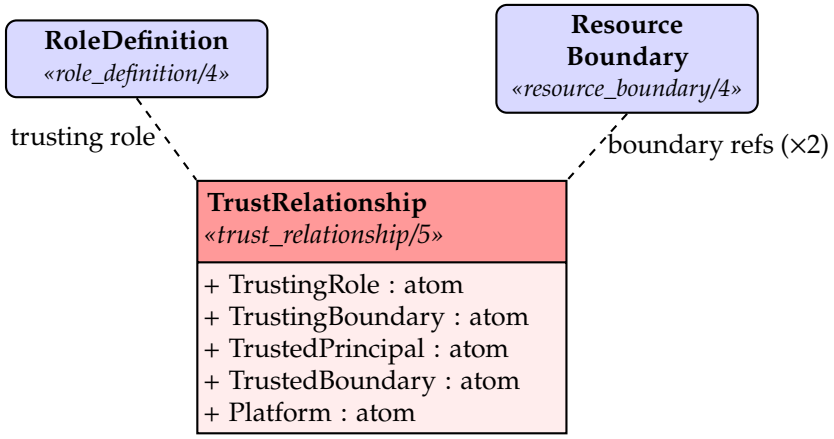


Figure 3.3: Cross-account trust extension as a UML class diagram. **TrustRelationship** represents `trust_relationship/5`: **TrustingRole** references **RoleDefinition** and both **TrustingBoundary** and **TrustedBoundary** reference **ResourceBoundary**. Representing the two boundary references as attributes rather than association lines keeps the diagram compact. `TrustedPrincipal` is represented as an attribute because it may contain different AWS principal forms, including an account root or role Amazon Resource Name (ARN), rather than a single metamodel concept.

Listing 3.4: Cross-account trust relationship facts and the `trust_chain/3` derived predicate (`derived.pl`). Two independent `trust_relationship/5` facts are shown: a root principal in `acc-dev` that can assume a cross-account role in production, and a separate CI/CD principal in `acc-dev` that can assume a staging deploy role. The two facts do not form a multi-hop chain. `trust_chain/3` gathers the directly trusted principals for one role. Transitive reachability requires a separate recursive traversal over multiple `trust_relationship/5` facts.

```
% AWS assume-role trust relationships:
%   trust_relationship(TrustingRole, TrustingBoundary,
%                      TrustedPrincipal, TrustedBoundary, Platform)

trust_relationship(
    'arn:aws:iam::111122223333:role/CrossAccountRole', 'acc-prod',
    'arn:aws:iam::444455556666:root',                  'acc-dev', aws).

trust_relationship(
    'arn:aws:iam::555566667777:role/StagingDeployRole', 'acc-staging',
    'arn:aws:iam::444455556666:role/cicd',               'acc-dev', aws).

%! trust_chain(+Role, -TrustedPrincipals, +Platform) is det.
%  All principals that can directly assume Role via trust policy.
trust_chain(Role, TrustedPrincipals, Platform) :-
    findall(Principal,
            metamodel:trust_relationship(
                Role, _, Principal, _, Platform),
            All),
    sort(All, TrustedPrincipals).

% Example queries:
% ?- trust_chain('arn:aws:iam::111122223333:role/CrossAccountRole', Ps, aws).
%    Ps = ['arn:aws:iam::444455556666:root'].
%
% ?- trust_chain('arn:aws:iam::555566667777:role/StagingDeployRole', Ps, aws).
%    Ps = ['arn:aws:iam::444455556666:role/cicd'].
```

3.3.3 Assignment Provenance and File Traceability

For an auditor, knowing that `Alice` has `Owner` on `prod` is incomplete without the source file that defines the assignment (for example, `rbac/prod/admins.yaml`). The predicate `assignment_source/5` associates an assignment's principal, boundary, role, and platform with a `source(Type, Detail, File)` term. `Type` and `Detail` record provenance metadata (e.g. direct, group-derived, or inherited sources) while `File` records the CasC path. The Heimdall parsing pipeline retains file paths but not source line numbers, so provenance is recorded at file level only.

The predicate `definition_path/3` associates an entity identifier (identity, boundary, or role) and a platform with its CasC definition path. These recorded file paths are surfaced as source links in the Explorer UI's provenance display (Section 5.4), enabling an auditor to jump from a reported assignment to the originating CasC file.

Figure 3.4 separates the provenance attached to an assignment from the definition paths attached to identities, resource boundaries, or role definitions.

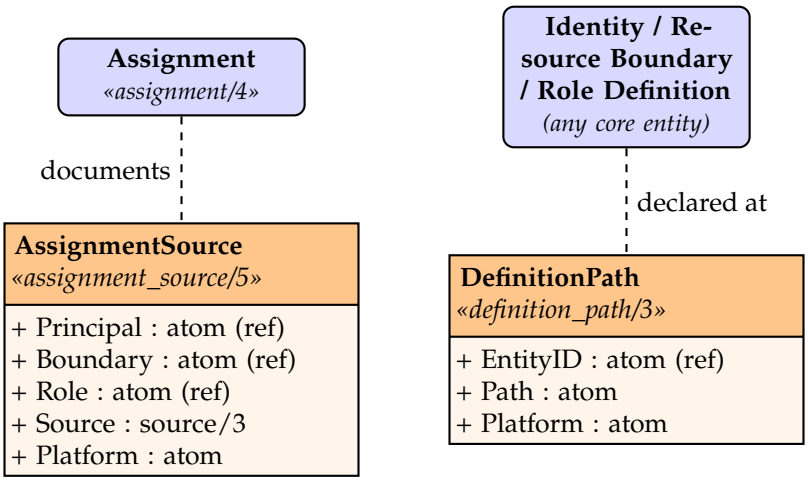


Figure 3.4: Assignment provenance and file traceability as a UML class diagram. **AssignmentSource** represents `assignment_source/5`. Its `Principal`, `Boundary`, `Role`, and `Platform` fields identify the documented assignment. **DefinitionPath** represents `definition_path/3` and associates an identity, resource boundary, or role definition with its CasC file path through `EntityID`. The shared blue stub avoids drawing three equivalent reference lines.

3.3.4 Boundary Hierarchy and Scope

Each provider has its own hierarchy primitives (Azure Management Groups, GCP Folders, AWS Organizational Units). The predicate `boundary_scope/3` records each boundary's level (`org`, `folder`, `project`, `subscription`, `account`, ...), and `boundary_parent/3` records directed child-to-parent containment edges.

The derived predicate `boundary_hierarchy/2` in `derived.pl` follows successive `boundary_parent/3` facts and returns ancestors from the immediate parent to the root. This traversal supports structural analysis such as reporting a boundary's full containment path, but it does not propagate permissions: an effective-permission lookup still requires the assignment's exact boundary, and no derived predicate walks `boundary_parent/3` to grant permissions assigned on an ancestor boundary.

Two further predicates attach per-boundary annotations rather than hierarchy edges. The predicate `boundary_installation/4` records a Heimdall installation identifier and deployment stage for a boundary. The predicate is generic by design and accepts a `Platform` argument, but in the evaluated

corpus it is populated only from GCP source data, as the other providers do not expose an equivalent installation-stage concept in their Heimdall exports. The predicate `boundary_admin/4` records the administrative identity responsible for a boundary.

Figure 3.5 distinguishes the hierarchy chain from the parallel boundary-administrator annotation.

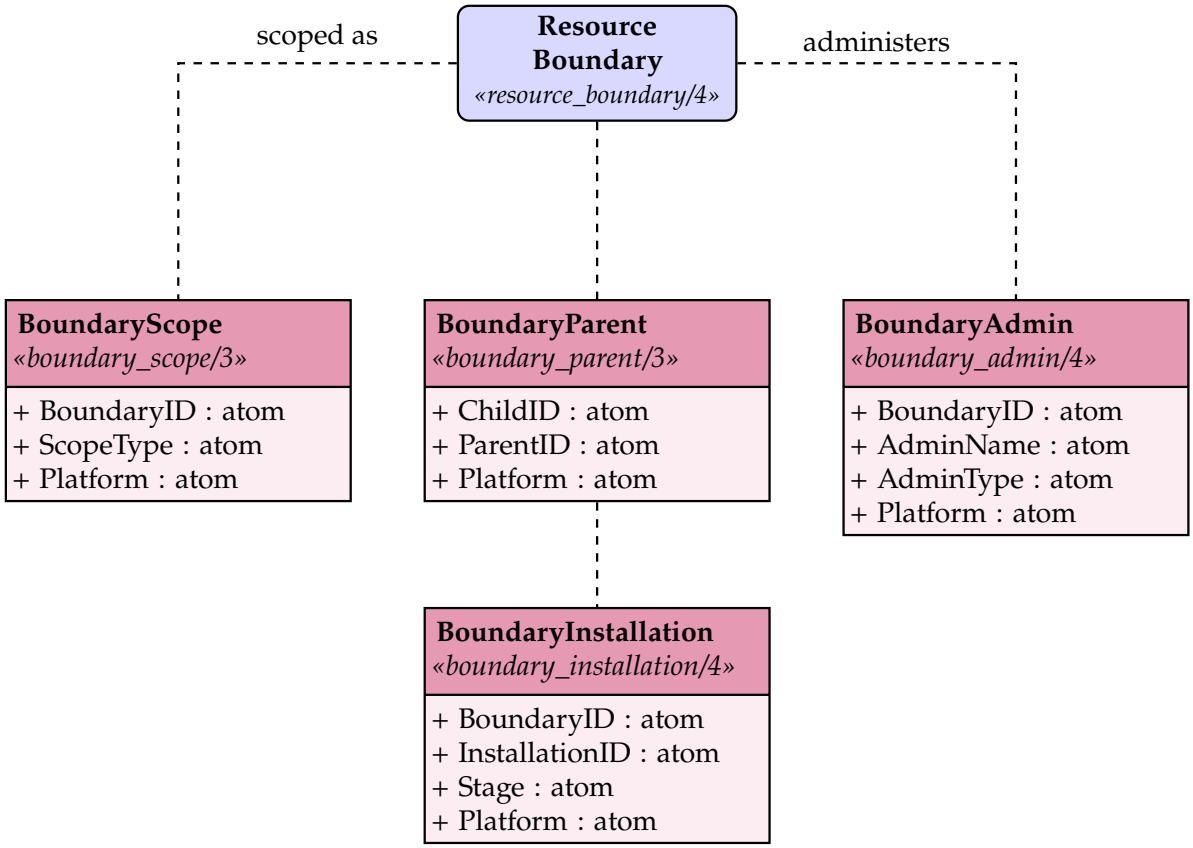


Figure 3.5: Boundary hierarchy and administration as a UML class diagram. **BoundaryScope** records a boundary's scope category. **BoundaryParent** represents the child-to-parent hierarchy edge. **BoundaryInstallation** attaches installation and stage metadata. It is generic by design but populated only from GCP data in the evaluated corpus. **BoundaryAdmin** is a separate administrative annotation.

3.3.5 Assignment Conditions and Identity Attributes

Some IAM role assignments carry conditions, such as GCP Common Expression Language (CEL) expressions and Azure ABAC rules. AWS policy `Condition` blocks are handled separately as permission-level annotations by `permission_condition/4`, as described above. Evaluating assignment conditions would require provider-specific expression evaluators and request-context attributes that the Heimdall export does not provide. Instead, `assignment_condition/5` stores the condition title and raw expression under `condition(Title, Expression)` so that a human can inspect them in the User Interface (UI).

The predicate `identity_attribute/4` adds free-form key–value metadata to identities, for example department, cost centre, or employee ID. The current implementation loads and displays these attributes but does not use them for general filtering or for joining to organisational data outside IAM.

Figure 3.6 shows these two enrichments attached to assignments and identities respectively.

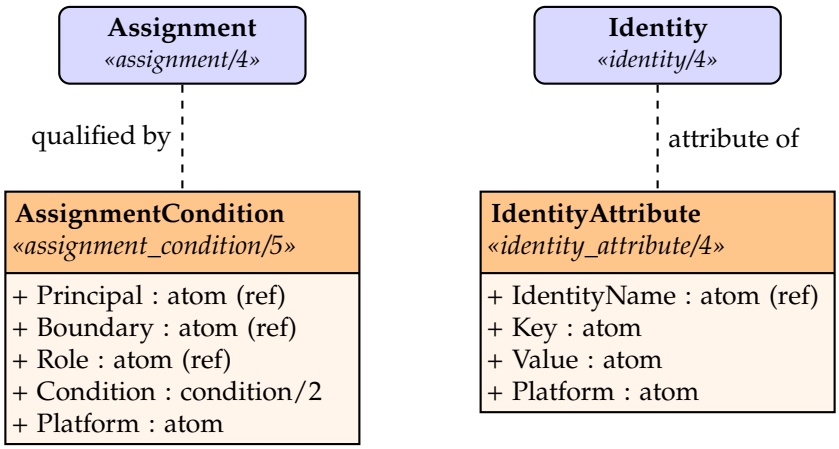


Figure 3.6: Assignment conditions and identity attributes as a UML class diagram. **AssignmentCondition** qualifies one **Assignment**, mirroring **AssignmentSource** in Figure 3.4. **IdentityAttribute** attaches free-form key–value metadata to **Identity**.

3.4 Query Design and Implementation

Deny-precedence semantics require `permission/4` to carry an `Effect` argument so that explicit deny rules override allow rules for the same action. Wildcard expansion requires `action/2` to hold the full provider action catalog so that patterns such as `s3:*` resolve to concrete actions. The loaded base facts store these core concepts, while derived predicates implement the permission-checking logic. Figure 3.7 illustrates this conceptual decision flow, which Listing 3.5 and Listing 3.6 encode.

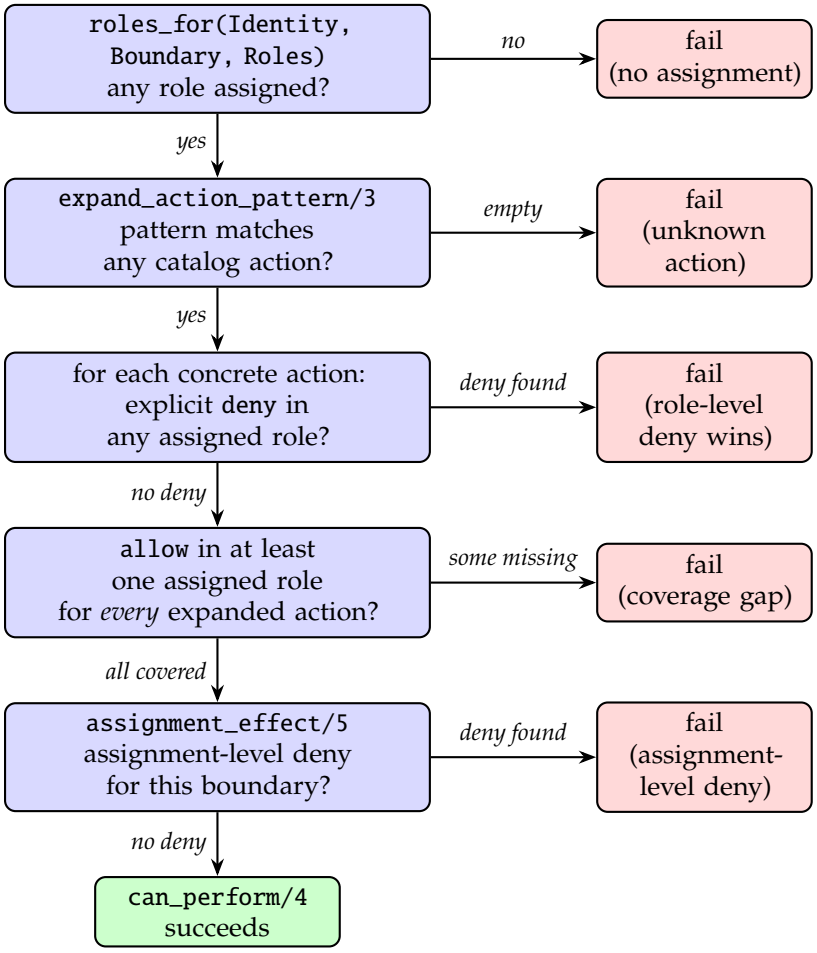


Figure 3.7: Permission-resolution decision tree for `can_perform/4`. Role-level deny precedence is checked before allow coverage: a single explicit deny in any assigned role fails the check before any allow is considered. After allow coverage succeeds, a separate assignment-level deny check through `assignment_effect/5` can still cause failure. The current GCP input contains only role allows because hierarchy-scoped deny policies are not gathered. Azure `notActions`/`notDataActions` are resolved as exclusions within each role definition, while explicit AWS denies are checked at the role level directly.

Listing 3.5: Permission checking with deny precedence (`derived.pl`). The predicate `can_perform/4` resolves all assigned roles, expands the requested action pattern into its concrete set, and verifies that every expanded action passes the role-level check, then applies the assignment-level deny check shown in Figure 3.7.

```
%! can_perform(+Identity, +Boundary, +ActionPattern, +Platform) is semidet.
%  True iff Identity is allowed ALL actions matching ActionPattern on Boundary.
can_perform(Identity, Boundary, ActionPattern, Platform) :-
    roles_for(Identity, Boundary, Roles),    % get assigned roles
    Roles \== [],
    expand_action_pattern(ActionPattern, ConcreteActions, Platform),
    ConcreteActions \== [],
    forall(                                  % ALL expanded actions must pass
        member(ConcreteAction, ConcreteActions),
        action_allowed_by_roles(Roles, ConcreteAction, Platform)
    ),
    % Assignment-level deny check: an assignment-scoped deny record overrides
    % an otherwise-granted role-level allow for the same identity/boundary/role.
    \+ ( member(ConcreteAction2, ConcreteActions),
         member(Role, Roles),
         metamodel:permission(Role, ConcreteAction2, allow, Platform),
         metamodel:assignment_effect(Identity, Boundary, Role, deny, Platform) ),
    !.

%! action_allowed_by_roles(+Roles, +Action, +Platform) is semidet.
%  Deny takes strict precedence: any explicit deny blocks the action.
action_allowed_by_roles(Roles, Action, Platform) :-
    ( member(Rd, Roles), metamodel:permission(Rd, Action, deny, Platform) ->
        fail                                 % explicit deny wins
    ; member(Ra, Roles), metamodel:permission(Ra, Action, allow, Platform) ->
        true                                 % at least one allow
    ; fail                                   % no permission found
    ).
```

The variables `Rd` and `Ra` iterate through assigned roles to evaluate the deny and allow branches, respectively. To handle assignment-level evaluation, `ConcreteAction2` and `Role` check role permissions against explicit effects in `assignment_effect/5`. The surrounding negation ensures that no matching deny combination exists for the requested action.

Listing 3.6: Wildcard expansion (`derived.pl`). The predicate `expand_action_pattern/3` returns a singleton list for a literal action, the complete platform action catalog for the global wildcard *, or the subset of catalog actions matching a partial wildcard pattern. The helper `action_matches_pattern/2` normalises both operands to lower case and delegates to `pattern_match_chars/2` for recursive matching over character lists.

```
%! expand_action_pattern(+Pattern, -ConcreteActions, +Platform) is det.
%  Expand a wildcard pattern (e.g. 'ec2:*') into matching concrete actions.
%  If Pattern has no wildcards, returns [Pattern].
expand_action_pattern(Pattern, ConcreteActions, Platform) :-
    ( \+ sub_atom(Pattern, _, _, _, '*') ->
        ConcreteActions = [Pattern]          % no wildcard: return as-is
    ; Pattern = '*' ->
        findall(A, metamodel:action(A, Platform), ConcreteActions)
    ; findall(A,                             % partial wildcard: ec2:*, s3:Get*
              ( metamodel:action(A, Platform),
                action_matches_pattern(A, Pattern) ),
              ConcreteActions)
    ).

%! action_matches_pattern(+Action, +Pattern) is semidet.
%  Case-insensitive wildcard pattern match.
action_matches_pattern(Action, Pattern) :-
    downcase_atom(Action, AL),
    downcase_atom(Pattern, PL),
    atom_chars(AL, AChars),
    atom_chars(PL, PChars),
    pattern_match_chars(AChars, PChars).

%! pattern_match_chars(+ActionChars, +PatternChars) is semidet.
%  Recursive character-level wildcard matching.
pattern_match_chars([], []) :- !.
pattern_match_chars(_, ['*']) :- !.
pattern_match_chars([C|Rest], [C|PRest]) :- !,
    pattern_match_chars(Rest, PRest).
pattern_match_chars(Action, ['*'|PRest]) :-
    ( pattern_match_chars(Action, PRest)
    ; Action = [_|RestAction],
      pattern_match_chars(RestAction, ['*'|PRest])
    ).
```

Table 3.5 traces the evaluation of the action `s3:getobject` against the pattern `s3:get*`. Each literal character matches through the third clause until only the trailing * remains in the pattern, at which point the second clause (terminal wildcard) cuts and succeeds regardless of the remaining action characters. The fourth clause, which handles a * followed by further pattern characters (an infix wildcard, e.g. *a*a*a*), is not exercised by this terminal-wildcard example. Its backtracking behaviour is discussed below.

Table 3.5: Execution trace of `pattern_match_chars/2` matching the action `s3:getobject` against the pattern `s3:get*`, detailing the argument state and active clause at each recursive step.

Step	Action Characters	Pattern Characters	Clause Fired
1	`s3:getobject`	`s3:get*`	Literal head (`s`)
2	`3:getobject`	`3:get*`	Literal head (`3`)
3	`:getobject`	`:get*`	Literal head (`:`)
4	`getobject`	`get*`	Literal head (`g`)
5	`etobject`	`et*`	Literal head (`e`)
6	`tobject`	`t*`	Literal head (`t`)
7	`object`	`*`	Terminal wildcard (match succeeded)

The mid-pattern * clause can backtrack exponentially for pathological patterns with many consecutive wildcards (such as `*a*a*a*`). In practice, provider action names follow a structured `service:Action` or `service.resource.verb` format with at most one mid-pattern wildcard per segment, so this worst case does not arise on real catalog inputs.

Beyond `can_perform/4`, the `derived.pl` module exports 45 predicate signatures layered on the same primitives. One exported predicate, `redundant_assignments/4`, detects a role assignment whose allow-permissions are fully subsumed by another role the same identity already holds on the same boundary. Because the check operates only on allow sets and ignores deny statements, its output identifies candidates for review rather than assignments that can be removed without further analysis.

Table 3.6 summarises the resulting per-provider contract. It shows what `can_perform/4` evaluates on each platform and what a positive answer means for an auditor.

Table 3.6: Per-provider semantics of `can_perform/4`. The effective-permission check evaluates a different negative-permission model on each platform because each provider's deny mechanism is represented differently in the loaded facts.

Provider	Negative permissions in loaded facts	What `can_perform/4` evaluates	Auditor interpretation of a positive result
AWS	Explicit `deny`-effect `permission/4` facts and assignment-level denies in `assignment_effect/5`.	Deny-overrides-allow at two levels. Role-level deny takes strict precedence over role-level allow. After allow coverage succeeds, the assignment-level deny check can still reject.	Identity has at least one allow and no deny within the in-scope role and assignment model. AWS SCPs and resource-based policies are not evaluated. Policy `Condition` blocks are stored as annotations (`permission_condition/4`) but not evaluated.
Azure	`notActions`/ `notDataActions` are resolved as set subtractions at gather time. Only `allow`-effect facts are stored.	Allow-only evaluation after subtraction. No deny facts exist in the loaded base.	Identity has an effective allow after `notActions` subtraction within the role definition. ABAC conditions attached to assignments are stored but not evaluated.
GCP	No deny facts are loaded. Hierarchy-scoped deny policies are not exported by the gathering pipeline.	Allow-only evaluation. The predicate cannot fail on a deny because no deny facts exist.	Identity has at least one allow in the gathered data. Hierarchy-scoped deny policies, if present in the live environment, are not reflected in the result. Assignment conditions (CEL expressions) are stored but not evaluated.

The exported predicate `privilege_escalation_risk/2` compares the allow-permissions granted by an assigned role against a provider-specific escalation-action list explicitly maintained in `derived.pl`. It flags identities who hold a role that grants one of those actions and for which that same role contains no role-level deny for that concrete action. Because the predicate does not perform the full cross-role and assignment-level deny evaluation implemented by `can_perform/4`, its output identifies candidates for focused review rather than definitive proof of exploitability.

The helper `action_impact/3` heuristically classifies actions as `read`, `write`, `delete` or `admin` to aid prioritisation, and `identity_group_membership/3` exposes direct team membership through `team_member/3` for downstream analyses.

The count of 45 covers the module's declared public interface, excluding non-exported helper predicates and SWI-Prolog-generated tabling wrappers. The full permission-profile query surfaced in the Prolog NLP layer is assembled by `find_full_permission_profile/2` in `queries.pl` rather than by a predicate in `derived.pl`.

Listing 3.7 makes the expansion concrete.

Listing 3.7: Expanding the wildcard pattern `s3:Get*` against the loaded AWS action catalog. Matching is case-normalised: the pattern contains `Get` while the returned actions are lowercase. After normalisation, the trailing wildcard selects all catalog actions beginning with the `s3:get` prefix. The result is an ordinary Prolog list that downstream predicates iterate. Captured 6 July 2026 from the production-derived anonymised corpus (25 983 AWS actions).

```
?- expand_action_pattern('s3:Get*', Actions, aws),
   length(Actions, N).
Actions = ['s3:getbucketpolicy', 's3:getbucketwebsite',
           's3:getacceleraterconfiguration', 's3:getaccessgrant',
           's3:getaccessgrantsinstance',
           's3:getaccessgrantsinstanceforprefix'|...],
N = 109.
```

Azure control-plane and data-plane permission calculations (Section 2.1.1) remain separate during gathering. The gatherer maintains distinct control-plane and data-plane operation sets, subtracting each negative array solely from its corresponding positive array: `notActions` from `actions`, and `notDataActions` from `dataActions`. The surviving operations are subsequently combined and written as flat allow permissions to the catalog snapshot consumed by GoStore and the Prolog baseline. The gatherers do not encode either Azure exclusion array as deny permissions.

Separate assignment-level deny records, where present in the organisational source, remain explicit. Figure 3.8 shows the two role-local subtraction operations.

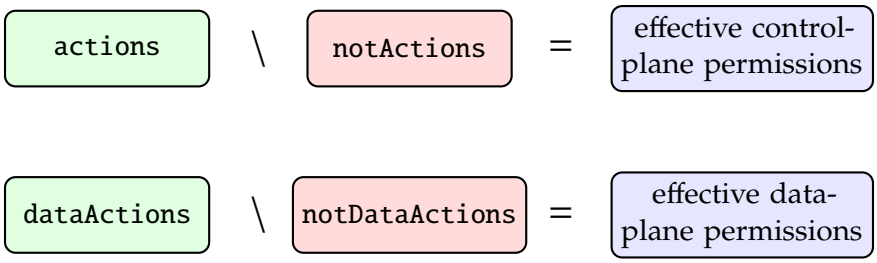


Figure 3.8: Azure role-definition permission calculations. Effective control-plane permissions are `actions` minus `notActions`, while effective data-plane permissions are `dataActions` minus `notDataActions`. The two `not*` arrays are role-local exclusions rather than deny rules, and an excluded operation that was not granted by its corresponding positive array has no effect. Before subtraction, wildcards in each array are expanded only against the corresponding control-plane or data-plane operation set.

4 Automated Metamodel Generation

The metamodel is only useful if the facts inside it are current and cover the supported query scope. Three engineering problems shape the design of the generation pipeline:

1. **Organisational facts and provider catalog information come from different sources and must be loaded together.** Heimdall supplies organisational assignments, and the provider-facing gatherers supply action-catalog snapshots and built-in role definitions. Both source-file families are loaded together into the common model (Section 4.1).
2. **A missing or stale cache forces parsing and normalisation of the entire JSON corpus at startup.** The retained Prolog baseline uses the `fastrw` binary cache to reduce repeated startup parsing, although decoding and fact assertion still remain (Section 2.4.2).
3. **The original execution path became a production bottleneck.** Aggregate-query latency and the separate Prolog/MQI serving architecture motivated the native GoStore reimplementation (Section 4.8).

This section describes the data pipeline from live cloud state to queryable model. The production engine is GoStore, with SQLStore as a raw-query fallback; the historical SWI-Prolog loader with `fastrw` caching and MQI protocol remains the comparison and benchmark baseline. Both engines consume the same source-file families from the gathering stage. The section proceeds bottom-up: gathering and normalisation (Sections 4.1–4.2), Prolog loading and caching (Sections 4.3–4.5), the MQI protocol (Section 4.6), and the GoStore reimplementation with its benchmark comparison (Section 4.8). Figure 4.1 shows the complete pipeline.

4.1 Two-Path Gathering

The Heimdall fetch runs first, one sequential request per platform. Once it completes, the three providers' SDK/CLI gatherers run concurrently. The results are written to separate provider source files that are loaded together into the common model.

Unless stated otherwise, every catalog, role, and record count reported in this section reflects a single recorded gathering run collected on 6 July 2026. Provider action catalogs, role definitions, and organisational CasC state change over time, so these figures are a snapshot from that date rather than a persistent property of the providers or the organisation.

4.1.1 Path 1: Heimdall API (Organisational CasC)

The Heimdall API [8] is an internal Dynatrace service that continuously parses the organisation's CasC repositories (Bitbucket-hosted) and exposes them through a JSON REST API: identities, role assignments, resource boundaries, trust relationships, teams, and assignment provenance with originating-file paths. Heimdall reads the provider-native CasC formats maintained by each team (custom JSON structures per provider) and normalises them into a unified export schema that is provider-independent at the API level. The gathering client reads this unified output rather than the raw provider-specific CasC files. As part of the same engineering environment, we contributed the GCP and Azure Heimdall parsers and parts of the unified export schema. This internal dependency means the gathering pipeline cannot be reproduced outside the organisation without replacing Heimdall with an equivalent CasC-parsing service for each provider.

Internally, Heimdall parses per-provider JSON into an in-memory Java/Kotlin representation and serialises it into one JSON document per provider (`aws`, `azure`, `gcp`) that the gathering client reads. It performs other organisational functions beyond this export, which are out of scope. Every entity

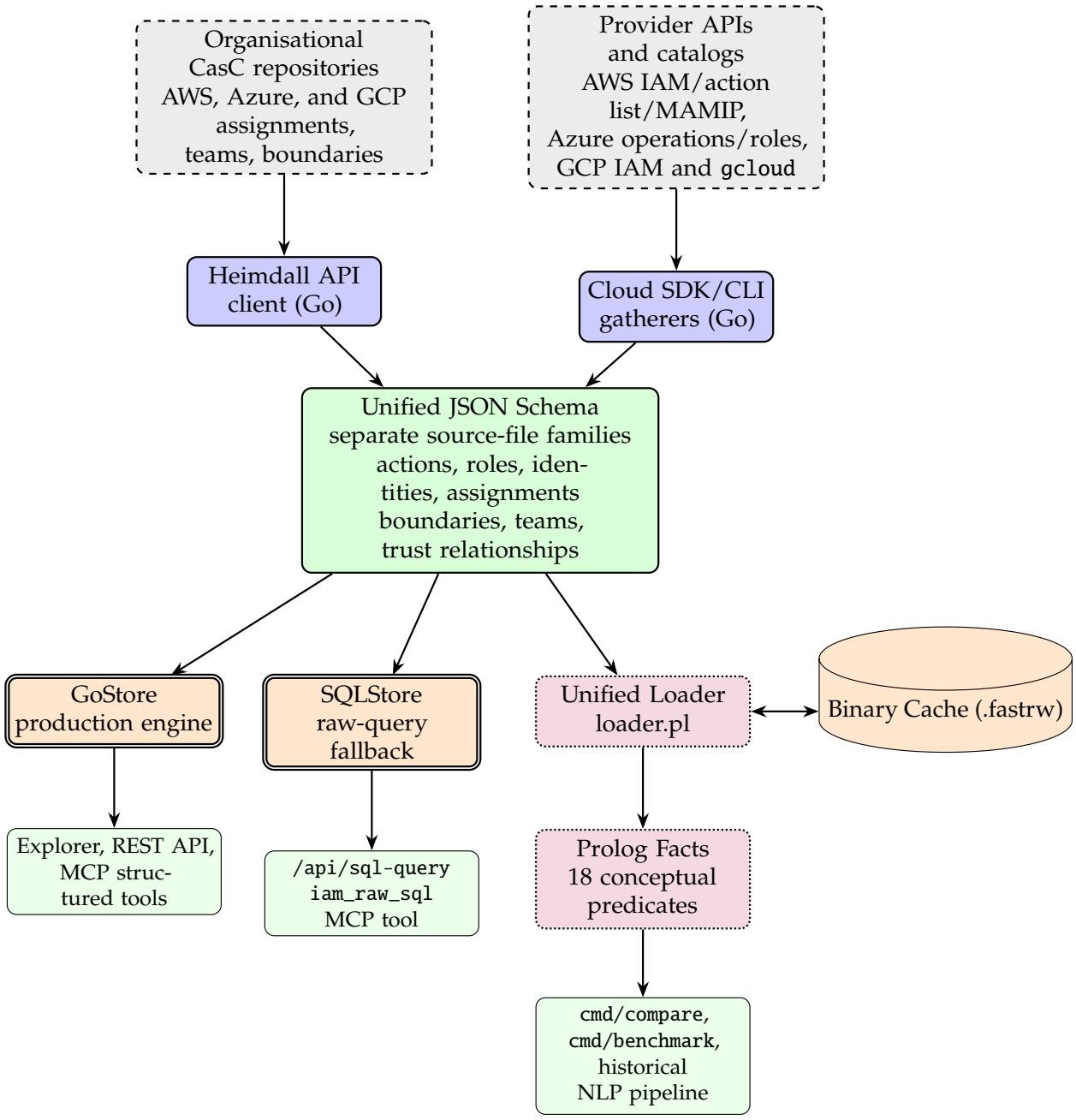


Figure 4.1: Complete data pipeline. The Heimdall API client reads organisational facts derived from the three CasC repositories, whereas the cloud SDK/CLI gatherers obtain action catalogs and role definitions from provider APIs and catalog sources. Both paths produce separate source files that conform to a unified JSON schema. Three independent engines load these files: GoStore (production, backing the Explorer, REST API, and MCP structured tools), SQLStore (the sandboxed raw-query fallback behind `/api/sql-query` and `iam_raw_sql`), and the Prolog loader, which maintains the `fastrw` binary cache for its own fast reloads and backs only the comparison/benchmark tooling and the historical NLP pipeline. All 18 conceptual predicates are populated on the Prolog side from this pipeline.

reaching the metamodel passes through this export; the gathering pipeline treats it as ground truth without independent reconciliation against the underlying CasC repositories.

The Go client authenticates using configured credentials and issues one unpaginated request per platform, writing the response to `{provider}_heimdall.json`, where `{provider}` is one of `aws`, `azure`, or `gcp`.

4.1.2 Path 2: Cloud SDKs/CLIs (Platform Catalogs)

The same Go process gathers provider action catalogs and built-in role definitions by combining provider APIs with maintained supplemental catalog sources. Each provider implements the same gathering interface. The shared output path writes the resulting data to `{provider}_roles_actions.json`. The following three strategy modes control which gathering path is used:

- **`auto`:** Attempts gathering through the provider SDK path and retries through the CLI fallback after any SDK-path error.
- **`sdk`:** Disables the CLI fallback but still loads required supplemental catalogs, particularly the AWS action list.
- **`cli`:** Avoids the provider SDK path entirely, making it useful in environments where the provider CLI is configured but SDK-based access is unavailable.

The fallback implementations combine supplemental catalogs with provider CLIs to assemble the action and role data:

- **AWS:** Combines action definitions from the `aws-iam-actions-list` repository [42] with a sparse checkout of MAMIP (now known as IAMTrail) [43] for managed-policy definitions.
- **Azure:** Combines the Azure PowerShell cmdlet `Get-AzProviderOperation` [44] with a local `azure-all-iam-roles.json` snapshot.
- **GCP:** Executes `gcloud iam roles describe` through a concurrent Go worker pool with retry and backoff, fetching each predefined role concurrently because every role requires a separate command invocation. The recorded corpus contains 2 363 such roles (as of 6 July 2026).

Heimdall supplies CasC-defined assignments and their organisational context, but its visibility is limited to permissions explicitly referenced by those assignments. The provider catalogs supply the action and built-in role definitions required to expand wildcard patterns such as `s3:*`. Both source-file families are therefore required to populate the common model.

The gathering architecture separates provider-dependent acquisition from shared processing. The provider-specific SDK and CLI files acquire the source data, whereas `gather.go` handles strategy selection, wildcard expansion helpers, and serialisation into the unified JSON source schema. The earlier Python loaders combined these responsibilities within the provider scripts.

Adding a fourth cloud provider would require implementing the shared `Gatherer` interface to map native provider concepts into the common entity model. Only three providers were implemented. Therefore, this extension path remains an architectural expectation and does not establish portability. Novel provider semantics could still require changes beyond the gathering layer, including modifications to the loader, schema, or query catalog.

Table 4.1 summarises the provider-specific implementations of the `Gather(ctx) (*GatherResult, error)` contract, where `GatherResult` contains the action and role slices. In `auto` mode, an error on the SDK path triggers a CLI attempt before failure is reported. These provider-specific `Gatherer`

implementations, together with the Heimdall CasC parsers described above, are the extension points that a fourth provider would need to implement. The shared processing described below is provider-independent and would not require changes.

Table 4.1: Data acquisition mechanisms for each provider-specific `Gatherer` implementation.

Implementation	Provider	Data acquisition
`aws_sdk.go`	AWS	External action catalog + `IAM.ListPolicies` + `IAM.GetPolicyVersion`
`aws_cli.go`	AWS	External action catalog + MAMIP sparse checkout
`azure_sdk.go`	Azure	`ProviderOperationsMetadata.List` + `RoleDefinitions.List`
`azure_cli.go`	Azure	`Get-AzProviderOperation` + local role snapshot (fetched with `Get-AzRoleDefinition` when absent)
`gcp_sdk.go`	GCP	`Roles.List` + `Roles.Get`
`gcp_cli.go`	GCP	`gcloud iam roles list` + `gcloud iam roles describe`

The gathering workflow writes each result to `{provider}_roles_actions.json` through the shared `SaveJSON` helper in `gather.go`. The resulting uniform schema contains a `metadata` block and the `actions` and `roles` arrays, allowing downstream loading steps to process every provider catalog identically regardless of the selected strategy.

Before saving the AWS output file, the workflow merges roles from the Heimdall export into the gathered catalog by reading inline policies from their `policies_inline` field. Wildcard action patterns within these inline policies are expanded against the gathered AWS action catalog. Merged roles are deduplicated by logical name, which is also stored as their role ID to match the normalised assignment references.

4.1.3 Azure Identifier Handling

Boundary and role identifiers are handled differently across the pipeline: boundary Globally Unique Identifiers (GUIDs) pass through unchanged, whereas role identifiers differ between input sources and require explicit normalisation.

For boundaries, the pipeline retains Heimdall's GUIDs without modification, making no attempt to reconcile them with human-readable subscription names. Therefore, the gathering and loading stages rely on Heimdall to maintain consistent boundary identifiers and names without independent validation.

Role identifiers differ between input sources. In the gathered catalog, Azure role IDs use canonical slugs prefixed with `azure:`, transforming display names such as `Reader` into `azure:reader`. In contrast, Heimdall's CasC assignment records reference bare human-readable names (such as `Reader` or `dtRoleContributor`) without an `azure:` prefix or consistent casing.

On ingestion, `normalizeRoleRef` in the Go loader and `normalize_role_ref/3` in `loader.pl` reconcile these formats. References that already begin with `azure:` pass through unchanged, whereas unprefixed references are lower-cased, hyphenated, and prefixed with `azure:` before evaluating the

assignment join. This normalisation ensures consistent matching across both built-in and custom Azure roles.

Listing 4.1 shows representative records from selected Heimdall collections, including nested assignment provenance under `role_assignments[].source` and boundary approvers inside `boundaries[].approvers`.

Listing 4.1: Abridged, structurally faithful example derived from the anonymised Heimdall organisational export (`aws_heimdall.json`). Arrays contain one representative record.


```
{
  "metadata": {
    "provider": "aws",
    "exported_at": "2026-07-06T11:36:55Z",
    "schema_version": "2.0",
    "source": "anonymized"
  },
  "identities": [
    {
      "name": "henry.thomas6@demo.example.com",
      "type": "user",
      "unified_type": "human",
      "boundary": null,
      "attributes": {
        "access_levels": [],
        "user_type": "NORMAL",
        "team": "team-storage-enablers"
      }
    }
  ],
  "role_assignments": [
    {
      "role": "networkcontributoraccess",
      "principal": "henry.thomas6@demo.example.com",
      "principal_type": "user",
      "boundary": "acc-100000000191-staging",
      "boundary_type": "account",
      "effect": "allow",
      "condition": null,
      "source": {
        "type": "team_member_override",
        "detail": "kevin.hall7@demo.example.com",
        "file": "accounts/acc-100000000191-staging_DynatraceD1NORAM.json"
      }
    }
  ],
  "boundaries": [
    {
      "id": "acc-100000000191-staging",
      "name": "Staging Network Account",
      "category": "Business Supporting",
      "scope_type": "account",
      "parent": null,
      "approvers": [ "will.garcia@demo.example.com" ],
      "administrators": [
        {
          "name": "ulrich.hill5@demo.example.com",
          "type": "user",
          "admin_type": "admin"
        }
      ]
    }
  ]
}
```

4.2 Permission Catalog Aggregation

Wildcard expansion requires an enumerated action universe against which wildcard patterns can be evaluated. Expanding patterns such as `s3:*` into concrete permissions relies on matching them against the gathered catalog snapshot, making the resulting expansion only as complete as that catalog. Each provider requires a dedicated catalog-acquisition method because action discovery mechanisms vary across platforms.

4.2.1 Azure

Azure supplies separate control-plane and data-plane operations across its services. In the recorded corpus (as of 6 July 2026), the gathered catalog contains 24 033 operations and 840 role definitions. Figure 3.8 visualises the separate control-plane and data-plane subtractions, and Equation 4.1 defines the combination of the resulting permission sets.

$$\mathit{EffectivePermissions} = (A \setminus A_{\text{not}}) \cup (D \setminus D_{\text{not}}) \tag{4.1}$$

The `IsDataAction` property of each provider operation partitions the gathered catalog into distinct control-plane and data-plane lookup sets. The Go helper `ExpandWildcardSet` expands the pattern arrays against their respective sets, expanding `actions` and `notActions` against the control-plane set to produce A and A_{not}, and `dataActions` and `notDataActions` against the data-plane set to produce D and D_{not}. The workflow performs the two subtractions separately and combines the resulting operations into a single set of flat allow permissions, as the `not*` arrays represent exclusions rather than explicit deny rules.

4.2.2 GCP

GCP's IAM Admin API [45] lists 2 363 predefined roles covering 13 413 distinct permissions in the recorded corpus (as of 6 July 2026). The `Roles.List` method retrieves role references and etags using the basic view, but fetching per-role permission lists requires a separate `Roles.Get` call for each uncached or modified role. The SDK path dispatches these detail calls through a bounded pool of 20 workers. No timing benchmark was recorded for this step, so no numerical speedup claim is made.

GCP IAM bindings reference roles directly rather than wildcard permission patterns. The gathered permission catalog is therefore used to resolve and validate those referenced roles, without performing wildcard expansion.

4.2.3 AWS

AWS IAM exposes managed-policy documents but does not provide an authoritative action-enumeration API. To support wildcard expansion, both gathering strategies load an action list from the `aws-iam-actions-list` repository [42]. The SDK strategy retrieves managed-policy documents through `IAM.ListPolicies` and `IAM.GetPolicyVersion`, while the CLI fallback obtains managed-policy definitions from a sparse checkout of MAMIP (now known as IAMTrail) [43]. Both paths expand wildcard actions against the shared catalog using the Go `ExpandWildcard` helper before writing the result. The recorded corpus contains 25 983 AWS actions and 1 904 role definitions (as of 6 July 2026).

The SDK policy-version cache stores a hash of the loaded action catalog snapshot. An unchanged policy containing a wildcard pattern can expand into a different set of concrete permissions if the

action catalog changes. To prevent stale outputs, a change in the catalog hash invalidates cached policy expansions before they are reloaded. This mechanism tracks modifications to the local catalog snapshot. It does not independently detect new actions released by AWS.

4.2.4 Catalog Count Provenance

The reported action and role counts describe the committed, production-derived anonymised corpus. They do not represent current provider catalog sizes. Gathering routines log the number of actions and roles written to each `{provider}_roles_actions.json` file, and thesis figures are read directly from the machine-generated evidence registry for that committed corpus. Section 4.4 defines what "demo" denotes for this dataset.

4.3 Loading and Caching

Once the JSON feeds are written, the source corpus (as of 6 July 2026) contains 141 392 entity and edge records excluding permissions (63 429 actions, 5 107 roles, 14 505 identities, 57 104 assignments, 857 boundaries, 390 trust relationships) and 1 066 044 expanded permission records, totalling 1 207 436 core and permission records. This subsection concerns the Prolog baseline, where a fresh `fastrw` binary cache (Section 2.4.2) avoids repeated JSON parsing and normalisation by reusing serialised metamodel facts. However, binary decoding and fact assertion into Prolog still occur. No specific performance magnitude is claimed because the accepted benchmark does not isolate these initialisation stages.

The loader (`loader.pl`) implements a two-path strategy for each provider section using `load_provider_section/3`:

1. **Fresh cache.** If the source JSON exists and the section cache is at least as new, the loader reads the serialised fact list from `{provider}_{section}.fastrw` and asserts the facts.
2. **JSON parse.** If the JSON exists but the cache is missing or stale, the loader parses the JSON and asserts its facts, then collects those facts and writes a new cache file.

If the source JSON is absent, the cache is not used. The loader issues a warning and skips the section. Loading demo data is handled separately through the explicit `load_demo_data/0` entry point.

The top-level load is platform-driven. The predicate `metamodel:all_platforms/1` returns `[aws, azure, gcp]`, and `load_all_providers/0` starts one Prolog thread per platform. Each thread loads two sections: `scripts`, which carries actions, role definitions, and permissions from the provider catalog, and `heimdall`, which carries the organisational CasC facts. The split is reflected in the cache names: `aws_scripts.fastrw`, `aws_heimdall.fastrw`, and so on. This separation allows the loader to evaluate each cache independently against the modification time of its corresponding JSON file.

The per-platform threading and two-section split are implemented by `load_all_providers/0`, `load_provider/1`, and `load_provider_section/3` in `loader.pl` (Listing 4.2).

Listing 4.2: Cache-selection excerpt from `loader.pl`. When the source JSON exists, fresh caches are decoded into fact lists and asserted, whereas missing or stale caches trigger a fresh JSON parse and cache regeneration.

```
load_all_providers :-
    metamodel:all_platforms(Platforms),
    maplist([P, T]>>(thread_create(load_provider(P), T, [])),
            Platforms, Threads),
    maplist([T]>>(thread_join(T, S),
                  ( S = exception(E) -> throw(E) ; true )),
            Threads).

load_provider(Provider) :-
    data_dir(Base),
    format(atom(ScriptsFile),  '~w/~w/~w_roles_actions.json',
           [Base, Provider, Provider]),
    format(atom(HeimdallFile), '~w/~w/~w_heimdall.json',
           [Base, Provider, Provider]),
    load_provider_section(Provider, scripts,  ScriptsFile),
    load_provider_section(Provider, heimdall, HeimdallFile).

load_provider_section(Provider, Section, JsonFile) :-
    ( exists_file(JsonFile) ->
        cache_file_path(Provider, Section, CacheFile),
        ( cache_is_fresh(CacheFile, JsonFile) ->
            format('  [loader] ~w/~w: cache hit~n',
                   [Provider, Section]),
            fast_read_file(CacheFile, Facts),
            assert_fact_list(Facts)
        ;
            format('  [loader] ~w/~w: loading from JSON~n',
                   [Provider, Section]),
            load_section_from_file(Section, JsonFile, Provider),
            cache_section(Provider, Section, CacheFile)
        )
    ;
        format(user_error,
               '  [loader] WARN: ~w not found (skipping ~w)~n',
               [JsonFile, Section])
    ).
```

Table 4.2 summarises the source-corpus record counts by provider.

Table 4.2: Source-corpus record counts by provider, as of the corpus collection date, 6 July 2026. The 1 167 AWS team-bearing source records contain 30 distinct anonymised team names. Team-bearing records and expanded permissions are reported separately from the entity-and-edge subtotal. The core-plus-permissions minimum reported above includes expanded permissions but excludes team-bearing records.

Record type	AWS	Azure	GCP	Total
Identities	12 010	1 417	1 078	14 505
Role definitions	1 904	840	2 363	5 107
Actions (catalog)	25 983	24 033	13 413	63 429
Assignments	41 698	3 664	11 742	57 104
Resource boundaries	365	163	329	857
Trust relationships	390	0	0	390
Team-bearing records	1 167	0	0	1 167
Expanded permissions	789 099	118 357	158 588	1 066 044

This subsection describes the Prolog-side loading and MQI boot sequence as it operated when Prolog served as the production backend. In the current production server, GoStore handles query execution directly without starting a Prolog subprocess (Section 4.8).

Historically, the Go supervisor coordinated the startup through three stages:

1. **Configuration and file preparation.** Loading application settings and optionally refreshing the Heimdall and provider source files before engine initialisation.
2. **Prolog startup and data loading.** Spawning the Prolog process, opening the configured MQI connections, and using the primary connection to consult the entry module and execute the cache-aware data load once.
3. **Server startup.** Configuring shutdown handling, mounting the MCP handler onto the HTTP server, enabling periodic reload outside demo mode, and starting the HTTP listener.

The deployed ECS task refreshed the source data every 60 minutes without requiring a server restart, leaving sufficient time for a full gathering cycle. The historical MQI connection pool was protected by a read-write mutex. Reloading acquired exclusive access and blocked queries, preventing them from seeing partially replaced data.

Today, the `cmd/compare` comparison tool (Section 4.8.1) always loads GoStore, but when supplied with the `-prolog` flag, it exercises Prolog using a reduced version of the historical boot sequence. It starts a single-connection MQI pool, consults `prolog/nlp_main.pl`, performs the selected data setup once, and pre-warms the Prolog query cache before executing comparison queries. The reduced path does not run the full server lifecycle and has neither a periodic reload loop nor an HTTP listener.

4.4 Data Anonymisation

The live Heimdall export contains confidential employee email addresses, internal project names, and organisational structures. Distributing this raw data would violate the confidentiality constraints noted in Section 1.5, making public archival unviable. This anonymisation mechanism is discussed here to define the dataset referenced throughout the remainder of this thesis.

An anonymisation script converts the production JSON source files into the redacted corpus shipped with the repository (a private repository, not publicly accessible) and used for all figures and screenshots. The corpus is committed to the repository rather than existing only transiently in memory, at `codeV2/prolog/demo/{aws,azure,gcp}/`, as two file families per platform: `{provider}_heimdall.json` (organisational data) and `{provider}_roles_actions.json` (provider catalogs). In this context, "demo" denotes a distributable dataset. The corpus retains the complete provider catalogs and scale-relevant production facts and is not a minimal illustrative fixture.

4.4.1 Two-Phase Deterministic Mapping

The script operates in two phases. The scan phase reads the JSON files and registers the sensitive values selected for replacement in typed registries. Separate registries collect user email addresses, service account names, group names, boundary identifiers, custom role identifiers, team names, and installation identifiers, and CasC file paths are transformed subsequently using the resulting token mappings.

The finalisation phase sorts each registry alphabetically and assigns stable one-based indices. These indices then feed into category-specific formatting schemes and multiple fixed word lists to generate the replacement values.

The resulting mappings and query-relevant contents are deterministic because sorting ensures that the index assignments remain stable for a given input set. Although repeated runs on the same production export do not produce byte-for-byte identical files due to updated `exported_at` timestamps, the stable identifiers support reproducible queries and help keep screenshots reproducible.

4.4.2 Fake Value Format by Category

Each category has a distinct format that preserves the structural role of the original value and removes organisational or sensitive identifiers:

- **Human users.** Real email addresses (e.g. `john.doe@corp.example`) map to `first.last@demo.example.com`. The script combines 40 random/artificial first names with 38 random/artificial last names to yield 1 520 combinations, appending a numeric suffix after exhausting this grid.
- **Service accounts and service principals.** Names map to provider-specific patterns: AWS service identities use `adjective-noun-N`, Azure service principals use `adjective-noun-sp-N`, and GCP service accounts use `adjective-noun-N@demo-project.iam.gserviceaccount.com`.
- **Resource boundaries.** Scope type drives the output structure so that structural meaning is preserved across providers (e.g. accounts, subscriptions, and projects). The display name is derived from the same stable index, adjective, noun, and scope type to remain consistent with the identifier.
- **Custom and internal roles.** Roles explicitly marked as custom or not recognised as built-in or provider-managed are anonymised, mapping to provider-specific replacements, keeping their permission sets unchanged, and setting their potentially sensitive descriptions to `null`.
- **Teams, installations, and CasC paths.** Teams map to `team-adjective-noun` patterns, installation identifiers become `inst-demo-N`, and CasC file paths have matching known components replaced. Embedded known account identifiers are handled separately.

4.4.3 What Is Kept Verbatim

Provider action catalogs and the built-in or provider-managed role entries within `roles_actions.json` remain unchanged, but custom roles are anonymised as described above. The separate full Azure and GCP role-catalog files are copied without modification. Keeping these definitions and standard role names, such as `Contributor` or `AdministratorAccess`, identical ensures that role references within assignments resolve against the retained definitions.

4.5 Edge Cases and Special Handling

These cases require dedicated loader logic to handle specific data structures correctly:

- **Team-derived assignments.** Heimdall materialises team-derived access as assignments for individual principals, with the loader recording their `source.type = team` provenance. A `DENY` team access level suppresses ordinary team-derived assignments, but explicit member overrides and non-team grants remain effective. Team membership itself is loaded separately to support team-oriented queries.
- **Internal super-admin role.** Dynatrace's CasC corpus tags certain boundary administrators with `admin_type: "super_admin"`, which the loader records as metadata in `boundary_admin/4`. Neither Prolog nor GoStore treats this flag as an authorisation bypass. Prolog evaluates permissions through the normal action-expansion path, and GoStore's

`isAdminRole` heuristic is restricted to discovery and security reporting. Therefore, neither engine avoids large action expansions merely because a role is classified as administrative.

- **AWS trust representation.** AWS assume-role policies identify principals that may assume a role rather than granting direct cross-account access. The loader captures these directed edges in `trust_relationship/5`, and Prolog's `trust_chain/3` returns directly trusted principals. GoStore handles transitive path traversal separately. These relationships remain distinct from ordinary role assignments.
- **GCP deployment stages.** Both projects and folders can carry installation metadata, which the loader stores as an `installation_id` and stage (`DEVELOPMENT`, `HARDENING`, `PRODUCTION`, or `UNKNOWN`) in `boundary_installation/4`. GoStore retains this same metadata and exposes it directly in boundary details. This explicitly recorded metadata avoids the need to infer deployment stages from resource names.

4.6 Go–Prolog Communication: The Machine Query Interface Protocol

Historically, the Go service communicated with the Prolog runtime through SWI-Prolog's MQI [9], a length-prefixed Transmission Control Protocol (TCP) protocol that remains relevant to comparison and benchmark tooling. Rather than embedding SWI-Prolog through `cgo`, Go launched the Prolog engine as a separate child process. Communication proceeded over a local TCP socket using the MQI connection.

Figure 4.2 shows the startup handshake between the Go supervisor and the Prolog subprocess.

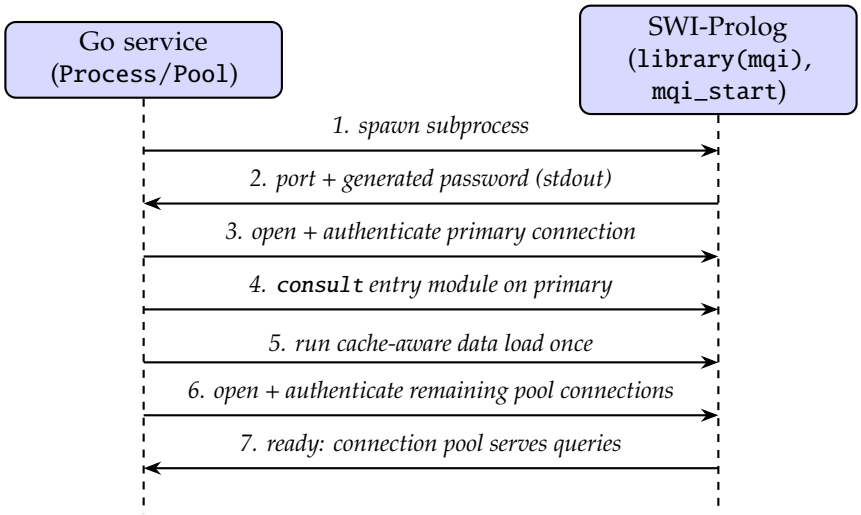


Figure 4.2: MQI startup handshake. The Go side spawns the Prolog subprocess, which loads `library(mqi)` and calls `mqi_start`. Go reads the auto-generated port and password from stdout, opens and authenticates the primary connection, consults the entry module, and loads the data once before opening the remaining pool connections. All connections share the resulting dynamic fact base.

In the historical ECS deployment, the Go supervisor launched a single shared Prolog subprocess and opened a pool of eight TCP connections, with incoming requests distributed across the pool.

Two dedicated goroutines continuously drained the Prolog subprocess's `stdout` and `stderr` pipes to prevent deadlocks during large query operations. If these output streams were not continuously consumed, Prolog would block on its next `format/1` call as soon as the observed 64 KiB pipe buffer filled.

This transport layer, its connection pooling, and its wire framing belong exclusively to the remaining Prolog comparison and benchmark tools. Production GoStore requests execute entirely in-process and never cross a process boundary (Section 4.8).

4.7 Performance Optimisations

Five narrower optimisations preceded the GoStore engine migration (Section 4.8). They addressed bottlenecks in startup, serialisation, concurrency, indexing, and response delivery. These measures retained the Prolog/MQI execution model and did not eliminate the underlying operational and aggregate-query costs. Table 4.3 summarises each optimisation: the problem it addressed, the mitigation applied, and its post-migration status.

Table 4.3: Prolog-era performance optimisations and their GoStore-era status.

Optimisation	Problem addressed	GoStore-era status
`fastrw` cache (Section 4.3)	Repeated JSON parsing and normalisation added startup work, but cache decoding and fact assertion still remained.	Superseded: GoStore parses JSON directly into Go structs, eliminating the separate Prolog-term construction stage.
Server-side pagination (`paginate_query_term/5`)	Large results incurred classification, serialisation, and transmission costs and could cause a stack overflow in the JSON writer.	Carried over: GoStore inherits the same offset/limit contract.
MQI connection pool (Section 4.6)	Multiple TCP connections to a shared Prolog process were required because a single connection would serialise concurrent requests behind one send/receive cycle.	Superseded: GoStore has no MQI transport. Concurrent reads share one in-process engine instance under a read/write lock instead.
Compact Full-Text Search (FTS) cache (`fts_token/3`)	Rebuilding the Prolog full-text index (tokenisation and phonetic codes) on every restart was slow.	Superseded: GoStore rebuilds its full-text search index from the loaded corpus during startup or reload and does not persist a separate Prolog-style token cache.
SSE result path	Emitting several paginated card batches per Explorer stream added flush overhead.	Superseded: GoStore fetches the full result in one in-memory call, so a single `cards` event replaces the batched ones.

The optimisations marked superseded above were resolved by replacing the Prolog serving path entirely, which the following subsection describes.

4.8 From Prolog Engine to GoStore: A Full Reimplementation

Prolog was initially useful because facts and query rules could be expressed and inspected within the same logic representation. Evaluating aggregate queries over 1 066 044 expanded permissions (Section 4.3) was the concrete driver for replacing the production Prolog path with GoStore. This

section covers the production migration to GoStore, the design alternatives considered, the validation methodology, and the benchmark results.

4.8.1 Alternatives Considered: Embedded Prolog and SQLite

We prototyped and evaluated two alternative in-process engine implementations alongside GoStore. An embedded SWI-Prolog engine linked directly into the Go process through `cgo` tested whether eliminating TCP/MQI transport while retaining Prolog's query logic was sufficient. SQLite tested whether a relational, in-process engine offered a simpler alternative to both Prolog and MQI. The benchmark harness in `cmd/benchmark` supports all three implementations. The registered quantitative comparison covers only GoStore and SQLite, while the embedded prototype is discussed through the integration problems it revealed.

Testing the embedded engine revealed two integration defects within the `cgo` wrapper path. Omitting the required `setup_demo_data` call caused the engine to run against an uninitialised dataset and return plausible-looking empty results. Similarly, failing to adjust foreign-call thread stack limits triggered a stack overflow when processing large result sets. Both defects were corrected.

These defects concerned integration wiring rather than the Prolog query predicates. In particular, the silent empty return was dangerous because it appeared deceptively valid instead of failing loudly. Retaining a second, lightly used execution path would require dedicated loading and stack-behaviour tests to detect regressions. That maintenance cost and correctness risk weighed against the embedded approach.

The registered quantitative comparison covers the 47-query catalog (excluding `search_keyword`, which uses a separate full-text engine) alongside dedicated point-query benchmarks, yielding 51 timed cases in total. Each timed case used 30 samples against the same AWS corpus in the benchmark environment described in Section 4.8.7. A separate result-count comparison between GoStore and SQLite confirmed zero count mismatches across the 41 applicable catalog queries, whereas the five verdict/object catalog queries and all five dedicated point queries were excluded from that count-based verification. The benchmark execution is recorded under evidence run `20260721T191513Z-e73c88c49ac8-enginecomp` (dataset hash `6eff501b2c0d`).

GoStore lost only on two cases (Table 4.4): GoStore required 818.869 ms versus SQLite's 694.207 ms on `privilege_escalation_risk`, and GoStore required 0.034 ms versus SQLite's 0.023 ms on `entity_info`. The margin on `entity_info` is sub-millisecond.

Peak memory differed as well (Table 4.4). Both measurements reflect the complete benchmarked engine behaviour rather than merely the intrinsic size of each data representation. The measurements differ in cache behaviour: GoStore's measurement includes its result cache being populated across the workload, whereas SQLite has no equivalent in-process result cache.

Although per-query routing to SQLite was considered, any candidate query shape would need to demonstrate more than a 2× measured speedup and preserve exact query semantics without scope simplification. Neither remaining SQLite win cleared this threshold.

The routing design is documented in `routing.go`, where `PreferredEngine` reads from the intentionally empty `sqlPreferred` map. Neither the production HTTP nor MCP service paths consult it during execution. Enabling hybrid routing in production would require integrating SQLite into reload handling, monitoring, and operational hardening. The empty routing map documents a possible future extension, but hybrid routing is neither active nor currently deployable.

The SQLite implementation built for the engine comparison already loaded the complete corpus into relational tables and could evaluate caller-supplied read-only `SELECT` queries. It was therefore

retained for a role outside the catalog benchmark: ad-hoc raw querying.

Read-only execution is enforced through two independent checks. The validator permits an optional trailing terminator but rejects any remaining semicolon, together with write, Data Definition Language (DDL), pragma, and transaction-control keywords. As a defense-in-depth measure, queries execute over a connection initialized with `PRAGMA query_only = ON`, ensuring that an unhandled edge case in the string filter cannot perform database writes.

This tool replaces the legacy Prolog raw-query sidecar (`iam_raw_prolog`), which previously relied on a persistent MQI pool and a spawned-subprocess fallback. Exposed through the existing MCP tool slot and the `/api/sql-query` endpoint, ad-hoc raw queries in production no longer spawn `swipl` subprocesses.

Memory efficiency did not drive the selection of GoStore, as SQLite had a lower measured footprint. GoStore was selected for its latency advantage despite the higher memory footprint. Retaining GoStore preserves the existing 18-predicate schema and 47-query catalog directly in Go without operating and maintaining the SQL catalog handlers as a second production engine.

Table 4.4: Trade-off summary for the three in-process engine candidates evaluated for the GoStore migration.

Engine	Transport	Query wins	Peak RSS (KiB)	Outcome
GoStore	in-process	49 of 51	3 072 228	Selected (production)
SQLite	in-process	2 of 51	782 468	Raw-query fallback only
Embedded Prolog (`cgo`)	in-process	—	—	Not selected: correctness/maintenance risk (defects fixed, later included in the four-way benchmark, Figure 4.3)

4.8.2 GoStore Architecture

GoStore is an in-memory typed store. The same conceptual entities represented as dynamic facts in the Prolog metamodel are held as Go structs in plain slices (`[]engine.Assignment`, `[]engine.Identity`, and related types), with one slice per predicate-equivalent record type. The engine ingests these entities directly from the same JSON source files used by the Prolog loader, keeping the conceptual data model intact.

To support fast query resolution without full slice scans, a fixed set of hash-map indexes is constructed once during loading. Primary maps include `assignsByPrincipal`, `permsByRole`, `boundaryByID`, `roleByID`, and `identityByNamePlat`. These are accompanied by derived reverse indexes, such as `trustsByTrusted` and `assignsByBoundary`, supporting cross-direction lookups directly.

Derived rankings that would otherwise require sorting the complete permission-count set are precomputed during loading. For example, top-N-by-permissions rankings are stored in `permCountSorted` and read directly by the corresponding query handler. This structure required no new data pipeline or schema redesign, as the 18-predicate conceptual model from Section 3 maps directly to Go struct types and index keys.

The join model differs fundamentally from Prolog's declarative resolution through goal conjunction, unification, and backtracking. GoStore instead implements each catalog query as an explicit Go handler that traverses typed slices and precomputed index maps directly, trading Prolog's concise declar-

ative representation for explicit, predictable access paths. The `find_identities_with_actions` query illustrates the cost: the Prolog version expresses the join between identities, roles, and expanded actions in under ten clauses, while GoStore requires explicit nested iteration over identity-role pairs combined with precomputed per-role action sets. More complex recursive or aggregate queries that would extend naturally from Prolog rules instead require corresponding additions to the explicit Go handlers and their indexes.

Query execution flows through a central entry point. `Store.Query` first inspects the result cache. On a cache miss, it calls the internal dispatch function, which acquires the store's read lock (`sync.RWMutex`). A compiled `switch` statement routes execution by `queryID` to the matching handler, enabling direct map lookups such as `permsByRole`. During a corpus update, the store acquires the write lock, preventing handlers from observing partially replaced data during a reload.

The result cache avoids repeated handler execution. Outputs from parameterless cacheable queries remain in the cache until a store reload occurs, whereas eligible parameterised query results use a five-minute Time to Live (TTL). A store reload clears the complete cache. A matching request can therefore return the cached response without traversing the indexes again.

Why GoStore reduces Prolog-serving overhead. Several architectural mechanisms explain the performance improvements observed relative to the Prolog implementation. The benchmark does not isolate their individual contributions:

- **Elimination of MQI transport overhead.** Relative to the MQI-based Prolog serving path (Section 4.6), GoStore executes within the same process and address space as the service handlers, avoiding a TCP socket, length-prefixed framing, and response parsing. This applies specifically to the MQI architecture, not to embedded execution or SQLite.
- **Explicit index lookups.** Point queries use direct map lookups, such as `permsByRole`, without unification and backtracking over Prolog's dynamic database clauses. Aggregate and discovery queries may still scan slices or combine multiple indexes, but point lookups avoid re-deriving joins at runtime.
- **Native result representation.** Query outputs are held as native Go structs from lookup through JSON marshalling, eliminating intermediate Prolog terms and their MQI term serialisation. JSON marshalling still occurs before response delivery, but GoStore removes the double-serialisation steps identified in Section 6.4.
- **Result caching.** The result cache accelerates repeated or warm requests by bypassing handler execution entirely.

These architectural explanations do not prove that GoStore is optimal for every query shape. SQLite achieved lower mean latency in 2 cases: `privilege_escalation_risk`, showing a query-planner-backed relational engine can remain competitive for a join-heavy aggregate query, and `entity_info`, where the difference was sub-millisecond. The empirical evidence therefore supports selecting GoStore for the current catalog and corpus, not that hash-indexed Go structs are always faster across all access patterns.

4.8.3 Migration Scope

The structured 47-query catalog and the 8-check security report were ported directly to GoStore. The DCG/NLP pipeline was not ported: production natural-language access instead uses the MCP server paired with an LLM (Section 5.5 explains why it superseded the pipeline documented in Section 5, which remains implemented but is no longer the production path).

Full-text search moved from the Prolog `fts.pl` index to Bleve [46], a full-text indexing and search library for Go. The maintained Prolog implementation remains available through `cmd/compare` and the benchmark tooling described in Section 4.8.1. The sandboxed SQL tool replaced the Prolog raw-query sidecar.

For reproducibility, the accepted evidence bundle records the source snapshot associated with each benchmark run. The decisions are summarised below.

- **DCG/NLP pipeline, retained in Prolog** (`grammar.pl`, `rules.pl`, `phrases.pl`). Production natural-language access uses the MCP server with an LLM, and the deterministic DCG pipeline remains available as the documented research implementation but is not part of the GoStore production path.
- **Full-text search, replaced by Bleve** (`fts.pl`). Bleve rebuilds from the loaded corpus on startup and reload. Its default search combines a boosted title prefix, fuzzy title matching, and subtitle matching, glob mode uses wildcard queries, and substring search is the fallback when Bleve is unavailable or a search fails. The Prolog phonetic tier was not ported.
- **Structured query catalog, ported to GoStore** (47 queries). Query IDs and parameter schemas were retained, with count parity confirmed except for `search_keyword` (Section 4.8.4), and the wider matched experiment favoured GoStore overall, with query-dependent variation.
- **Security report, ported to GoStore** (8 checks), implemented as explicit Go aggregations and filters over the loaded indexes.
- **Raw-query MCP tool, replaced.** `iam_raw_prolog` was replaced by `iam_raw_sql`, since SQLStore already loads the corpus and provides sandboxed, in-process read-only SQL without a Prolog subprocess.
- **Web UI, MCP structured-query tools, gathering, Heimdall ingestion, and JSON source files, interfaces retained.** These retain their contracts as the engine changes behind query dispatch. The raw-query tool is the documented exception above.

4.8.4 Correctness Validation

The `cmd/compare` tool evaluates count-level parity by executing every catalog query serially against both the Prolog implementation and GoStore using the same corpus, standardising the item-level verification already done during initial GoStore development across the complete catalog at the count level. AWS produced 46 matching result counts out of 47 queries, Azure 46 out of 47, and GCP 46 out of 47. In each case, the sole non-matching query is the same documented, expected discrepancy in `search_keyword`, caused by differing text analysers and ranking rules between Prolog's custom full-text index and GoStore's Bleve index, with 0, 0, and 0 unexpected differences beyond it for AWS, Azure, and GCP respectively.

This scorecard establishes count equality for the 46 non-search queries on each platform under the sampled parameters. During defect investigation, selected queries were compared by normalised keys. However, no exhaustive item-level comparison was run across all 141 query-platform combinations. Equal result counts can therefore still conceal differences in the returned records.

4.8.5 Historical Prolog Defects Found and Fixed

Parity comparison runs revealed three classes of Prolog-side defects, all of which were subsequently resolved:

- **Pattern case mismatch.** In `privilege_escalation_risk`, evaluating mixed-case curated patterns against lower-cased loaded actions produced false negatives. Both curated patterns and loaded actions are now normalised to lowercase before comparison.
- **Admin classification and trust filtering.** Admin-role classification produced mismatched counts, and trust queries did not consistently apply the requested platform filter. The trust lookup now receives and applies that platform parameter.
- **Over-broad file cache keying.** The precomputed file cache selected list-query files using only the query ID and platform, ignoring active filters such as `identity_type` and `role_kind`. For example, an AWS human-identity request returned all 11 199 cached identities instead of the smaller human-identity-only subset. File-cache eligibility is now restricted to default-filter requests, but non-default parameters use the full-parameter in-memory cache key. Regression tests cover the affected identity and role filters.

Following these fixes, both engines report matching sampled counts for the 46 non-search queries on each platform, including 256 999 AWS privilege-escalation identity/action pairs. Both engines also represent synthetic `svc_<name>` roles consistently as the internal join point for service-account inline policies.

4.8.6 Historical GoStore Defects Found and Fixed

Comparison and harness runs identified seven GoStore-side defects across four distinct categories, all of which were resolved before their respective final evidence runs:

- **Ignored query parameters.** The `redundant_assignments` query ignored its identity parameter, and `identity_groups` ignored its platform filter.
- **Incorrect aggregation.** The `team_role_assignments` query counted duplicate source records, and `cross_platform_summary` omitted known but unassigned platforms.
- **Incomplete query semantics.** The initial port of `duty_separation_violations` implemented only one of three specified detection paths, and `action_provenance` deduplicated results without considering the permission effect.
- **Incomplete result-cache key.** The production-memory harness revealed that result-cache keys omitted pagination and `count_only` parameters, allowing a bounded request following an unbounded request to receive the full cached response. Response-shaping parameters are now included in the cache key, and a dedicated regression test covers this sequence.

Following these corrections, the final duty-separation counts agree between engines across all platforms: 563 on AWS, 231 on Azure, and 141 on GCP. As with the catalog scorecard (Section 4.8.4), this establishes count equality only, not item-level agreement.

4.8.7 Benchmark Results

Benchmark environment. All local system-performance benchmarks reported in this chapter and Chapter 6 ran on a company laptop with an Intel Core Ultra 7 265H (16 cores and 16 logical processors: six performance, eight efficiency, and two low-power efficiency cores; 2.2 GHz base frequency, with advertised maximum turbo frequencies of 5.3 GHz for the performance cores and 4.5 GHz for the efficiency cores), 64 GB of DDR5-6400 memory, and a 1 TB Kioxia XG10d SED NVMe SSD. The benchmarks ran inside Ubuntu 24.04.4 LTS under WSL 2 (Linux kernel 6.6.87.2-microsoft-standard-WSL2), with 31 GiB of memory and 8 GiB of swap available to WSL. Windows used the Balanced power plan. The software versions were Go 1.26.5 and SWI-Prolog

10.0.2. Benchmark clients and servers ran within the same WSL instance, so the HTTP and MQI measurements include local protocol overhead but no external network latency. The catalog-stress runs additionally used the per-run cgroup limits stated below.

The accepted evidence run executed the engines serially on the same AWS corpus and recorded 10 samples per engine and query as a narrow two-query baseline. In this baseline, GoStore produced a mean latency of 0.044 ms for `trust_depth` and 861.992 ms for `security_report`. Prolog's corresponding means were 6.925 ms and 4318.180 ms.

The wider matched catalog experiment covered all 47 query IDs, evaluating Prolog-to-GoStore latency ratios across the broader catalog. This experiment reported case-level median ratios of 3.20 cold and 3.07 warm. Its broad interquartile ranges (13.02 cold and 35.97 warm) demonstrate substantial variation across query shapes, showing that a single speedup ratio does not represent every query.

The narrow serial baseline measured the complete process tree, not only the Go heap. Peak process-tree RSS was 2853256 KiB for GoStore and 1697836 KiB for Prolog, so GoStore used 1.681 times as much memory in this narrow comparison.

The separate catalog cgroup experiments measured serving topologies under the same 12 GiB, swap-disabled limit at concurrency 16. The median overall peak was 2.712 GiB for the Go-only HTTP process and 7.755 GiB for active pool-16 protected-run Prolog. Here, *active* denotes an MQI pool connection currently serving requests, and *standby* denotes an additional, fully loaded pool connection kept idle in reserve. The historical ECS deployment kept such a standby connection to support hot reloading of a second data set while the active connection kept serving requests, to recover cleanly from occasional Prolog process crashes, and to leave the door open for load balancing across two instances, a use case that was never implemented before the GoStore migration replaced this topology entirely. Adding a fully loaded standby raised the protected-run Prolog median to 8.386 GiB.

The separate Go Explorer SSE experiment used a distinct Go-only workload without a Prolog sidecar, reaching a peak of 4.323 GiB and returning 0.330 GiB decoded across 16 distinct requests.

These figures describe complete serving topologies, not a controlled, language-only comparison. The shared HTTP catalog run and the SSE transport run allocated different response representations, and the Prolog connection pool and loaded standby deliberately added serving capacity. As in Section 7.3, the historical 8.1 GB ECS incident is likewise a separate operational event and must not be treated as cross-engine evidence.

A four-way comparison subsequently extended this picture by benchmarking all four engine implementations across all 40 query identifiers on the same AWS corpus (Figure 4.3). GoStore and SQLite ran 30 iterations each, while Embedded Prolog and MQI-Prolog ran 10 and 5 iterations respectively. Iteration 0 served as a cold start and was excluded from the warm statistics reported below. Three queries were initially excluded from the MQI-Prolog run because their existing interface serialises the full result before returning it, but a retest on the development laptop confirmed they complete successfully with sufficient process-level stack space. All 40 queries are therefore included for all four engines.

Two families of query behaviour emerge. The analysis queries, `security_report`, `privilege_escalation_risk`, `duty_separation_violations`, and `actions_for_role`, each require substantial derivation. GoStore and SQLite complete `security_report` in roughly 630–710 ms by executing compiled Go against pre-indexed structs, versus approximately 2.3 s for both Prolog engines. The gap widens for `privilege_escalation_risk`: roughly 600 ms

for GoStore and SQLite against approximately 34 s (embedded) and 49 s (MQI) for Prolog, though those two Prolog figures rest on only 10 and 5 iterations respectively, fewer than GoStore and SQLite's 30, so Figure 4.3's per-iteration spread should be read alongside them. `duty_separation_violations` follows the same pattern, with GoStore's result-cache hit under 0.001 ms against multi-hundred-millisecond and multi-second times for SQLite and Prolog respectively.

The catalog count queries behave oppositely. All 35 queries that go through the `count_only` path exhibit Prolog query-cache hits from iteration 1 onwards. For these queries Embedded Prolog returns counts in 0.1–0.6 ms and MQI-Prolog in 0.6–2.6 ms (adding TCP round-trip overhead). GoStore CatalogCount is sub-millisecond for essentially all catalog queries. One exception illustrates the cache effect: `boundary_identity_density` takes 5.8 ms on GoStore but only 0.3 ms on Embedded Prolog and 1.0 ms on MQI-Prolog, because the Prolog engine's query cache stores the aggregated result after the first iteration while GoStore recomputes the value on each call.

The MQI transport overhead is quantifiable from this data. Catalog count queries use the `count_only` path, which returns a minimal count response without serialising the full result list on the Prolog side. Consequently, the difference between Embedded (median 0.21 ms) and MQI-Prolog (median 1.07 ms) for cached catalog queries isolates approximately 0.9 ms of per-query TCP round-trip and connection-framing cost, with no full JSON serialisation on either side.

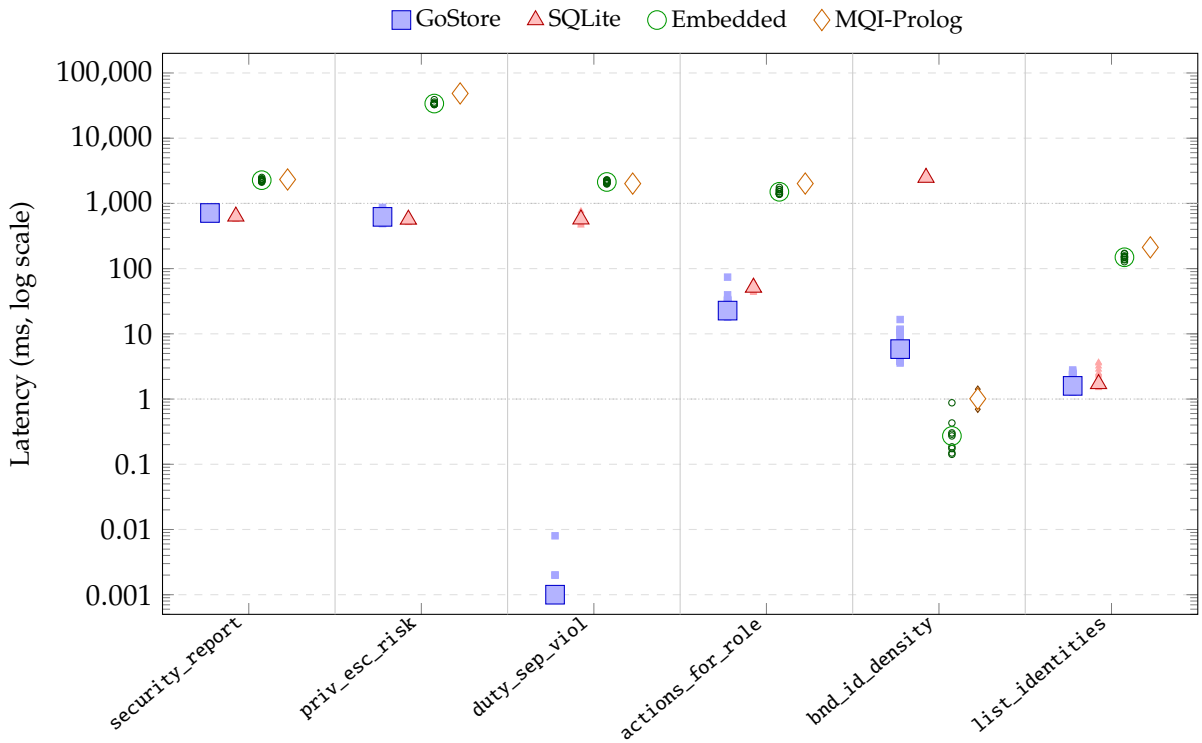


Figure 4.3: Four-way engine comparison on the AWS corpus: six representative queries on a log scale. Large marks show the warm median (p50), and small faint marks show every warm iteration. Iteration counts differ by engine: 30 for GoStore and SQLite, 10 for Embedded Prolog, 5 for MQI-Prolog. Shapes and colours both identify engines for legibility in greyscale. Dotted reference lines mark 1 ms and 1 000 ms. `duty_sep_viol` for GoStore is below 0.001 ms after the cold start (result-cache hit) and shown at the floor.

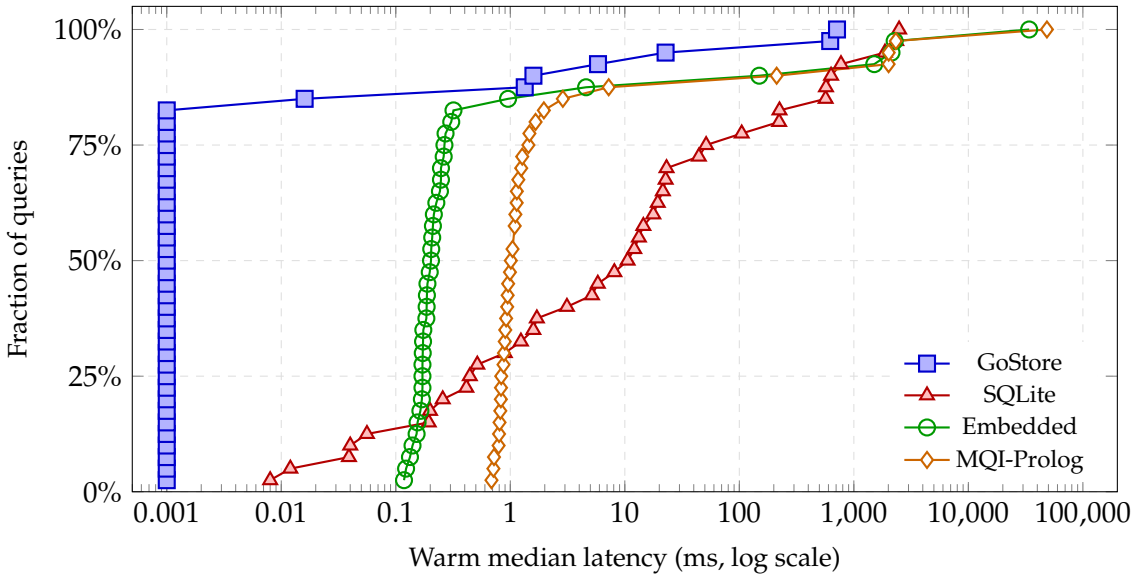


Figure 4.4: Cumulative distribution of warm median latency across all 40 benchmarked queries per engine (log X axis). Each mark is one query's warm p50, sorted by latency, and the Y axis shows the fraction of queries at or below that latency. GoStore completes 83% of queries in under 0.001 ms (result-cache hits from the CatalogCount path), and the remaining 17% are the analysis queries that require full derivation. Embedded Prolog completes 85% of queries under 0.35 ms (Prolog query-cache hits via the `count_only` path) before jumping to seconds for analysis queries. MQI-Prolog shows the same two-regime shape, shifted right by approximately 0.9 ms due to TCP round-trip overhead per query. SQLite has no query-level result cache and therefore distributes more gradually across the full range.

4.8.8 What This Means for the Thesis Claim

The measured speedup reflects the complete serving topology: GoStore executing in-process versus Prolog served through the MQI TCP boundary. The four-way comparison additionally isolated the MQI transport contribution. Embedded Prolog, which eliminates the TCP boundary by linking SWI-Prolog directly into the Go process via `cgo`, adds roughly 0.9 ms per catalog-count query relative to MQI-Prolog. Because catalog-count queries use the `count_only` path and therefore do not serialise the full result list on either side, this 0.9 ms represents TCP round-trip and connection-framing overhead rather than serialisation cost. For the expensive analysis queries, which do serialise their full result payloads via MQI, the transport cost is negligible relative to the derivation time.

The metamodel defined in Section 3 is independent of the engine that executes it. The 18-predicate conceptual schema was defined once and implemented twice: in SWI-Prolog as the research baseline, and in Go as the production engine.

Prolog was appropriate during the metamodel design phase because it kept the logic schema and derivation rules concise and legible. For the bulk-aggregation and reporting workload that the system evolved to serve, the measured comparisons favour GoStore. This conclusion is limited to the evaluated corpus and machine and does not apply to the engines in general.

The empirical comparison is a result in its own right: repeated cross-engine runs exposed defects in both implementations before the final serial run reached count parity (Section 4.8.4).

Figure 4.5 summarises the resulting architecture: both engines consume the same JSON source files, `cmd/compare` compares their result counts during validation, and GoStore alone serves production traffic.

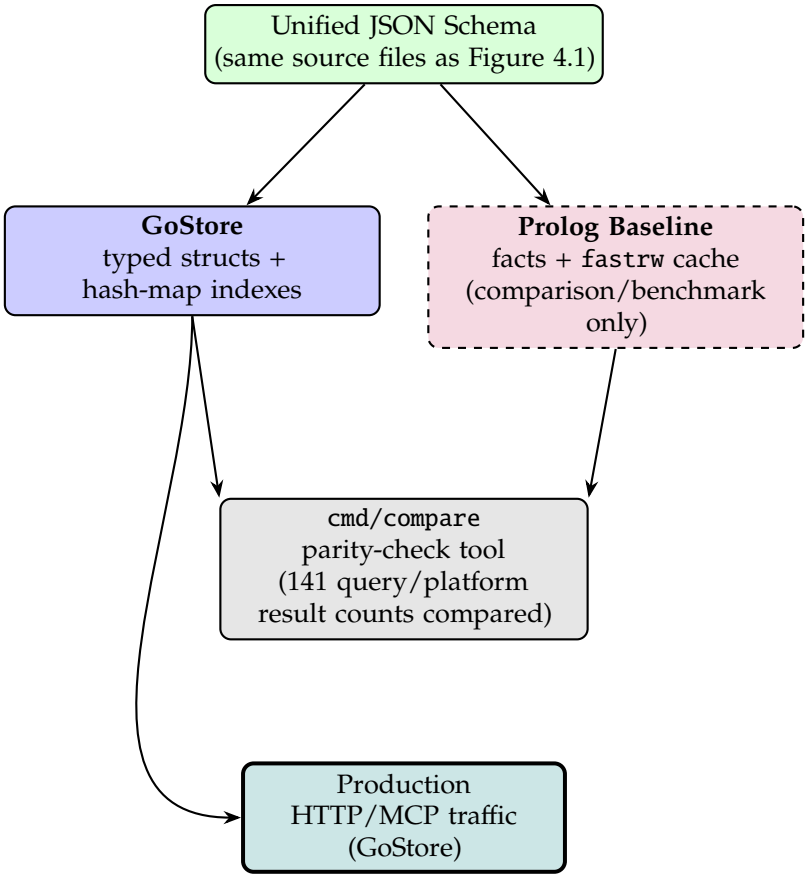


Figure 4.5: GoStore and Prolog comparison architecture. Both engines load the same JSON source files (Figure 4.1). The `cmd/compare` tool compares catalog result counts between the two engines, while production HTTP and structured MCP traffic use GoStore. Prolog remains available to the comparison and benchmark commands. Raw ad-hoc querying uses the sandboxed SQL tool described in Section 4.8.1 instead.

The production server has no engine-selection flag: GoStore is loaded unconditionally. `cmd/compare` starts Prolog separately for the correctness-baseline experiments in Section 4.8.4. No Prolog process runs in production any more, including for raw queries.

5 Metamodel Application and Querying

The generic metamodel (Section 3) is query-agnostic, separating core permission logic from concrete application interfaces. In production, human engineers using the Explorer UI or REST API and LLM agents using typed MCP tools all use the shared query catalog and execute its queries through the GoStore backend. The Prolog NLP grammar remains runnable for natural-language queries supported by its fixed grammar as a comparison baseline, though it is not registered as a production route.

This section details the seven advanced query patterns, the Go web application, the retained Prolog NLP interface, the Explorer UI, and the MCP integration. Section 6.6 in the evaluation chapter later reports an exploratory observational study comparing the MCP agent against an LLM reading raw CasC files directly.

Table 5.1 summarises which interfaces are part of the production path and which are retained research baselines, so that the status qualifiers used throughout this chapter do not require tracking across sections.

Table 5.1: Interface status overview for Chapter 5. Production interfaces dispatch through GoStore. The retained NLP baseline runs separately against the frozen Prolog engine.

Interface	Status	Engine	Notes
Explorer UI	Production	GoStore	Structured catalog queries, SSE streaming
REST API	Production	GoStore	OpenAPI, shared catalog dispatch
MCP server	Production	GoStore	6 grouped tools + raw-query fallback
NLP / DCG pipeline	Retained baseline	Prolog (frozen)	Separately runnable, not a production route

Effective-permission boundary. All query results reflect the gathered CasC state at the time of the last corpus reload. GCP hierarchy-scoped deny policies, AWS Service Control Policies, resource-based policies, and inline session policies are not collected by the gathering pipeline and are therefore absent from effective-permission computations (Section 1.5). A result of "access granted" or "no findings" indicates that the gathered role and assignment facts permit the action or find no violation, without accounting for these excluded enforcement layers. Operators relying on query results for access decisions should be aware that the model covers the gathered permission state, not the provider's complete runtime enforcement.

5.1 Advanced Query Patterns

The following seven examples are representative analyses selected to demonstrate different capabilities of the metamodel from Section 3. They exercise wildcard expansion, cross-platform unification, graph traversal, provenance inspection, privilege classification, and team-membership resolution. Some correspond directly to a production catalog query, while others describe internal helpers or map to broader catalog queries that subsume them. In each case, Explorer and MCP clients invoke a catalog entry without implementing the underlying joins, expansion, or traversal themselves. The metamodel was designed from its schema concepts rather than derived from these use cases.

1. Cross-Platform Summary. A security auditor wants the full cloud footprint of one iden-

tity in one place. The production catalog query `cross_platform_summary` aggregates boundaries, roles, assignments, allow-permission counts, and deny-permission counts for a selected identity across AWS, Azure, and GCP. The retained Prolog implementation dispatches to `find_cross_platform_identity_summary/2`. Calling the query for `'alice@company.com'` returns one record per platform where Alice is known: for example, an AWS record showing `AdministratorAccess` on `prod-eu` with its allow and deny counts, an Azure record showing `Contributor` on `sub-1`, and a GCP record showing `Viewer` on `project-x`.

Figure 5.1 shows the Explorer rendering of a `cross_platform_summary` query for a different demo identity. GoStore emits one result card for every platform on which the identity is known, each carrying its own boundary, role, assignment, and permission-count summary. A platform on which the identity does not exist produces no card.

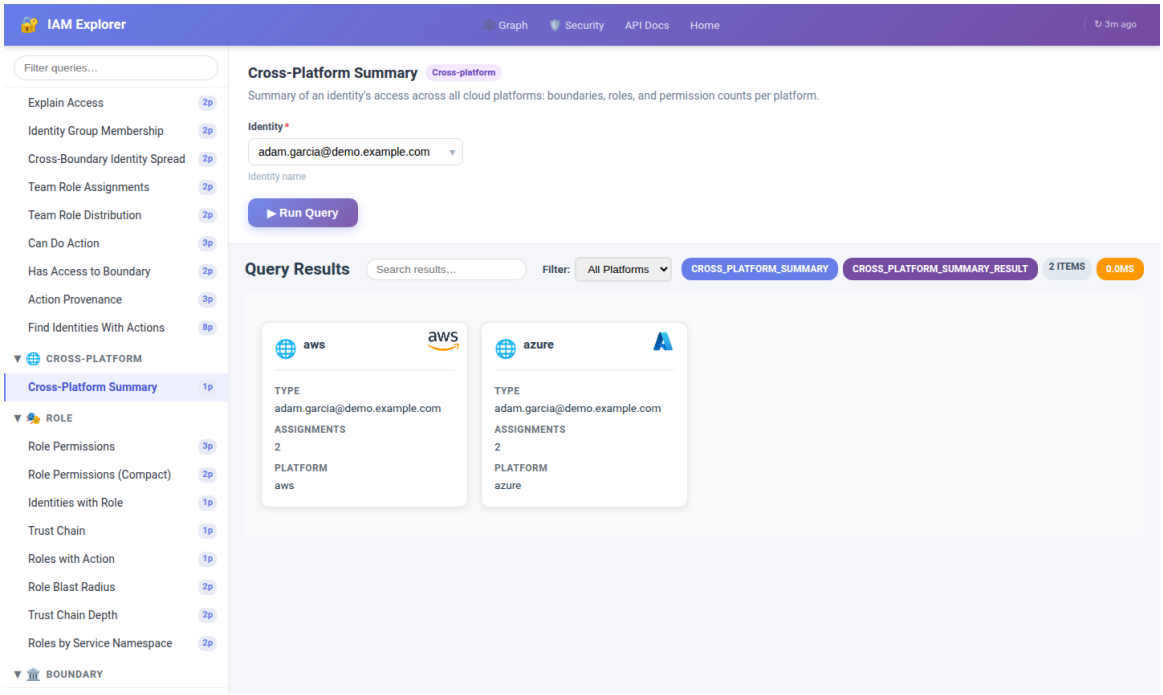


Figure 5.1: Explorer result of a `cross_platform_summary` query for one demo identity. The identity is known on AWS and Azure, so the result contains one card per platform with boundary, role, assignment, and permission counts. No GCP card appears because the identity does not exist on that platform.

2. Permission Comparison. When onboarding an engineer, the question is what access drift exists relative to a peer. The retained Prolog helper `compare_permissions/4` computes the common, only-in-first, and only-in-second sets of unique allowed, non-denied concrete actions for two identities on one platform. Calling it as `compare_permissions('alice', 'bob', aws, Comparison)` yields the exact AWS actions both share, the actions only Alice has, and the actions only Bob has, expanded to individual API calls.

The production interface uses the broader `compare_any` catalog query, which supports multiple entity types and modes including identity-permission and role-assignment comparisons. Figure 5.2 shows two demo identities compared in `role_assignments` mode, reporting coverage percentages and the only-in-A, shared, and only-in-B split at the role-assignment level rather than at the individual action level.

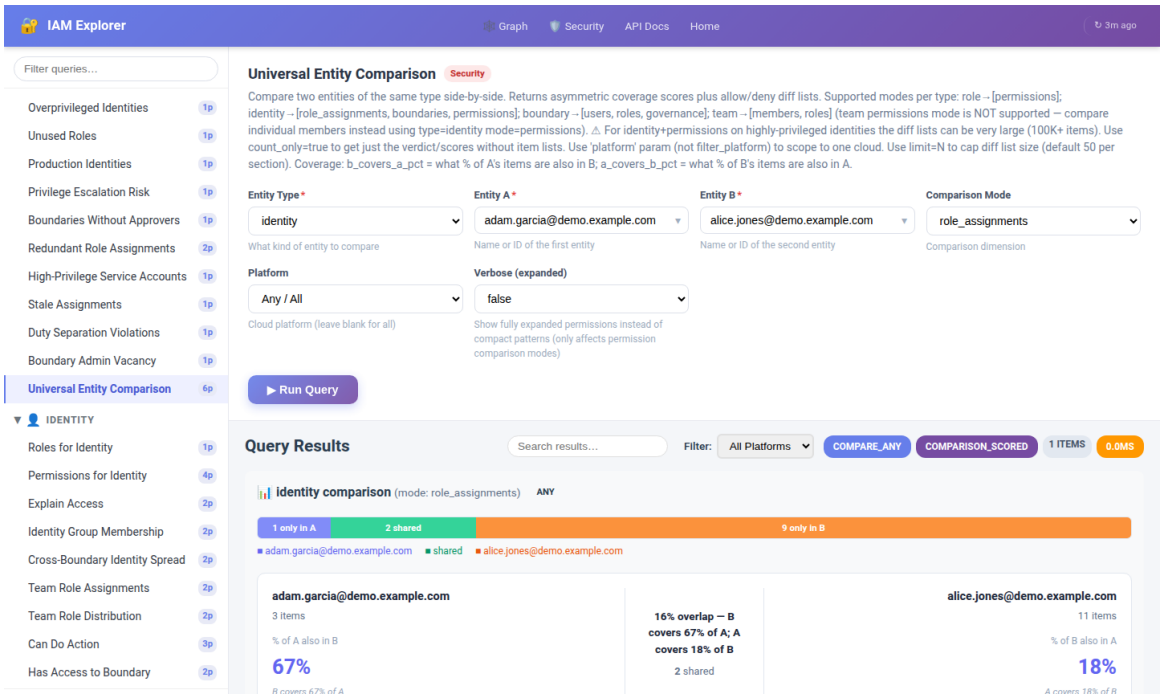


Figure 5.2: Explorer rendering of the `compare_any` query for two demo identities in `role_assignments` mode. The result shows coverage percentages and the only-in-A, shared, and only-in-B split. Permission mode compares at the individual action level, while other modes compare their respective units such as role assignments, boundaries, or governance attributes.

Patterns 3 through 7 map each scenario to its corresponding production catalog query and underlying Prolog predicate, as summarized in Table 5.2.

Table 5.2: Advanced query patterns 3 through 7, detailing the associated production catalog query, retained Prolog predicate, and primary use case for each pattern.

Pattern	Production Query	Retained Prolog Predicate	Use Case
Wildcard Coverage	None (internal helper)	`expand_action_pattern/3`	Expose destructive actions hidden behind wildcards such as `Microsoft.Compute/*`
Trust Chain	`trust_chain`	`trust_chain/3`	Trace cross-account assume-role pathways
Overprivileged Identities	`overprivileged_identities`	`find_overprivileged_users/1`	Flag identities holding administrative roles across multiple boundaries
Why Does X Have Access?	`explain_access`	`explain_user_access/3`	Trace access grants back to their originating CasC definitions
Team Membership	`team_members`	`list_team_members/2, /3`	Resolve the members and owners belonging to a team

Several implementation details complement the summary in Table 5.2. While the base Prolog predicate is non-recursive, the production `trust_chain` query executes a recursive traversal limited to five edges, detects cycles, and returns maximal inbound and outbound paths. The

`overprivileged_identities` query identifies administrative roles heuristically using keyword matching for terms such as `admin`, `owner`, `root`, `superuser`, or `full access`, so its output is a heuristic review signal rather than formal verification of excessive privilege. The `explain_access` query yields structured provenance records instead of a binary decision, and team analysis is expanded by `team_role_assignments` (role assignments held by team members) and `team_role_distribution` (how those roles are distributed across the membership).

The metamodel incorporates organisational team membership into standard RBAC [18]. Because the core user–role–permission model lacks a native team or group construct, role assignments remain attached to individual members rather than to the team as a distinct entity, and the Team Membership pattern above resolves membership through the retained `list_team_members/2, /3` predicates. The dataset coverage and platform distribution of team records are described in Section 3.3.1.

5.2 Go Web Application

To avoid requiring users to learn an engine-specific query language, three structured production interfaces share a single GoStore backend: the Explorer UI, a structured REST API, and the MCP server (Section 5.5). All three rely on a central catalog registry that defines query identifiers, parameter schemas, and descriptions. However, registering a query only establishes its metadata. Execution additionally requires a corresponding dispatcher implementation within GoStore.

The server renders the initial HTML templates and a small vanilla-JavaScript client builds catalog-driven forms that users fill and submit as a JSON `POST` to `/explorer/stream`. The Go handler executes the selected catalog query in GoStore and, for normalised results, emits an `init` event containing the result shell, a single `cards` event with all normalised items (to minimise network flushes), and a `done` event. An `error` event is emitted if loading or query execution fails. The client buffers the received cards, initially renders up to 50 items, and then establishes virtual pagination over the buffered result set. For result sets exceeding 1 000 items, active search and platform filtering can be offloaded to a Web Worker, whereas smaller sets are filtered synchronously. Highlighting remains on the main thread.

For example, the `Identities with Role` query can generate large result sets. Specifically, for `PlatformDeployerAccess`, it returns 4 220 distinct identity–role–boundary results after duplicate source rows are collapsed. The Explorer presents each result as an individual card and paginates them in groups of 50.

The repository's OpenAPI 3.0 specification documents the 47-entry query catalog and related operations: catalog discovery and structured-query execution, autocomplete, graph and security-report APIs, health and reload operations, and the optional raw SQL fallback endpoint. The specification is maintained by hand alongside the route table and can drift from the implementation if either is edited without the other. Figure 5.3 shows the specification rendered by the served Swagger UI.

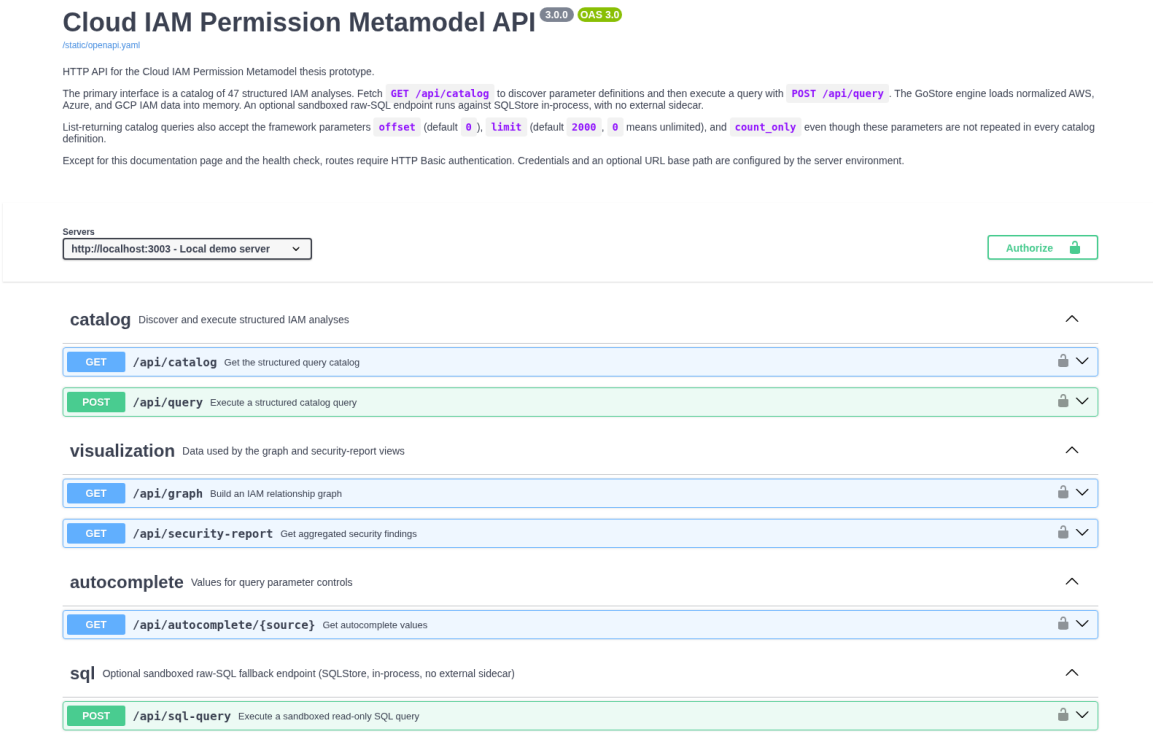


Figure 5.3: Swagger UI rendering of the OpenAPI 3.0 specification served by the demo server with GoStore loaded. The page documents the 47 structured catalog operations and marks the optional raw SQL endpoint as a compatibility fallback.

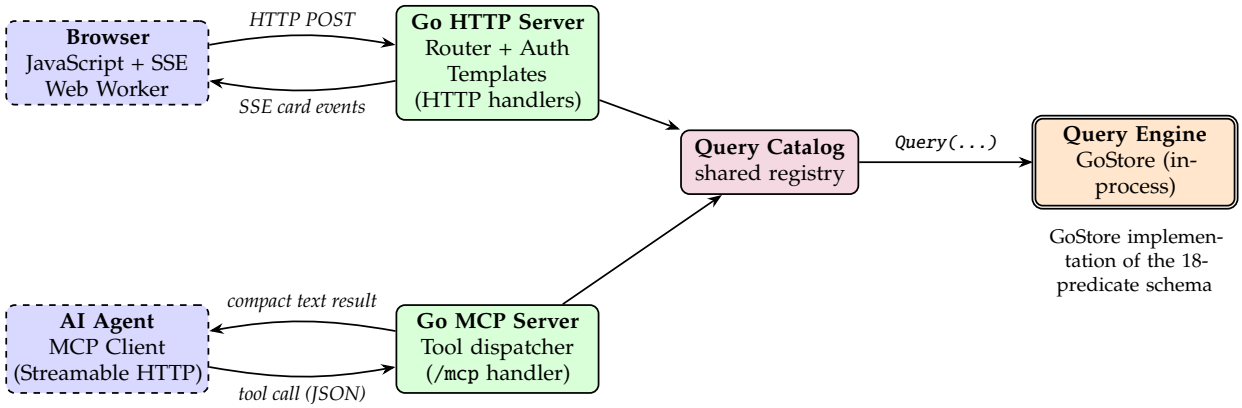


Figure 5.4: Web application architecture. The Explorer (browser HTTP handlers) and the agent-facing MCP handler consult a shared query catalog and execute eligible structured calls in the same in-process GoStore instance. The separate raw-SQL fallback path (SQLStore) is omitted from this diagram. No Prolog sidecar participates in the production execution paths.

Figure 5.4 focuses on Explorer and MCP request paths: the HTTP handlers and the `/mcp` handler run in the same Go process and dispatch eligible structured calls through the shared query catalog to the same in-process GoStore instance, while the raw SQL fallback (SQLStore) operates outside this structured catalog and is omitted from the diagram.

The catalog registry, shown in Listing 5.1, is a Go slice of `QueryDef` entries. Each entry carries an

identifier, human-readable name, category, parameter schema, descriptive text, and a required Prolog goal template with %s placeholders. The current registry contains 47 named queries spread across six categories: discovery, security, identity, cross-platform, role, and boundary. Explorer consumes this metadata to construct forms and validate input, HTTP handlers use it for request-time parameter validation, and the MCP server generates tool schemas directly from registered entries.

Selected entries marked `NoMCP` are excluded from MCP tool exposure. GoStore supplies the execution behaviour for each query through its dispatcher implementation. Contract tests exercise representative GoStore queries and verify selected execution behaviour and result structures.

Listing 5.1: Query catalog registry (`catalog.go`), excerpt. `QueryDef` carries the identifier, name, description, category, parameter schema, `NoMCP` flag, and the retained Prolog goal template. Explorer constructs forms, HTTP handlers validate parameters, and the MCP server constructs tool schemas excluding entries marked `NoMCP`.

```
// QueryDef describes a single curated analysis query.
type QueryDef struct {
    ID          string     `json:"id"`
    Name        string     `json:"name"`
    Description string     `json:"description"`
    Category    string     `json:"category"`
    Parameters  []ParamDef `json:"parameters"`

    NoMCP bool `json:"no_mcp,omitempty"`

    prologGoal string
}

// Registry is the master list of all curated queries.
var Registry = []QueryDef{
    {
        ID:          "cross_platform_summary",
        Name:        "Cross-Platform Summary",
      Description: "Summary of an identity's access across all cloud platforms:
                boundaries, roles, and permission counts per platform.",
        Category:    "cross-platform",
        Parameters: []ParamDef{
            {Name: "identity", Label: "Identity", Type: ParamTypeString,
          Required: true, AutocompleteSource: AutocompleteIdentities,
          Description: "Identity name"},
        },
      prologGoal: "nlp:structured_query_json(cross_platform_summary, json([identity=%s
                ]), JsonString)",
    },
    // Remaining catalog entries omitted.
}
```

5.3 Natural Language Interface

Some intended users require a natural-language interface rather than learning Prolog or the structured catalog. The retained SWI-Prolog NLP pipeline resolves supported natural-language phrases into queries over the same metamodel data. It is a separately runnable, deterministic SWI-Prolog implementation and is not registered as a production HTTP route. Production natural-language access is provided by the MCP server paired with an LLM.

The pipeline has five deterministic layers, uses no embedding model, and makes no remote inference calls.

1. **Tokenisation.** The input string is split into a list of atom tokens.
2. **Fuzzy matching.** `match_word/2` applies tiered comparisons when a grammar or vocabulary rule compares an input token with an expected word. Expected words of two characters or fewer permit only an exact case-insensitive match. Longer expected words use the following fallback order: exact match, Porter stemming [47], Double Metaphone phonetic key [48], and iSub string similarity [49] with a score strictly greater than 0.85 (selected empirically during development to balance false positives against false negatives on the controlled-language query corpus, with no systematic calibration study performed).
3. **DCG grammar.** 40 exported non-terminal signatures implemented by 48 clauses collapse phrasing variants ("who can", "which users can", "is anyone able to") into abstract phrase categories. Multiple clauses may implement one signature.
4. **Query rule selection.** 41 top-level `query//1` clauses pattern-match the parsed structure to a query type and bind the result variable. The double-slash notation (`name//arity`) is standard Prolog DCG convention and distinguishes grammar non-terminals from regular predicates (`name/arity`). These comprise 40 built-in clauses and one clause that delegates to installed plugins.
5. **Execution and serialisation.** The NLP entry point, `nlp_query_json/2`, invokes `rules:query//1` to produce a result term, which `result_to_json_string/3` then classifies, normalises where applicable, and serialises to JSON.

The separate non-NLP structured interface uses `structured_query_json/3`. Both paths share the same outer response-envelope structure, though their result payloads may be either normalised items or dedicated JSON shapes.

Table 5.3 illustrates representative natural-language inputs and the query types the DCG parser resolves them to. The NLP search bar that originally served these queries was removed in the GoStore migration (Section 4.8).

Table 5.3: Representative accepted natural-language inputs, the query functor selected by the DCG parser, and the nominal result type returned by `nlp_query_json/2` for a successful, non-empty result.

Natural Language Input	Query Type	Nominal Result Type
"who are the admins"	`admins`	`identity_list`
"can alice delete on azure-sub-1"	`can_do`	`boolean`
"what actions does AdministratorAccess have"	`actions_for_role`	`role_actions_detail_list`
"list users in aws"	`platform_identities`	`identity_detail_list`
"show trust chain for role dev-role"	`trust_chain`	`trust_chain_list`
"members of cloud-team"	`team_members`	`team_members_list`
"find overprivileged users"	`overprivileged`	`overprivileged_list`
"compare alice and bob"	`compare`	`comparison`

As the table illustrates, each supported phrasing maps deterministically to one query, which gives the pipeline predictable behaviour but limits coverage to the pre-defined patterns. The grammar defines a fixed set of phrase patterns. It does not parse arbitrary free-form English. Inputs outside the grammar's coverage fall back to the retained Prolog FTS implementation in `modules/nlp/fts.pl`, which is separate from the production Explorer's Bleve-backed GoStore index described in Section 5.4.

The DCG grammar was chosen because it makes phrase variants explicit and auditable in Prolog, which suited the research context. Adding a structured catalog query does not add a phrase to the grammar or rules files automatically.

The tokeniser, the 48 grammar clauses (`modules/nlp/grammar.pl`), and the 41 query-dispatch clauses (`modules/nlp/rules.pl`) are hand-written Prolog. The Porter stemming [47], Double Metaphone [48], and iSub [49] algorithms are applied through their SWI-Prolog standard library implementations (`library(porter_stem)`, `library(double_metaphone)`, `library(isub)`). The four-tier orchestration that sequences them is hand-written in `matching:match_word/2`.

5.3.1 Grammar Word Matching and Entity Resolution

"Fuzzy matching" in the grammar refers to word-level comparison through `matching:match_word/2`, which applies the exact/stem/phonetic/iSub four-tier sequence when grammar or vocabulary terminals compare an input token with an expected keyword (worked example in Table 5.4).

Entity resolution relies on type-specific DCG non-terminals (the double-slash notation is the same standard Prolog DCG convention noted above) that consume and resolve tokens during parsing. Identity parsing through `user_name//1` consumes a single token and requires an exact match against known identities in `identity/3`, performing no fuzzy matching. Action parsing through `action_verb//1` compares a single token against configured action synonyms using `match_word/2` or treats any non-reserved token as a literal action keyword.

Boundary parsing through `resource_name//1` accepts one to three tokens and delegates resolution to `resolve_boundary/2`. This helper evaluates candidates using a strict precedence sequence: wildcard matching over IDs and names, direct boundary ID lookup, exact case-insensitive display-name matching, and finally case-insensitive substring matching against IDs or names. Platform parsing uses the aliases defined in `vocabulary/platforms.pl`, with query rules that treat the platform phrase as optional supplying a default of `any` when no platform is specified.

5.3.2 Word Matching and Entity Resolution Examples

Table 5.4: Representative word-matching and entity-resolution examples. The iSub tier succeeds when its Stoilos string-similarity score [49] is strictly greater than 0.85. Exact, Porter stemming, and Double Metaphone tiers produce Boolean success rather than numeric scores.

User text	Candidate	Resolution mechanism	Reason
`destroy`	`delete`	action synonym	Configured verb mapping in `actions.pl`.
`administrtor`	`administrator`	Double Metaphone	Both produce phonetic code `ATMN`.
`users`	`user`	Porter stem	Plural collapses to singular.
`monitoz`	`monitor`	iSub > 0.85	Typo resolved after stemming and phonetic tiers fail.
`analytics-pr`	`analytics-prod`	boundary substring	Boundary resolver checks ID and name substrings.

On the Prolog side, three implementation safeguards keep large result sets from destabilising the retained baseline engine: a tail-recursive accumulator-based normaliser that avoids one stack frame per result item, pagination enforced before normalisation runs, and a minimum stack limit applied before query execution. The normaliser is shared by both the retained NLP and structured Prolog serialisation paths. The stack-limit prelude is applied by the MQI and embedded comparison tooling before executing Prolog queries. Pagination and the stack-limit floor are general Prolog-process safeguards summarised in Section 4.7.

Most list-producing queries normalise their results into a shared envelope: item type and platform, title, subtitle, and identifier, generic card fields, optional CasC provenance, optional drill-down links that submit follow-up structured queries, and optional relevance score and documentation URL. Generic normalised items use the shared card renderer.

Dedicated result shapes bypass this envelope and use separate template branches: Boolean answers, comparisons, grouped results, errors, and other skip-listed result types retain their own JSON structures and rendering paths. Listing B.3 in Appendix B gives the full Go type and an example JSON output.

Within the Prolog serialisation path, `classify_result/7` selects the result type and initial JSON representation. Normalisable list types are subsequently passed through `normalize_result_item/3` to produce the shared envelope. Skip-listed types retain their dedicated structures.

Figure 5.5 summarises how the tokeniser, fuzzy matcher, DCG grammar, query rules, and serialisation stages combine into the retained NLP pipeline, including the FTS fallback described next.

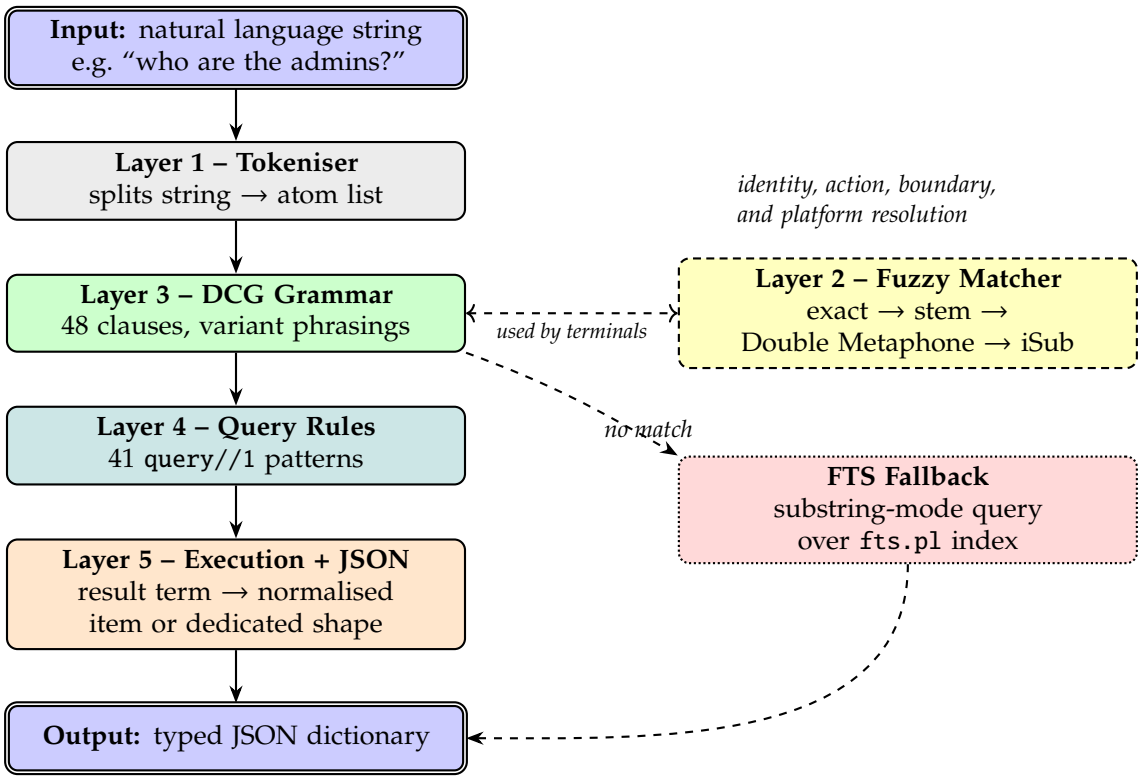


Figure 5.5: Five-layer NLP pipeline. Natural-language input passes through tokenisation, DCG grammar parsing that invokes the four-tier fuzzy matcher where needed, and query-rule selection before being executed against the retained Prolog metamodel. When the grammar produces no match, a substring-mode fallback over the retained Prolog FTS index returns results if matching documents exist.

For unmatched input, the fallback searches a retained Prolog FTS index built from six entity types: actions, role definitions, assignments, identities, resource boundaries, and teams. Permission-derived terms enrich role documents rather than forming standalone FTS documents. The NLP fallback invokes substring mode. If no documents match, `nlp_query_json/2` returns an unknown or error response. In production, the separate `search_keyword` structured catalog query uses a Bleve index and is accessible through the Explorer UI, while the `/fts-test` diagnostic endpoint calls GoStore's FTS functionality directly without invoking the Prolog NLP pipeline.

Testing uses a dedicated `plunit` suite, runnable with `make test-nlp` from the repository root. The tests use synthetic AWS, Azure, and GCP fixture data asserted directly into the metamodel rather than production data. Section 6.3 reports the suite's block structure, pass count, and coverage limitations (Table 6.1).

Listing 5.2 shows two representative `query//1` clauses and the `rest_as_role//1` helper. The grammar terminals such as `w(Expected)` and phrase-pattern matchers delegate token comparison to `match_word/2`, which runs the four-tier fuzzy matcher. Phrasing variants like "list admins" and "show all admins" therefore select the same `admins` query rule and execute `find_admins/1`.

Listing 5.2: DCG query rules (`modules/nlp/rules.pl`), excerpt: a zero-parameter rule, a multi-parameter rule with a fixed answer shape, and a helper that consumes one to three remaining tokens as a role name joined with spaces. Grammar terminals delegate individual token comparisons to `match_word/2` through the four-tier fuzzy matcher, so phrasing variants select the same query rule. The goal inside `{...}` executes when the preceding DCG symbols have matched. Listing B.2 reproduces these two rules and adds the remaining five (who-can-do-X lookup, user roles, explain access, trust chain, and team membership).

```
% 1. List admins (cross-platform)
query(admins(Admins)) -->
    grammar:list_admins_phrase,
    { queries:find_admins(Admins) }.

% 3. Can USER perform ACTION on BOUNDARY?
query(can_do(User, Action, Res, Answer)) -->
    grammar:can_phrase_with_user(User),
    vocabulary:action_verb(Action),
    grammar:on_phrase,
    vocabulary:resource_name(Res),
    { queries:check_can_do(User, Action, Res, Answer) }.

% Consume remaining tokens as role name (may be multi-word).
rest_as_role(Role, [T1, T2, T3 | _], []) :-
    atomic_list_concat([T1, T2, T3], ' ', Role).
rest_as_role(Role) --> [T1, T2],
    { atomic_list_concat([T1, T2], ' ', Role) }.
rest_as_role(Role) --> [Role].
```

5.4 Explorer UI

While the retained Prolog NLP pipeline accepts only phrases supported by its fixed grammar, the production Explorer UI (`web/templates/explorer.html`) provides a structured query interface. Users select a catalog query from the registry and supply its parameters through schema-generated controls.

When parameter metadata declares an autocomplete source, the interface renders dynamic suggestions. GoStore executes the query, returning formatted results for immediate inspection or drill-down navigation.

5.4.1 Autocomplete

The HTTP endpoint `/api/autocomplete/{source}` supplies candidates for parameters whose catalog definitions declare an autocomplete source. It accepts optional `q` and `platform` filters. The handler calls GoStore's `Autocomplete` method, which returns up to 300 matches from five dynamic sources: identities, boundaries, roles, teams, and actions. Platforms are returned as a fixed list. The response is a lightweight JSON envelope containing `source`, `items`, and `count`, where `items` is a string list rather than a `NormalizedItem` result. When `q` is supplied, GoStore ranks prefix matches before word-boundary-prefix and general substring matches.

The client debounces typed searches by 100 ms and caches results by source, platform, and query.

In the `Explain Access` query, typing a partial boundary name such as `hub` surfaces matching Hub-prefixed boundaries across all three loaded platforms in a single dropdown, rather than requiring the user to already know the exact identifier.

5.4.2 Provenance and Drill-Down

A central compliance feature is CasC file provenance. Heimdall records repository path, source type, and optional source detail for assignments, and definition paths for entities, stored in the retained Prolog model as `assignment_source/5` and `definition_path/3` facts. GoStore loads the equivalent fields into its indexes for Explorer queries, and the `explain_access` result cards display them, including the stored source classification, source detail, and the recorded file with its Bitbucket link when available, closing the loop from "who has access?" back to file-level evidence. Figure 5.6 shows these fields rendered on role cards, though the generated Bitbucket links identify the file without source line numbers or a commit reference.

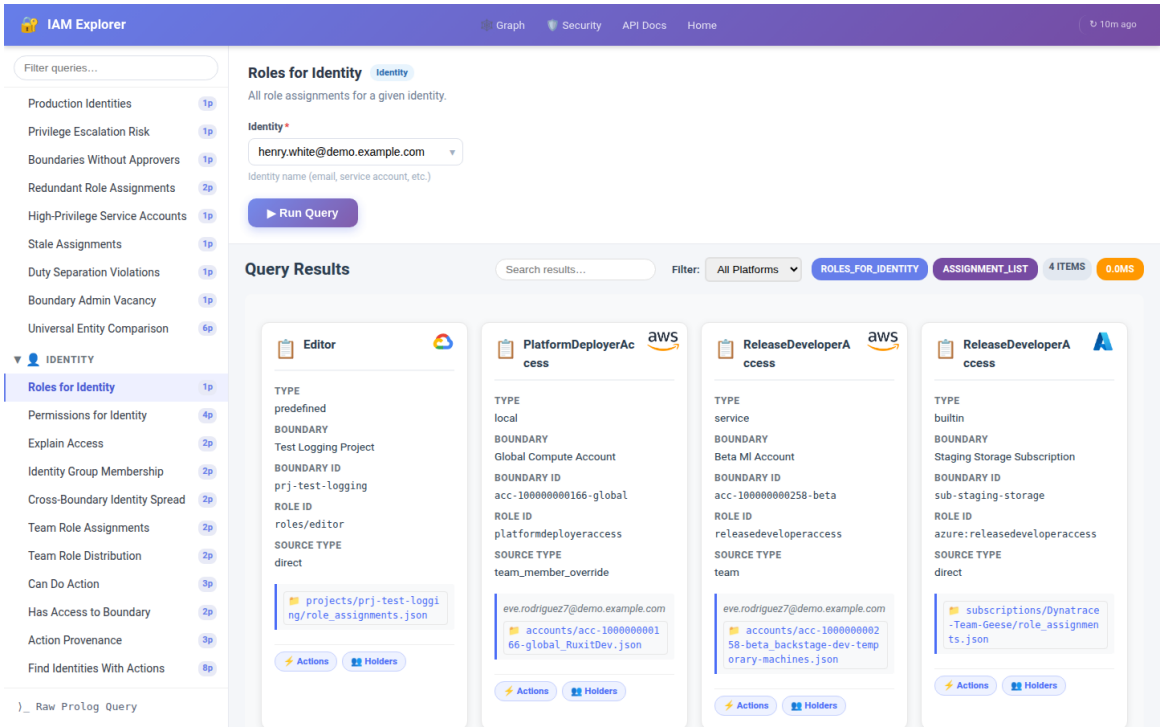


Figure 5.6: Explorer result of a `Roles for Identity` query on the anonymised demo corpus. Each card shows the stored source classification (`direct`, `team_member_override`, or `team`) and the recorded CasC file path, with drill-down links to the role's actions and holders.

Result cards contain drill-down links that select another catalog query, prefill its parameters, update the Explorer URL, and execute the new query within the interface. A boundary card may link to its assigned identities, an identity card to its roles and team memberships, and a role card to its granted actions and trust relationships, depending on the result type and query. Boundary metadata displays installation identifiers and stages when corresponding data exists, stored as `boundary_installation/4` facts in the retained Prolog model and loaded equivalently by GoStore for production queries. AWS trust-chain results (AWS-only) are rendered as cards representing complete maximal inbound or outbound paths, each showing the hop-by-hop assume-role edges.

5.4.3 Graph Navigation

The Graph Explorer, accessible at `/graph`, is an additional visualisation mode complementing the card-based Explorer. It renders a force-directed graph centred on a single entity: identity, role, boundary, or team. Traversal depth is configurable from 1 to 3 hops. Node-category toggles let

the user include or exclude identities, boundaries, roles, actions, teams, and trust relationships from the traversal, independently of the centre entity's own type. The graph queries route through a dedicated `/api/graph` endpoint backed by a Breadth-First Search (BFS) traversal in GoStore, separate from the general catalog registry, so the graph view can follow multi-type edges in one call that the catalog's per-category queries cannot.

Dense neighbourhoods can produce overlapping labels in the current force-directed rendering. The view is useful for topology discovery but not a substitute for the card view when exact provenance must be read.

5.4.4 Permission Checks

The Graph Explorer answers the question of how an identity is connected. The yes/no `Can Do Action` query answers the narrower compliance question of whether this identity can perform this action here. Figure 5.7 shows the resulting card. A boolean answer is not left to stand alone. Every role/boundary pair that grants the action is listed underneath it, each as its own drill-down link. Role assignments with conditional grants appear in this list. The conditions themselves are not evaluated, so a `yes` answer indicates that the gathered role and assignment facts permit the action without accounting for any conditional expressions (Section 1.5).

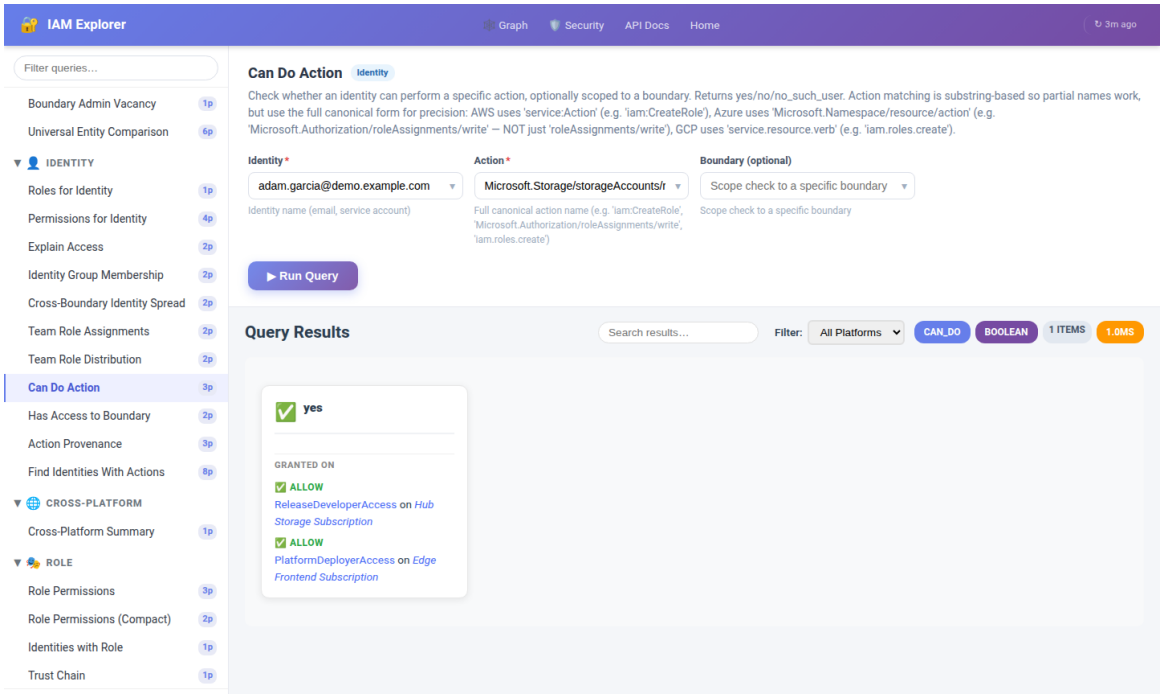


Figure 5.7: Result card for a `Can Do Action` query: a boolean `yes` plus every role/boundary pair that grants the action, each a drill-down link.

The permission-check queries above address individual access decisions. The Security Report dashboard aggregates across all identities and boundaries to produce a portfolio-level risk view.

5.4.5 Security Report Dashboard

The Security Report dashboard, accessible at `/security-report`, aggregates all eight security checks into a single page and is distinct from calling the individual checks through the `iam_security`

MCP tool. A single request runs every check, computes a weighted overall score on a 0–100 scale, and renders a per-platform breakdown alongside per-check finding counts, each linked directly to the Explorer for drill-down. The dashboard's identity counts are lower than Table 4.2's because it deduplicates identities by name (once per platform for the per-platform tiles, once globally for the header total), whereas that table reports raw per-platform loader fact counts without deduplication. Figures 5.8 and 5.9 show the dashboard output on the anonymised demo corpus.

Each check's score is $100 \times (1 - \sqrt{\text{findings}/\text{total}})$, a square-root-dampened penalty so a small finding rate still costs noticeable score, and the overall score is the weighted mean of the eight check scores. `Overall Score` ≥ 80 renders "Low Risk", ≥ 50 "Moderate Risk", otherwise "High Risk". The square-root form avoids the sharp single-finding drop a linear penalty would cause, which is too severe where some violations (such as admin and approver being the same person on a playground boundary) are expected and acceptable. The formula, weights, and thresholds are heuristic design choices intended to produce a prioritisation signal, not a calibrated risk measurement against independent ground truth. Table 5.5 gives the per-check weights, none of which are shown in the dashboard UI itself.

Table 5.5: Security Report check weights (`security_report_json.go`). Weights sum to 1.0. All weights are heuristic design choices, not empirically calibrated values.

Check	Weight
Privilege Escalation Risk	0.20
Overprivileged Identities	0.15
High-Privilege Service Accounts	0.15
Boundaries Without Approvers	0.10
Boundary Admin Vacancy	0.10
Unused Roles	0.10
Stale Assignments	0.10
Duty Separation Violations	0.10

Two security checks illustrate important data-quality limitations discussed elsewhere in this thesis. Privilege Escalation Risk carries the highest weight (0.20) but depends on a manually curated action list and does not evaluate conditional grants, so its finding count is a heuristic upper bound rather than a verified escalation path. The list contains 16 actions across the three providers: 7 AWS IAM actions covering role creation, policy attachment, user-to-group addition, assume-role policy modification, and `iam:PassRole`, 3 Azure Authorization actions covering role-assignment writes, role-definition writes, and elevated-access grants, and 6 GCP actions covering role creation and update, service-account impersonation, access-token generation, and project- and organisation-level IAM policy writes. The list was assembled by manual review of documented privilege-escalation techniques for each provider's IAM model. No systematic comparison against published community lists (such as those from Cloudsplaining or PMapper) was performed.

Boundary Admin Vacancy and the other coverage-style checks inherit the same CasC-completeness caveat as every other query in this thesis (Section 1.5): a boundary with no recorded admin may reflect an incomplete CasC export rather than a genuine governance gap. Neither check's weight is discounted for this.

5.5 Agentic Integration (Model Context Protocol, MCP)

The agent-facing MCP interface targets autonomous LLMs rather than humans directly. The server exposes typed tools over standard input/output streams and Streamable HTTP following the MCP

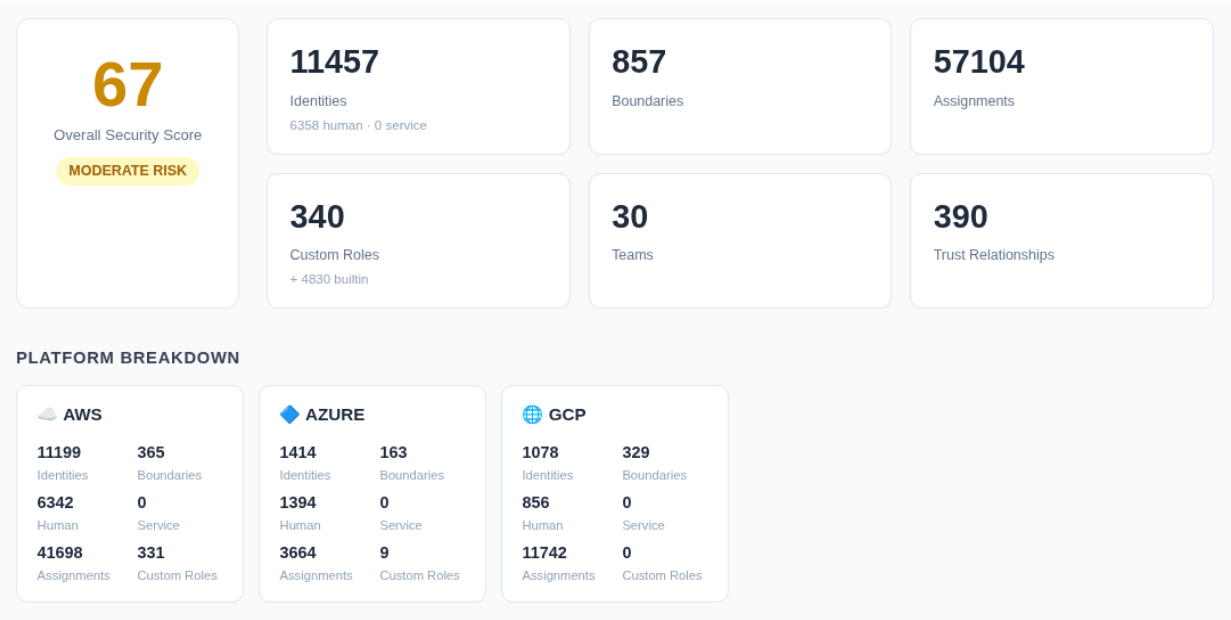


Figure 5.8: The Security Report dashboard summary: overall score (67, "Moderate Risk"), the six dataset totals, and the per-platform breakdown.

standard [50]. Going through MCP instead of the REST API matters for two reasons. The agent receives a strict tool schema with parameter types and descriptions, and it does not have to guess request shapes from prose the way it would reading REST API documentation.

This adds a machine-oriented interface alongside the web UI. A human uses the Explorer to browse and verify. An agent uses MCP to request bounded joins such as roles for an identity, trust chains for a role pattern, or identities holding an action on a boundary. GoStore supplies the semantics for every structured production call. The Prolog implementation is invoked separately by the experiment harness. The raw-query fallback path (`iam_raw_sql`) runs in-process against SQLStore and needs no sidecar.

Figure 5.10 shows the tool layout and why the grouping matters for the agent's context budget.

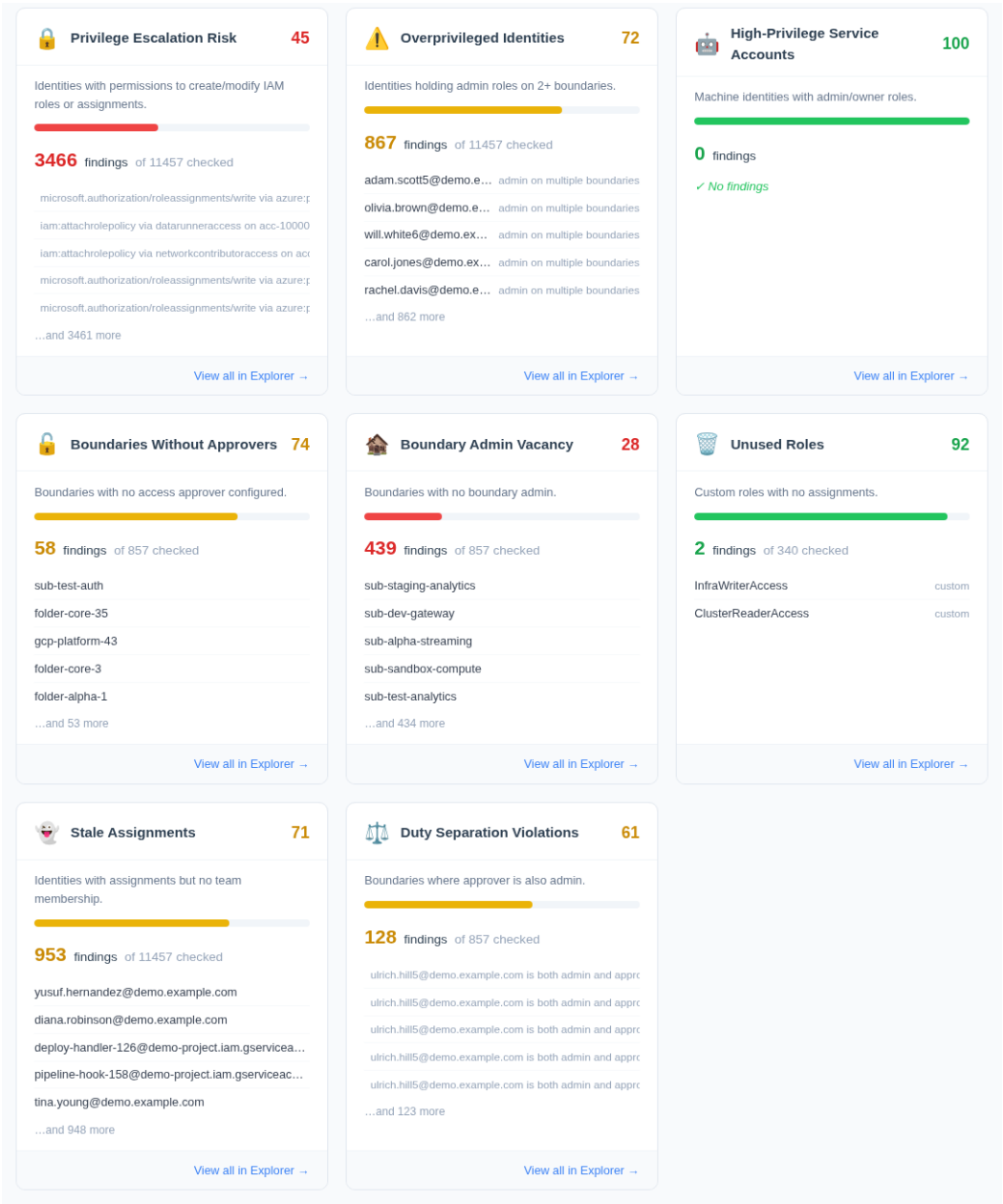


Figure 5.9: The 8 individual checks of the Security Report dashboard, each with a finding count, sample findings, and a link back into the Explorer.

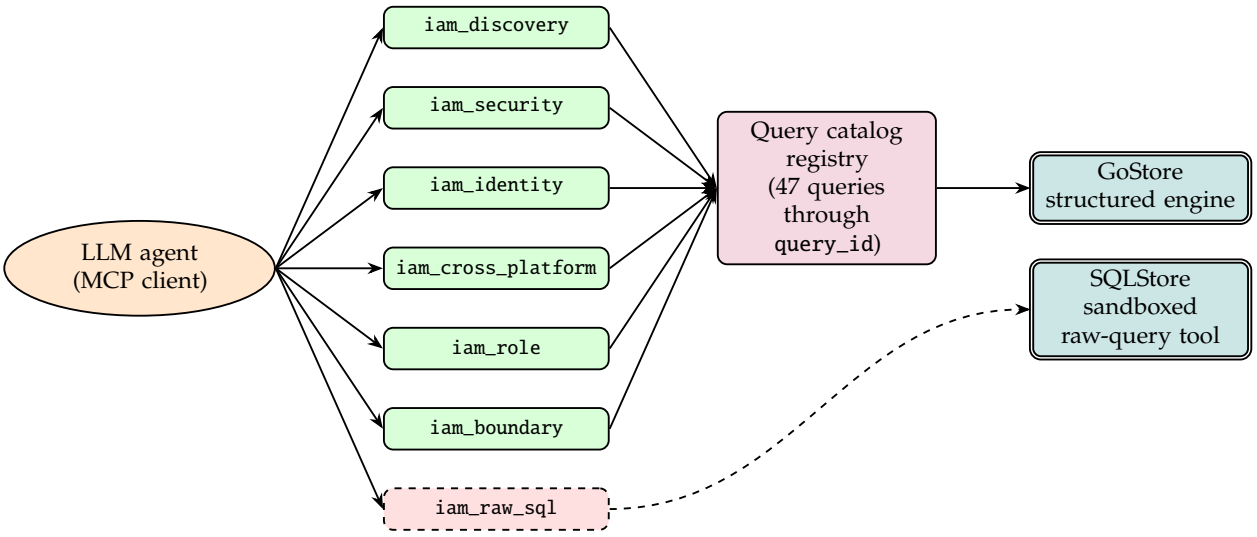


Figure 5.10: MCP tool architecture. Six grouped tools cover 46 of the 47 catalog queries through a `query_id` parameter and dispatch through the shared registry to GoStore. The remaining query, `list_actions`, is Explorer-only and not exposed over MCP. The seventh schema, `iam_raw_sql`, bypasses the catalog and runs a sandboxed read-only SQL query against SQLStore in-process, with no external sidecar to configure. Grouping keeps the advertised tool list and its repeated parameter schemas smaller than a one-tool-per-query design.

5.5.1 Why MCP Is Not Just Another API

Designing MCP tool surfaces is a different problem from designing a REST API. A REST client developer reads the documentation, picks an endpoint deliberately, and constructs a well-formed request. An LLM agent instead sees a short tool list inside its context window and must infer the correct tool to call from the schema and description alone. That constraint amplifies poor design choices. A long tool list, vague parameter names, or tools that return large, unbounded results all increase prompt-engineering overhead and make the agent less reliable before it begins the audit [51]. Section 6.6 provides partial empirical support for this behaviour.

Our first MCP prototype exposed one tool per catalog query because it was mechanically simple to generate. In practice that repeated every query's schema and common parameters in the agent's startup context and made tool selection less reliable. To reduce that cognitive load we now group the catalog into six high-level tools while still exposing individual queries through a `query_id` parameter. This grouped design keeps the MCP tool list short but lets agents reach 46 of the 47 catalog queries through the grouped interface. We also advertise `iam_raw_sql` as a sandboxed, last-resort fallback. See Section 4.8.1 for details. Table 5.6 lists the resulting six grouped tools alongside their typical query IDs and agent use cases.

Table 5.6: MCP tool grouping. The grouped design keeps the tool list short while still exposing 46 of the 47 catalog queries through the `query_id` parameter. `list_actions` is Explorer-only.

Tool	Typical query IDs	Agent use case
`iam_discovery`	list identities, roles, boundaries	Browse the model with filters and counts.
`iam_security`	overprivileged identities, unused roles, redundant assignments	Audit high-risk or redundant access.
`iam_identity`	roles for identity, permissions for identity, explain access	Investigate one principal.
`iam_cross_platform`	cross-platform summary, compare identity coverage	Compare access across providers.
`iam_role`	actions for role, roles with action, trust chain	Investigate policies and role reachability.
`iam_boundary`	users in boundary, approvers, boundary metadata	Investigate one resource boundary.
`iam_raw_sql`	caller-supplied SQL `SELECT`	Sandboxed, in-process last-resort query when no structured query fits.

5.5.2 Read-Only Tool Contract

Every MCP tool (a typed, callable function exposed by the server to the LLM agent) is read-only with respect to both cloud state and the loaded fact base, enforced through four layers summarised in Table 5.7.

Table 5.7: Read-only enforcement layers for MCP tools. The first three layers strictly enforce the contract, whereas the final layer advertises capabilities to clients through metadata annotations.

Layer	Mechanism
Structured tool dispatch	All six grouped tools route exclusively through read-only GoStore query methods with no available write paths.
SQL string validation	Strips a single trailing semicolon, rejects internal semicolons, enforces an initial `SELECT` or `WITH` clause, and blocks write, DDL, pragma, or transaction-control keywords.
Engine level	Executes queries over a dedicated SQLite connection configured with `PRAGMA query_only = ON`.
Tool annotations	Applies `ReadOnlyHint` and `IdempotentHint` flags across all seven tools for client-side discovery.

5.5.3 Shared Catalog, Single Semantics

Beyond the shared registry (Section 5.2), the MCP server derives its tool surface directly from that metadata: at startup the MCP builder groups entries by category and generates one tool schema per group. When a structured tool call arrives, the server validates the `query_id` and parameters, injects pagination defaults, and calls the same GoStore `Query` method as the HTTP handlers.

Therefore, the HTTP, Explorer, and MCP interfaces dispatch through the same GoStore `Query` computation path rather than three independent implementations, although Chapter 6 evaluates that path directly rather than verifying output equivalence across all three adapters (Section 6.7).

5.5.4 Pagination and Context Control

IAM result sets can be large relative to an LLM's context window. The Azure action catalog in the anonymised demo dataset contains 24 033 entries (Section 4.3, Table 4.2), and the AWS catalog is comparable at 25 983 entries. Returning either set in full would consume a substantial portion of an agent's context and provide a poor default tool response, which is why `list_actions` is marked `NoMCP` in the catalog registry and available only through Explorer and the REST API.

The same hazard applies at smaller scale to queries that remain MCP-exposed. The `list_identities` query, unfiltered on the demo dataset, returns 13 691 deduplicated results. The Explorer paginates those at 50 per page, but handing the full set to an LLM in one response is a poor default. Table 5.8 reconciles this figure against other identity counts that appear elsewhere in this thesis. Each count applies a different deduplication key to the same underlying records, and the table exists to explain why the numbers differ rather than contradict one another.

Table 5.8: The demo dataset's identity counts, reconciled. Each count uses the same underlying source records but a different deduplication key, which is why the four figures differ.

Count	Value	Deduplication key
`list_identities` (all-platform)	13 691	By name and platform together, so a principal on two platforms counts twice
Autocomplete registry (Section 5.4.1)	11 457	By name alone across all platforms, so a principal on two platforms counts once
Raw AWS-only source records (Section 4)	12 010	No deduplication, one AWS source record each
Raw all-platform source records (Table 4.2)	14 505	No deduplication, one source record each, all platforms

The all-platform `list_identities` total exceeds the AWS-only raw figure because it spans three providers, but falls below the all-platform raw figure because same-platform duplicate rows collapse into a single entry. The autocomplete registry applies a coarser name-only key that collapses further, so its count sits below the by-name-and-platform total whenever a principal appears on more than one platform.

To defend against unbounded responses, the MCP server injects `offset=0` and `limit=20` into every structured query by default. A caller may override the limit, but the server clamps it at 200. Every response includes a pagination envelope with a `has_more` flag, so an agent can detect remaining pages and retrieve them deliberately rather than assuming it received the complete set.

Server-side instructions further steer the model toward targeted, inexpensive calls through a decision tree:

1. If the exact entity name is known, call the specific query rather than a global list.
2. If only a keyword is known, call `search_keyword` first.
3. If the question is yes/no access, call `can_do`, `has_access`, or `explain_access`, and do not enumerate all permissions.
4. If only the size matters, set `count_only=true`.
5. If browsing is necessary, set `filter_platform` first and page through results.

These are prompt-level nudges rather than enforced constraints. The evaluation in Section 6.6 treats the MCP interface as a whole and does not isolate how much of the observed effect is attributable to these instructions specifically versus the tool grouping itself.

5.5.5 Output Formatting

Listing 5.3 shows the compact text format the MCP formatter emits, used as the running example for the rest of this subsection.

Listing 5.3: Compact MCP text output, captured 2026-07-28 against the anonymised demo dataset. Top: `roles_for_identity` for `adam.garcia@demo.example.com` with no platform filter. The identity holds assignments on two platforms, so the formatter groups results under a `## platform - N` sub-header per platform; within each group, fields constant across that platform's rows are hoisted into the sub-header and rows are pipe-delimited. Bottom: same query for a non-existent identity, where a zero-result response carries a `# SUGGEST:` recovery hint instead of an empty body.

```
# 3 result(s)
## gcp - 2 (Boundary ID: prj-alpha-data, Boundary: Alpha Data Project, Source Type:
    direct)
  Editor | predefined | Role ID: roles/editor
  Kubernetes Engine Admin | predefined | Role ID: roles/container.admin
## azure - 1 (Boundary ID: sub-platform-media, Boundary: Platform Media Subscription,
    Role ID: azure:platformdeployeraccess, Source Type: direct)
  PlatformDeployerAccess | builtin
# NEXT: identity_permissions identity=<same> -- full allow+deny breakdown; add effect=
    allow for only what they can do

# 0 results for roles_for_identity | identity=alice.johnson@demo.example.com
# SUGGEST: verify the value exists with: search_keyword keyword=<term>
```

The Explorer needs icons, drill links, HTML classes, and source buttons. An agent does not. The MCP formatter strips presentation fields and emits compact text grouped by query result, as in the listing above. Constant-valued fields are hoisted into the parenthesised group header and removed from per-item rows. Fields whose value duplicates the title are suppressed. CasC source traceability is opt-in using `include_source=true`. This keeps normal reasoning calls small while still allowing an agent to request provenance when it needs to cite a file path.

The result footer can include a `NEXT:` hint (see Listing 5.3). For example, after `roles_for_identity`, the server can suggest `identity_permissions` as a natural drill-down. This is deliberately simple. The server does not plan the audit. It exposes the next query edge so the model does not have to rediscover the graph structure at each reasoning step.

Listing 5.4 shows the JSON shape behind that footer. Query metadata and bounded result rows sit alongside the pagination envelope and the optional `next` hint.

Listing 5.4: Slim illustrative MCP response shape. The full implementation renders compact text for the model, but the same concepts apply: query metadata, bounded result rows, optional pagination, and optional next-step hints.

```
{
  "query_id": "roles_for_identity",
  "count": 2,
  "pagination": {
    "total_count": 2,
    "offset": 0,
    "limit": 20,
    "has_more": false
  },
  "items": [
    { "platform": "aws", "role": "S3ReadOnly", "boundary": "prod" },
    { "platform": "gcp", "role": "roles/viewer", "boundary": "analytics" }
  ],
  "next": "Call role_permissions for the role that looks risky."
}
```

5.5.6 Agentic Audit Workflow

A typical agentic workflow is narrower than a human browsing session. Suppose the prompt is: "Check whether the deployment service account can delete storage objects in production, and explain why." The model first uses `iam_identity` with `query_id=can_do` and the identity, action, and production boundary. A positive result can be traced with `query_id=explain_access`. If the responsible role appears broad, `iam_role` with `query_id=role_permissions` reveals whether the grant is targeted or part of an administrator role. These operations are identifiers within the grouped tools, not standalone tool schemas. The output is a short explanation with evidence, not a dump of every assignment.

This workflow demonstrates how the metamodel can be reused by another reasoning system. The LLM handles decomposition and explanation. GoStore handles the joins, wildcard expansion, and deny precedence for structured production calls.

5.5.7 Limits of the Agent Interface

The MCP server can answer only what is encoded in the metamodel. Where the correct answer depends on a ticket workflow, an approval policy that lives outside Heimdall, a resource-based AWS policy, a service control policy, or a conditional expression that the engine stores but does not evaluate, the tool can surface the facts it holds but cannot complete the governance judgment. Unless the agent is informed of that boundary, a precise-looking tool result risks creating false confidence.

The experiment in Section 6.6 exposed two concrete tool-design defects (since corrected) and one scope-boundary confusion:

1. Evaluated agents expected `identities_with_role` to accept a boundary filter. By design, the query accepts only a role and a platform, whereas boundary-scoped questions belong on `users_in_boundary`. To resolve this confusion, the parameter description for `search_keyword` was updated to direct assignment searches explicitly between these two queries.

2. Boundary-name resolution now normalises slash and underscore separators to the same form as dashes, so that `sub/edge_frontend` and `sub-edge-frontend` resolve identically.
3. `get_casc_path` now declares `entity_type` and `entity_id` as required parameters, and the grouped MCP adapter's missing-parameter check returns query-specific hints (valid options and an example value) rather than a generic error.

Items 2 and 3 were implementation defects in the evaluated build. Item 1 was a scope boundary that the experiment surfaced rather than a defect. None reflect fundamental limitations of the MCP protocol.

Three output adapters exist by design: JSON serves the REST API, event-framed cards serve Explorer SSE, and compact text serves MCP. During development, selected queries and result counts were manually compared across these interfaces, though this did not constitute a systematic equivalence test of all adapter output. Automated tests cover the shared GoStore computation beneath the adapters, evaluated in Section 6.5.

This thesis evaluates the MCP interface on the questions it is built for, namely cross-platform lookup, permission checks, trust-chain traversal, role-action explanation, and provenance lookup, in the LLM workflow comparison reported in Section 6.6.

5.5.8 Access-Control and Security Considerations

The MCP tool surface exposes the same enumeration queries that security auditors use to map privilege-escalation paths and trust relationships. In the evaluated deployment, the intended users of the MCP endpoint already have access to the underlying CasC repositories. However, the MCP interface reduces the effort required to aggregate and query security-relevant state across these repositories. Its authentication and internal-network restriction therefore remain security-relevant even though it does not expose a new underlying data source.

The MCP server is read-only, requires the same HTTP Basic Authentication as the Explorer and REST API, and is reachable only from the internal network.

The model approximates the effective permission state rather than computing it fully. Where AWS Service Control Policies, resource-based policies, or GCP hierarchy-scoped deny policies apply, a `can_do` result may be more permissive than the provider's actual enforcement. Results should be treated as indicative, and agents must not act autonomously on gathered-state conclusions without manual verification against live provider APIs.

5.5.9 Error and Disconnection Handling

The server distinguishes two failure classes. Foreseeable input mistakes, such as a missing required parameter, return a structured text hint using the same `# ERROR: ... # Hint: ... # Example: ...` shape as the zero-result `# SUGGEST:` hint shown in Listing 5.3. This is produced by the parameter-validation branch in `internal/mcp/server.go` and returned as tool content rather than a protocol-level error, so the agent can interpret the hint and retry within the same conversation turn. Genuine backend failures, such as an unknown `query_id` or a downstream query error, propagate as standard MCP protocol errors.

Transport-level session and reconnection handling is provided entirely by the MCP Go SDK's `StreamableHTTPHandler`, configured with its defaults (the server passes `nil` for options). No custom retry, backoff, or session-recovery logic is layered on top, and this thesis does not independently evaluate the SDK's transport lifecycle.

Connecting either workflow condition to an external, cloud-hosted LLM raises a data-privacy and reconnaissance risk, since CasC metadata (identities, boundaries, roles, and assignments) leaves the internal network as part of the request. This risk is mitigated in the reported experiment by using only the anonymised demo corpus (Section 4.4), which contains no production secrets, real identities, or credentials. Section 6.6 reports the head-to-head experiment. It uses the same LLM (Claude Sonnet 4.6), the same audit questions, and two conditions, one reading raw CasC files directly and one using the MCP server.

6 Evaluation

This chapter evaluates the system using the production-derived anonymised corpus and synthetic edge-case fixtures across four main areas: representation coverage, permission-query correctness, performance across serving topologies, and the shared interface computation path. A separate exploratory experiment compares raw CasC repository access against MCP-structured queries. When an experiment does not cover a configuration, metric, or interface comparison, the corresponding limitation is stated.

6.1 Evaluation Questions

To address the central research question (Section 1.1), four operational questions structure the evaluation around representation, correctness, performance, and interface evidence.

EQ1: Representation coverage. How do the selected cross-provider permission and organisational concepts map into the 18-predicate metamodel? Tables 3.2 and 3.3 trace the 18 conceptual predicates to the source JSON fields used by both loaders. Additional implementation annotations and normalisation rules are documented alongside these mappings, while explicit scope exclusions are stated separately.

EQ2: Permission correctness. Do permission queries return expected results across direct grants, wildcard matches, explicit deny rules, and boundary-scoped assignments? These scenarios are validated across both execution paths: the Prolog unit-test suite (specifically `core_tests.pl` and `nlp_query_tests.pl`) and the production GoStore test suite, which includes dedicated tests for deny logic, assignments, boundaries, and query results. Section 6.3 reports the detailed test coverage and results.

EQ3: Query performance. What latency and complete-serving-topology memory differences were measured between GoStore and Prolog on the production-derived anonymised corpus? Measurements encompass a serial two-query baseline, a deterministic full-catalog run across all 47 query IDs, cgroup-v2 (Linux control group v2, the kernel memory-accounting mechanism used for the concurrency trials) concurrency trials from 1 to 16, and a Go-only Explorer SSE run. This setup does not evaluate historical deployment replays or alternative corpus scales, and the SSE streaming run has no direct Prolog counterpart.

EQ4: Interface computation path. Do Explorer, REST, and MCP dispatch structured queries to the same GoStore implementation? This question exists because AI tools assisted in porting the dispatch code across the three interfaces (Appendix E), making independent verification worthwhile. Source-code inspection establishes this shared execution path, while a live spot-check corroborates it for REST and Explorer (Section 6.5). This establishes shared row computation rather than equivalence of formatting, pagination, or truncation; the retained Prolog NLP pipeline operates separately.

6.2 Dataset and Test Fixtures

The production-derived anonymised corpus combines full provider catalogs with scale-relevant facts and a reduced team subset, serving as the primary dataset for all recorded benchmarks, examples, and user-interface screenshots. Confidential live Heimdall data supplied the organisational portion of this corpus rather than serving as a separately evaluated dataset. Alongside this corpus, hand-written synthetic test fixtures (`test_aws`, `test_azure`, and `test_gcp`) isolate manually inspectable edge cases such as explicit deny conflicts, orphan roles, trust relationships, and hierarchy behaviour.

This split addresses two distinct evaluation needs. Synthetic fixtures supply known expected

answers for precise unit-level correctness checks, whereas the larger production-derived corpus exposes scale-sensitive behaviour absent from trivial fixtures, including JSON parsing overhead, `fastrw` cache size, large role-action result sets, and MQI payload sizes. Table 4.2 provides the canonical recorded counts for the production-derived corpus. The production-derived anonymised corpus (the shipped demo corpus) retains production-scale catalogs and assignment volume without publishing source identifiers, and serves query-latency measurements, examples, and screenshots. The synthetic test platforms encode denials, orphan roles, trust edges, and boundary hierarchy independently of confidential data, and back the edge-case tests in `nlp_query_tests.pl` and `core_tests.pl`.

6.3 Correctness Evaluation

6.3.1 Test Coverage and Results

Correctness is evaluated directly at the engine query layer rather than inferred from rendered interface output. The retained Prolog path is tested directly at its predicate boundary using synthetic facts asserted under distinct provider atoms (`test_aws`, `test_azure`, and `test_gcp`). The production GoStore path is tested separately by Go tests covering explicit deny precedence, assignments, boundaries, and query results.

The test file `nlp_query_tests.pl` covers the higher-level Prolog query layer. One notable case concerns the keyword matcher shared by `find_users_for_action/3` and `identities_with_pattern_actions/2`. It expands service-level wildcard actions (such as `s3:*` or `lambda:*`) into concrete actions before keyword matching, allowing administrator-style wildcard roles to match queries such as "write".

The suite contains 159 test cases across 25 test blocks. For this evaluation, their purposes are grouped into three categories:

- **Query-logic correctness.** These tests cover representative discovery, access, search, boundary, comparison, and analysis queries.
- **JSON classification behaviour.** The `nlp_json_not_found` block verifies that empty query results produce a typed `not_found` result rather than a successful empty payload.
- **Grammar-level phrase parsing.** The `nlp_identity_type_filter` and `nlp_security_plugin_phrases` blocks invoke `phrase/2` directly against the DCG rules to verify wording variants such as "service accounts" and "who can escalate privileges". This includes a regression test verifying that an unbound platform argument does not throw an `instantiation_error`.

This suite establishes parsing and query-dispatch correctness on controlled fixtures. It does not evaluate fuzzy-matching tiers (typo tolerance, phonetic matching, and iSub thresholds) against a large adversarial input corpus. The examples in Table 5.4 are illustrative rather than systematic robustness evidence. The NLP layer therefore remains a proof of concept within the scope defined in Section 1.5. Table 6.1 reports the complete suite results.

Table 6.1: Results from the complete test-evidence run. All suites passed without blocked tests. The Prolog correctness and normalizer suites use synthetic facts. The loader and Go suites load the committed production-derived corpus.

Suite	Purpose	Tests	Passed	Blocked
`core_tests.pl`	Base predicates, permission logic, raw-query sandbox	86	86	0
`loader_tests.pl`	Loader integration against the demo corpus, parser contracts	16	16	0
`nlp_query_tests.pl`	Query dispatch, phrasing variants, JSON classification	159	159	0
`integrity_tests.pl`	Referential integrity of loaded facts	32	32	0
`normalizer_tests.pl`	`NormalizedItem` schema: type mapping, drill links, Bitbucket URLs	55	55	0
`fts_tests.pl`	Index construction and match modes	7	7	0
Go test suite	GoStore queries, gatherers, MCP formatter	188	188	0
Total		543	543	0

`trust_chain` is excluded from the parity comparison because the Prolog and GoStore implementations are not equivalent: Prolog returns direct trusted principals only, whereas GoStore performs bounded multi-hop traversal. This diverged after the Prolog path was frozen as an offline comparison and benchmark baseline (Section 4.8); back-porting multi-hop semantics to Prolog was not undertaken because Prolog is not reachable from any request-serving code path and doing so would not have changed any production behaviour. This is the one exception to the research question's common query interface requirement (Section 1.1): the same query identifier invokes semantically different traversal depth in each engine.

Item-level parity. A title-set comparison verified that 28 of 34 checked AWS queries, 29 of 34 Azure queries, and 27 of 31 GCP queries returned identical result sets in both engines, excluding large result sets, single-object results, and known implementation gaps. The remaining differences fall into serialisation format (`cross_platform_summary` uses lower-case in GoStore and upper-case in Prolog), tie-breaking in ranking queries, the `list_trust_chains` implementation divergence described above, and small normalisation differences on AWS where GoStore returns marginally more items. Accordingly, the title-set evidence establishes exact agreement only for the reported subsets and leaves these bidirectional representation, tie-breaking, traversal, and normalisation differences explicit.

6.3.2 Selected Corpus Findings

The following findings come from catalog queries against the evaluated corpus snapshot. They describe the recorded CasC declarations and exporter coverage rather than persistent properties of the organisation or cloud providers.

Recorded approver coverage differs across providers. The `boundaries_without_approvers` query returned 57 of 329 GCP boundaries and 1 of 163 Azure boundaries with no approver recorded in the CasC model, compared with 0 results across 365 AWS boundaries. Determining whether a gap reflects incomplete CasC declarations or a genuine absence of designated approvers requires comparison with an authoritative source outside this metamodel. One common cause is employee turnover: when a departing approver is removed from the CasC declarations, the gap persists until a successor is designated and their assignment is added.

The privilege-escalation query surfaces candidate identity/action pairs across all providers. In the final serial comparison, both query engines returned 256 999 results on AWS, 3 238 on Azure, and 6 934 on GCP. These results are pattern-matched identity/action pairs rather than distinct identities. They represent candidates rather than confirmed escalation paths because detection relies on a manually curated action list and does not evaluate conditional grants. Human review is required to determine whether a viable escalation path exists. Likewise, an empty result would not establish that no escalation path exists.

6.4 Performance Evaluation

6.4.1 Benchmark Environment

The benchmarks used the hardware and software environment specified in Section 4.8.7. In the serial evidence run (20260721T184930Z-780ceb1b2e2b), both engines processed the same committed production-derived anonymised corpus in serial execution (concurrency 1), with Prolog configured for an MQI pool size of 1. For reproducibility, the harness rebuilds both binaries, records source and dataset hashes, and measures peak RSS across the complete process tree.

Three accepted catalog-stress runs evaluate performance using a single deterministic manifest with 382 cases in the same benchmark environment. After 34 documented platform and parameter exclusions, 348 applicable cases spanning all 47 catalog query IDs execute once cold and once warm, in that fixed order within each fresh-process trial: the *cold* phase is the first pass through the manifest against a just-started process, and the immediately following *warm* phase repeats the identical manifest against the same still-running process. For GoStore, warm-phase requests can hit the in-process result cache (Section 4.8). Prolog has no equivalent request-level result cache, so its warm-phase gain, if any, comes only from incidental effects such as OS page-cache warmth, not from a comparable cached-response mechanism. Each configuration runs in an isolated cgroup-v2 scope capped at 12 GiB of memory and 0 bytes of swap for each of its 3 fresh-process trials. Following cgroup-v2 accounting definitions [52], the harness records `memory.peak`, samples `memory.current`, drains every response, and validates result shape and pagination.

Both Go and Prolog receive the workload through the same HTTP adapter. The GoStore process operates without an SWI-Prolog sidecar, whereas the Prolog topology includes both the Go HTTP wrapper and its SWI-Prolog process tree. These measurements compare complete serving topologies rather than isolated language heaps. The optional raw-Prolog sidecar is excluded, as including it in a hybrid Go deployment would add memory to the total footprint.

The benchmark selected two representative AWS query paths for repeated comparison: the bounded graph query `trust_depth` and the aggregate `security_report`. The engines ran sequentially to avoid overlapping memory footprints, with each executing 10 samples per query. Table 6.2 reports the resulting latency distributions.

Table 6.2: Accepted repeated cross-engine latency measurements on the AWS corpus, in milliseconds.

Query	Engine	n	Mean	p50	p95	Std. dev.
`trust_depth`	GoStore	10	0.044	0.039	0.083	0.014
	Prolog	10	6.925	6.698	8.410	0.908
`security_report`	GoStore	10	861.992	933.671	1 012.944	118.983
	Prolog	10	4 318.180	4 013.264	7 491.780	1 061.561

The preceding table measures query latency at the shared HTTP boundary. In the accepted serial baseline, peak process-tree memory (RSS) reaches 2 853 256 KiB for GoStore against 1 697 836 KiB for Prolog, a GoStore/Prolog ratio of 1.681×. This larger process-tree peak for GoStore in the narrow serial baseline serves as a sanity check on process-level accounting rather than the primary memory comparison. GoStore's peak includes the complete catalog resident in memory. The two-query Prolog run may be captured before SWI-Prolog has loaded the complete catalog and can therefore understate its footprint under full load. The catalog-wide benchmark below is more representative because both engines serve the complete catalog, and it reports a larger peak for Prolog.

6.4.2 Catalog-Wide Latency

The Prolog serving path includes the complete inter-process communication overhead caused by MQI and the measured speedup therefore reflects the full architectural difference, not the query engines themselves. Section 4.8's four-way engine comparison isolates that transport cost directly at approximately 0.9 ms per cached catalog-count query and shows the expensive analysis queries are dominated by derivation cost instead, so the catalog-wide speedup below is corroborated by, not merely disclaimed against, that independent result. For every applicable manifest case, the three Go and three protected-Prolog trials are matched by query ID, platform, parameters, result shape, and cold/warm phase. The baseline is the protected Prolog topology at pool `1/c1` (single Prolog worker, concurrency 1, admission guard enabled), matching GoStore's single-process, single-request configuration in this comparison. Each engine's case latency is taken as the median across its three trials. Speedup is calculated as the Prolog median divided by the GoStore median. Table 6.3 reports case-level distributions in which every applicable manifest case receives equal weight. Because query IDs contribute different numbers of parameterised cases, the query-level comparison reported below also aggregates once per query ID.

Table 6.3: Matched end-to-end catalog speedup: GoStore versus the MQI-Prolog serving path across 348 applicable cases and all 47 query IDs. IQR is the interquartile range of case-level ratios. *Bounded* cases request a page-limited result (`limit=200`), *unbounded* cases request the full uncapped result set (`limit=0`), and *count-only* cases request just the result count (`count_only=true`).

Phase	Cases	Median	IQR	Bounded	Count-only	Unbounded
Cold	348	3.20×	13.02	4.68×	2.40×	12.60×
Warm	348	3.07×	35.97	23.51×	2.44×	21.74×

The case-level median speedup was 3.20× cold and 3.07× warm. Aggregating once per query ID yields query-level medians of 5.44× and 6.31×, respectively. The interquartile ranges (13.02 cold and 35.97 warm) are wide relative to the medians. The broader case-level distribution extends from below 1× to double-digit multiples, showing that relative performance varies substantially by query shape. In the cold phase, GoStore was faster for 46 of 47 query-ID medians. In the warm phase, GoStore was faster for all 47 query-ID medians.

The only cold-phase query-level median below 1× was `find_identities_with_actions` at 0.70×, meaning that GoStore was slower for this query. The Prolog implementation precomputes the matching role set once per action/platform condition in O(#permissions) time. It then checks each identity's assignments against this set in O(#identities × #assignments) time. GoStore instead re-evaluates assignments and permissions for every identity, resulting in an O(#identities × #assignments × #permissions) pass. The implementations are therefore not optimisation-equivalent for this query. The result identifies a porting gap rather than an inherent limitation of GoStore.

GoStore catalog queries reach the 0.01 ms measurement floor. The largest speedup ratios are therefore best interpreted as comparisons between sub-millisecond execution times and multi-second baselines rather than exact multiplication factors.

6.4.3 Catalog Memory and Concurrency

Table 6.4 uses the fully admitted overload run with the Prolog admission guard disabled, in which both engines completed every request without out-of-memory events or busy rejections. GoStore's median kernel peak remained stable near 2.7–2.9 GiB across all test configurations. As the worker pool size and concurrency scaled from pool `1/c1` (a single Prolog worker handling one concurrent request) to pool `16/c16` (16 Prolog workers, each handling one of 16 concurrent requests), the complete Prolog topology's median kernel peak increased from 4.104 GiB to 7.447 GiB.

Table 6.4: Median cgroup-v2 `memory.peak` (GiB) across 3 fresh-process trials in the fully admitted *overload* run (Prolog admission guard disabled, where the guard, when enabled, immediately rejects a heavy query as busy instead of queueing it once the configured concurrent-heavy-query limit is reached). The `c1` ratio evaluated in the text uses the *protected*-run Prolog peak (admission guard enabled) instead. These values come from different run configurations and are not interchangeable.

Concurrency	GoStore, pool 1	Prolog, matching pool
1	2.898	4.104
2	2.873	4.717
4	2.921	5.327
8	2.800	6.625
16	2.712	7.447

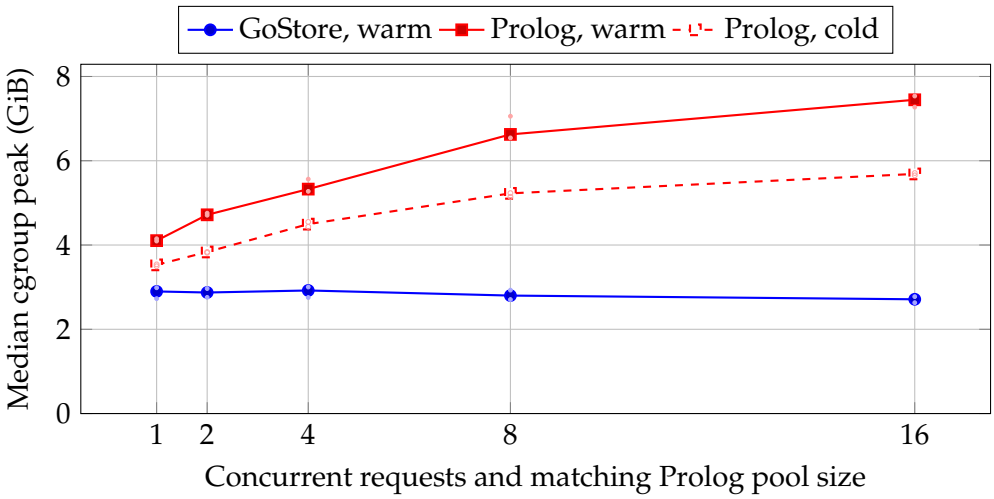


Figure 6.1: Peak memory as concurrency and the Prolog pool size increase in the fully admitted catalog run, for both the warm and cold phases. Each point is the median of 3 fresh-process trials. The minimum and maximum are shown as faint marks. The GoStore configuration retains one Go engine instance, and its cold- and warm-phase medians coincide within 0.005 GiB at every point, so only the warm series is plotted.

Figure 6.1 shows the Prolog topology's resident worker state growing with pool size in both phases, unlike GoStore's flat footprint.

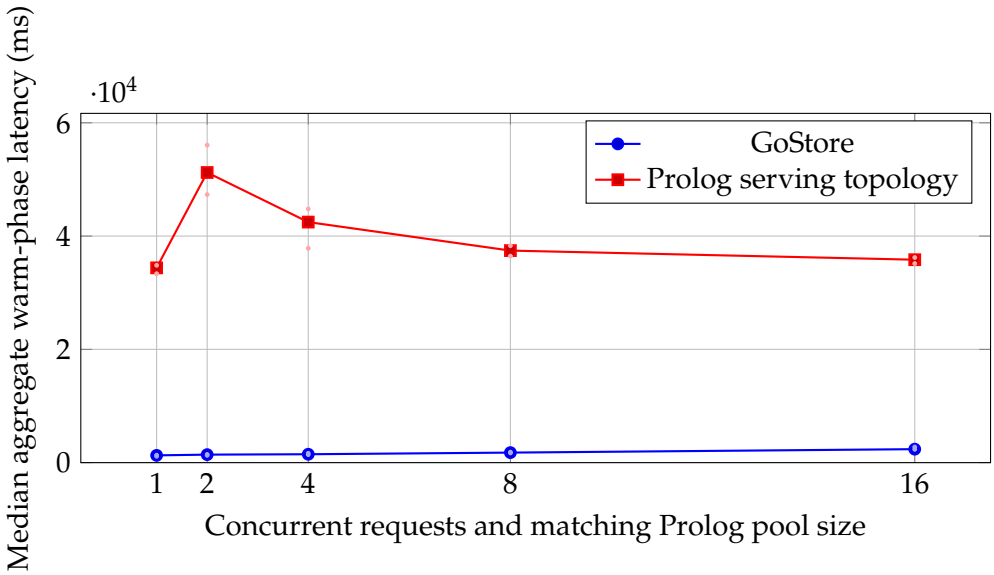


Figure 6.2: Each point reports median aggregate warm-phase latency across 3 trials in the fully admitted run, with the Prolog admission guard disabled, as concurrency and the Prolog pool size increase together. Aggregate latency is the sum of elapsed times across all 348 warm-phase requests in a trial. The minimum and maximum are shown as faint marks. This evaluation uses the same manifest and trials as Figure 6.1.

Figure 6.2 shows that GoStore maintains lower aggregate warm-phase latency at every matched pool size, extending the concurrency-one results from Table 6.3. Prolog's aggregate warm-phase latency spans from 34390.395 ms at `c1` to 35829.224 ms at `c16`, with variability between these endpoints. Cold-phase elapsed time (not plotted) follows the same direction.

The latency gap narrows from `c1` to `c16` because GoStore's aggregate latency increases from

1282.991 ms at c1 to 2376.734 ms at c16. The experiment does not isolate the cause of GoStore's latency growth, so contention on the single shared engine instance remains a plausible hypothesis rather than a measured result.

At c1, GoStore used 0.706 times the protected Prolog topology's peak memory (2.898 versus 4.108 GiB). At c16, GoStore used 0.364 times the active overload-Prolog peak. These ratios draw from distinct protected and overload run configurations and do not form a single interchangeable series. In both cases, GoStore achieved lower median latency and lower peak memory than the corresponding Prolog topology.

The protected Prolog run keeps the admission guard enabled. Median cold/warm busy rejections were 3/3 at c2, 6/6 at c4, and 2/2 at c8. The c16 configuration recorded zero rejections. This zero result does not indicate that admission control is irrelevant. At c16 the pool size matches the concurrency level, so each concurrent request has a dedicated worker and the guard's heavy-query limit is not reached under the deterministic batch schedule used here.

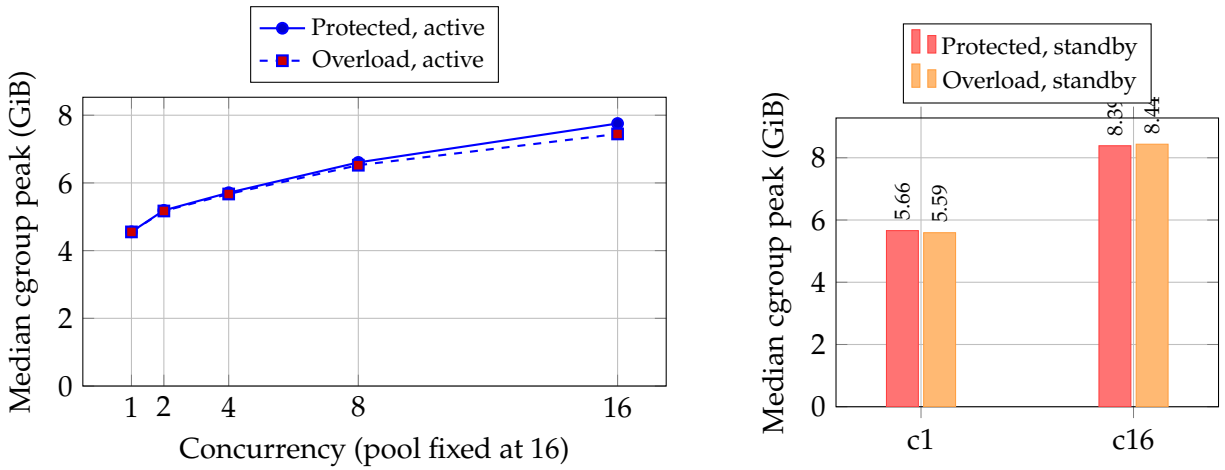


Figure 6.3: Median peak memory with the Prolog pool fixed at 16 (active and standby pool connections defined in Section 4.8). Left: the admission-guard-enabled (protected) and guard-disabled (overload) active-only runs across all five measured concurrency levels, aligning within 0.31 GiB at every point and indicating that this memory-scaling behaviour occurs independently of the admission guard. Right: the loaded-standby variant of each run, measured only at c1 and c16, shown as bars rather than a connected line to avoid implying intermediate measurements. Introducing a standby elevates the peak in both runs.

Figure 6.3 offers a plausible mechanism for multi-gigabyte production memory footprints without claiming to reconstruct the historical task exactly.

A supplemental cache-hostile workload (distinct, non-repeating results) kept pool-8/c8 peak memory within 5–17 % of the standard-workload baseline (Figure 6.4).

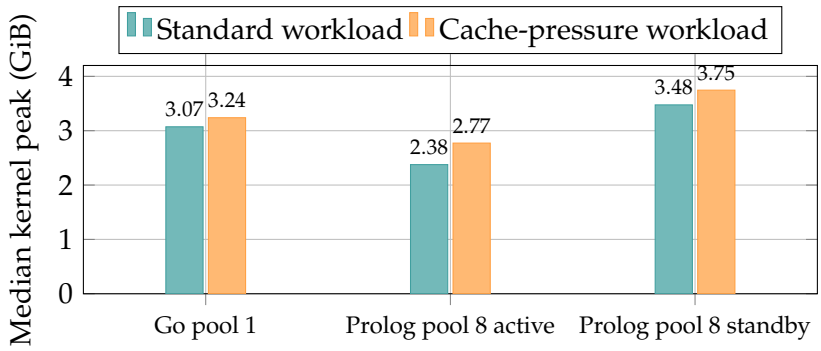


Figure 6.4: Peak memory at `c8` under the standard catalog workload versus a supplemental cache-hostile workload with distinct, non-repeating results per request, for the three pool-8-scale configurations. The close agreement between the two workloads indicates that cross-request result caching is not a major contributor to the measured peaks.

6.4.4 Explorer Streaming Path

Run 20260714T143801Z-ddb898cd, recorded on the same host as the serial baseline and catalog-stress runs above, separately evaluates the deployed Go Explorer SSE endpoint using 16 distinct, large-result requests at concurrency levels 1–16. This run sets `DEMO_MODE=true` and `MQI_POOL_SIZE=0`, meaning no SWI-Prolog process or standby is present. Figure 6.5 reports results for GoStore's query, card-conversion, gzip, and streaming path under this configuration. The streaming endpoint is unique to the Go implementation and has no Prolog counterpart.

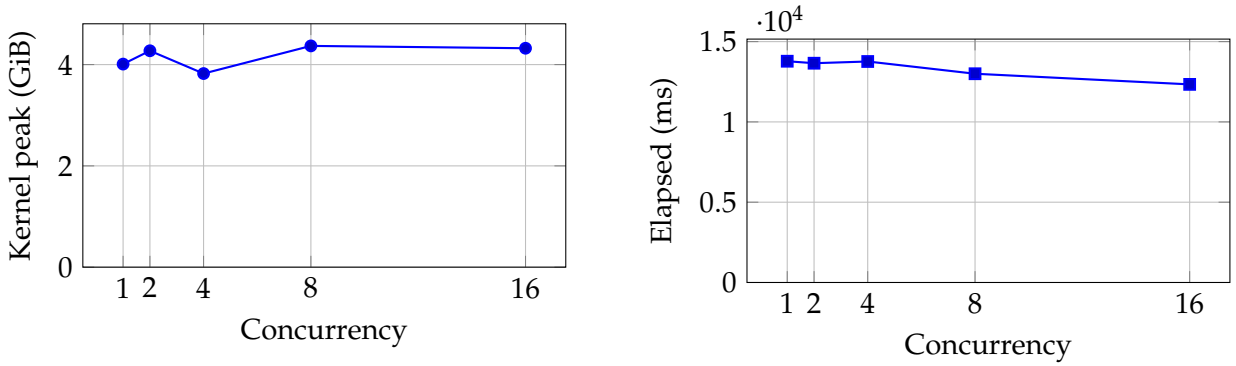


Figure 6.5: Go Explorer SSE stress medians across 3 fresh-process trials, not distinguishable from trial-to-trial noise at this trial count. Kernel peak (left) ranges non-monotonically between 3.823 GiB and 4.369 GiB, and elapsed time (right) is the total batch-completion time for a fixed request set, not a throughput measurement.

Each fixed-work trial decoded approximately 0.330 GiB of card data (the normalised Explorer result cards streamed via SSE, Section 5.2) while transferring 0.010 GiB over the wire with gzip compression, and this volume held constant to within a few bytes across all five concurrency levels. The cause of the non-monotonic memory pattern was not isolated. Shared startup overhead and overlapping I/O waits are plausible contributors but were not measured separately.

6.5 Interface Evaluation

The Explorer, REST API, and MCP server compute catalog rows through the same GoStore engine instance. Prolog is not reachable from any request-serving code path in the production server. It was frozen as an offline comparison and benchmark baseline (Section 4.8), invoked only through

separate comparison and benchmark commands, while raw ad-hoc queries use the sandboxed SQL tool. Presentation differs across the three entry points: Explorer streams card events, the REST API returns JSON, and MCP returns compact pipe-delimited text.

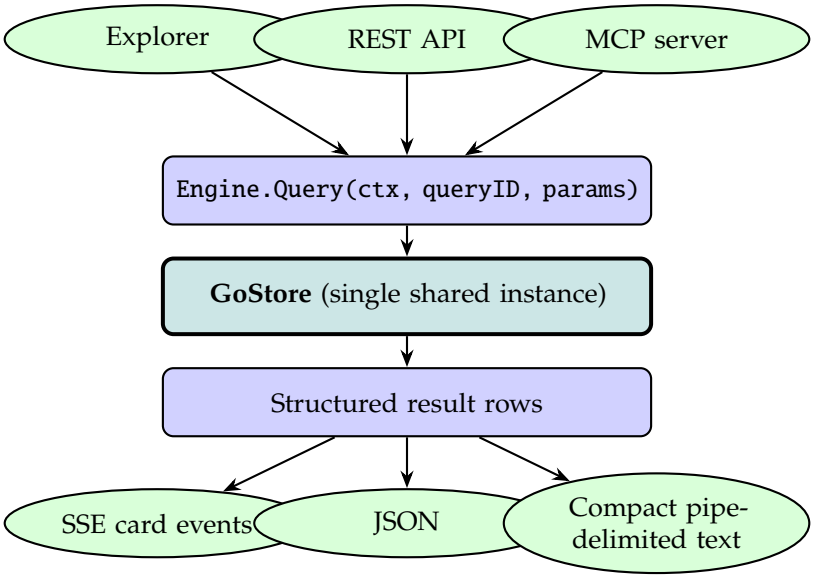


Figure 6.6: Shared dispatch path for the three structured-query interfaces. Explorer, the REST API, and the MCP server all resolve a `query_id` through the same query catalog and invoke `Engine.Query` on the single shared GoStore instance. Presentation formatting is applied only after this shared call returns, so the three interfaces differ in adapter code and output shape but not in the underlying computation.

Evidence: a shared dispatch path, not three parallel implementations. Source inspection directly establishes the shared dispatch path within the codebase. The REST endpoint (`APIQuery`), the Explorer stream handler (`ExplorerStream`), and the MCP per-category tool handler resolve `query_id` through the same query catalog. All three invoke `Engine.Query(ctx, queryID, params)` on the single GoStore instance supplied at startup, with none independently implementing query dispatch or result construction. Differences across entry points are limited to adapter-specific parameter handling and the presentation applied after `Engine.Query` returns.

A live spot-check against a running instance using the demo dataset (`DEMO_MODE=true`) confirmed that `POST /api/query` and the retained non-streaming `/explorer/query` adapter returned the same three identities for `query_id=list_identities, platform=aws, limit=3`. The standard browser path uses `/explorer/stream`, so this check covered only the non-streaming adapters.

The retained Prolog NLP pipeline runs on the frozen Prolog baseline rather than GoStore (Section 5). It is not registered as a production server route. Its correctness evidence comes from the Prolog test suite (Table 6.1). The shared-dispatch finding for the GoStore interfaces does not extend to this pipeline.

The recorded live check provides one reproducible example in which the REST endpoint and the retained non-streaming Explorer adapter returned the same three identities. Source tracing establishes that the primary SSE and MCP paths invoke the same `Engine.Query` implementation on the shared GoStore instance. The adapters intentionally apply distinct pagination defaults and formatting, so this evidence shows shared computation, not byte-identical output.

Source inspection answers EQ4 by establishing shared dispatch across the GoStore interfaces. No automated cross-adapter suite verified row-count equivalence across all 47 catalog query IDs.

6.6 Exploratory LLM Workflow Observation: Direct CasC Access vs. MCP

Guiding question. Does giving a fixed LLM MCP-structured access to the pre-loaded metamodel, instead of raw CasC repository file access, change answer quality and cost on a fixed set of IAM audit questions? This section does not compare the LLM against the logic-based GoStore/Prolog system, and it does not evaluate the MCP protocol in isolation. Both conditions use the same LLM and start from the same frozen, anonymised CasC repository state, but the two conditions do not see identical data. The MCP condition queries the pre-loaded metamodel through structured tool calls, which additionally includes provider SDK/CLI action catalogs the CasC repository does not embed, and the RAW condition reads the CasC repository directly via shell tool calls, which can see repository-only governance files (such as `CODEOWNERS` and CI/CD configuration) that the metamodel does not expose. The study is observational and exploratory. It does not evaluate tool-augmented LLM methods in general [53], [54], only whether the metamodel interface changes answer quality and cost on this prompt set.

6.6.1 Experimental Setup

The model was Claude Sonnet 4.6 under both conditions. RAW used a detailed system prompt (`agents.md`, roughly 30 kB) covering inheritance traversal, permission exclusions, and CasC schema conventions, without MCP access. MCP used a strict MCP workflow configuration and tool-use hint without access to the full `agents.md`. The two conditions therefore used different instruction sets, not the same base prompt.

The 18 prompts span five categories: lookup (A1–A4), reasoning (B1–B6), real-world operational (C1–C3), robustness (C4–C5), and stress tests (D1–D3). Operational prompts were drawn from `#gotc` with documented resolutions as ground truth. Each prompt was executed 5 separate times under each condition. The configuration was fixed before data collection and not changed between repetitions, yielding 90 scored outputs per condition overall.

Each output is scored on a six-label rubric mapped to five numerical weights: `correct` and `correct_with_richer_context` both receive 1.0, `mostly_correct` 0.75, `partial` 0.5, `structurally_incomplete` 0.25, and `wrong` 0. The full rubric and all per-repetition scores are in `experiments/llm-comparison/prompts.json`. Appendix D reproduces the anonymised prompts. Each repetition also records per-prompt API cost, recorded task duration, and total operations.

6.6.2 Results

Correctness and stability. Across all 5 repetitions the descriptive mean weighted score was 91 % for MCP and 88 % for RAW. Full-credit outputs were 71/90 (79 %) and 69/90 (77 %). Head-to-head by majority vote, MCP scored higher on 4 prompts, RAW on 3, and 11 were tied. Because the 90 scored outputs are repetitions of only 18 distinct prompts that we curated, we examine the prompt-level win record separately from the descriptive weighted-score difference. The near-even 4–3 split among the seven non-tied prompts provides no descriptive indication of a consistent advantage for either condition. Given the small prompt set we curated, these results are exploratory and do not support population-level inference. Both conditions were highly stable: 16/18 prompts had identical classifications across all repetitions under MCP and 16/18 under RAW. Only four prompt-condition combinations varied, driven by two wording ambiguities (A1, both conditions), a scope-of-search question (B3 RAW), and one isolated wrong answer (C1 MCP).

Resource use. Figure 6.7 shows all four metrics per repetition. MCP was cheaper in every repetition (total \$16.03 vs \$22.68, per-repetition means \$3.20 vs \$4.54). The gap reflects RAW loading large CasC files into context on each prompt, confirmed by the 12.2× difference in uncached input tokens.

Mean recorded task duration was 55 s per prompt for MCP and 121 s for RAW (2.2×), with 5.3 vs 13.1 mean operations per prompt.

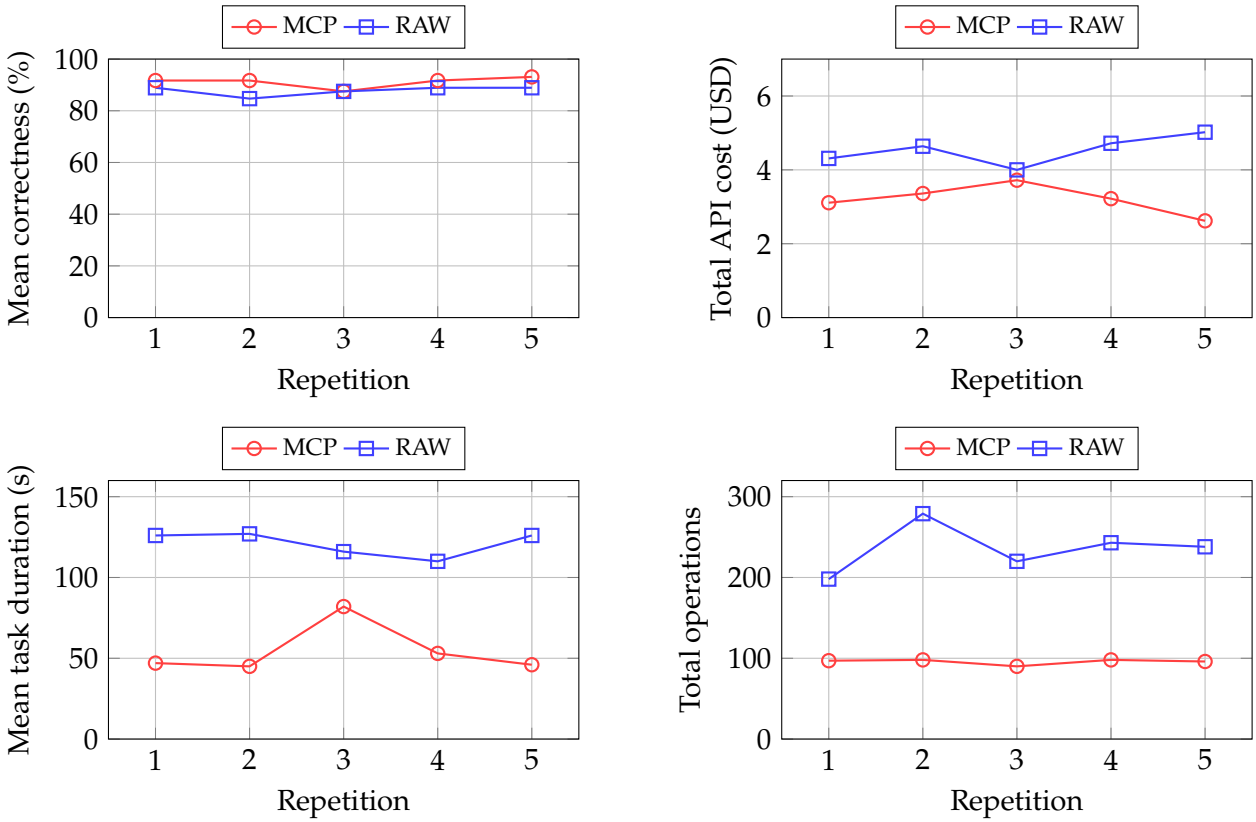


Figure 6.7: Per-repetition metrics across 5 separate repetitions, 18 prompts each. Each paired point shows the MCP (circle) and RAW (square) value for the same repetition, and faint lines connect the pair. *Top left:* mean weighted correctness score. *Top right:* total API cost per repetition in USD. *Bottom left:* mean recorded task duration per prompt in seconds. *Bottom right:* total recorded operations (structured tool calls for MCP, and tool-call and file-access operations for RAW).

6.6.3 Interpretation

Table 6.5 breaks the results down by category. For most categories the difference between conditions reflects data visibility rather than reasoning quality: MCP accesses the metamodel, which includes provider SDK/CLI action catalogs, while RAW accesses the CasC repository directly. The two data sources are not identical.

Table 6.5: Correctness by prompt category across 5 repetitions. Mean score uses the six-level rubric weights.

Category	MCP		RAW	
	Mean	Full credit	Mean	Full credit
Lookup (A1–A4)	96 %	17/20	86 %	15/20
Reasoning (B1–B6)	92 %	25/30	89 %	24/30
Real-world (C1–C3)	77 %	9/15	83 %	10/15
Robustness (C4–C5)	100 %	10/10	100 %	10/10
Stress test (D1–D3)	92 %	10/15	83 %	10/15

MCP scored higher on lookup and reasoning because it can query the pre-loaded identity index

directly (A1) and has access to the provider-managed-policy catalog that CasC does not embed (B2). Conversely, RAW scored higher on real-world prompts because it can read governance metadata, such as `CODEOWNERS` and CI/CD configuration, that the metamodel does not expose, and on B5 because it reads the full role definition from the CasC file and correctly identifies a conditional deny on instance types that the compact MCP summary omitted. On A4, the compact output produced a wildcard-heavy permission list that was difficult to interpret. This motivated adding the `compact_permission/4` annotation store to the metamodel (Section 3.2). Robustness was tied at 100 % for both conditions. Stress-test differences turned on whether D1 auto-revocation was visible in CasC files (RAW advantage) or whether D2 blast-radius ranking required a full identity-assignment index (MCP advantage).

Traceability. MCP execution traces record structured tool calls with explicit `query_id` values, parameters, and intermediate results, keeping the evidentiary path visible and allowing individual calls to be rerun. RAW provenance is spread across file reads, shell commands, and model-side aggregation, making the reasoning chain harder to reconstruct independently even when the final output is correct. This difference is an operational property of the workflows, discussed further in Section 7.

The B6 loader defect. An early pilot run of B6 disagreed between conditions: RAW found an AWS role grant reached through a team-membership override, and MCP did not, because the loader's team-deny gate had incorrectly suppressed the override entirely instead of gating only the team's own blanket grants. RAW's advantage was direct file access, not superior reasoning. The defect was fixed and reverified before the five scored repetitions, which is why it no longer appears in Table 6.5.

6.7 Threats to Validity

6.7.1 LLM Workflow Comparison

Designing both the prompt set and the MCP server creates a risk of selection and outcome-confirmation bias. The results describe a deliberately varied corpus that we curated rather than an unbiased sample of IAM questions. We scored the outputs without blinding to condition. A single model from one vendor was used, which limits generalisation. The 90 scored outputs are 5 re-executions of the same 18 task designs, not 90 independent questions, so stochastic stability should be interpreted accordingly.

The two conditions differ in instruction set, data visibility, and operation type, not only in the presence or absence of MCP. Results cannot be attributed to MCP as a protocol in isolation. Correspondingly, MCP structured tool calls and RAW shell or file operations are not equivalent units: the recorded operation counts, costs, and durations are specific to Claude Code, its caching model, and the local execution environment and should not be generalised to other MCP deployments. The server ran on the same machine as the model client, so the recorded task durations do not represent a remote production deployment. Three tool-design defects identified during development were corrected before the scored repetitions, as documented in Section 5.5.7.

6.7.2 Broader Evaluation

The evaluation uses an anonymised export rather than live organisational data. The underlying anonymisation pipeline (Section 4.4) is designed to preserve the structural properties relevant to the benchmark, specifically scale, role hierarchy, and assignment density. The results therefore apply to this transformed CasC corpus, which is the system's actual input, rather than the provider's complete live authorisation state.

Six broader limitations affect the interpretation of the evaluation:

1. **Test coverage.** The correctness suite evaluates only the structural patterns represented within its synthetic fixtures. An operational CasC repository containing unmodelled edge cases or non-standard configurations could therefore expose behaviour not covered by the suite.
2. **Measurement environment.** All latency and memory figures were measured on the single documented machine (Section 6.4). Different hardware architectures or operating-system scheduling could alter both the absolute measurements and the relative performance comparisons between engines.
3. **Performance-study scope.** The performance experiments rely on a single anonymised corpus and three fresh-process trials per stress configuration, reporting descriptive medians across trials for the concurrency measurements and case-level medians with interquartile ranges for the catalog speedup. The cgroup `memory.peak` metric measures the footprint of the complete serving topology rather than isolated language heaps. The study also does not measure sustained-arrival throughput or performance across alternative corpus scales. The catalog-wide speedup figures include the full MQI transport overhead by construction (Section 6.4). The four-way engine comparison in Section 4.8 isolates that transport cost at approximately 0.9 ms per cached catalog-count query and shows the expensive analysis queries are dominated by derivation cost rather than transport, which corroborates but does not eliminate this scope limitation.
4. **Corpus representativeness.** The corpus represents a single snapshot from one organisation. Its role structures, CasC patterns, and overall scale may differ from those in other enterprise repositories. The findings therefore describe this specific dataset rather than general CasC usage.
5. **Controlled-language evaluation.** The NLP evaluation assesses the implemented controlled grammar, fuzzy matching, and full-text fallback engine. It tests this deterministic querying pipeline rather than open-domain language understanding or an LLM-based parser.
6. **Representation boundaries.** The evaluation covers the source concepts formalised within the metamodel rather than every provider enforcement mechanism. Policy conditions, SCPs, resource policies, GCP hierarchy-scoped deny policies, and Azure management-group inheritance remain outside the evaluated permission decisions (Section 1.5).

These limitations bound the evaluation claims to the implemented metamodel and query engine operating on the modelled CasC data and evaluated configurations.

7 Conclusion and Future Work

Within the stated CasC and enforcement boundaries, the implemented artefact shows that the shared permission primitives of AWS, Azure, and GCP IAM can be normalised into a single logical model and queried through a common interface. The central result is a metamodel comprising 18 predicates, backed by an automated loader pipeline that ingests multi-cloud configuration data. This model is implemented twice: once in SWI-Prolog as a research baseline and once in Go (GoStore) as the production engine. GoStore executes the production structured-query catalog used by the Explorer UI, REST API, and MCP server, while the Prolog baseline remains separately runnable for the retained NLP pipeline and engine comparisons.

Beyond this central result, the exploratory LLM comparison (Section 6.6) surfaced a secondary, methodological finding: running the same audit question through two independent access paths, an LLM reading raw CasC versus the same LLM through MCP, exposed a genuine loader defect (B6) that neither path alone would have exposed as a bug rather than a disagreement in wording. This comparison-as-defect-finding technique was not part of the original design. It emerged as a side effect of building two independent systems against the same schema, the same pattern that surfaced the historical defects on both sides of the Prolog/GoStore comparison, and in hindsight we would include it from the start.

7.1 Answers to the Evaluation Questions

The four operational evaluation questions introduced in Section 6.1 are answered below based on the evidence presented in Chapter 6.

EQ1 (representation coverage). The 18 conceptual predicates represent the selected core of cross-provider permission and organisational concepts. Additional retained source fields are handled through implementation annotations or normalisation rules, while known unsupported mechanisms are documented as scope exclusions. Tables 3.2 and 3.3 establish the source-field mappings for the conceptual predicates but do not claim exhaustive coverage of the complete provider schemas.

EQ2 (permission correctness). The Prolog baseline and GoStore are evaluated independently at their respective engine boundaries (Table 6.1). Both implementations return the expected results for direct grants, wildcard matching, explicit deny rules, and boundary-scoped assignments within their respective test suites. This evidence establishes correctness for the tested policy patterns and fixture scenarios, though it does not constitute exhaustive verification of every possible provider policy. The one exception to the common query interface requirement is `trust_chain`. GoStore introduced bounded multi-hop traversal after the Prolog production path had already been frozen as an offline comparison baseline, so the two implementations now invoke semantically different traversal depths under the same query identifier, a deliberate scoping decision rather than an unresolved gap (Section 6.3).

EQ3 (query performance). On the production-derived corpus, GoStore ran faster than the Prolog serving path for 46 of 47 cold query-ID medians and all 47 warm query-ID medians, yielding case-level median speedups of 3.20× (cold) and 3.07× (warm). The one exception is `find_identities_with_actions`, where GoStore is slower because it re-evaluates permissions per identity rather than precomputing the matching role set once as Prolog does. This identifies a porting gap rather than an inherent GoStore limitation (Section 6.4). Memory consumption is topology-dependent: the narrow two-query serial baseline gives GoStore the higher peak, whereas catalog-wide runs serving the full catalog show the opposite. Neither comparison represents an intrinsic per-language memory ratio outside its measured serving topology.

EQ4 (interface computation path). Source-level tracing confirms that the Explorer UI, REST API, and MCP server share a single query dispatch by invoking the same `Engine.Query` implementation on GoStore. A live spot-check (Section 6.5) returned the same three identities through the REST API and the non-streaming Explorer adapter. Together, this evidence shows shared computation rather than identical adapter output, as formatting, pagination defaults, and truncation intentionally differ across interfaces. Row-count or output equivalence across all 47 catalog queries was not verified in an automated cross-adapter suite.

LLM workflow comparison (exploratory). Beyond the four evaluation questions, the head-to-head comparison between MCP-structured access and direct CasC access (Section 6.6) found that, across 5 frozen repetitions per condition (90 scored outputs per condition) using Claude Sonnet 4.6 on a non-independent prompt set that we designed, MCP scored full credit more often than RAW (79 % versus 77 %) at lower recorded API cost. Because the two conditions differ in data visibility, instruction set, and operation type, not only in the presence of MCP (Section 6.7), this result supports MCP as the more effective interface for this agent and prompt set, without establishing a general, protocol-level effect isolated from that visibility difference.

Structured MCP calls also made the supporting query IDs, parameters, pagination, and intermediate results easier to inspect and rerun than equivalent answers assembled through longer chains of shell commands and repository file reads. This is an observed traceability and verifiability property of the two evaluated workflows on this prompt set, not a proof that every MCP workflow is inherently more auditable than every raw-access workflow.

7.2 Summary of Contributions

The primary contribution of this thesis is the cross-provider schema and query interface, which addresses the central challenge introduced in Section 1.1. Without such a model, auditing access across AWS, Azure, and GCP requires security engineers to manually reconcile three distinct provider vocabularies, interpret provider-specific wildcard and negative-permission semantics, and combine separate provider-specific results by hand.

The remaining contributions operationalise and evaluate the model described above. Engine comparisons evaluate result-count parity and performance, while the LLM experiment compares observed outcomes under the MCP and RAW conditions.

Table 7.1 summarises each core objective alongside its corresponding contribution and recorded outcome. Those outcomes show that the selected IAM concepts were represented and queried at production-derived scale. However, building and maintaining the abstraction adds loader and engine complexity. This trade-off is primarily justified for recursive queries, wildcard expansion, and cross-provider joins. For the LLM comparison row in particular, the two conditions exposed unequal information to the model, so the scores reflect an aggregate system outcome rather than an isolated protocol effect.

Table 7.1: Summary of the core metamodel contribution and its supporting implementation, interfaces, and evaluation results.

Purpose	Contribution	Recorded Outcome
Unify provider-specific models	18-predicate metamodel (Section 3)	One structured query model and catalog across AWS, Azure, and GCP
Automate metamodel population	Go gathering and loading pipeline (Section 4)	Loads 141 392 entity/edge records (excluding permissions) and 1 066 044 expanded permission facts into the metamodel
Remove the Prolog serving boundary	GoStore: native Go in-memory engine (Section 4.8)	All non-search result counts matched for each provider, with one expected full-text-search difference. Median speedups of 3.20× cold and 3.07× warm
Evaluate controlled-language access	Retained DCG NLP baseline (Section 5.3)	41 query-dispatch clauses over controlled-language queries
Support interactive exploration	Explorer UI (Section 5.4)	Predicate-level navigation with CasC provenance
Provide structured tools for LLMs	MCP server with grouped tools (Section 5.5)	Exposes 46 of 47 catalog queries without defining one tool per query
Exploratory, secondary to the central research question:		
Compare structured MCP and RAW CasC access	18-prompt exploratory observational study (Section 6.6)	Across 5 frozen repetitions (90 scored outputs per condition), MCP achieved 79 % full credit (total $16.03) versus 77 % for RAW (total $22.68), scoring higher by majority vote on 4 prompts against 3 for RAW and tying on 11, though the two conditions exposed unequal information, so the scores are not protocol-isolated (Section 6.7)

The contributions above did not emerge without missteps. The following subsection records the design choices that required correction and the lessons each produced.

7.3 Engineering Mistakes and Lessons Learned

Early design choices revealed limitations during implementation and evaluation. Correcting those limitations shaped the final architecture and produced the lessons described below.

- **One tool per MCP query consumed unnecessary model context.** The initial MCP server exposed each catalog query as an individual tool, requiring separate tool schemas before the first user prompt. 46 of the 47 catalog queries were regrouped behind six category tools accepting a `query_id` parameter, with `list_actions` restricted to the Explorer interface and

the raw-query tool (`iam_raw_sql`) kept on a separate non-catalog schema. No controlled token-count artefact was retained, so this efficiency gain is reported qualitatively.

- **The MQI process boundary introduced both serialisation overhead and pipe-handling risk.** The early architecture double-serialised query results, converting Prolog terms into JSON strings before re-serialising them for HTTP responses, a path later streamlined using structured Go types. Large result sets also required continuous stdout and stderr draining to avoid deadlocks (Section 4.6). Neither issue exists in the production GoStore path, which executes queries entirely in-process.
- **The initial metamodel was too provider-specific.** The first schema version multiplied predicate types by provider, requiring new predicates rather than new facts whenever a platform was added. The schema was refactored around platform-tagged predicates representing six shared concepts, seven of which form the core model, within a complete conceptual metamodel of 18 predicates (Section 3). Provider-specific mapping now belongs in the loader, so supporting an additional provider primarily requires a new loader mapping rather than schema changes.
- **A nested permission index raised ECS Fargate RSS to 8.1 GB.** A nested map structure (`map[platform][identity][action]bool`) storing Boolean entries across platform, identity, and action combinations became costly for identities holding broad roles. Replacing it with a compact bitset reduced that deployment's footprint to approximately 1 GB, showing that indexing expanded permission relations requires compact representations. This historical intra-Go result comes from a different deployment and does not establish that GoStore uses less memory than Prolog. The controlled comparison of both serving topologies is in Section 6.4.
- **A deterministic DCG grammar is technically feasible but practically superseded by LLMs for open-ended querying.** The grammar maps each sentence to a single query and cannot chain tool calls without pre-programming each chain, whereas an LLM agent chains multiple tools adaptively. The pipeline's research value is establishing where deterministic NLP stops being practical and LLM-backed interfaces become the appropriate choice.
- **Response caches initially ignored parameters that alter query output.** In GoStore, cache keys omitted pagination and `count_only`, letting a bounded request receive a full cached list from a preceding unbounded query. In Prolog, precomputed-file lookups used only the query ID and platform, ignoring additional filters. Both defects, found during workload design, confirm that a cache key must include every parameter capable of altering the result. GoStore now incorporates response-shaping parameters into its key, and Prolog restricts precomputed files to default-filter requests, falling back to its full-parameter in-memory cache otherwise. Focused regression tests cover both cases.
- **Cross-engine comparison exposed defects in both implementations.** Initial runs revealed Prolog-side defects, including case-sensitivity issues in privilege-escalation queries and incomplete admin and trust filtering. Subsequent runs found GoStore-side defects, including ignored filters, duplicate-source counting, incomplete duty-separation logic, effect-blind provenance deduplication, and the cache-key defects above. All were resolved. In the final comparison, AWS recorded 46/47 matching result counts, Azure 46/47, and GCP 46/47, each platform with one expected full-text-search difference. This demonstrates count equivalence across all non-search catalog queries under the evaluated parameters, though matching counts alone do not establish identical record sets.

7.4 Limitations

The following limitations define the current scope boundary. For an audit tool, an excluded mechanism is not uniformly risky: omitting an unevaluated *deny*-type constraint over-reports access (a conservative false "can access" that adds noise but hides no real risk), while omitting an unevaluated *grant*-type constraint under-reports access (a false "cannot access" that silently hides a real risk). Each item below is labelled with its dominant direction, mixed where the same mechanism can attach to either an allow or a deny.

- **Azure management-group inheritance** (*under-reports access*). Azure is flattened to the subscription level, so role assignments made at management-group scope are absent from effective-permission answers. Because permissions granted by these role assignments are inherited by child scopes, omitting them can under-report access by reporting that an identity has no access even though it receives access through the management-group hierarchy.
- **AWS organisation policies** (*over-reports access*). SCPs are not evaluated when determining effective permissions. Because an SCP limits the permissions available to principals in member accounts rather than granting permissions itself, omitting it can leave access reported that the organisation policy would restrict.
- **AWS resource-based policies** (*mixed direction*). Resource-based policies are not evaluated as effective-policy sources. Omitting an allow can under-report real access, including access granted to another account, whereas omitting an explicit deny can over-report access. The direction therefore depends on the statements attached to the relevant resource.
- **GCP hierarchy-scoped deny policies** (*over-reports access*). Deny-policy resources attached to organisations, folders, or projects are not gathered, so their denies are absent from effective-permission answers, again in the conservative, over-reporting direction.
- **Conditional policy evaluation** (*mixed direction*). AWS condition keys, Azure ABAC expressions, and GCP CEL expressions are preserved as annotations but not evaluated against request context. The associated allow or deny is therefore applied without determining whether its condition holds. An unresolved condition on an allow over-reports access, since the grant may not hold at request time, while an unresolved condition on a deny under-reports access, since the restriction may not hold at request time. Which direction dominates for a given answer cannot be determined without inspecting the specific condition.
- **Intended state without drift detection.** The metamodel represents CasC assignments and identities. Live-provider gathering supplies generic catalog data rather than live assignment and identity state. The system therefore cannot compare declared access with access enforced in the deployed environment.
- **Administrative role classification by name.** The `overprivileged_identities` query and the security report classify roles as administrative by matching role names or IDs against a fixed keyword list (admin, owner, root, superuser, full access). Organisations that use non-standard or business-domain names for privileged roles may receive false negatives for these checks.

The under-reporting exclusions (Azure management-group inheritance, resource-policy allows within the mixed-direction AWS item, and conditional denies within the mixed-direction condition item) warrant the most caution, since they can silently hide real over-privilege. The purely over-reporting exclusions add false positives that a reviewer can rule out by inspecting the omitted constraint.

7.5 Extensions of the Current System

The current architecture suggests several extensions for future work, structured around four themes:

- **Permission coverage and model portability.** Extending permission coverage requires incorporating Azure management-group inheritance, AWS SCPs and resource-based policies, GCP hierarchy-scoped deny policies, and context-aware conditional evaluation. Supporting additional cloud providers or Kubernetes RBAC requires dedicated gathering and mapping logic. While existing entity types provide reusable mapping targets, each extension still requires gathering, mapping, semantic validation, and potentially provider-specific schema additions.
- **Pipeline validation and drift detection.** Capturing live identities, assignments, and relevant policy state alongside catalog data would enable automated drift detection, comparing declared CasC intent directly against live provider configuration. Incorporating a validation stage between gathering and loading would verify data integrity before metamodel population, while periodic ticker reloads could be supplemented or replaced by event-driven updates triggered by provider state changes.
- **Temporal and continuous security analysis.** Storing periodic metamodel snapshots and introducing timestamped queries would enable point-in-time privilege analysis over historical states. Scheduled integrity rules executed over the metamodel could support continuous compliance monitoring, using standards such as NIST SP 800-53 [55] as a starting point for defining rules. Replacing the curated privilege-escalation action list would require constructing and validating an action-dependency relation from provider documentation or another maintained source.
- **Alternative engines and query interfaces.** Evaluating Datalog could offer a useful performance and expressiveness middle ground between Prolog and GoStore [14]. Interface research could explore a read-only, allowlisted, and resource-limited free-form query mechanism alongside an LLM translation layer that produces either a catalog query ID with validated parameters or controlled-language input for the retained DCG pipeline. Token-optimised serialisation formats such as Token-Oriented Object Notation (TOON) [56] could be evaluated for the retained Go-to-Prolog evaluation channel and MCP tool outputs.

7.6 Broader Applications of the Metamodel Approach

The modelling approach developed in this work combines a compact shared schema, source-specific loader mappings, and derived query rules and is not limited to cloud IAM. The following candidate applications were not implemented or evaluated as part of this work. It could be applied in domains where heterogeneous schemas represent shared concepts and where the required questions involve recursive traversal, including transitive closure.

Network policy analysis is one potential application: cloud security groups, Kubernetes `NetworkPolicy` manifests, and traditional firewall rules define reachability using distinct syntaxes. Loaders could map these heterogeneous policy sources into shared reachability facts, enabling derived queries to evaluate whether one workload can reach another across multiple infrastructure layers. Threat modelling represents another candidate domain, as frameworks such as MITRE ATT&CK [57] publish relational structures connecting threat groups, software, techniques, and mitigations. Mapping these structures into a metamodel would allow derived queries to evaluate multi-step relationships, such as identifying which mitigations cover techniques associated with a specific group or software component.

Software-dependency analysis provides a third candidate: package ecosystems define dependency graphs across diverse manifest and lock-file formats. Dedicated loaders could map these ecosystem-specific dependency edges into a shared relational structure, enabling recursive rules to traverse indirect dependencies and identify, for example, which applications transitively depend on a vulnerable Log4j release. This mechanism directly mirrors the transitive trust-chain traversal implemented for IAM role assumptions in Section 5.1.

A shared metamodel is justified when heterogeneous sources represent comparable concepts and recurring queries require cross-source or recursive reasoning. Its benefits must be balanced against the effort needed to maintain source-specific loaders and the shared abstraction. Section 4.8.1 compares GoStore with a relational SQLite implementation using the same corpus and query contracts, but this thesis does not compare the metamodel against an independently designed relational or graph data model. The results therefore support the selected implementation for the evaluated catalog rather than establishing the general superiority of the metamodel approach, while demonstrating the practical feasibility of the artefact within its evaluated scope.

References

[1] Flexera. "2026 state of the cloud report", Flexera, Accessed: Apr. 29, 2026. [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud

[2] Cartography Contributors, *Cartography: Infrastructure asset inventory and relationship mapping*, GitHub repository, version 0.139.0, 2019. Accessed: Jul. 15, 2026. [Online]. Available: https://github.com/cartography-cncf/cartography/releases/tag/0.139.0

[3] Microsoft. "Azure role-based access control documentation", Accessed: Apr. 29, 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/role-based-access-control/

[4] Google. "Google Cloud identity and access management documentation", Accessed: Apr. 29, 2026. [Online]. Available: https://cloud.google.com/iam/docs

[5] Amazon Web Services. "AWS identity and access management user guide", Accessed: Apr. 29, 2026. [Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/

[6] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts", in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 164–175.

[7] E. Bobrov et al., "DevOps and its philosophy: Education matters!", in *Microservices: Science and Engineering*. Springer, 2019, pp. 349–361.

[8] Dynatrace, *Heimdall IAM export API schema and responses*, Internal API documentation and evaluated response corpus, Private snapshot, schema version 2.0.0, archived at thesis-repository commit b72ae7d6777a, 2026. Accessed: Jul. 14, 2026.

[9] SWI-Prolog. "MQI – Python and other programming language integration for SWI-Prolog", Accessed: Apr. 29, 2026. [Online]. Available: https://www.swi-prolog.org/pldoc/doc_for?object=section('packages/mqi.html')

[10] ISO/IEC, "Information technology — database languages — SQL — part 1: Framework (SQL/Framework)", International Organization for Standardization, Geneva, CH, Standard ISO/IEC 9075-1:2023, 2023. Accessed: Jul. 15, 2026. [Online]. Available: https://www.iso.org/standard/76583.html

[11] M. Gelfond and Y. Kahl, *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[12] C. Barrett and C. Tinelli, "Satisfiability modulo theories", in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[13] I. Robinson, J. Webber, and E. Eifrem, *Graph databases: new opportunities for connected data*. O'Reilly Media, 2015.

[14] E. Pasarella and J. Lobo, "A datalog framework for modeling relationship-based access control policies", in *Proceedings of the 22nd ACM Symposium on Access Control Models and Technologies (SACMAT)*, ACM, 2017, pp. 91–102.

[15] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley, Reading, 1995.

[16] SWI-Prolog. "A.18 library(fastrw): Fast reading and writing of terms", Accessed: Apr. 29, 2026. [Online]. Available: https://www.swi-prolog.org/pldoc/man?section=fastrw

[17] V. C. Hu et al., "Guide to attribute based access control (ABAC) definition and considerations", National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-162, 2014. DOI: 10.6028/NIST.SP.800-162

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models", *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[19] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control", *ACM Transactions on Information and System Security*, vol. 4, no. 3, pp. 224–274, 2001. DOI: 10.1145/501978.501980

[20] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog", *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[21] F. C. Pereira and D. H. Warren, "Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks", *Artificial Intelligence*, vol. 13, no. 3, pp. 231–278, 1980.

[22] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies", in *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 196–205. DOI: 10.1145/1062455.1062502

[23] F. Turkmen, J. den Hartog, S. Ranise, and N. Zannone, "Formal analysis of XACML policies using SMT", *Computers & Security*, vol. 66, pp. 185–203, 2017. DOI: 10.1016/j.cose.2017.01.009

[24] J. Backes et al., "Semantic-based automated reasoning for AWS access policies using SMT", in *2018 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, 2018, pp. 1–9.

[25] J. Backes et al., "Stratified abstraction of access control policies", in *Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 12224, Springer, 2020, pp. 165–176. DOI: 10.1007/978-3-030-53288-8_9

[26] A. Gouglidis, A. Kagia, and V. C. Hu, *Model checking access control policies: A case study using Google Cloud IAM*, 2023. arXiv: 2303.16688 [cs.CR].

[27] C. Parisel, *Quantifying Azure RBAC wildcard overreach*, 2025. arXiv: 2506.10755 [cs.CR]. Accessed: Jul. 15, 2026. [Online]. Available: https://arxiv.org/abs/2506.10755

[28] E. Zahoor, A. Ikram, S. Akhtar, and O. Perrin, "A formal approach for the identification of authorization policy conflicts within multi-cloud environments", *Journal of Grid Computing*, vol. 20, 2022, Art. no. 18. DOI: 10.1007/s10723-022-09606-1

[29] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan, "Quacky: Quantitative access control permissiveness analyzer", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2022, 163. DOI: 10.1145/3551349.3559530

[30] M. I. H. Sukmana, K. A. Torkura, H. Graupner, F. Cheng, and C. Meinel, "Unified cloud access control model for cloud storage broker", in *2019 International Conference on Information Networking (ICOIN)*, IEEE, 2019, pp. 60–65. DOI: 10.1109/ICOIN.2019.8717982

[31] Open Policy Agent, *Open Policy Agent (OPA) documentation*, Project documentation, 2026. Accessed: Jul. 15, 2026. [Online]. Available: https://www.openpolicyagent.org/docs

[32] J. W. Cutler et al., "Cedar: A new language for expressive, fast, safe, and analyzable authorization", *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, 2024, Art. no. 118. DOI: 10.1145/3649835

[33] NCC Group, *Principal mapper (PMapper): A tool for quickly evaluating IAM permissions in AWS*, GitHub repository, 2018. Accessed: Jul. 15, 2026. [Online]. Available: https://github.com/nccgroup/PMapper

[34] Salesforce, *Cloudsplaining: AWS IAM security assessment tool*, GitHub repository, 2020. Accessed: Jul. 15, 2026. [Online]. Available: https://github.com/salesforce/cloudsplaining

[35] NCC Group, *ScoutSuite: Multi-cloud security auditing tool*, GitHub repository, 2016. Accessed: Aug. 3, 2026. [Online]. Available: https://github.com/nccgroup/ScoutSuite

[36] Prowler Cloud, *Prowler: Open source security tool for AWS, Azure, GCP, and more*, GitHub repository, 2016. Accessed: Aug. 3, 2026. [Online]. Available: https://github.com/prowler-cloud/prowler

[37] Turbot, *Steampipe: Select where cloud meets SQL*, GitHub repository, 2021. Accessed: Jul. 29, 2026. [Online]. Available: https://github.com/turbot/steampipe

[38] CloudQuery, *CloudQuery: Open-source cloud asset inventory*, GitHub repository, 2021. Accessed: Jul. 29, 2026. [Online]. Available: https://github.com/cloudquery/cloudquery

[39] Microsoft. "Cloud Infrastructure Entitlement Management (CIEM) — Microsoft Defender for Cloud", Accessed: Jul. 22, 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/defender-for-cloud/permissions-management

[40] N. Li and J. C. Mitchell, "DATALOG with constraints: A foundation for trust management languages", in *Practical Aspects of Declarative Languages (PADL)*, 2003, pp. 58–73. DOI: 10.1007/3-540-36388-2_6

[41] M. Y. Becker, C. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language", *Journal of Computer Security*, vol. 18, no. 4, pp. 619–665, 2010. DOI: 10.3233/JCS-2009-0364

[42] TryTryAgain. "AWS IAM Actions List", Accessed: Jul. 26, 2026. [Online]. Available: https://github.com/TryTryAgain/aws-iam-actions-list

[43] zoph.io. "IAMTrail, previously known as MAMIP", Accessed: Jul. 15, 2026. [Online]. Available: https://github.com/zoph-io/IAMTrail

[44] Microsoft. "Get-azprovideroperation (az.resources)", Accessed: Jul. 6, 2026. [Online]. Available: https://learn.microsoft.com/en-us/powershell/module/az.resources/get-azprovideroperation

[45] Google. "REST resource: Roles", Accessed: Jul. 6, 2026. [Online]. Available: https://cloud.google.com/iam/docs/reference/rest/v1/roles

[46] Couchbase, Inc. "Bleve: A modern text indexing library for go", Accessed: Aug. 8, 2026. [Online]. Available: https://github.com/blevesearch/bleve

[47] M. F. Porter, "An algorithm for suffix stripping", *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[48] L. Philips, "The double metaphone search algorithm", *C/C++ Users J.*, vol. 18, no. 6, pp. 38–43, Jun. 2000, ISSN: 1075-2838.

[49] G. Stoilos, G. Stamou, and S. Kollias, "A string metric for ontology alignment", in *International Semantic Web Conference*, Springer, 2005, pp. 624–637.

[50] Anthropic. "Model context protocol (MCP) documentation", Accessed: Apr. 29, 2026. [Online]. Available: https://modelcontextprotocol.io

[51] N. F. Liu et al., "Lost in the middle: How language models use long contexts", *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. DOI: 10.1162/tacl_a_00638 [Online]. Available: https://aclanthology.org/2024.tacl-1.9/

[52] Linux Kernel Developers. "Control group v2", Accessed: Jul. 15, 2026. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v2.html

[53] S. Yao et al., *ReAct: Synergizing reasoning and acting in language models*, 2023. arXiv: 2210.03629 [cs.CL]. Accessed: Jul. 15, 2026. [Online]. Available: https://arxiv.org/abs/2210.03629

[54] T. Schick et al., "Toolformer: Language models can teach themselves to use tools", *Advances in Neural Information Processing Systems*, vol. 36, pp. 68 539–68 551, 2023.

[55] Joint Task Force, "Security and privacy controls for information systems and organizations", National Institute of Standards and Technology, Tech. Rep. NIST SP 800-53 Rev. 5, 2020.

[56] TOON Format Contributors, *TOON: Token-oriented object notation*, GitHub repository, 2025. Accessed: Jul. 15, 2026. [Online]. Available: `https://github.com/toon-format/toon`

[57] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, *MITRE ATT&CK: Design and philosophy*, Technical report, 2018.

A Acronyms

ABAC Attribute-Based Access Control

API Application Programming Interface

ARN Amazon Resource Name

ASP Answer Set Programming

AWS Amazon Web Services

BFS Breadth-First Search

CasC Configuration as Code

CD Continuous Delivery

CEL Common Expression Language

CI Continuous Integration

CIEM Cloud Infrastructure Entitlement Management

CLI Command Line Interface

CTE Common Table Expression

DCG Definite Clause Grammar

DDL Data Definition Language

ECS Elastic Container Service

FTS Full-Text Search

GCP Google Cloud Platform

GUID Globally Unique Identifier

HTTP Hypertext Transfer Protocol

IAM Identity and Access Management

JSON JavaScript Object Notation

LLM Large Language Model

MCP Model Context Protocol

MQI Machine Query Interface

NIST National Institute of Standards and Technology

NLP Natural Language Processing

RBAC Role-Based Access Control

REST Representational State Transfer

RSS Resident Set Size

SCP Service Control Policy

SDK Software Development Kit

SMT Satisfiability Modulo Theories

SQL Structured Query Language

SSE Server-Sent Events

SSO Single Sign-On

TCP Transmission Control Protocol

TOON Token-Oriented Object Notation

TTL Time to Live

UI User Interface

UML Unified Modeling Language

XACML eXtensible Access Control Markup Language

YAML YAML Ain't Markup Language

B Code Listings

This appendix contains complete or extended listings that support the main body of the thesis. It includes the query catalog registry shared across the Explorer, REST API, and MCP server (Section 5), the historical DCG query rules, and the `NormalizedItem` Go type used by the template rendering layer across the retained Prolog and production GoStore paths. Organisation-specific example records and identifiers are anonymised or synthetic. No production identifiers appear.

Query Catalog Registry Excerpt

Listing B.1 expands on Listing 5.1 from Section 5.2. While the main body defines `QueryDef` and shows a single registry entry, this appendix excerpt provides three entries spanning the cross-platform, identity, and role categories. These additional entries demonstrate how the same registry schema represents varied parameter sets and retained Prolog goal templates across query types. The complete registry contains 47 query definitions in total, so this listing remains a representative excerpt.

Listing B.1: Query catalog registry (`catalog.go`), abridged three-entry excerpt spanning three categories. See Section 5.2 for the `QueryDef`/`ParamDef` struct definitions and discussion.

```
var Registry = []QueryDef{
    {
        ID:          "cross_platform_summary",
        Name:        "Cross-Platform Summary",
        Description: "Boundaries, roles, and permission counts for an identity across
                      all clouds.",
        Category:    "cross-platform",
        Parameters: []ParamDef{
            {Name: "identity", Label: "Identity", Type: ParamTypeString,
             Required: true, AutocompleteSource: AutocompleteIdentities},
        },
        prologGoal: "nlp:structured_query_json(cross_platform_summary," +
                    "json([identity=%s]), JsonString)",
    },
    {
        ID:          "explain_access",
        Name:        "Explain Access",
        Description: "Why an identity has access to a boundary: role chain + CasC file
                      provenance.",
        Category:    "identity",
        Parameters: []ParamDef{
            {Name: "identity", Label: "Identity", Type: ParamTypeString, Required: true
                },
            {Name: "boundary", Label: "Boundary", Type: ParamTypeString, Required: true
                },
        },
        prologGoal: "nlp:structured_query_json(explain_access," +
                    "json([identity=%s, boundary=%s]), JsonString)",
    },
    {
        ID:          "trust_chain",
        Name:        "Trust Chain",
        Description: "Cross-account trust relationships for roles matching a pattern.",
        Category:    "role",
        Parameters: []ParamDef{
            {Name: "role_pattern", Label: "Role / Principal Pattern",
             Type: ParamTypeString, Required: true},
        },
```

```
        prologGoal: "nlp:structured_query_json(trust_chain," +
                    "json([role_pattern=%s]), JsonString)",
    },
    // ... further entries omitted
}
```

Definite Clause Grammar (DCG) Query Rules Excerpt

Listing B.2 extends Listing 5.2 from Section 5.3 into a seven-rule excerpt. It demonstrates how different natural-language query shapes extract intent and bind parsed values directly to the underlying Prolog predicates. This listing represents the retained historical deterministic NLP pipeline rather than the active production REST, Explorer, or MCP execution path.

Listing B.2: DCG query rules (`modules/nlp/rules.pl`), seven representative rules. See Section 5.3 for the grammar-terminal discussion.

```
% 1. List admins (cross-platform)
query(admins(Admins)) -->
    grammar:list_admins_phrase,
    { queries:find_admins(Admins) }.

% 2. Who can perform ACTION on BOUNDARY?
query(users_for_action(Action, Res, Users)) -->
    grammar:who_can_phrase,
    vocabulary:action_verb(Action),
    grammar:on_phrase,
    vocabulary:resource_name(Res),
    { queries:find_users_for_action(Action, Res, Users) }.

% 3. Can USER perform ACTION on BOUNDARY?
query(can_do(User, Action, Res, Answer)) -->
    grammar:can_phrase_with_user(User),
    vocabulary:action_verb(Action),
    grammar:on_phrase,
    vocabulary:resource_name(Res),
    { queries:check_can_do(User, Action, Res, Answer) }.

% 14. What roles does USER have?
query(user_roles(User, Roles)) -->
    (grammar:w(what), grammar:w(roles), grammar:does_phrase),
    vocabulary:user_name(User),
    (grammar:w(have) | []),
    { queries:list_user_roles(User, Roles) }.

% 19. Why can USER access BOUNDARY? (provenance / explain)
query(explain_access(User, Boundary, Explanation)) -->
    grammar:explain_access_phrase,
    vocabulary:user_name(User),
    ( grammar:w(can), ( grammar:w(access) | [] )
    | grammar:w(access)
    | grammar:w(have)
    | grammar:w(do)
    | []
    ),
    (grammar:on_phrase | []),
    vocabulary:resource_name(Boundary),
    { queries:explain_user_access(User, Boundary, Explanation) }.
```

```
% 21. Show members of team X
query(team_members(TeamPattern, Results)) -->
    grammar:team_members_phrase,
    rest_as_role(TeamPattern),
    { queries:list_team_members(TeamPattern, Results) }.

% 22. Cross-account trust for role X
query(trust_chain(RolePattern, Results)) -->
    grammar:trust_chain_phrase,
    rest_as_role(RolePattern),
    { queries:find_trust_chain(RolePattern, Results) }.

% Consume remaining tokens as role name (may be multi-word).
rest_as_role(Role, [T1, T2, T3 | _], []) :-
    atomic_list_concat([T1, T2, T3], ' ', Role).
rest_as_role(Role) --> [T1, T2],
    { atomic_list_concat([T1, T2], ' ', Role) }.
rest_as_role(Role) --> [Role].
```

`NormalizedItem` Go Type and Example JSON Output

Listing B.3 presents the `NormalizedItem` Go struct, which serves as the unified rendering envelope for normalisable result cards in the template layer, alongside an example JSON representation for a role assignment. The `ItemType` field identifies the result kind during enrichment, `Platform` identifies the cloud provider, and `Icon` supplies the rendered icon, while `Fields` holds the card body's key-value pairs. `Source` supplies CasC provenance metadata (repository, file path, and detail string) when present, and `DrillLinks` populates contextual follow-up queries that users can trigger directly from the UI, as described in Section 5.3.

Listing B.3: `NormalizedItem` type (`response.go`) and one example JSON output for a role-assignment card.

```
// NormalizedItem is the unified envelope for every normalizable result item.
type NormalizedItem struct {
    ItemType      string      `json:"item_type"`
    Platform      string      `json:"platform"`
    Icon          string      `json:"icon"`
    Title         string      `json:"title"`
    Subtitle      string      `json:"subtitle"`
    ID            string      `json:"id"`
    IsProfileLink bool        `json:"is_profile_link"`
    Fields        []ItemField `json:"fields"`
    Source        *ItemSource `json:"source,omitempty"`
    DrillLinks    []DrillLink `json:"drill_links,omitempty"`
    Score         *float64    `json:"score,omitempty"`
    DocURL        string      `json:"doc_url,omitempty"`
}

// Captured JSON output (identities_with_role, icons JSON-escaped):
// {
//   "item_type": "assignment",
//   "platform": "azure",
//   "icon": "\ud83d\udccb",
//   "title": "adam.garcia@demo.example.com",
//   "subtitle": "",
//   "id": "",
//   "is_profile_link": true,
//   "fields": [
//     {
```

```
28 //       "label": "Role",
29 //       "value": "PlatformDeployerAccess",
30 //       "code": false,
31 //       "link": "/explorer?query_id=entity_info\u0026type=role\u0026id=azure%3
            Aplatformdeployeraccess"
32 //     },
33 //     {
34 //       "label": "Boundary",
35 //       "value": "Platform Media Subscription",
36 //       "code": false,
37 //       "link": "/explorer?query_id=entity_info\u0026type=boundary\u0026id=sub-
            platform-media"
38 //     }
39 //   ],
40 //   "drill_links": [
41 //     {
42 //       "label": "Roles",
43 //       "icon": "\ud83c\udfad",
44 //       "query": "/explorer?query_id=roles_for_identity\u0026identity=adam.garcia%40
            demo.example.com"
45 //     },
46 //     {
47 //       "label": "Permissions",
48 //       "icon": "\ud83d\udd11",
49 //       "query": "/explorer?query_id=identity_permissions\u0026identity=adam.garcia%40
            demo.example.com"
50 //     }
51 //   ]
52 // }
```

C Artefact and Reproducibility Record

The machine-readable source of the quantitative metrics presented in this thesis is the consolidated summary file `experiments/thesis-evidence/results.json`. This file is generated from validated repository sources and accepted evidence bundles containing the underlying experimental records. Depending on the harness, an individual bundle contains `environment.json`, per-trial `samples.jsonl`, and run-specific configuration or result files. Together, these files record host and tool versions, exact commands, binary and dataset hashes, per-process peak resident-set-size samples, and generated summaries. The newer harnesses also record the pinned Git commit and working-tree state. The accepted evidence bundles comprise the following runs:

- **Serial benchmark and parity** (run 20260721T184930Z-780ceb1b2e2b, commit 780ceb1b2e2b, dataset SHA-256 prefix 6eff501b2c0d): provides the two-query serial baseline (Table 6.2) and the catalog parity comparison. Recorded with a clean working tree.
- **GoStore/SQLStore engine comparison** (run 20260721T191513Z-e73c88c49ac8-enginecomp): provides the underlying measurements for Section 4.8.1. Recorded with a clean working tree.
- **Catalog-stress and SSE transport runs** (20260714T160405Z-catalog-f21d3541, 20260714T145042Z-catalog-3bfbe1c6, 20260714T161453Z-catalog-fc8d89d3, 20260714T143801Z-ddb898cd): support the stress and transport evaluations. These runs predate per-run working-tree tracking in the harness, meaning clean-tree status cannot be established retrospectively.
- **LLM workflow comparison**: the full prompt set, per-run scores, and cost figures are recorded in the repository under `experiments/llm-comparison/`. Appendix D provides a redacted, self-contained prompt listing; the private experiment record retains the exact submitted wording. The exact CasC repository commit used by the RAW condition was not captured, so exact replay of that condition is unavailable from the retained artefacts. The recorded prompts, outputs, and scores still support auditing the reported comparison.

The production-shaped catalog-stress evidence comprises three catalog runs and a separate SSE transport run: the Go-only HTTP catalog run, Prolog under the production admission policy, Prolog as an overload-sensitivity case admitting up to 16 heavy requests, and the distinct Go Explorer transport experiment. Raw artefacts for these runs are committed to the repository in `experiments/production-memory/raw-evidence.tar.gz`, a compressed archive containing per-trial `samples.jsonl` memory traces, manifests, and server logs from `experiments/production-memory/runs/` and `experiments/production-memory/sse-runs/`. Incidentally captured benchmark binaries are excluded because they add no evidentiary value and can be rebuilt from source. Portable aggregates and source hashes are also incorporated into `experiments/thesis-evidence/results.json`.

Each configuration runs in a fresh user cgroup-v2 scope configured with `MemoryMax=12G` and `MemorySwapMax=0`. During execution, the harness samples `memory.current` and records the kernel's `memory.peak`. The catalog manifest encompasses 348 applicable cases across all 47 query IDs, distinguishing bounded, count-only, and unbounded response shapes. The matched pool-1/concurrency-1 runs form the catalog-wide latency comparison, yielding case-level median Prolog-to-Go request ratios of 3.20 cold and 3.07 warm. These are end-to-end request ratios that incorporate query execution, cache state, response shaping, and serialisation.

The SSE run evaluates the Go-only Explorer process with the Prolog sidecar and MQI pool disabled. Its 16 distinct large-result requests represent a transport-specific memory experiment rather than a direct counterpart to the matched engine benchmark. Expanding the Prolog pool or adding a warm

standby process measures additional serving capacity and introduces duplicate resident state. Such topologies cannot establish an intrinsic Go-versus-Prolog memory ratio.

Table C.1 lists the reproduction command for each artefact, executed from the repository root. The benchmark step runs the Go and Prolog measurements and parity comparisons serially so their peak-memory phases do not overlap. The test-evidence step refreshes the complete test results across all suites (Table 6.1), and the generation step validates the artefacts and regenerates both `experiments/thesis-evidence/results.json` and `thesis/generated/metrics.tex`.

The benchmark and generation steps use the anonymised corpus versioned within the private submission repository, whereas the test-evidence runner primarily executes test suites. Because the corpus currently has no public artefact URL or redistribution licence, public reproduction requires an archived, licensed release. While the production-memory raw evidence is versioned as described above, the serial-benchmark and test-evidence raw run directories under `experiments/thesis-evidence/` remain local and unversioned due to their large combined size and mixture of accepted and superseded diagnostic runs. Although `experiments/thesis-evidence/results.json` retains their selected aggregates and source hashes, exact auditing or reanalysis of those specific local runs requires providing them to an authorised reviewer on request.

The catalog-stress script runs the default protected topology when invoked without additional arguments, whereas the accepted Go and overload variants are reproduced using the exact parameter values retained in their manifests. The accepted catalog manifests preserve configuration names, workload hash, trial count, admission policy, and the fully expanded server-command template. The harness executes configurations serially, for the same reason as the benchmark step above, and to avoid exhausting available memory.

Table C.1: Reproduction commands for the quantitative evidence, executed from the repository root.

Artefact	Command
Serial benchmark and parity	`RUN_ID=20260721T184930Z-780ceb1b2e2b BENCH_N=30 BENCH_PL=10 VARIANCE_N=100 make thesis-bench`
Test evidence refresh	`python3 experiments/thesis-evidence/run_test_evidence.py`
Metrics and macro regeneration	`python3 experiments/thesis-evidence/generate.py`
Catalog-stress topology	`python3 experiments/production-memory/catalog_stress.py`
Explorer SSE transport experiment	`python3 experiments/production-memory/sse_stress.py`

D LLM Workflow Evaluation Prompt Set

This appendix reproduces the user-level prompts used in the LLM workflow comparison from Section 6.6. Personal and organisation-specific identifiers have been replaced with identities from the anonymised demo corpus or with synthetic resource names. These substitutions preserve each question's structure and evaluation intent but are not intended to reproduce the original production state. The private machine-readable experiment record retains the exact submitted wording, ground truth, scoring rules, and run outputs.

Table D.1: Redacted user-level prompts used in the MCP-versus-RAW workflow comparison.

ID	Category	Redacted prompt
A1	Lookup	How many distinct users are defined on each cloud platform (AWS, Azure, GCP)?
A2	Lookup	What roles does user `julia.garcia` have on GCP?
A3	Lookup	Is there a user named `ivan.robinson` on the Azure platform?
A4	Lookup	What actions are allowed by the role `OrgDevOpsRole` on AWS?
B1	Reasoning	Can user `adam.green` perform the action `s3:PutObject` on AWS?
B2	Reasoning	Which AWS roles include the `ec2:TerminateInstances` action?
B3	Reasoning	Are there any AWS roles that have zero users assigned to them?
B4	Reasoning	What is the smallest role (fewest total actions) on AWS that still includes `s3:GetObject` and `s3:ListBucket`?
B5	Reasoning	Can user `ivan.robinson` create virtual machines on both AWS (`ec2:RunInstances`) and Azure (`Microsoft.Compute/virtualMachines/write`)?
B6	Reasoning	On which platforms does user `hana.martin` have access? List the platforms and their respective roles.
C1	Operational	I am `hana.martin@demo.example.com` and I need access to GCP project `project-observability-demo`. I need `roles/cloudfunctions.admin`, `roles/cloudtasks.admin`, `roles/container.admin`, `roles/iam.serviceAccountUser`, `roles/storage.admin`, `roles/artifactregistry.reader`, and `roles/artifactregistry.writer`. The self-service page only shows `roles/custom_function_deployment`, `roles/container.admin`, `roles/editor`, and `roles/iam.serviceAccountUser`. Should I open a manual PR to the GCP CasC repository to add the missing roles?
C2	Operational	I am `hana.martin@demo.example.com` and I need to assign Contributor on resource group `rg-assistant-demo` and AcrPush on registry `assistantdemoacr` to my service principal `gh-actions-assistant-demo` (application ID `00000000-0000-4000-8000-000000000001`) in subscription `Engineering-North`. I lack `Microsoft.Authorization/roleAssignments/write`. How can I do this?
C3	Operational	I have AWS access with `OrgReadOnlyRole`. It gives Support Center access but not AWS Health events. `OrgDevOpsRole` has Health access. Should I switch to `OrgDevOpsRole`, or add `health:*` permissions to `OrgReadOnlyRole`?
C4	Robustness	What roles does user `nora.nonexistent` have on AWS?
C5	Robustness	What roles are available on the Oracle Cloud Infrastructure (OCI) platform?

Continued on the next page

Table D.1 continued

ID	Category	Redacted prompt
D1	Stress test	User `ivan.robinson@demo.example.com` is leaving the company. Compile a complete offboarding checklist: (1) list every role assignment on all three platforms (AWS, Azure, GCP), (2) list every team membership, and (3) for each access grant, indicate whether the access-management system would automatically revoke it when the user's IdP account is deleted or whether it requires a manual PR to the CasC repository.
D2	Stress test	Which AWS users hold at least one role that grants `iam:PutRolePolicy` or `iam:AttachRolePolicy`? For each such user, how many distinct AWS account boundaries do they have access to (their blast radius)? Rank the results from highest to lowest blast radius.
D3	Stress test	`julia.garcia@demo.example.com` has access to multiple GCP projects. For each GCP project she has access to, list which other users share at least one of her roles in that project. Identify any role–project combinations where she is the only holder, since these are potential orphaned-ownership risks.

E Declaration of AI Tool Usage

During the process of writing this thesis and developing the software artefacts, I used AI tools for code generation, debugging assistance, data analysis, and brainstorming and editing of text structure. The AI tools used include Claude (Anthropic), Codex (OpenAI), and GitHub Copilot.

Software implementation. For the Go and Prolog code implementation, Claude (Anthropic) and GitHub Copilot were used to generate boilerplate code, draft individual functions, and assist in porting query logic between languages during the GoStore migration. AI tools were also used to write test scaffolding and to provide debugging assistance, particularly for diagnosing deployment failures on AWS (such as stack overflows in JSON serialisation and the 64 KiB pipe-buffer deadlock) and for memory profiling of the Go engine. The web UI styling (HTML/CSS templates) was developed with AI assistance. At the beginning of the Prolog implementation, I also used these AI tools to assist in understanding the Prolog language and its features. Generated code was reviewed, adapted, and validated by me through the test suites and the `cmd/compare` parity tool described in Section 4.8.

Data analysis. AI tools were used to write and adapt prototype scripts for data analysis, as well as to sanity-check derived numbers, such as fact counts and benchmark tallies, against the raw result files. All analysis code was reviewed, modified, and executed by me, and the outputs were verified against the raw source data.

Thesis text and typesetting. Claude and Codex were used for brainstorming the content structure and section scope, as well as to check grammar, consistency in writing and style. For typesetting, AI assisted in generating and debugging LaTeX constructs and TikZ diagrams, which I specified and verified. AI helped cross-check chapters against the implementation code and review the correctness and clarity of the written text.

Research support. AI tools were used to help discover candidate literature and brainstorm design alternatives, including the engine comparison. These AI suggestions were treated strictly as leads rather than authoritative sources. I personally read and verified every source cited in this thesis.

What AI was not used for. I selected the research problem, designed the 18-predicate metamodel, decided to replace the production Prolog engine with GoStore, designed the evaluation, and made the final interpretation of all results. No AI editing or assistant tool fabricated, selected, or altered experimental data or reported results. This is distinct from the LLM (Claude Sonnet 4.6) evaluated as the experimental subject in Section 6.6. All analysis was executed and verified by me. No AI-generated text appears verbatim in this thesis.