UNITEC INSTITUTE OF TECHNOLOGY

School of Computing

Master's Thesis

Enhancing RAG Agent Performance through Structured Knowledge Bases from NLP-Parsed Data Warehouses

Duc Phu Nguyen

Principal Supervisor: Dr. Lei Song

Associate Supervisor: Dr. Hamid Sharifzadeh

A thesis submitted in partial fulfilment of the requirements
for the degree of **Master of Applied Technologies**

Date: 25 January 2026

Abstract

Enterprise teams need correct, explainable answers from governed data warehouses (DWs), but Natural-Language-to-SQL (NL2SQL) systems often fail when schemas evolve, constraints are complex, and policies must be enforced consistently. Current RAG-only approaches and unconstrained NL2SQL do not reliably enumerate eligible options or enforce procurement policies under schema change. In ten controlled IT procurement user cases, schema-aware execution recovered the full budget-compliant device pools and enforced auto-approval rules with 0% violations, while a RAG-only baseline omitted eligible devices and occasionally exceeded budget at rank 1. This thesis investigates how a schema-aware hybrid pipeline can provide grounded, auditable answers for budget-constrained and policy-constrained IT procurement decisions over a realistic warehouse slice. The proposed design scopes each interaction to a safe, read-only query surface, retrieves schema and policy context, and generates a single validated `SELECT` statement. It combines dense vector retrieval over documentation and metadata with a Qdrant index, a lightweight knowledge graph for entity and relationship hints, and a per-session PostgreSQL warehouse slice that encodes request budgets and departmental auto-approval rules.

To assess effectiveness, three pipelines are implemented and compared: a RAG-only pipeline that answers directly from retrieved text; a SQL-only pipeline that queries the session warehouse through a fixed, validated `SELECT` template; and the proposed Hybrid pipeline that couples semantic retrieval with constrained, warehouse-backed execution. Using ten controlled user cases (A–J) spanning departments, roles, quarters, and new versus replacement requests, the evaluation measures (Q1) correctness of the returned budget-fitting device pool and (Q2) correctness and efficiency of standard auto-approvable shortlists. Against a deterministic SQL baseline, the SQL-only and Hybrid pipelines

recovered the exact Q1 device sets for all cases (65 devices total), while the RAG-only pipeline returned 49 devices and matched the expected set in 2 of 10 cases (mean absolute count error 1.6). Top-1 budget compliance was 10 of 10 cases for the SQL-only and Hybrid pipelines and 8 of 10 for the RAG-only pipeline. For Q2, only the SQL-backed pipelines (Experiments 2 and 3) were evaluated because auto-approval is represented as governed warehouse attributes (for example, `auto_approve_ok`) that Experiment 1 cannot access or enforce reliably from text alone; both SQL-only and Hybrid matched the baseline auto-approved pools in all cases and returned 0% of devices that were outside budget or not auto-approved. Although they return identical Q2 decisions, compared with SQL-only, the Hybrid pipeline reduced median end-to-end latency from 9.45 s to 5.77 s and median total tokens from 7,457 to 3,656 (about 51% lower).

These results show that constraining generation to a schema-scoped and policy-scoped context can deliver accurate, constraint-respecting answers at lower operational cost, without sacrificing auditability. The thesis contributes: (i) a governance-first NL2SQL pattern with static name, join, type, and policy checks plus evidence tracing; (ii) an empirical comparison of RAG-only, SQL-only, and Hybrid pipelines on a realistic IT procurement workload; and (iii) reusable per-session design patterns for organisations seeking to deploy schema-aware RAG in regulated, data-intensive environments.

Acknowledgments

This thesis has been completed with the help, guidance, and support of many people. I am grateful to all who contributed in different ways to the development and completion of this work.

I am deeply indebted to my principal supervisor, Dr Lei Song, whose guidance, support, and expertise enabled me to complete my Master's degree within the time available to me as an international student. His timely advice and encouragement helped me stay focused on the core objectives of the project and navigate challenges constructively. Throughout the research, he showed patience and provided clear direction that greatly supported the progress of this work. I feel very fortunate to have had him as my mentor. I also wish to acknowledge my associate supervisor, Dr Hamid Sharifzadeh, whose thoughtful advice, careful critiques, and steady support have been invaluable throughout this research.

I extend my thanks to Unitec Institute of Technology, in particular the School of Computing, for providing a supportive academic setting and the facilities required to complete this work. I am grateful to the library staff for their help and for ensuring access to the literature and resources needed for this thesis.

Completing this Master's degree would have been far more difficult without the support of a few important people in my life. My partner, Timmie, has put up with my stress, late nights, and occasional panic with patience, kindness, and steady encouragement. I am deeply grateful for her understanding, especially in the final months of writing, when her support helped me keep going. I am also grateful to my daughter, Mia, whose smiles and affection eased the pressure of this journey and reminded me why this effort was worthwhile.

The deepest thanks go to my parents, whose love, care, and steady support have made all of my achievements possible and whose belief in my abilities has motivated me to pursue this academic path with determination. It is with heartfelt gratitude that I dedicate this thesis to my mum and dad. I am also deeply thankful to my parents in law for their continued encouragement, support, and prayers. These spiritual values are extremely precious to me and reminded me that I was never facing this journey alone.

Above all, I am grateful to God Almighty for granting me the strength, knowledge, perseverance, and direction needed to complete this research.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

ANN Approximate Nearest Neighbour

ANSI American National Standards Institute

API Application Programming Interface

AQL Automatic Query Language

BERT Bidirectional Encoder Representations from Transformers

CDC Change Data Capture

CPU Central Processing Unit

DBMS Database Management System

DW Data Warehouse

ELT Extract, Load, Transform

ETL Extract, Transform, Load

GPU Graphics Processing Unit

HNSW Hierarchical Navigable Small World

IQR Interquartile Range

JSON JavaScript Object Notation

LLM Large Language Model

MRR Mean Reciprocal Rank

NL2SQL Natural Language to SQL

NLP Natural Language Processing

NZ New Zealand

NZD New Zealand Dollar

OCR Optical Character Recognition

OLAP Online Analytical Processing

ORM Object Relational Mapping

PDF Portable Document Format

PK Primary Key

QAA Query AutoAwesome

RAG Retrieval-Augmented Generation

REST Representational State Transfer

RFID Radio Frequency Identification

RL Reinforcement Learning

RPA Robotic Process Automation

RQ1 Research Question 1

RQ2 Research Question 2

RQ3 Research Question 3

SQL Structured Query Language

UI User Interface

USD United States Dollar

VRAM Video Random Access Memory

XAI Explainable Artificial Intelligence

Chapter 1

Introduction

1.1 Background and Inspiration

Enterprise data warehouses (DWs) store key operational and analytical data in structured, governed environments. They support decisions in IT procurement, budget governance, compliance, and service operations. Accessing DW data typically requires structured query language (SQL) and detailed knowledge of schemas, joins, and business rules. This creates a gap between expert users, who can translate questions into valid SQL, and non-technical users, who understand the decision they need to make but not the technical structures.

Recent advances in large language models (LLMs) and Natural Language to SQL (NL2SQL) systems promise to reduce this gap: users state questions in natural language and models generate candidate SQL queries. Retrieval-Augmented Generation (RAG) extends this idea by introducing semantic search over documentation, schemas, and examples so that models can ground their outputs in relevant context.

Deploying such systems in real enterprise environments remains challenging. In governed settings such as IT procurement, errors are costly: queries must respect department budgets, purchasing policies, role-based eligibility, and approval rules, while staying aligned with the authoritative device catalogue and budget tables. Schemas evolve, source systems are noisy, and policy definitions can be incomplete or distributed across documents and tables. A practical solution must operate robustly under these conditions, not only on static benchmarks. This thesis is motivated by the distance between promising NL2SQL/RAG methods and the demands of governed enterprise DWs.

1.2 Research Gap

Current NL2SQL and RAG-based systems are difficult to apply reliably to enterprise workloads. Three limitations are central:

(a) Separation of retrieval and generation. Many pipelines treat retrieval (over text or mixed artefacts) and SQL generation as loosely coupled steps. Retrieved context is not tightly aligned with the active schema, join paths, or constraint set (for example, request budgets and auto-approval rules), which weakens grounding and leads to brittle queries.

(b) Limited schema and policy awareness. Many approaches assume stable schemas and weak or implicit business rules. In realistic DWs, table design, naming conventions, joins, and mandatory constraints (such as budget caps, department policies, and eligibility filters) are essential to correct answers. Systems that do not bind generation to the actual schema and rules tend to produce invalid joins, omit required predicates, or select from inappropriate tables.

(c) Lack of governance-first, auditable pipelines. Most implementations do not provide (i) an explicit evidence trace linking the question, retrieved context, and final SQL; (ii) pre-execution checks for schema and policy compliance; and (iii) a controlled query surface that restricts models to safe, read-only statements. This is problematic in governed procurement scenarios, where outputs must be explainable, defensible, and cost-aware.

This thesis addresses these gaps by designing and evaluating a *schema-aware hybrid pipeline* that couples semantic retrieval, a lightweight knowledge graph, and a per-session slice of the warehouse into a single, controlled query-generation process.

1.3 Research Questions

We investigate whether a schema-aware and policy-aware hybrid pipeline improves robustness and practicality compared with simpler baselines (RAG-only and SQL-only). This study addresses three research questions:

1. **RQ1 (Retrieval and grounding):** Does schema-aware retrieval improve evidence recall and grounding compared with RAG-only or SQL-only approaches?
2. **RQ2 (SQL validity and answer quality):** Does the hybrid pipeline generate a higher proportion of executable, constraint-respecting SQL and more accurate final answers than both baselines?
3. **RQ3 (Efficiency and cost):** What latency and token-cost differences arise when using the hybrid pipeline compared with a baseline NL2SQL approach without schema-aware retrieval?

These questions align with three themes that run through the thesis: trust and governance (grounded, auditable outputs), correctness of SQL under realistic constraints, and practical efficiency for interactive use.

1.4 Motivation

Many organisations want non-technical staff, such as team leads, service desks, and administrators, to obtain reliable answers from complex DWs without learning SQL or internal schema details. This pressure is strong in procurement workflows, where each query may inform purchasing decisions, budget compliance, and approval routing, with direct operational and budget consequences.

Text-only RAG can surface policies and examples but often lacks precise links to live tables, keys, and per-request budget rules. NL2SQL without strong schema and policy binding can produce syntactically valid but semantically incorrect or non-compliant queries. Both patterns lead to recurring problems: fragile queries that fail under real schemas, inconsistent enforcement of constraints, and answers that are difficult to audit.

A schema-aware hybrid approach offers a more disciplined alternative. By constraining each interaction to a per-session slice of the schema, device catalogue, and policy rules, the system can (i) restrict what the model sees and is allowed to query, (ii) retrieve only entities and constraints that matter for the current request, and (iii) generate a single read-only `SELECT` statement that is statically validated before execution. This design is directly relevant for IT procurement, where users ask both for the complete budget-fitting device pool (Q1) and for standard auto-approvable choices for a role and department (Q2), while expecting consistent decision logic and traceable evidence.

1.5 Methodology

This thesis follows a design, build, and evaluate methodology grounded in the gaps, research questions, and evaluation dimensions developed in Chapter 2. Section 2.5 identifies four themes in the literature: scalability and efficiency; multilingual and multimodal support; trustworthy domain-aware AI; and usability and adaptivity. Section 2.6 then links the reviewed literature back to the three research questions on retrieval grounding, SQL validity, and efficiency, supported by the evaluation framework in Section 2.8. Together, these themes and questions define the requirements for the proposed pipeline and guide how it is designed and assessed in later chapters.

The first stage is the design of a schema-aware RAG pipeline built on a PostgreSQL data warehouse and a Qdrant vector store. The system ingests structured procurement tables (budgets, employees, departments, requests, and device catalogue rows) together with policy and decision metadata into a linked knowledge base that supports grounded question answering and SQL generation. Each user session operates within a scoped schema view so that only relevant tables and columns are exposed to the model, which reduces token usage and limits hallucinated field names. Chapter 3 presents this methodology as an eight-stage workflow: requirement analysis and data-warehouse construction, schema and document ingestion, embedding and indexing, user query normalisation, dynamic sub-warehouse construction, retrieval and re-ranking, prompt construction, and SQL generation with validation against the scoped schema.

The second stage is implementation and controlled experimentation based on this pipeline design. Using the same warehouse slice and policy logic, three configurations are instantiated that differ only in how they retrieve and filter candidates: (i) a RAG-only configuration that answers from vector-retrieved device context without executing SQL, (ii) a SQL-only configuration that issues a single validated `SELECT` over a session-scoped warehouse schema, and (iii) the proposed Hybrid configuration that combines semantic retrieval and a lightweight knowledge representation with the same constrained execution path. The detailed experiment setup is reported in Chapter 4.

The final stage is evaluation against realistic procurement-style cases. Ten controlled user scenarios (A–J) vary department, role, quarter, and request type (new or replacement) to exercise different budget caps and approval conditions. For Q1, a deterministic SQL baseline defines the expected budget-eligible device set under the per-case request budget. For Q2, the expected outputs are assessed in terms of whether the system selects devices that are both within budget and marked as standard auto-approvable for the role and department under the current rules. The three configurations are compared on: (i) device pool correctness for Q1 using order-agnostic expected-set comparison, count error, and Top-1 budget compliance; (ii) Q2 behaviour for auto-approval shortlists in the schema-aware designs; and (iii) efficiency measured by end-to-end latency and token counts summarised with Tukey's five-number statistics. Chapter 4 describes the dataset, experimental design, and setup. Chapter 5.2 reports the quantitative results for RQ1–RQ3 and discusses their implications in relation to the themes developed in Chapter 2.

1.6 Contributions

In response to the identified gap and questions, this thesis makes the following contributions:

1. **Schema-aware hybrid pipeline for NL2SQL in IT procurement.** An end-to-end architecture that integrates semantic vector retrieval, a lightweight knowledge graph, and a per-session data warehouse slice into a single NL2SQL pipeline for budget-fit retrieval and auto-approval recommendations. The design constrains model visibility and output to a controlled, read-only query interface.

2. **Governance-first validation and evidence tracing.** Static checks over names, joins, types, and policy rules are applied before execution, coupled with an evidence trace from question to retrieved context to final SQL, supporting auditability and explainability.

3. **Empirical comparison with RAG-only and SQL-only baselines.** Three pipelines, namely RAG-only, SQL-only, and the proposed Hybrid, are evaluated on retrieval grounding, answer correctness under budget and approval rules, and efficiency (latency

and token cost) using controlled procurement-style user cases.

4. **Reusable implementation patterns.** A concrete per-session schema, prompt templates, validation rules, and workflow patterns that can be adapted by practitioners aiming to deploy schema-aware RAG for governed enterprise querying in other regulated or data-intensive domains.

1.7 Thesis structure and roadmap

This thesis is organised as follows. Chapter 2 reviews data warehousing, database querying, and AI-based querying approaches, and synthesises the literature to identify the key gaps that motivate this study. Chapter 3 presents the proposed methodology and the schema-aware hybrid pipeline, including how the per-session warehouse slice, retrieval index, and validation steps work together to support grounded NL2SQL. Chapter 4 describes the dataset, experimental design, assumptions, and the three configurations (RAG-only, SQL-only, and Hybrid) used for comparison. Chapter 5 presents the quantitative results for RQ1–RQ3 and discusses their implications for retrieval grounding, constraint and policy enforcement, and efficiency in governed procurement workflows. Chapter 6 concludes by summarising the main findings, stating the thesis contributions, and outlining directions for future work. The appendices provide supporting artefacts and technical details, including SQL templates and additional workflow assumptions.

Chapter 2

Literature Review

This chapter reviews the foundations needed to deploy Natural Language to SQL (NL2SQL) and retrieval-augmented generation (RAG) in governed enterprise data warehouses. It first surveys data-warehouse creation strategies, including architecture and schema design choices, then examines how rules, structures, and governance practices shape query safety, constraint enforcement, and auditability in practice. It then reviews traditional approaches to querying databases and how AI-based methods, including NL2SQL and RAG, extend these approaches to support natural-language access, with attention to grounding, common failure modes under schema change, and evaluation practices. The chapter concludes with a brief synthesis that links the reviewed literature back to the research gap and research questions defined in Chapter 1, and summarises the evaluation dimensions used in later chapters.

2.1 Strategies for Creating a Data Warehouse

A well-structured data warehouse depends on clear strategies that guide its design, integration, and maintenance. The literature presents several complementary approaches for achieving reliability, scalability, and governance in data-driven systems. These include process-driven modelling that links business workflows to data structures, architectural choices such as Kimball, Inmon, and Data Vault 2.0, disciplined ETL / ELT and DataOps practices, adoption of managed cloud services for automation, and frameworks for data governance, quality, and analytics. Studies also extend these strategies to real-time and IoT integration and to hybrid environments that combine NoSQL and Big Data platforms. The following subsections summarise representative methods and findings from prior work, outlining common principles, strengths, and trade-offs that underpin modern data-warehouse development.

2.1.1 Integrated Process-Driven Strategy

A process-driven strategy aligns DW structures with the business processes that produce and consume data. The *AS-IS* view from source systems (data-driven) and the *TO-BE* view from business goals (demand-driven) are modelled together so that process changes propagate to entities, keys, and facts in a controlled way. The Integrated Process-Driven (IPD) approach couples process models with data models to keep lineage and accountability visible during change (Kaldeich & Sá, 2004).

In practice, core processes are scoped (e.g., intake, pricing, approvals) and mapped by actors, events, and documents. Each process yields candidate entities, measures, and constraints, which are validated against available source data to avoid speculative modelling. Gaps are recorded and handled through staging logic when source data needs reshaping. This cycle, moving from the process view to the data view and then back again, helps reduce later mismatches in business intelligence and analytics.

IPD improves maintainability. When rules change (for example, thresholds or compliance attributes), updates flow from the process map to data contracts and then to physical objects. Studies of process-aware DW methods report that documenting flows, responsibilities, and checkpoints shortens debugging time and lowers defect rates in ETL (Dhaouadi et al., 2022).

2.1.2 Select Appropriate Architecture and Schema Design Strategy

Architecture determines how sources are integrated, governed, and queried. This part compares established approaches and notes common analytic schema patterns.

Kimball (bottom-up). Dimensional data marts are delivered for priority domains and integrated over time using conformed dimensions. Benefits include fast BI delivery, simple star schemas, and strong performance for aggregation-heavy workloads. Risks include fragile cross-domain integration when conformance slips and governance overhead as many marts evolve in parallel (Kimball & Ross, 2013).

Inmon (top-down). An integrated, normalised enterprise data warehouse is designed first; dimensional marts are then published downstream. Benefits include enterprise consistency and centralised governance from the start. Risks include longer time-to-value and more upfront modelling while the core stabilises (Inmon, 2005).

Data Vault 2.0 (DV2.0). The raw integrated layer is organised into *Hubs* (business keys), *Links* (relationships), and *Satellites* (context/history), fed from a Landing/Staging area and exposed through dimensional marts in a Presentation layer. Reported benefits include change tolerance, auditability, parallel loads, and strong automation potential on cloud ELT stacks; added indirection and more joins in the raw layer are typical trade-offs, which require modelling discipline and automation to remain efficient (Naderi, 2024).

Schema patterns (star vs. snowflake). For analytics, star schemas reduce join depth and are easy to understand. Snowflaking normalises some dimensions to reduce redundancy and can help maintain conformance at scale. Selection should follow query patterns and maintenance goals. Across approaches, layered ETL/ELT with explicit orchestration remains essential for data quality, lineage, and maintainability (Dhaouadi et al., 2022).

Table 2.1 summarises common trade-offs often cited when selecting layered architectures and scoped schemas.

Table 2.1. *Architectural and schema strategies and their main trade offs for governed data warehouses.*

Approach (source)	Integration path	Strengths	Trade-offs / fit
Kimball (Bottom-up) [K]	Dimensional marts → bus (conformed dims)	Fast BI; simple stars; strong aggregation performance	Conformance can drift across domains; governance overhead with many marts; good when quick wins matter.
Inmon (Top-down) [I]	EDW (3NF) → dimensional marts	Enterprise consistency; centralised governance	Longer upfront modelling; slower initial value; best when integration is the primary goal.
Data Vault 2.0 [DV]	Landing/Staging → Raw DV (H/L/S) → Presentation	Change-tolerant; auditable; automation-friendly; parallel loads	More joins in raw layer; needs modelling discipline and automation; strong fit for evolving/compliant sources.
Star schema [K]	Dimensional (denormalised)	Shallow joins; easy for BI/NL2SQL; performant scans	Some redundancy; manage SCDs and conformance.
Snowflake schema [K]	Dimensional (normalised dims)	Less redundancy; can help conformance at scale	Deeper joins; may lower query speed; use when maintenance wins outweigh cost.
Layered ETL/ELT [D]	Staging → Core → Marts	Quality gates; lineage; safer changes	More moving parts; needs orchestration and testing; essential in regulated settings.

Notes (sources): [K] (Kimball & Ross, 2013); [I] (Inmon, 2005); [DV] (Naderi, 2024); [D] (Dhaouadi et al., 2022).

2.1.3 Reliable ETL/ELT and DataOps practices

Reliable ETL and ELT processes underpin a data warehouse. The literature treats extraction, staging, transformation, and loading as the core of the lifecycle and notes that weak design in this area is a frequent source of cost, delay, and quality defects

(Dhaouadi et al., 2022). In ETL, data is transformed before loading, whereas in ELT raw data is loaded first and heavier processing is carried out downstream on scalable engines. Regardless of pattern, clear modelling of flows, explicit metadata, lineage, and auditability are emphasised. Incremental movement via change data capture (CDC) is preferred after the initial load to reduce latency and redundancy, and load steps should be idempotent (e.g., MERGE/upsert) to support safe reruns (Dhaouadi et al., 2022).

In practice, implementations specify the extraction mode (full or incremental) and the CDC technique, such as audit columns, log mining, or keyed deltas, based on the source system's capabilities and the required freshness of the data. A staging or data preparation area isolates cleaning and harmonisation (type standardisation, code mapping, deduplication, outlier handling) before loading analytic targets (star/snowflake) and managing history through slowly changing dimensions (Dhaouadi et al., 2022). DataOps introduces engineering discipline across these steps: version control for pipeline code and configuration; automated tests at multiple levels (schema/contract checks, rule-level unit tests, data-quality assertions); CI/CD to build–test–promote; central orchestration with observability (run status, durations, row counts, freshness and completeness alerts); and environment parity to reduce drift (Bahaa et al., 2021).

These practices improve dependability and maintainability. Incremental loads cut processing time and cost, layered staging localises risk and eases rollback, and systematic tests and monitoring surface defects early. Formalised change management, which includes classifying schema changes as compatible or breaking, planning deprecations, and assessing downstream impacts, helps reduce disruption and supports reproducible analytics. Reviews converge on the view that combining disciplined ETL/ELT with DataOps controls yields more predictable delivery cycles and higher analytical data quality (Dhaouadi et al., 2022; Bahaa et al., 2021).

2.1.4 Use Managed Cloud Services To Automate Data Pipelines

Managed platforms reduce friction in ingest, storage, and publish. In a Fabric-style setup, data lands in a Lakehouse and can be "loaded to tables," then organized into staged schemas (e.g., stage, control, data vault, and information mart). This layered arrangement supports repeatable ELT and clearer lineage, and it aligns well with DV2.0 patterns (Hubs, Links, Satellites) for auditable change and parallel loads (Naderi, 2024).

To meet scale and scan-heavy analytics, column-oriented NoSQL stores are a good fit as warehouse back-ends. They provide straightforward data scalability and are well suited to decisional queries, while OLAP functionality is commonly added through engines such as Hive or Kylin for cube computation, which enables distributed, high-throughput analysis over very large datasets (Dehdouh et al., 2020).

Modern cloud warehouses also separate concerns at the execution layer. Storage-separated designs (e.g., Snowflake) scale compute and storage independently, while state-separated designs (e.g., Microsoft POLARIS) fully decouple state from compute. On-demand state separation allows a running query to migrate between workers without losing progress, improving elasticity and fault tolerance for variable workloads (Winter et al., 2022). In our later methodology, these principles motivate a layered, automation-friendly stack with controlled ELT, and they explain our choice to isolate session-bound warehouses while keeping final SQL read-only and reproducible.

2.1.5 Data Governance, Quality, and Advanced Analytics Strategy

Data governance and data quality make a warehouse dependable. Governance clarifies roles, ownership, access, and lifecycle; quality controls ensure correctness, completeness, timeliness, and conformance. Reviews of industrial practice note persistent gaps in methodology, metadata, and traceability, and argue that systematic governance and quality management improve reliability and auditability across the data lifecycle (Freitas et al., 2025).

In practice, implementations align three threads. First, *metadata and lineage*: sources, keys, and business definitions are catalogued, and lineage is recorded from ingestion through publish so that changes remain explainable (Freitas et al., 2025). Second, *quality gates*: freshness thresholds, row-count deltas, integrity checks (primary/foreign keys), duplicate control, and outlier screening are applied at staging and again before publish (Freitas et al., 2025). Third, *operational visibility and feedback*: run logs and validation summaries are surfaced to BI/OLAP so stakeholders can monitor processes and enact continuous improvement, as reported in case-based work on data-driven process enhancement (Rogers et al., 2017). For analysis, dimensional models and OLAP cubes support monitoring and drill-down, while predictive components (e.g., scenario and forecasting models) provide forward-looking views for planning in complex domains (Ekeh et al., 2025). Studies on cloud warehousing also describe forms of automation, such as AI-assisted routines or RPA-style tasks, that streamline data preparation, anomaly handling, and compliance reporting. These approaches help reduce manual effort at scale (Machireddy, 2023).

These practices improve dependability and usefulness. Consistent governance and explicit lineage raise auditability; quality gates reduce downstream defects; visibility links data operations to business actions; and OLAP plus predictive analytics extend decision support beyond descriptive reporting. Across the surveyed literature, a common pattern emerges: formalise governance and metadata, enforce quality checks throughout the pipeline, and augment analysis with OLAP and predictive methods while leveraging cloud-era automation where appropriate (Freitas et al., 2025; Rogers et al., 2017; Ekeh et al., 2025; Machireddy, 2023).

2.1.6 Real-Time and IoT Data Integration Strategy

Real-time and IoT-style integration emphasises low-latency ingestion and up-to-date visibility, where operational events must be captured, structured, and delivered reliably from edge systems into the analytical store (Sahara & Aamer, 2021). In these settings, higher volume and velocity increase the risk of missing events, inconsistent identifiers, and schema drift, so validation, standardised event structure, and lineage capture become important before data are exposed for downstream analytics (Sahara & Aamer, 2021). At scale, platforms that support distributed ingestion and processing mainly matter because they intensify data-quality and management challenges, which strengthens the need for governance controls and quality gates (Silva et al., 2021). The key implication is that real-time pipelines make freshness, traceability, and controlled publication core requirements, which aligns with constraints faced by governed schema-aware querying and NL2SQL/RAG systems.

2.1.7 NoSQL and Big Data Warehousing Strategy

NoSQL stores (e.g., document and column families) handle high volume and variety, but analytical use still needs structure. The literature proposes extracting implicit structure from semi-structured records and then designing a dimensional schema for analysis (Bouaziz et al., 2019). MapReduce-based extraction over JSON collections can recover candidate attributes and relationships, which are then organised into facts, dimensions, and hierarchies for a warehouse schema (Bouaziz et al., 2019).

In practice, implementations follow a simple path: select the NoSQL source type; run schema extraction/profiling to capture attributes (including embedded documents); build an intermediate structure graph to identify entities and links; derive multidimensional concepts (facts, measures, dimensions); and load into analytic targets (star/snowflake). This approach treats schema discovery as a reverse-engineering step and supports later conformance and history management for analytics (Bouaziz et al., 2019).

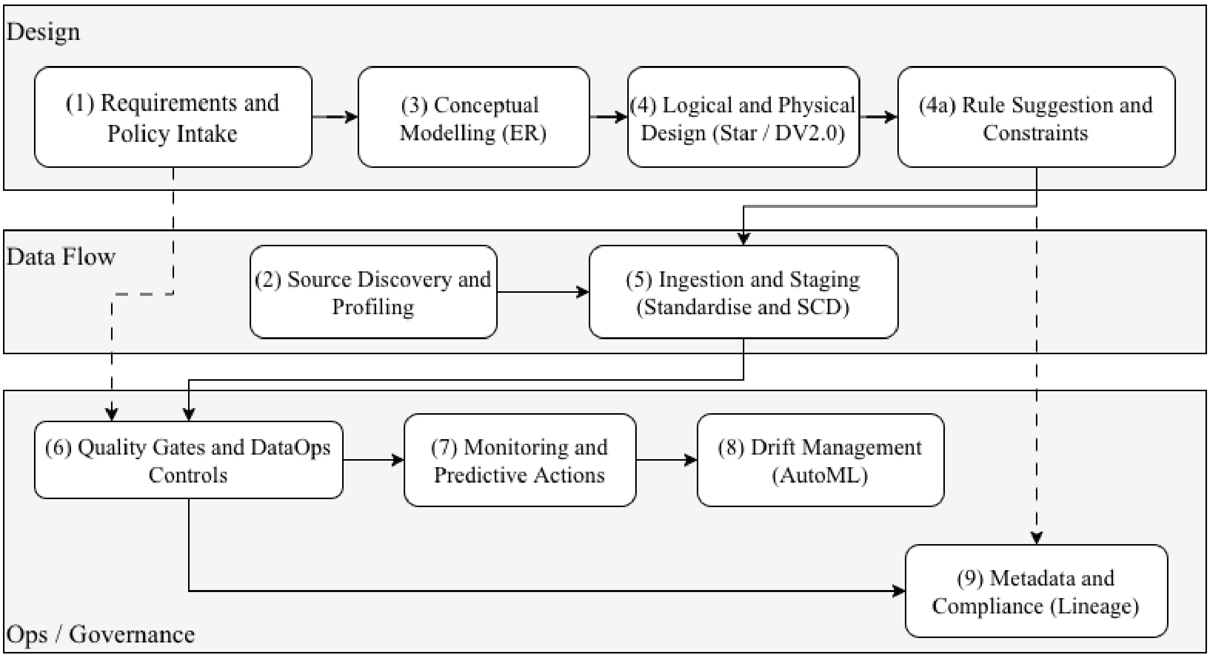
At platform level, Big-Data ecosystems combine NoSQL with distributed SQL/OLAP engines to meet throughput needs. Demonstration work shows that integrating Hive with MongoDB and Cassandra provides flexible schemas, parallel processing, and distributed storage suitable for multi-source, high-volume analytics; OLAP and BI are layered on top to serve varied users and queries (Ngo et al., 2019). Across the studies, a consistent pattern emerges: extract structure from NoSQL data, model it dimensionally, and deploy it on scalable engines, while retaining the governance and quality controls outlined earlier.

2.2 Building Data Warehouse with AI (Rules & Structures)

Research reports describe how artificial intelligence supports the *rules and structures* of data warehousing. Studies cover methods that infer schemas from heterogeneous sources, translate requirements into logical and physical designs, suggest and validate data-quality constraints, and maintain conformance, lineage, and documentation. Other strands address operations: monitoring pipelines in real time, detecting drift, triggering retraining, and automating routine changes. Reported benefits include faster modelling cycles, more consistent rule application, and improved traceability, with trade-offs around validation on real data, portability across technologies, and the need for human oversight.

We organise this review into four threads: (i) adaptive integration and quality pipelines; (ii) real-time monitoring and predictive controls; (iii) scalable AutoML orchestration for warehouse operations; and (iv) metadata-driven governance. Figure 2.1 provides a synthesis view of the patterns reported across these threads.

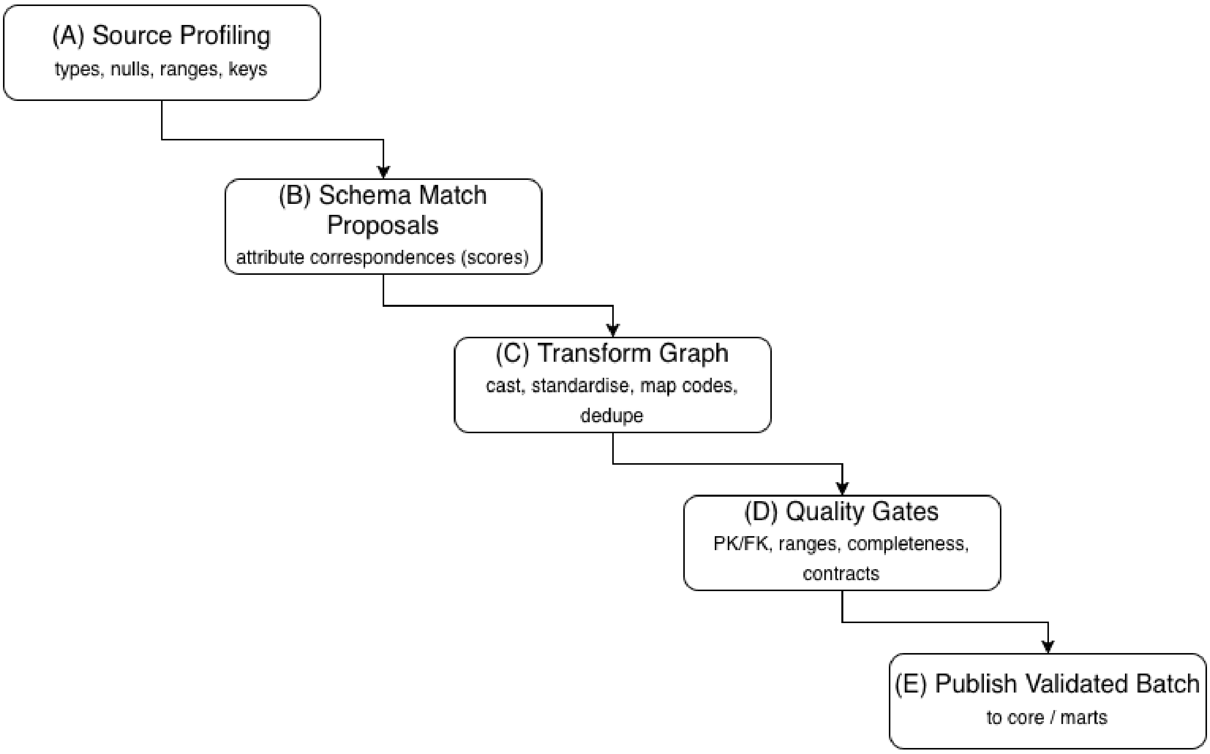
Figure 2.1. *Design, data flow, and governance for AI assisted data warehouse rules and structures*



Notes. Nodes 2, 4a, 5, and 6 highlight the adaptive integration and quality pipeline, node 7 the real time monitoring and predictive control loop, node 8 the AutoML orchestration, and node 9 the metadata intelligence and governance pipeline that Sections 2.2.1 to 2.2.4 describe.

2.2.1 Adaptive Integration and Quality Pipelines

Figure 2.2. *Adaptive integration and data quality pipeline*



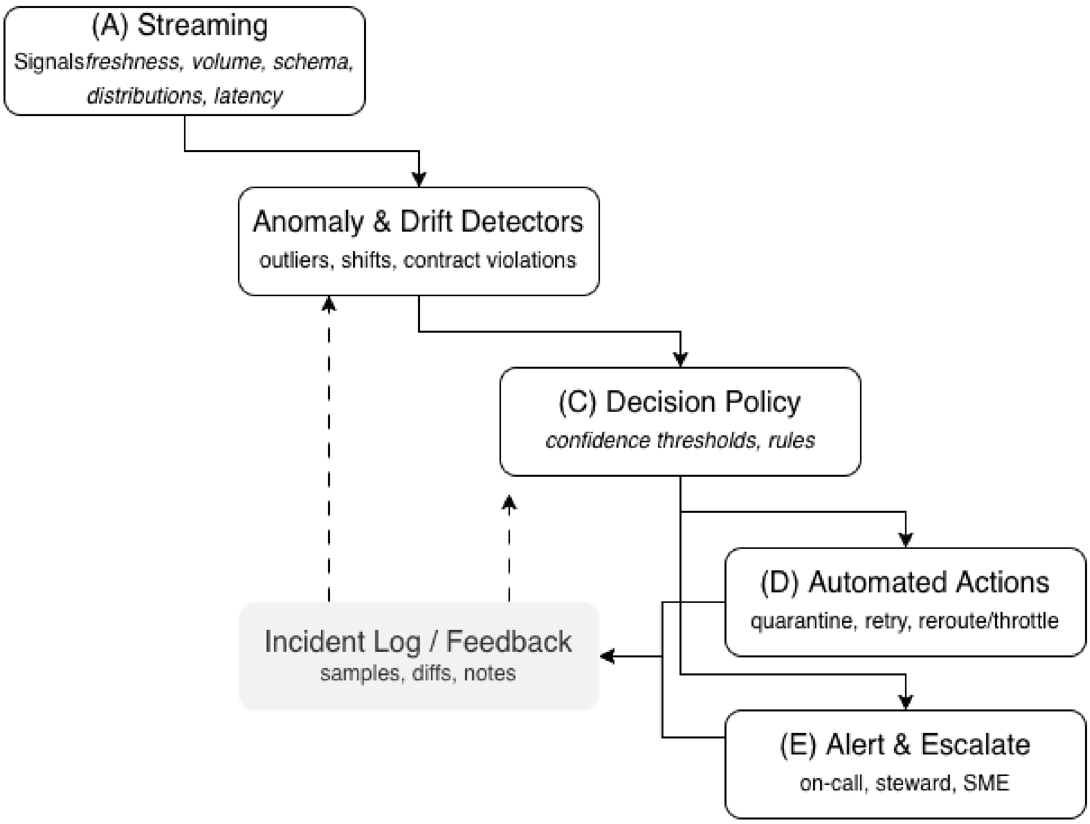
Notes. Sources are profiled (A), matches are proposed (B), a transform graph is generated (C), quality gates are applied (D), and validated batches are published (E). Steward review for low confidence matches, incident logging, and lineage capture at publish correspond to nodes 2, 4a, 5, and 6 in Figure 2.1.

Work in this strand examines how learning methods are embedded in ETL/ELT so that source data are profiled early, mappings are suggested rather than hand-coded, and quality checks run close to ingestion (see Fig. 2.1 and the internal flow in Fig. 2.2). Studies report systems that learn schema correspondences from names, types, and structural context; assist cleaning through detection of duplicates, missingness, and inconsistent codes; and surface anomalies or distribution shifts during load (Rachakatla et al., 2021; Gadde, 2020). A common effect is to shorten time-to-analytics in near–real-time settings by proposing transforms and thresholds that reflect current data characteristics rather than fixed rules (Gadde, 2020).

Across implementations, the main focus falls on the ingestion and staging boundary. Source systems are profiled to capture data types, null patterns, ranges, and key candidates. Possible attribute matches are scored and reviewed, and a transform graph is produced or refined to standardise types and units, harmonise enumerations, map codes, and remove duplicates. Quality checks covering integrity, completeness, ranges, and contract rules are applied before the data are published to core layers or marts. Feedback loops use operator decisions and incident logs to refine matchers and adjust thresholds so that subsequent batches need less manual retuning (Rachakatla et al., 2021; Gadde, 2020). Reported benefits include faster onboarding of new feeds, more consistent rule application, and earlier detection of bad data (Gadde, 2020; Rachakatla et al., 2021). Typical caveats are dependence on representative training data, variation across tools and engines, and the need for human review where confidence is low or business rules change; in practice, authors combine learned suggestions with explicit contracts and an approval step for mappings and transforms that affect downstream analytics.

2.2.2 Real-Time Monitoring and Predictive Action

Figure 2.3. *Real time monitoring and predictive action*

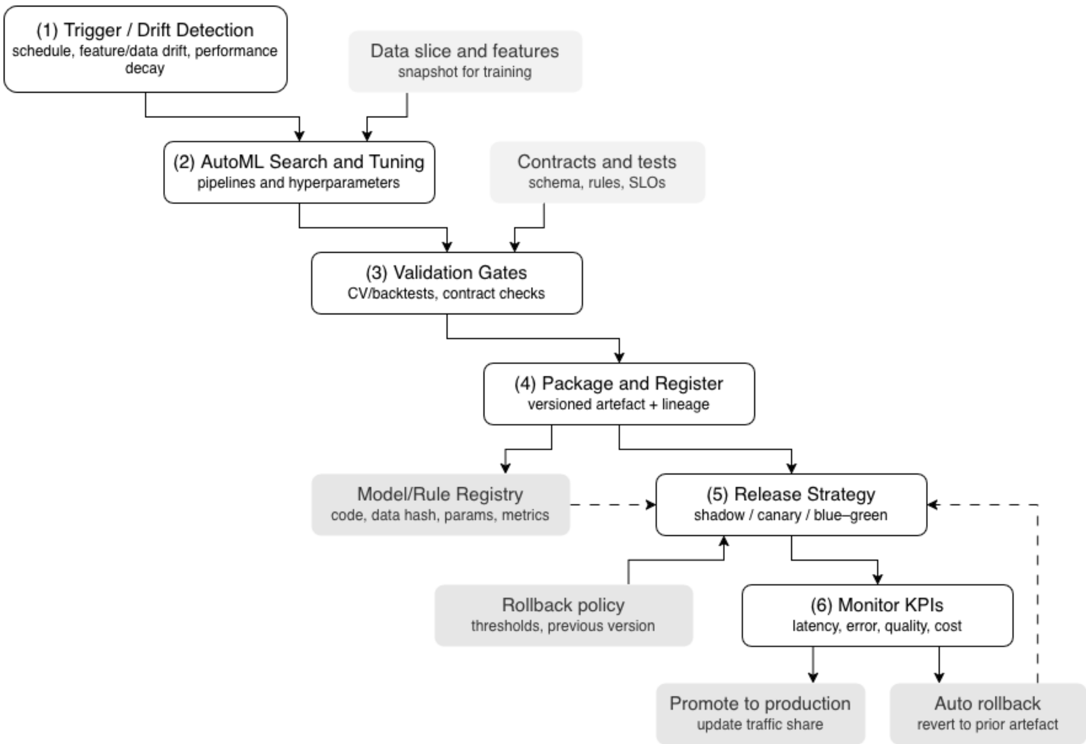


Notes. Pipelines emit signals (A), detectors flag anomalies and drift (B), a policy layer decides (C) and routes to automated actions (D) or human escalation (E), and incidents feed back to improve detectors and policies. This monitoring loop corresponds to node 7 in Figure 2.1.

Prior work embeds machine-learning monitoring into data pipelines to track freshness, volume, schema conformance, distribution drift, and latency, then routes anomalies to a policy layer that can trigger controlled actions or human review (Gadde, 2020). A common pattern is a feedback loop where incidents are logged with evidence and outcomes, and thresholds or contracts are updated to reduce repeated failures over time (Gadde, 2020; Seethala, 2020). The key relevance for governed querying is that monitoring outputs support audit trails and conservative, policy-gated decisions before downstream analytics or NL2SQL/RAG components consume the data (Gadde, 2020).

2.2.3 Scalable AutoML Orchestration

Figure 2.4. *AutoML orchestration for drift management*



Notes. The pipeline triggers search and tuning, validation, packaging and registration, release (shadow, canary, or blue green), and KPI monitoring, with registry and rollback policies controlling promotion and rollback outcomes. This orchestration corresponds to node 8 in Figure 2.1.

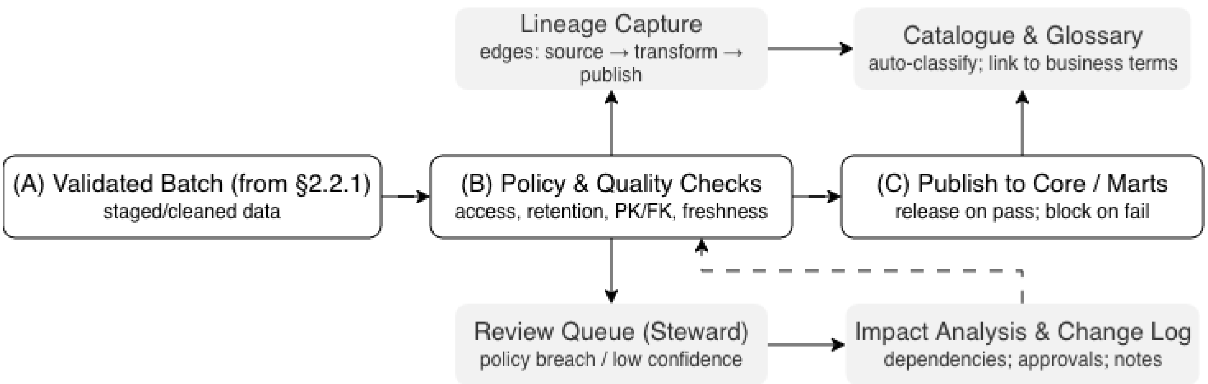
This strand of work reviews automation frameworks that use AutoML to retrain and deploy models or rule sets as data and workloads change (see the orchestration component in Fig. 2.1 and the internal flow in Fig. 2.4). Descriptions emphasise CI/CD integration in which triggers (time-based schedules, feature/data drift, or performance decay) start searches over pipelines and hyperparameters; candidate solutions are validated with cross-validation/backtests and data-contract checks before any release is considered (Machireddy, 2022).

A common pattern is to package a versioned artefact with full lineage, including code, training data slice, parameters, and metrics, into a registry, and then use controlled rollout strategies such as shadow, canary, or blue-green deployments to manage risk. Live KPIs (latency, error, quality, cost) are monitored, with automatic rollback to a prior artefact when thresholds are breached. Reports also note elastic resource orchestration so that training and inference scale to meet budgeted time or cost targets while preserving reproducibility through the registry and build logs (Machireddy, 2022).

Across these accounts, benefits include faster iteration, more consistent releases, and traceability from data to decision. Common cautions include search cost, dependency drift across environments, and the need for human approval when changes affect business rules. Because of these issues, authors emphasise the importance of governance guardrails in the AutoML loop, including promotion reports, explicit rollback policies, and audit trails (Machireddy, 2022).

2.2.4 Metadata Intelligence and Governance

Figure 2.5. *Metadata intelligence and governance pipeline*



Notes. After staging, policy and quality checks gate publish; lineage is captured and assets are catalogued with automatic classification and glossary links. Breaches enter a steward review loop with impact analysis and change logging. This flow corresponds to node 9 in Figure 2.1.

This strand of work reviews AI-assisted metadata platforms that automate how assets are described, traced, and checked against policy (see the governance component in Fig. 2.1

and the internal flow in Fig. 2.5). Contemporary systems apply NLP and machine learning to auto-classify new assets, link them to business glossaries, maintain end-to-end lineage graphs, and run continuous validation that flags policy or quality issues early (Yang et al., 2025). In regulated settings such as healthcare, studies describe metadata-centred pipelines that help manage sensitivity and compliance while supporting near–real-time operations (Seethala, 2020). Reviews from heavy-industry domains likewise emphasise that robust metadata repositories and provenance records are practical prerequisites for scaling AI and analytics beyond pilots (B. Yu et al., 2024).

Across reports, a typical pattern emerges. Technical and statistical metadata are extracted on ingest (schema, types, nulls, distributions, sensitivity hints); assets are classified and linked to business terms; and lineage is updated from source through transformations to publish as jobs run. Before release to core layers or marts, executable policy and data-quality checks (access tiers, retention, sensitivity, freshness, completeness, PK/FK) gate publication, with low-confidence cases routed to steward review. Catalogue entries, lineage views, validation results, and change notes are retained to support audit and reproducibility (Yang et al., 2025; Seethala, 2020). Reported benefits include lower manual effort, improved transparency and auditability, and stronger compliance readiness through continuous checks and lineage, with caveats around accurate metadata extraction, training data, and governance overhead in legacy estates; many authors therefore recommend clear escalation paths and lightweight approval workflows for policy-relevant changes (Yang et al., 2025).

2.3 Methods of Querying Databases

This section surveys three families of methods reported in the literature. First, *direct (low-level) SQL* provides full control over selection, joins, and execution plans; studies report strengths in precision and auditability, and note risks around schema dependence and safety. Second, *abstraction and high-level querying* raises the interface above raw SQL using ORMs, functional and visual builders, and AI-assisted refinements; prior work highlights productivity gains alongside performance and transparency trade-offs. Third, *NoSQL and modern querying* extends beyond relational stores with document, key–value, wide-column, and graph databases, and introduces API layers such as GraphQL and hybrid routing; publications emphasize flexibility and scale with interoperability and tuning challenges.

The following subsections summarize representative techniques and findings from prior studies. Our aim is to clarify terminology and typical trade-offs, and to provide background that motivates the research questions and later evaluations in this thesis.

2.3.1 Direct (Low-Level) Querying Approaches

Direct SQL (including parameterised queries and stored procedures) provides deterministic control over selection, filtering, and joins, which supports analytical precision and strong auditability in data warehouse environments. Because logic is explicit and executable, it is well-suited to repeatable operations such as validation, aggregation, and audit logging, and it can enforce constraints that are difficult to guarantee through automated tools (Pasimeni, 2019). However, direct querying depends on accurate schema knowledge and stable join paths, and it is sensitive to schema evolution and vendor-specific syntax, which limits accessibility for non-expert users and motivates higher-level NL2SQL approaches that must still preserve traceability and correctness.

2.3.2 Abstraction and High-Level Querying Approaches

High-level querying tools abstract the complexity of raw SQL by allowing users to work through code frameworks, functions, or visual interfaces rather than writing statements directly. Object–Relational Mapping (ORM) systems are one of the most common forms of abstraction. They translate objects in programming languages into relational data structures, which allows developers to manipulate data through native class methods instead of SQL statements. Hule and Ranawat (2023) highlight that ORM tools reduce the need for developers to manage low-level database access, improving productivity and code maintainability. By mapping entities and relationships automatically, ORMs simplify the connection between application logic and data storage, enabling faster development cycles and cleaner code structure. However, their study also notes that ORMs can introduce additional overhead, leading to slower query execution, increased memory usage, and longer load times when interacting with large datasets. The authors conclude that while ORMs are beneficial for general development efficiency, their performance cost becomes noticeable in data-intensive or time-critical systems.

Beyond ORMs, abstraction has also evolved through functional and visual query builders. These tools offer predefined functions such as `select()`, `where()`, or `join()` that construct SQL statements safely and programmatically. Query builders lower the chance of syntactic or injection errors, helping maintainers enforce consistent patterns across projects. A further step toward accessibility is visual query building, where users compose queries through drag-and-drop interfaces rather than text-based commands. Russell-Rose and Shokraneh (2020) propose that such visual paradigms improve transparency and reduce human error, particularly when constructing long Boolean expressions or multi-table searches. Their 2DSearch framework allows users to visualize query components as objects on a canvas, combine them through graphical operators, and immediately see the effects in real time. The study reports that visual abstraction reduces syntax errors and enhances understanding of query semantics, particularly for non-expert users or knowledge workers who rely on structured search for decision support.

Comparative research supports these findings. Svarre and Russell-Rose (2025) evaluated visual and form-based query builders across professional information specialists. Their user study found that visual interfaces encouraged exploratory behavior, reduced query formulation time, and increased completeness by including more facets and terms per query. Participants rated the visual interfaces as easier to adjust and more pleasant to use, as they offered better overview and flexibility. However, experienced users still valued the transparency of form-based systems because they could directly inspect and verify the underlying Boolean syntax. This suggests that effective abstraction should balance automation with visibility, allowing users to see or edit the generated SQL when needed.

More recently, artificial intelligence has been introduced to enhance query abstraction. Query AutoAwesome (QAA) is an AI-driven system that automatically improves existing SQL statements using schema and data context. It analyses a query's structure, generates multiple enhanced variants, and ranks them using objective functions that measure consistency or diversity of results Suryavanshi et al. (2019). For example, QAA can replace literals with alternative parameters, combine related queries, or add semi-joins based on foreign-key relationships. Experimental results show that these enhancements can expand coverage or improve retrieval efficiency without human intervention. This model illustrates that AI can extend abstraction beyond simple syntax reduction, because it can actively optimise and generalise queries while preserving their semantic intent.

Overall, abstraction and high level querying approaches aim to make data access more intuitive and efficient. ORMs and function based builders help developers integrate databases into application logic in a safe and structured way, while visual and AI assisted tools support understanding, collaboration, and flexible exploration. These benefits, however, come with costs in performance, transparency, and dependency management. In modern data warehouse environments, abstraction works best as a hybrid layer that offers simplicity for routine tasks but still allows users to inspect, validate, and tune SQL for mission critical analytical workloads.

2.3.3 NoSQL and Modern Querying Approaches

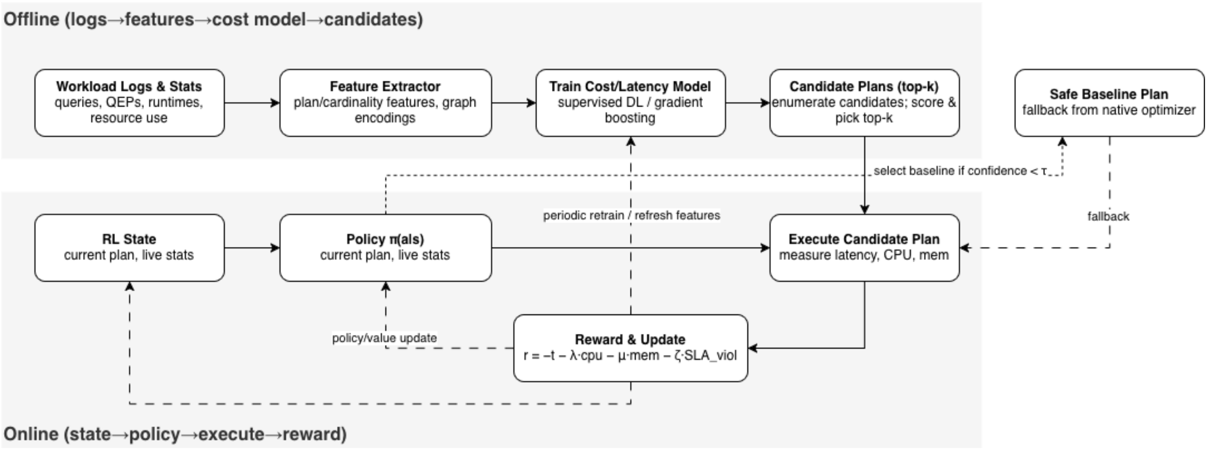
NoSQL systems support high-volume, heterogeneous data by relaxing rigid relational modelling, but this flexibility often shifts complexity to querying and governance because structures are implicit, inconsistent, or only partially standardised across platforms (Asaad, 2023). Compared with relational warehouses, weaker or evolving schema constraints can make it harder to ground generated queries to valid fields and relationships, increasing the risk of missing join logic, mis-typed attributes, or unvalidated assumptions when using NL2SQL methods (Asaad, 2023). Prior work therefore emphasises explicit metadata layers and validation mechanisms that recover structure, map user intent to executable queries, and enforce constraints across heterogeneous back ends (Asaad, 2023; Ren et al., 2023). From a governed NL2SQL/RAG perspective, the key implication is that schema flexibility increases grounding difficulty, so controlled context and schema-aware checks become more important as data models diversify.

2.4 Querying Databases with AI

Recent studies report three complementary trends for making databases easier to query and faster to execute: (i) learning-based plan and resource optimisation; (ii) natural-language interfaces that translate user intents into executable SQL; and (iii) retrieval-centric methods that bring schema and policy context into the generation loop. The subsections summarise reported evidence across each trend and, where helpful, include compact workflows or formulas.

2.4.1 AI-Driven Query Optimisation

Figure 2.6. *AI driven query optimisation pattern with offline learning and an online policy controlled loop*



Notes. An offline loop learns cost and latency models from workload logs to rank candidate plans, while an online loop selects among these plans using a policy with a confidence gate and a safe fallback to the native optimiser.

Figure 2.6 shows a two loop pattern reported across recent systems. In the offline loop, a cost and latency model is learned from workload logs and used to propose top-k candidate plans. In the online loop, a policy $\pi(a \mid s)$ chooses among these plans at runtime, guarded by a confidence gate and a safe fallback to the native optimiser when confidence is low or a service level agreement is breached. Rewards from observed execution update the policy and periodically retrain the cost model.

Confidence gate and fallback. Execute the policy plan only when confidence exceeds τ and there is no SLA risk:

$$a_t = \begin{cases} \arg\max_a \pi(a \mid s_t), & \text{if } \mathrm{conf}(\pi, s_t) \geq \tau \ \wedge \ \neg \mathrm{SLA\_viol}_t, \\ a_{\mathrm{baseline}}, & \text{otherwise}, \end{cases} \tag{2.1}$$

where $\mathrm{conf}(\pi, s_t)$ is a model confidence score, $\mathrm{SLA\_viol}_t$ is the SLA-violation indicator, and a_{baseline} is the plan from the native optimiser.

Reward shaping.

$$r_t \;=\; -\,\text{latency}_t \;-\; \lambda\,\text{CPU}_t \;-\; \mu\,\text{MEM}_t \;-\; \zeta\,\mathbf{1}\{\text{SLA}_t \text{ violated}\}, \tag{2.2}$$

with non-negative weights λ, μ, ζ penalising CPU, memory, and SLA breaches. Rewards update the policy online; the offline cost model is periodically refreshed from new logs.

Learning-based optimisers use workload logs to train cost/latency predictors and generate better execution plans than static heuristics. Reported results include up to a 40% reduction in query time and 30% lower memory use (with CPU also reduced) when machine learning is applied to plan selection (Panwar, 2024), and up to 40% faster responses in a deep-learning–guided planner (Gadde, 2023). To adapt online, reinforcement learning (RL) policies choose among candidate plans and update from live rewards; recent work shows RL can optimise query execution and resource allocation while remaining explainable, and workload generators can simulate future demand to prime the policy (Noor & Abbas, 2024). Together, offline cost modelling and online RL yield both steady-state and adaptive gains in evolving workloads.

2.4.2 AI-Enhanced SQL Query Processing

AI-enhanced SQL query processing introduces learning mechanisms to improve how queries are planned, tuned, and executed. Traditional optimisers rely on fixed cost models and static rules; recent studies show that machine learning can make these processes more adaptive and accurate by learning from workload behaviour and historical performance data. Learned models capture relationships between query structure, data distribution, and execution time to refine plan selection and cost estimation (Panwar, 2024).

Research on adaptive indexing extends this learning approach to access paths. Instead of predefined or manually maintained indexes, AI-driven systems analyse query frequency and access patterns to create or drop indexes dynamically. This self-tuning process adjusts to workload drift and helps sustain throughput without manual intervention. Experimental findings report performance gains of around 30% over static methods (Gadde, 2022).

In distributed and cloud-based engines, RL has been explored for join ordering, operator choices, and resource allocation based on feedback such as latency or node utilisation.

Generative models simulate future workloads to train and stress-test policies ahead of peaks. Explainable AI (XAI) surfaces the rationale behind actions to support operator trust and governance (Noor & Abbas, 2024). Across the literature, common themes include data quality for model training, low-overhead integration with existing engines, and safeguards to avoid regressions (Panwar, 2024).

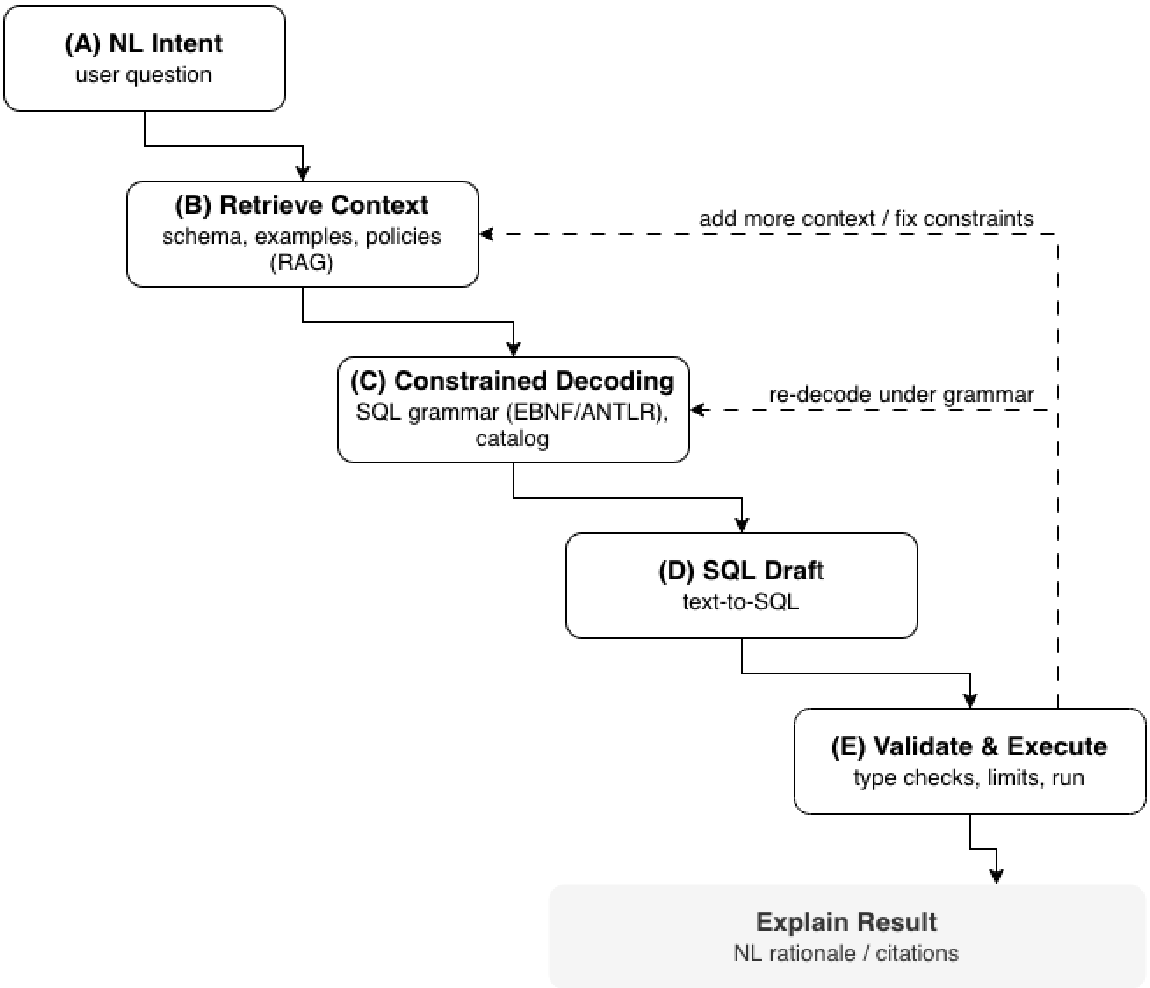
2.4.3 Conversational Querying

Conversational querying applies natural-language interfaces to data lakes and warehouses so that users can retrieve information without knowing SQL syntax. AI/NLP components interpret intent and translate requests into executable statements, narrowing the gap between technical access and business information needs (Gadde, 2020). By mapping natural expressions to query structures that run on existing platforms, studies report improved usability and broader participation in routine analytical tasks (Gadde, 2020).

Recent work demonstrates integration patterns in which a conversational layer connects to warehouse or lake engines via APIs for text-based or voice-based interaction. The system interprets input, constructs the corresponding query, executes it, and returns results in readable form, supporting near–real-time responses and lowering the barrier to self-service access (James, 2024). Reported challenges include handling ambiguity in user requests and ensuring privacy and governance when exposing sensitive data through natural-language endpoints (James, 2024).

2.4.4 Natural Language to SQL

Figure 2.7. *NL2SQL core workflow with retrieval, constrained decoding, and validation*



Notes. The system retrieves schema and rule context via RAG, decodes under a constrained SQL grammar to draft a query, then validates and executes it. Dashed paths indicate refinement loops when checks fail, and the system returns a natural language explanation of the results.

Constrained decoding. Let $\mathcal{G}_t$ be the set of grammar-allowed tokens at step t (from an EBNF/ANTLR parser), $\mathcal{C}$ the catalogue-allowed tokens (tables, columns, operators), $\mathcal{A}_t = \mathcal{G}_t \cap \mathcal{C}$ the admissible set, $\mathcal{V}$ the full vocabulary, and ℓ_v the model logit for token v. We apply a masked softmax:

$$p(y_t = v \mid y_{<t}, x) = \frac{\exp(\ell_v)\,\mathbf{1}[v \in \mathcal{A}_t]}{\sum_{u \in \mathcal{V}} \exp(\ell_u)\,\mathbf{1}[u \in \mathcal{A}_t]},$$

which sets illegal tokens to zero probability and reduces syntactic errors (Troy et al., 2023).

Natural Language to SQL (NL2SQL) allows users to express information needs in plain language and obtain answers from relational data. Recent systems pair large language models with retrieval so that generation is grounded in the target schema and domain rules. A common design vectorises tables, columns, relationships, and business policies; semantic search supplies this context to the generator before decoding. The model then proposes SQL from the retrieved context and runs checks, which helps on large schemas or when business logic is complex (Steve Jeffrey, 2024).

A two-step validation loop is typical. First, a *syntactic* check ensures the statement parses and respects engine constraints. Second, a *semantic* check evaluates whether results align with user intent and business rules (e.g., join paths, filters, groupings); on mismatch, the system retrieves additional context and refines the query (Steve Jeffrey, 2024). Grammar-constrained decoding works alongside retrieval by enforcing a SQL grammar during generation, which helps stabilise models against malformed output. The catalogue mask then restricts tokens to tables and columns that are in scope (Troy et al., 2023). At execution time, systems commonly apply guardrails such as read-only mode (SELECT only), parameterised literals, row limits, and timeouts before sending the statement to the database.

Across the literature, the strongest results come from combining (i) schema/rule retrieval to ground the model, (ii) iterative validation to correct or refine outputs, and (iii) grammar/parser constraints to enforce SQL syntax. Reported caveats include sensitivity to ambiguous intents, added latency from refinement loops on complex prompts, and the need to refresh domain rules and catalogues as they evolve (Steve Jeffrey, 2024).

2.4.5 Autonomous, Self-Tuning Databases

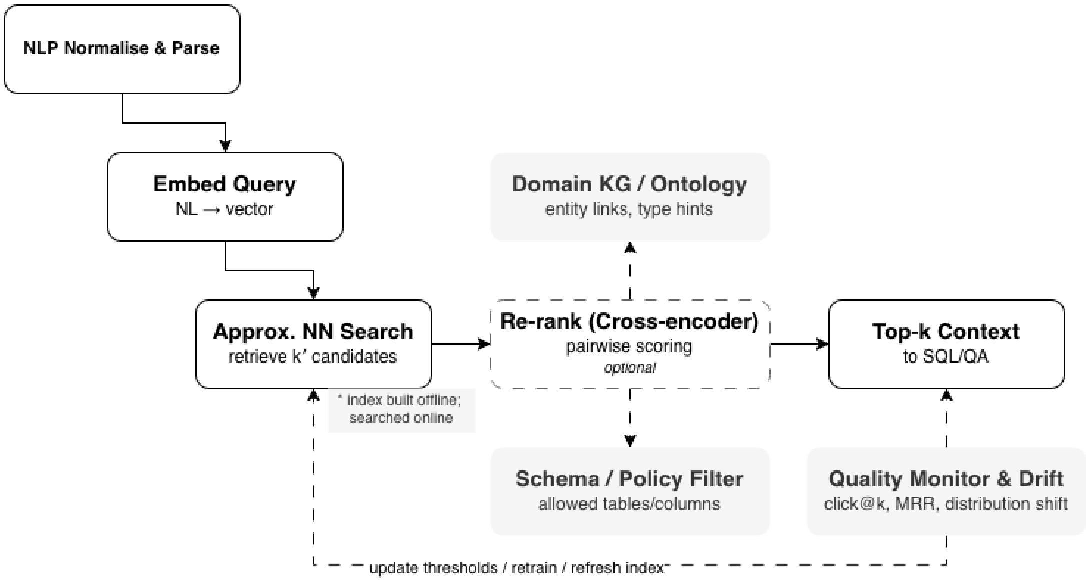
Autonomous, self-tuning databases aim to adjust configuration and optimisation strategies without manual intervention. Recent work describes systems that monitor workload and adapt plans in near real time, moving routine tuning into the engine (Panwar, 2024).

A common pattern is continuous analysis of query performance with automatic parameter adjustment. This includes updating optimiser settings as workloads change and using learned signals to keep execution efficient (Panwar, 2024). Studies also describe predictive-maintenance features that surface emerging bottlenecks and hardware issues earlier, reducing downtime and stabilising service levels (Panwar, 2024). In addition, automated access path management, which creates, modifies, or drops indexes in response to observed queries, is presented as a key element of scalable, self-managing operation (Panwar, 2024).

These approaches promise more consistent performance with less manual effort. Reported limits include privacy considerations around detailed logs, computational overhead for training and inference, and integration effort in existing deployments. The literature recommends balancing model complexity against operational cost and adopting changes in phases to avoid regressions (Panwar, 2024).

2.4.6 Semantic Search with AI-Driven Vector Embeddings

Figure 2.8. *Semantic retrieval pipeline for schema and policy aware context*



Notes. Natural language questions are mapped to embeddings, compared against embedded documents in a vector index, and combined with domain context to return semantically similar, context aware content.

Semantic search focuses on meaning rather than exact keywords. Vector embeddings map text into high-dimensional representations so that queries and documents can be compared by semantic similarity, which has become a common foundation in modern retrieval systems (Singh, 2024).

In practice, natural-language processing (NLP) interprets user questions and prepares them for retrieval. Systems analyse the query, map it into an embedding, and fetch semantically similar content using learned similarity measures (Srivastava et al., 2023). This shifts matching from token overlap to meaning, which helps with paraphrases and varied phrasing.

Domain context improves results further. Knowledge graphs and ontologies make entity relationships explicit; combined with embeddings and NLP, they guide retrieval towards context-aware matches instead of surface keyword hits (Srivastava et al., 2023). This is useful when terms have multiple senses or when schema and business rules must be respected.

At scale, deployments report familiar concerns: preparing high-quality training data, keeping models reliably deployed, and monitoring retrieval quality over time. Addressing data preparation, retraining, and drift detection early helps maintain stable performance as content and queries evolve (Srivastava et al., 2023). Beyond search, the same NLP techniques support automated question answering and customer-support workflows where intent and domain vocabulary matter (Olujimi & Ade-Ibijola, 2023).

2.4.7 Generation-Oriented: Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) couples retrieval with generation so that a model answers using grounded context rather than only its parametric memory. A retriever selects relevant passages (e.g., from documents or prior records), and the generator conditions on those passages to produce the final response. Surveys report that this two-stage design reduces unsupported claims and makes outputs easier to verify (Zhao et al., 2024).

In enterprise settings, RAG typically indexes internal materials (policies, manuals, knowledge articles) as embeddings in a vector store, then retrieves and supplies the most relevant chunks to the model at inference time. Descriptions of production architectures emphasise chunking strategies, embedding/vector stores, and simple orchestration from stored content to grounded answers (Jeong, 2023).

For data access, RAG can retrieve schema facts and domain rules before generation, helping a model produce SQL aligned with table structures and business logic. Recent work stores schema descriptors and rules in a vector store, retrieves pieces matched to a user question, generates SQL, and returns a natural-language summary of the result; some designs add validation loops to catch syntactic or semantic errors (Steve Jeffrey, 2024).

Operationally, RAG quality depends on the retrieval layer: chunk size, indexing/refresh cadence, and retriever choice affect what the model can condition on. Access control and provenance are also required so that retrieved context is appropriate and auditable. These concerns recur in both survey and system papers and are central to reliable deployments (Zhao et al., 2024; Jeong, 2023).

2.4.8 Reasoning-Oriented: Neural Databases

Neural databases shift part of query answering into transformer models that reason over facts expressed in natural language. Instead of issuing SQL against a fixed schema, users pose questions and the model composes evidence from multiple textual statements to produce an answer (Thorne et al., 2021). Early systems show that language models can resolve select–project–join style needs when the relevant facts are available as sentences, while noting open challenges in scaling and query coverage (Thorne et al., 2021).

Recent work extends this idea beyond text. *Multimodal neural databases* combine text and images under a unified representation so that a single natural-language query can retrieve and reason across modalities; reported architectures use a retriever-reasoner-aggregator pattern to handle large corpora (Trappolini et al., 2023).

It is useful to distinguish these approaches from natural-language interfaces that keep the relational engine in place. NL2SQL systems translate a user's request into executable SQL and run it over existing databases. Examples include LLM-prompted generation of SQL (NLSQL) (Attawar et al., 2023) and hybrid rule/NER pipelines tailored to specific languages and domains (Mandal et al., 2022). These systems improve accessibility while preserving schemas, constraints, and transactional guarantees; neural databases, in contrast, answer directly from model-read facts (and retrieved artefacts), reducing schema exposure but introducing different concerns about provenance and refresh.

Across the literature, practical considerations recur: curating and updating the fact store so answers remain current, providing auditable supporting evidence, and managing the computational cost of large-model inference (Thorne et al., 2021; Trappolini et al., 2023). In settings that require repeatability and strong governance, authors often recommend hybrid patterns that use neural reasoning for understanding and explanation, and NL2SQL when results must be executed and checked against the database.

2.5 Research Gaps

Section 2.4 reviewed recent advances in AI-assisted database querying, including learning-guided optimization, retrieval-augmented generation, and NL2SQL systems. While these approaches demonstrate measurable gains in efficiency and usability, the literature also reveals persistent limitations that align with the research gaps defined in Section 1.2 of the Introduction (Chapter 1). Rather than restating that gap, this section synthesises the reviewed literature and organises it into focused gap areas that motivate the experimental design and evaluation conducted in this thesis.

Across database and AI research, AI-driven optimization techniques reduce execution time and resource consumption, yet practical deployment remains constrained by cold-start effects, integration cost, and instability under workload shift (Panwar, 2024; Zhou et al., 2022). Retrieval-augmented and constrained generation approaches reduce hallucination by grounding outputs in retrieved context, but their effectiveness depends on careful chunking, re-ranking, schema awareness, and governance mechanisms (Zhao et al., 2024; Steve Jeffrey, 2024). Neural and multimodal querying techniques further expand the range of supported questions, while exposing unresolved issues in scalability, semantic fidelity, and evaluation practice (Thorne et al., 2021; Zhao et al., 2024). Taken together, these strands reinforce the need for schema-aware and governed pipelines that balance correctness, efficiency, and auditability.

2.5.1 Scalability and efficiency under governance constraints

Prior work demonstrates that AI-assisted query planning and execution can deliver substantial performance gains, including reductions of up to 40% in execution time (Panwar, 2024). These gains can extend to larger datasets and higher throughput as models learn improved execution strategies from historical workloads (Gadde, 2022). However, surveys and system studies consistently report challenges when such techniques are deployed in production environments. These include sensitivity to workload or data drift, dependence on specific hardware or DBMS implementations, and ongoing training and maintenance overhead (Zhou et al., 2022; Li et al., 2021). As a result, a practical gap remains in delivering pipelines that maintain predictable latency and token cost while preserving accuracy and governance guarantees. The literature motivates schema-aware designs that bound the model's visible context, minimise redundant retrieval, and allow safe fallback to deterministic execution paths when confidence is low or service-level objectives are at risk.

2.5.2 Multilingual and multimodal querying limits

Surveys of retrieval-augmented generation highlight growing interest in multilingual and multimodal querying, alongside persistent concerns around robustness, evaluation methodology, and cost control (Zhao et al., 2024). In database-style querying, systems such as NeuralDB demonstrate that transformer-based models can answer simple selection and projection queries over short textual facts, but do not scale reliably to richer schemas, joins, or aggregation operations (Thorne et al., 2021). Enterprise case studies therefore remain predominantly text-first and prioritise correctness, schema alignment, and operational reliability over modality breadth (Jeong, 2023). These observations support the scoping decisions adopted in this thesis and clarify why extensions beyond text must preserve schema awareness, validation, and provenance in governed settings.

2.5.3 Trust, grounding, and policy compliance

A recurring theme across the literature is the need for grounded, auditable, and policy-compliant answers in enterprise environments. Surveys on database-centric AI systems treat governance, security, and lineage as first-class concerns for deployment (Zhou et al., 2022). RAG-focused reviews similarly emphasise uneven retrieval quality, alignment risks, and the lack of robust provenance and refresh strategies (Zhao et al., 2024). Recent NL2SQL prototypes propose concrete mechanisms, including dual syntactic and semantic validation, rule-aware retrieval, and iterative repair loops, to align outputs with domain policy (Steve Jeffrey, 2024). While effective, these safeguards often introduce additional latency, and their behaviour under realistic, production-like workloads remains underexplored. This limitation motivates the schema-aware and policy-aware pipeline evaluated in this thesis, which binds generation to governed schema surfaces and produces an auditable evidence trail.

2.5.4 Enhancing usability and adaptivity

Finally, usability studies of natural language interfaces report recurring issues such as ambiguous phrasing, domain-specific language, and user misunderstanding (Olujimi & Ade-Ibijola, 2023). Recent NL2SQL systems address these issues through multi-step validation and rule-aware retrieval, but the associated refinement loops must remain lightweight to preserve interactive response times (Steve Jeffrey, 2024). Database and AI studies further impose strict latency constraints and highlight cold-start and robustness costs when workloads evolve (Li et al., 2021; Zhou et al., 2022). The literature therefore supports lightweight interaction patterns that combine guided prompts, minimal corrective feedback, and confidence-gated execution while preserving predictable latency and robust enforcement under ambiguity and drift.

2.6 Research Questions

The literature reviewed in this chapter highlights recurring challenges in applying RAG and NL2SQL techniques to governed enterprise data warehouses. Prior work shows that retrieval-based approaches often lack awareness of live schemas and policy constraints, while standalone NL2SQL models depend heavily on accurate schema specification and can struggle to incorporate broader contextual evidence. These limitations align with the research gaps defined in Section 2.5, particularly around governance, grounding, and operational efficiency.

To address these gaps, this study asks three research questions.

1. **RQ1:** Does schema-aware retrieval improve evidence recall and grounding compared with RAG-only or SQL-only approaches?
2. **RQ2:** Does the hybrid pipeline produce a higher rate of executable, constraint-respecting SQL and more accurate final answers than both baselines?
3. **RQ3:** What latency and token-cost differences arise when using the hybrid pipeline compared with a baseline NL2SQL approach?

Collectively, RQ 1 and RQ 2 examine whether schema-aware grounding and constraint enforcement improve reliability in a governed setting, while RQ 3 evaluates the associated efficiency trade-offs. Each research question is evaluated using the metrics specified in Section 2.8, enabling a systematic comparison of retrieval quality, SQL correctness, and system efficiency in a governed procurement context. Together, these questions provide a structured basis for assessing whether the proposed approach addresses the limitations identified in prior work while remaining practical for enterprise deployment.

2.7 Motivation

The motivation for this study is introduced in Section 1.4. Building on that framing, the literature reviewed in this chapter consistently emphasises that enterprise analytics in governed environments requires more than producing a plausible response. Studies across data warehousing, metadata management, and AI-assisted querying highlight recurring requirements for (i) grounding outputs in verifiable sources, (ii) enforcing schema and policy constraints during query formulation and execution, and (iii) maintaining predictable efficiency for interactive use. These requirements are especially salient when users expect answers to be auditable and when incorrect results can lead to operational or compliance risk.

Across prior work, a common failure mode is that retrieval-oriented approaches may provide relevant context without guaranteeing that outputs are consistent with the live warehouse schema or business rules, while purely generation-based NL2SQL approaches may produce syntactically plausible SQL that does not reliably respect joins, constraints, or policy conditions. This motivates approaches that strengthen schema awareness and constraint checking, and that make the evidence used for answering transparent and reviewable.

Accordingly, this thesis focuses on three motivation themes that follow directly from the gaps in Section 2.5. First, answers should be grounded in evidence that can be inspected, including the schema elements and governance artefacts that justify the query and its result. Second, generated queries should be executable and constraint-respecting, so that policy and budget rules are enforced consistently rather than being treated as optional text. Third, the approach should remain efficient in practice, with manageable latency and token cost, so that it can be used iteratively and scaled beyond prototypes. These motivations inform the design choices and the evaluation reported in subsequent chapters.

2.8 Evaluation

Prior work evaluating schema-aware RAG and NL2SQL systems commonly reports results across three evaluation dimensions: retrieval quality, SQL generation accuracy, and system efficiency. This section summarises these dimensions and the standard metrics used to describe them in the literature, while study-specific operational settings (e.g., k, baselines, aggregation, and any decision thresholds) are defined in the methodology chapter.

2.8.1 Retrieval Quality Evaluation

Retrieval quality is typically assessed by how well the system retrieves relevant and diverse documents that support downstream reasoning or SQL generation. Standard retrieval metrics therefore capture both top-ranked relevance and ranking quality.

- **Precision@k (%)** measures the proportion of relevant documents within the top k retrieved results. It focuses on the quality of the highest-ranked outputs, which are most impactful to end users (H. Yu et al., 2025).

$$\text{Precision@k} = \frac{|\text{Rel}_k|}{k} \tag{2.3}$$

Where: Rel_k is the number of relevant items in the top k results.

- **Recall@k (%)** measures the proportion of all relevant documents that are included within the top k results returned by the retrieval system. It assesses the completeness of retrieval by determining how well the system captures the full set of possible relevant documents (H. Yu et al., 2025; Liu et al., 2023). This metric is particularly useful when multiple documents can satisfy a query, making it a standard for evaluating document retrieval systems, including RAG pipelines (Lyu et al., 2024).

$$\text{Recall@k} = \frac{|\text{Rel}_k|}{|\text{Rel}|} \tag{2.4}$$

Where: Rel is the total number of relevant items.

- **MAP@10** calculates the average precision of retrieval results across all queries, considering only the top 10 ranked documents. It provides a single-figure measure of overall retrieval quality by averaging the precision values after each relevant document is retrieved within the top-10 results (H. Yu et al., 2025). MAP remains one of the most widely adopted metrics for retrieval tasks as it balances both precision and recall by incorporating rank-based relevance into its scoring formula (Manning et al., 2008).

$$\text{MAP@10} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{m_q} \sum_{j=1}^{10} P_q(j)\, \text{rel}_q(j) \tag{2.5}$$

Where: $m_q = \min(|\text{Rel}_q|, 10)$; $P_q(j)$ is precision at rank j; $\text{rel}_q(j) \in \{0, 1\}$.

- **Normalised Discounted Cumulative Gain (nDCG@10)** evaluates the usefulness of documents based on their positions in the ranked retrieval results. Highly relevant documents appearing earlier in the list contribute more to the score. This position-weighted metric is widely used in information retrieval and NLP-IR pipeline evaluations to assess ranking quality (Manning et al., 2008; Sauchuk et al., 2022).

$$\text{DCG@10} = \sum_{i=1}^{10} \frac{2^{\,\text{rel}_i} - 1}{\log_2(i+1)}, \tag{2.6}$$

$$\text{nDCG@10} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{\text{DCG}_q}{\text{IDCG}_q} \tag{2.7}$$

Where: rel_i is the graded relevance at rank i; IDCG_q is the best possible DCG for query q.

- **Mean Reciprocal Rank (MRR)** measures how quickly the first relevant document appears in the ranked list of retrieved results. It is defined as the average reciprocal rank of the first relevant document across all queries. MRR is widely adopted for evaluating retrieval systems and RAG pipelines where a single highly relevant item exists per query, providing a simple yet powerful measure of ranking quality (Mengmeng et al., 2024; Galitsky et al., 2025; Y. Chen et al., 2025).

$$\text{MRR} = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{\text{rank}_q} \tag{2.8}$$

Where: rank_q is the position of the first relevant item for query q.

- **Semantic Diversity** measures the redundancy among retrieved documents by computing the pairwise similarity between their embeddings. A lower similarity indicates higher diversity, which helps avoid redundant information in multi-document retrieval tasks (Rezaei & Dieng, 2025; Filice et al., 2025).

$$\text{Diversity} = 1 - \frac{2}{n(n-1)} \sum_{i<j} \cos\big(\mathbf{e}_i, \mathbf{e}_j\big) \tag{2.9}$$

  Where: n is the number of retrieved chunks; $\mathbf{e}_i$ is the embedding of chunk i; $\cos(\cdot, \cdot)$ is cosine similarity between two embeddings.

These measures are standard in information retrieval and have been commonly used in RAG system evaluation.

2.8.2 SQL Generation Accuracy

SQL generation quality is typically assessed using both strict structural agreement and functional correctness, as well as schema-consistency indicators that capture grounding to valid schema elements.

- **Exact-Match Accuracy (%)** measures the percentage of generated SQL queries that exactly match the gold reference SQL for a given natural language question. It provides a strict measure of syntactic and structural correctness, ensuring that the generated query aligns perfectly with the expected output (Sauchuk et al., 2022; Thorne et al., 2021).

$$\text{EM} = \frac{\#\text{Exact Matches}}{\#\text{Queries}} \times 100\% \tag{2.10}$$

  Where: #Exact Matches is the number of predicted SQL queries that match the gold query exactly.

- **Execution Accuracy (%)** measures the percentage of generated SQL queries that, when executed, produce the same results as the gold reference query. This assesses the functional correctness of the query beyond syntactic similarity (Panwar, 2024; Gadde & Raza, 2024).

$$\text{ExecAcc} = \frac{\#\text{Correct Executions}}{\#\text{Queries}} \times 100\% \tag{2.11}$$

Where: #Correct Executions is the number of generated SQL queries that return the same output as the gold query on execution (ignoring row order).

- **Join-Path Accuracy (%)** measures the percentage of generated SQL queries that use valid join paths and table relationships corresponding to the database schema. It evaluates whether correct foreign key joins and table combinations are applied to generate the intended results (Ma et al., 2024).

$$\text{JoinAcc} = \frac{\#\text{Valid Joins}}{\#\text{Join Queries}} \times 100\% \tag{2.12}$$

Where: #Valid Joins is the number of join queries using correct tables and keys based on the schema; non-join queries are excluded.

- **Hallucination Rate (%)** measures the percentage of generated SQL queries that reference schema elements not present in the retrieved database schema. This reflects the system's ability to avoid introducing spurious or non-existent tables, columns, or attributes (Xie et al., 2025; Qu et al., 2024).

$$\text{HallucRate} = \frac{\#\text{Invalid Schema Refs}}{\#\text{Queries}} \times 100\% \tag{2.13}$$

Where: #Invalid Schema Refs is the number of queries that include table or column names not found in the retrieved schema context.

- **Average Refinement Loops** tracks the number of regeneration attempts performed by the system to fix invalid or failed SQL queries before producing an accepted query. This builds on previous work which proposed interactive correction loops alternating between repair and synthesis (Yaghmazadeh et al., 2017; Tian et al., 2024).

$$\text{AvgLoops} = \begin{cases} \frac{\#\text{Refinement Iterations}}{\#\text{Queries with Refinement}}, & \text{if } \#\text{Queries with Refinement} > 0; \\ 0, & \text{otherwise.} \end{cases} \tag{2.14}$$

Where: #Refinement Iterations is the total number of retry attempts; the metric only applies to queries that required refinement.

- **Groundedness Score (%)** measures the proportion of generated SQL queries whose elements (e.g., tables, columns) are fully grounded in the retrieved database schema context. It assesses the system's ability to align outputs with the source schema and avoid introducing unsupported elements (Stolfo, 2024; Deng et al., 2021; Fan et al., 2024).

$$\text{Grounded} = \frac{\#\text{Fully Grounded Queries}}{\#\text{Queries}} \times 100\% \tag{2.15}$$

Where: #Fully Grounded Queries refers to queries where every table and column used appears in the retrieved schema/document chunks.

These metrics are commonly used to describe SQL correctness and schema grounding in text-to-SQL and hybrid systems; any study-specific pass/fail criteria are specified in the methodology chapter.

2.8.3 System Efficiency

System efficiency is commonly reported using latency and resource-cost indicators, since multi-stage retrieval and validation can improve correctness but may increase response time and compute cost.

- **Median Latency (ms)** measures the typical end-to-end query response time. Paul et al. (2021) used plan encoders to predict query latency in database workloads, demonstrating its importance in workload profiling. Gadde (2023) showed an AI-augmented DBMS reduced query response times by up to 40%, confirming latency as a key measure of system efficiency.

$$\tilde{\mathrm{t}} = \text{median}\Big\{\, t_1, \ldots, t_N \,\Big\} \tag{2.16}$$

  Where: t_i is the end-to-end latency for query i; N is the number of runs.

- **p95 Latency (ms)** measures the maximum response time below which 95% of system responses occur. This metric is widely used to assess system consistency and tail latency performance under load. Deeksha et al. (2025) report p95 response times for evaluating LLM applications. Bappy et al. (2025) apply p95 latency to assess small language model inference times.

$$p_{95} = P_{0.95}(t_1, \ldots, t_N) \tag{2.17}$$

Where: P_q is the empirical q-percentile.

- **p99 Latency (ms)** measures the maximum response time below which 99% of system responses occur. It is a standard metric to assess tail latency and detect extreme outliers. Bappy et al. (2025) directly report p99 inference latency for small language models. Bambhaniya et al. (2025) define p99 latency thresholds for evaluating multi-stage AI pipelines, and Shen et al. (2024) use p99 latency to analyze tail performance variability in retrieval-augmented generation inference.

$$p_{99} = P_{0.99}(t_1, \ldots, t_N) \tag{2.18}$$

- **GPU-seconds per Query** measures the average GPU compute time per query, providing a system-level indicator of computational cost and efficiency. Gadde and Raza (2024) and Gadde (2022) highlight resource utilization and query-level efficiency as critical evaluation metrics in AI-augmented database systems. The iServe framework similarly applies GPU utilization and serving time tracking for optimizing large language model inference (Liakopoulos et al., 2025).

$$\text{GPUsec} = \frac{\sum_{i=1}^{N} g_i}{N} \tag{2.19}$$

Where: g_i is GPU compute time in seconds for query i.

- **Peak GPU Memory (MB)** measures the maximum GPU memory usage during query execution. High memory demand can limit scalability and affect system throughput. Recent RAG system research has focused on memory-efficient designs to mitigate these issues. W. Yu et al. (2025) proposed offloading strategies in RAGDoll to reduce VRAM pressure during LLM inference. Quinn et al. (2025) highlighted memory bandwidth limitations as a bottleneck in GPU-bound RAG workloads, and Kim and Mahajan (2025) introduced adaptive vector index partitioning to minimise

GPU memory overhead during retrieval tasks.

$$\text{PeakMem} = \max_t \Big(\text{Mem}(t)\Big) \tag{2.20}$$

Where: $\text{Mem}(t)$ is GPU memory usage at time t.

- **Cost per 1 000 Queries (USD)** estimates the average monetary cost to process 1,000 queries using the retrieval and generation system. It reflects the practical cost impact of system resource consumption. L. Chen et al. (2023) introduced FrugalGPT to study and reduce LLM API query costs, reporting that query-level pricing varies significantly between providers and that optimising query allocation can reduce total inference costs by up to 98%.

$$\text{Cost}_{1000} = \frac{C_{\text{GPU}} + C_{\text{storage}} + C_{\text{API}}}{N} \times 1000 \tag{2.21}$$

  Where: C_{GPU} = GPU-time cost; C_{storage} = storage cost; C_{API} = any external API fees.

Operational targets and any decision thresholds used in this study are specified in Chapter 3.

Note: In this thesis, we report median latency (with five-number summary & IQR) and total tokens (in+out). Additional efficiency metrics defined above (p95/p99, GPU-seconds, peak memory, and cost per 1k queries) are discussed as relevant measures in prior work but were not instrumented in our experiments.

In summary, the literature supports evaluating governed schema-aware querying along retrieval quality, SQL correctness and grounding, and efficiency. The remainder of this thesis adopts these dimensions, while Chapter 3 defines the operational procedure used for comparison across baselines and the proposed pipeline.

Chapter 3

Methodology

This chapter presents the end-to-end methodology used to design and implement a schema-aware hybrid pipeline for governed IT procurement querying. It explains how each user interaction is scoped to a session-bounded warehouse slice, how relevant schema and policy context is retrieved, and how execution is constrained to a single validated, read-only `SELECT` when required. It then details the supporting components, including vector retrieval over documentation and metadata, lightweight graph-based relationship hints, prompt construction, and static validation checks that enforce schema and policy compliance. Together, these elements define a controlled and reproducible workflow for grounded NL2SQL in the IT procurement setting.

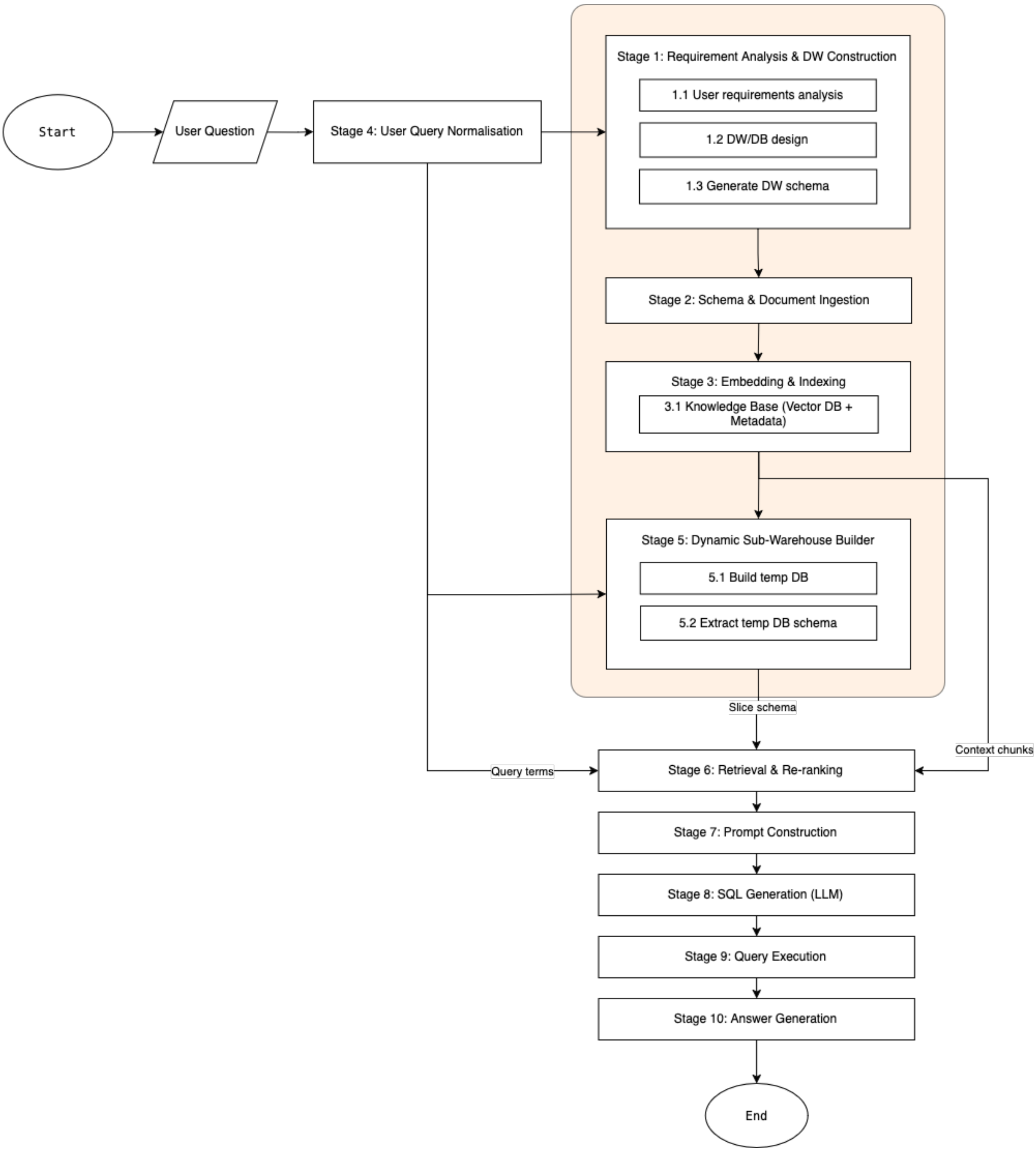
3.1 Proposed Approach

This research adopts a modular, schema-aware Retrieval-Augmented Generation (RAG) pipeline built on a PostgreSQL data warehouse and a Qdrant vector store. The system ingests both structured tables and unstructured enterprise documents, transforming them into a linked knowledge base that supports grounded question answering and SQL generation.

The core idea is to let the agent retrieve contextual evidence, such as schema fragments, descriptions, and sample records, before producing a PostgreSQL-compatible SQL query or a knowledge-based response. Each user session operates within a scoped schema view, ensuring that only relevant tables and columns are exposed to the model. This reduces token usage and minimises the risk of hallucinated field names.

The pipeline supports multilingual input, dynamic schema updates, and iterative prompt refinement. Figure 3.1 illustrates the complete workflow, showing the sequence from data ingestion and embedding to retrieval, SQL generation, and final recommendation output.

Figure 3.1. *Overview of the schema aware RAG based multi agent pipeline used in this study*



Notes. The pipeline spans ingestion and embedding, retrieval over the session warehouse slice and knowledge graph context, governed SQL generation and validation, and final recommendation output for the in-budget device procurement use case.

Stage 1: Requirement Analysis and Data-Warehouse Construction

The first stage of the RAG-based pipeline focuses on generating a structured data-warehouse schema from unstructured or loosely structured enterprise sources. It involves analysing user natural-language queries to identify required data elements and comparing these needs with the current warehouse structure. When existing schema elements are insufficient, new tables or views are proposed to capture the missing information while maintaining alignment with the overall data model.

Inputs

- Normalised user questions and task definitions Q.
- Enterprise documentation: entity–relationship diagrams (ERDs), data dictionaries, and technical manuals.

Process

We extract required entities/attributes $\mathcal{R} = \{r_1, \ldots, r_n\}$ from normalised user intents and enterprise documents. Each requirement r_i is matched to a warehouse attribute $a_j \in S$ using a combined similarity:

$$\mathrm{sim}(r_i, a_j) = \lambda_1 \, \cos\Big(\mathbf{e}_{r_i}, \mathbf{e}_{a_j}\Big) + \lambda_2 \, \mathrm{Jac}\Big(\mathrm{Ctx}_{r_i}, \mathrm{Meta}_{a_j}\Big). \tag{3.1}$$

Here, $\cos(\cdot)$ is cosine similarity between embedding vectors and $\mathrm{Jac}(\cdot)$ is the Jaccard coefficient over contextual keywords/metadata. Accepted matches satisfy:

$$\mathrm{sim}(r_i, a_j) \;\geq\; \theta. \tag{3.2}$$

We then compute requirement coverage of the current schema:

$$\mathrm{Cov}(S, \mathcal{R}) = \frac{\Big|\{\, r_i \in \mathcal{R} : \exists a_j \in S,\ \mathrm{sim}(r_i, a_j) \geq \theta \,\}\Big|}{|\mathcal{R}|}. \tag{3.3}$$

Gaps ($\mathrm{Cov} < \tau$) are recorded for resolution via minimal views or attribute additions, preferring derived/view-level solutions over base-schema changes.

We register constraints as symbolic predicates without binding experiment-specific constants:

$$\Gamma = \{\gamma_1, \ldots, \gamma_u\}. \tag{3.4}$$

Each predicate has the form

$$\gamma_j : \varphi_j(A; \vartheta_j), \tag{3.5}$$

where φ_j is a quantifier-free formula over comparisons on attributes A and unknown parameters ϑ_j (e.g., $a \odot \vartheta$ with $\odot \in \{<, \leq, =, \geq, \in, \neq\}$).

Output

A validated logical fragment S' and a gap list $\mathcal{M}$ with proposed minimal views/attributes, plus a symbolic constraint set Γ for later filtering and validation. These artefacts are passed to Stage 2 for ingestion and indexing.

Stage 2: Schema and Document Ingestion

This stage prepares both structured database metadata and unstructured enterprise documents for later retrieval and query generation. Its goal is to provide the RAG system with access not only to formal schema definitions, but also to contextual domain knowledge captured within documentation text.

Input

Catalog metadata S (tables, attributes, constraints, relationships) derived from Stage 1; enterprise documents D (manuals, ERDs, policies, reports).

Process

We first serialise catalog metadata into a structured form that preserves links between tables, attributes, and constraints, along with lightweight descriptions (names, aliases, data types, glossary text). Enterprise documents are normalised and segmented into sections, then split using fixed-size sliding windows (window w, overlap o, stride $s = w - o$) to create fragments, in line with modern passage-retrieval practice (Karpukhin et al., 2020). Each fragment f_k carries section labels and offsets; named entities and key phrases are recorded as auxiliary annotations. *(Optionally)* the number of fragments for a document with T_d tokens is

$$N_d \;=\; \max\Bigl(1,\; \Bigl\lceil \frac{T_d - w}{s} \Bigr\rceil + 1\Bigr), \qquad s > 0, \tag{3.6}$$

anchoring the last window to cover the tail.

To link text fragments to schema, for each f_k and attribute $a_j \in S$ we compute a combined lexical/semantic score

$$\ell(f_k, a_j) \;=\; \mu_1 \,\mathrm{Jac}\Bigl(\mathrm{terms}(f_k), \mathrm{Meta}(a_j)\Bigr) \;+\; \mu_2 \,\cos\Bigl(\mathbf{v}_{f_k}, \mathbf{v}_{a_j}\Bigr), \tag{3.7}$$

and accept links when

$$\ell(f_k, a_j) \;\geq\; \varphi. \tag{3.8}$$

Here $\cos(\cdot)$ is cosine similarity over tf–idf (or equivalent) vectors and $\mathrm{Jac}(\cdot)$ is the Jaccard coefficient over keyword/meta sets (Manning et al., 2008). Weighted combination with a threshold is consistent with recent schema-matching practice that mixes lexical and semantic features (Pan et al., 2022; Liu et al., 2025).

We suppress near-duplicates via shingling: if

$$\mathrm{Jac}\Bigl(\mathrm{shingles}(f_i), \mathrm{shingles}(f_j)\Bigr) \;\geq\; \delta_{\mathrm{dup}}, \tag{3.9}$$

the lower-quality fragment is dropped, following scalable near-duplicate detection practice (Rodier & Carter, 2020).

Finally, each output record ("chunk") is packaged with provenance:

$$c_k \;=\; \Bigl\langle \mathrm{text} = f_k,\; \mathrm{schema\_refs} = \mathcal{A}_k,\; \mathrm{provenance} = \Pi_k \Bigr\rangle, \tag{3.10}$$

where $\mathcal{A}_k = \{\, a_j : \ell(f_k, a_j) \geq \varphi \,\}$ and Π_k stores document id, section, offsets, content hash, and source type.

Output

A collection $\mathcal{C} = \{c_k\}_{k=1}^{M}$ of schema–text chunks with explicit schema links and full provenance, ready for embedding and indexing in Stage 3.

Stage 3: Embedding and Indexing

This stage transforms the mixed schema–document chunks from Stage 2 into dense vectors and builds an Approximate Nearest Neighbour (ANN) index with bound metadata for schema-scoped retrieval.

Input

Cleaned, structured chunks from Stage 2 (text, schema refs, provenance).

Process

We map each fragment f_k to a d-dimensional vector using a sentence-level encoder f_θ (e.g., SBERT/DPR-style):

$$\mathbf{e}_k = f_\theta(f_k). \tag{3.11}$$

Vectors are ℓ_2-normalised so cosine similarity equals a dot product; this matches the similarity used later in retrieval (Reimers & Gurevych, 2019; Karpukhin et al., 2020; Manning et al., 2008). Each vector is stored together with its payload (source, table/attribute references, section/offsets) as $(\hat{\mathbf{e}}_k, \Pi_k)$ for traceability into Stage 6.

We then build an HNSW-style ANN index over $\{\hat{\mathbf{e}}_k\}$ and persist payloads alongside vectors. Construction/search knobs (graph degree M, beams ef_{build}, ef_{search}) control the recall–latency trade-off and are fixed in the experiment setup (§4) (Malkov & Yashunin, 2020; Johnson et al., 2017). At query time, a predicate ψ over payloads restricts results to the session scope S'' from Stage 5; we retrieve ANN top-k by cosine within items satisfying ψ, with higher ef_{search} improving recall at higher cost (Malkov & Yashunin, 2020). (No BM25 or cross-encoder re-ranking is used in our experiments; these are optional in general.)

Index quality (definition). For diagnostics we may report

$$\text{recall@}k = \frac{\left|\widetilde{N}_k(\hat{\mathbf{q}}) \cap N_k^\star(\hat{\mathbf{q}})\right|}{\left|N_k^\star(\hat{\mathbf{q}})\right|}, \tag{3.12}$$

where $N_k^\star$ are exact neighbours. Our main RQ1 metric in §4 uses top-k *coverage* of required tables/attributes, not exact-neighbour recall.

Output

An HNSW ANN index of normalised vectors with bound metadata, ready for schema-scoped retrieval in Stage 6.

Stage 4: User Query Normalisation

This stage processes the user's natural language question and converts it into a structured form that can be matched with the indexed knowledge base or used directly for SQL generation. Its aim is to normalise the query text, identify key entities and filters, and, when relevant, produce a semantic vector embedding.

Input

Raw query text q from the user interface or user cases.

Process

We normalise the raw query using standard IR preprocessing techniques, including lowercasing, tokenisation, light stop-word removal, and optional lemmatisation, to reduce surface variation (Manning et al., 2008). Named-entity recognition and lightweight rules then identify dates, numeric values/ranges, categorical values, and target entities; detected phrases are mapped to canonical *predicate templates* (attribute–operator–value).

To hint schema elements, for each extracted phrase ϕ and attribute $a \in S$ we compute a combined lexical/semantic score

$$\mathrm{sim}_q(\phi, a) = \lambda'_1 \, \mathrm{Jac}\Big(\mathrm{terms}(\phi), \mathrm{Meta}(a)\Big) + \lambda'_2 \, \cos\Big(\mathbf{v}_\phi, \mathbf{v}_a\Big), \tag{3.13}$$

and accept candidates when $\mathrm{sim}_q(\phi, a) \geq \theta_q$, where $\mathbf{v}_{(\cdot)}$ are tf–idf (or equivalent) vectors and $\mathrm{Meta}(a)$ are attribute descriptions/aliases (Manning et al., 2008). Using a weighted score with a threshold follows established schema-matching practice (Peukert et al., 2010).

When retrieval is enabled, we also produce a dense query representation with the same sentence-level encoder f_θ used in Stage 3:

$$\mathbf{e}_q = f_\theta\Big(\mathrm{norm}(q)\Big) \in \mathbb{R}^d, \tag{3.14}$$

then ℓ_2-normalise to obtain $\hat{\mathbf{q}}$ for cosine scoring against the index (Reimers & Gurevych, 2019; Karpukhin et al., 2020). *Parameter values* $(\lambda'_1, \lambda'_2, \theta_q)$ are set in Chapter 4 (Experiment Setup).

Output

A structured query package containing: (i) tokens and extracted predicate templates; (ii) a candidate attribute set $\mathcal{A}_q = \{a : \exists\,\phi,\ \mathrm{sim}_q(\phi, a) \geq \theta_q\}$ for schema-aware retrieval/generation; and (iii) an optional semantic query vector $\hat{\mathbf{q}}$ for retrieval stages.

Stage 5: Dynamic Sub-Warehouse Builder

This stage creates a temporary, session-scoped version of the data warehouse that contains only the schema elements relevant to the user's question. The aim is to minimise prompt size, reduce SQL generation errors, and ensure that subsequent reasoning occurs within a clearly defined schema boundary.

Input

Extracted query metadata from Stage 4 (predicate templates, candidate attributes); catalog metadata from Stage 2; optional retrieval signals from Stage 3 (schema-linked chunks and similarities).

Process

We score candidate attributes using both query–attribute similarity and retrieval support from Stage 3. For each extracted phrase ϕ (Stage 4) and schema attribute a, define

$$s(a) = \alpha\,\mathrm{sim}_q(\phi, a) \;+\; \beta \max_{k:\,a \in \mathcal{A}_k} \cos\Big(\hat{\mathbf{q}}, \hat{\mathbf{e}}_k\Big), \tag{3.15}$$

and accept when $s(a) \geq \theta_s$, where sim_q is the Stage 4 score, $\hat{\mathbf{q}}$ the (normalised) query vector, and $\hat{\mathbf{e}}_k$ chunk embeddings from Stage 3. Using a weighted score with a threshold follows established matcher practice (Peukert et al., 2010); cosine scoring is standard in IR/dense retrieval (Manning et al., 2008; Karpukhin et al., 2020). Let $A^\star = \{a : s(a) \geq \theta_s\}$.

We then select a minimal table cover: map each $a \in A^\star$ to its base table $t(a)$ and choose the smallest table set T' that covers all accepted attributes, preferring fewer tables/columns while ensuring any columns referenced by Stage 4 predicate templates (and required key columns) are included. Next, enforce join connectivity by restricting edges to primary/foreign-key relations so the induced subgraph over T' is connected, preserving referential integrity (joins follow FK→PK relationships). Columns are pruned to $A' = A^{\star} \cup$ {PK/FK used in the chosen joins} plus any columns required by symbolic constraints Γ recorded earlier, yielding a concise projection sufficient for filtering, joining, and presentation.

Finally, we serialise the session-scoped fragment as

$$S'' = (T', A', E', \Gamma),$$

listing selected tables, kept columns, PK/FK join edges, and constraint forms for downstream prompt construction and SQL generation.

Output

A minimal, connected, session-scoped logical schema S'' with (i) selected tables T', (ii) necessary columns A', (iii) join edges E', and (iv) attached constraint forms Γ. This fragment bounds retrieval and generation to the exact schema elements required for the current question.

Stage 6: Retrieval and Re-ranking

This stage identifies and ranks the most relevant information from the knowledge base so that the model can construct accurate, grounded SQL queries. It performs semantic retrieval to gather a concise, context-rich set of evidence from both schema and documentation sources.

Input

- Vector-store chunks from Stage 3 (schema + text embeddings).
- Scoped schema definition from Stage 5.
- Query vector and extracted tokens from Stage 4.

Process

We query the ANN index with the normalised query vector $\hat{\mathbf{q}}$, applying the Stage 5 payload predicate to restrict results to the session scope; the index returns top-k_0 candidates by cosine similarity (equivalently inner product after ℓ_2 normalisation) (Manning et al., 2008; Karpukhin et al., 2020; Malkov & Yashunin, 2020). To balance evidence types, we retrieve from both schema-linked and documentation pools, normalise scores within each pool, and merge in descending order with a light prior favouring schema-linked items so the context remains compact yet covers attributes and relations needed for SQL. We then suppress redundancy by dropping near-duplicates using a cosine-similarity threshold on embeddings and prefer fragments that improve table/attribute coverage under the Stage 5 scope, keeping the window diverse rather than filled with paraphrases (Rodier & Carter, 2020). Finally, we preserve provenance for every retained fragment (source type, table, column, section/offsets) and order the final top-k by the fused score to produce a small, self-contained context window for Stage 7.

Output

A context window with the top-k most relevant, non-redundant schema and text fragments (with provenance), ready for Stage 7 prompt construction.

Stage 7: Prompt Construction

This stage converts the retrieved context into a structured prompt that guides the language model to generate a valid, executable PostgreSQL query. The prompt integrates both technical schema information and domain-specific business rules so that the generated SQL remains accurate, complete, and grounded within the scoped schema.

Inputs.

Top-k context from Stage 6 (with provenance); session schema $S'' = (T', A', E', \Gamma)$ from Stage 5.

Process

We select the smallest fragment set from Stage 6 that covers the required attributes in A' and explains the joins E' and constraint forms Γ, removing paraphrases and duplicates to conserve tokens. From this, we compose a compact *schema block* that lists tables T', enumerates A' per table with PK/FK marks, and names the join edges in E' with their key columns; type hints are included only when needed to avoid ambiguity. We add a concise *constraint block* that lists only the forms in Γ (e.g., uniqueness, bounds, membership) without any experiment-specific constants. We then append clear *instructions/guardrails*: return exactly one read-only ANSI-SQL `SELECT`; follow the joins in E'; apply the listed constraint forms; respect declared data types; avoid DDL/DML; include a `LIMIT`; and return *only* the SQL text. Finally, we note that the generated SQL will be validated against $S'' = (T', A', E', \Gamma)$ for names, keys, and basic types; on validation failure, a reduced schema/context is issued and the step is retried.

Output

A single prompt string that (i) summarises S'' and the relevant constraint forms, and (ii) instructs the model, under explicit guardrails, to generate one ANSI-SQL `SELECT` statement suitable for Stage 8 validation.

Stage 8: SQL Generation

This stage uses the final prompt from Stage 7 to produce and validate a single, executable PostgreSQL `SELECT`. The goal is to express the user's intent while remaining syntactically correct and restricted to the scoped schema.

Input

Final prompt string from Stage 7 (schema context, business rules, formatting instructions).

Process

We obtain exactly one read-only ANSI-SQL `SELECT` statement as plain text from the model, with no DDL or DML operations and no multi-statement batches. The statement is then parsed and validated against the session scope $S'' = (T', A', E', \Gamma)$: every referenced table must be in T' and every column in A' (name check); joins must follow edges in E' and use PK/FK keys (join check); comparisons and functions must respect declared types (type check); required predicate forms from Γ must be present (constraint check); statements that modify state are rejected (read-only check); and an explicit `LIMIT` is enforced (output bound). If any check fails (e.g., unknown column, invalid join), we narrow the schema/context and request a revised query, repeating up to a small fixed number of retries. When execution is enabled, the validated query runs against PostgreSQL and returns rows for downstream evaluation (e.g., latency, correctness); analysis-only runs skip execution.

Output

A single validated ANSI-SQL `SELECT` consistent with S'' (and, when enabled, its result set for evaluation).

In summary, this chapter defined the pipeline architecture and the workflow stages that connect governed warehouse structures, retrieval signals, and constrained NL2SQL execution into one controlled process. It also established the enforcement and validation surfaces that make outputs traceable to structured evidence. Chapter 4 applies this methodology to define the experimental setup, including the dataset slice, assumptions, controlled user cases, and the three pipeline configurations used for comparison.

Chapter 4

Experiments

This chapter describes the experimental design used to compare three approaches to answering employee questions about IT devices under departmental budgets and approval rules. All experiments use the same sponsor-provided device catalogue and budget data, but differ in how context is retrieved and how constraints are enforced at answer time. The chapter specifies the execution environment, dataset slice, shared assumptions, controlled user cases, and evaluation procedure used to ensure a fair and repeatable comparison across the three configurations.

The experiments use data from the sponsor's operational style database. For each session, Phase 1 collects department, role, quarter, and request type, then derives a per request budget `request_budget_nzd`. This information is written into a per session data warehouse with a ***capacity*** table and an ***inventory_candidates*** table. The global device catalogue is also exposed through a vector index, and for one method, a session scoped knowledge graph and RAG text surface. Together, these components let the system reason about budgets and approval rules, filter candidate devices, and explain its choices.

We compare three agent designs. All three draw from the same device pool, use the same hard budget constraint `price_nzd` ≤ `request_budget_nzd`, and respect the same departmental auto approval rules encoded in the warehouse. Only the way candidates are retrieved and combined differs.

- **Experiment 1 - RAG only.** All relevant context is passed to the model. It uses semantic search over the device index, filters candidates against the per request budget and approval hints, ranks them, and returns a shortlist. No SQL is executed at answer time.

- **Experiment 2 - SQL only.** The model issues a single safe `SELECT` statement against the per session warehouse. The query is validated, restricted to read only, and must include a `LIMIT`. No vector search is used.

- **Experiment 3 - RAG + KG + DW.** The per session warehouse is extended with a small knowledge graph and RAG text built from the device candidates. Retrieval uses semantic search over this session's chunks with metadata filters derived from the budget and policy mode.

Across all methods, the ten user cases A to J, their request budgets, and the expected device pools are kept fixed. Each answer is returned as a ranked list of devices that includes the original `device_id` and key specifications so that results can be checked against the warehouse.

We evaluate both correctness and efficiency. Correctness includes:

- **Top 1 budget compliance**: the first device must satisfy `price_nzd` ≤ `request_budget_nzd`.

- **Match to the expected budget-eligible set**: order agnostic equality with a deterministic SQL baseline for each user case, using only devices that meet the budget constraint.

- **Policy compliance and follow up accuracy**: correct treatment of the auto approval rule when the user asks for options that are auto approved only, and correct reporting of which devices would require manual approval.

Efficiency is assessed using end to end latency (median, five number summary, and interquartile range using Tukey's hinges) and total token usage (input plus output) per run. Additional efficiency metrics defined in Chapter 2 (p95/p99, GPU-seconds, peak memory, cost per 1k tokens) were scoped but not instrumented in this study.

4.1 Hardware

All experiments ran locally on a single workstation. Software stack details (Docker Desktop versions, active containers, and image tags) are reported in Section 4.2.

Host Machine

Table 4.1. *Host machine specifications used to run all experiments.*

Component	**Specification**
Model	MacBook Pro 16-inch (2019)
CPU	2.3 GHz 8-core Intel Core i9
GPU (discrete)	AMD Radeon Pro 5500M, 8 GB
GPU (integrated)	Intel UHD Graphics 630, 1.5 GB
Memory	32 GB 2667 MHz DDR4
Storage	1 TB SSD (≈242 GB free during tests)
Operating system	macOS Sequoia 15.4.1
Role	Local runtime host for all experiments

4.2 Software

All experiments were executed in Docker Desktop on macOS, using three core containers: `n8n`, `postgres`, and `qdrant`. Each experiment followed the same orchestration workflow; only the retrieval or query logic differed. Table 4.2 summarises the runtime environment, followed by container and development tool details.

Runtime Environment

Table 4.2. *Docker Desktop and runtime configuration for the experimental environment.*

Component	Version / Setting
Docker Desktop (application)	4.46.0 (build 204649)
Engine	28.4.0
Compose	v2.39.2-desktop.1
Credential Helper	v0.9.3
Kubernetes (local)	v1.32.2
Docker memory pool	∼7.65 GiB available to containers
Docker CPU limit	None (full host access)
Network mode	Bridge (default)
GPU support	None (LLM via external API)

Notes. Each container used the default Docker Desktop bridge network. No GPU or device passthrough was configured. All model inference occurred externally through API calls to the provider.

Active Containers During Experiments

Table 4.3. *Containers used in all experiment runs and their roles in the pipeline.*

Name	Image & Tag	Purpose
`n8n`	`n8nio/n8n:1.107.4`	Workflow orchestration, logging, and latency measurement
`postgres-db`	`postgres:16`	Per-session data warehouse (`user_dw_<sessionId>`)
`qdrant-1`	`qdrant/qdrant:1.15.3`	Vector database for retrieval and session knowledge graph

Notes. Container CPU and memory were dynamically allocated within the 7.65 GiB Docker Desktop limit. CPU utilisation during active inference stayed below 30%. All latency measurements were recorded inside the `n8n` container for consistency across runs.

Development and Tooling

Table 4.5. *Development tools and auxiliary software used for implementation, orchestration, and analysis.*

Tool	Version / Role
JetBrains WebStorm	2024.3 - Code authoring and workflow editing
DBeaver Community	25.2.0 - Database management and query inspection
Node.js	20 LTS - Utility scripts and n8n plugin support
PostgreSQL client	psql 16.2 - Schema inspection and direct SQL testing
Curl / HTTPie	Latest - API validation and endpoint testing
macOS Terminal	Built-in - Docker control and log monitoring

This configuration provided a controlled, reproducible software environment across all experiments. The combination of n8n, PostgreSQL, and Qdrant ensured consistent orchestration, structured storage, and semantic retrieval, while WebStorm and DBeaver supported workflow debugging and data inspection.

4.3 Datasets

This study uses an IT device procurement data warehouse and a request log supplied by the industry sponsor. The warehouse is a de-identified copy of a test environment that mirrors the sponsor's real operational database structure. No personal information is processed in this research.

The data used in this study were provided by the sponsor, and the database structure was designed to reflect the sponsor's real operational database. In addition, the query questions were refined in consultation with the sponsor to ensure alignment with potential future implementation.

The experiments work with four main data sources inside this warehouse: a device configuration table, employee and department dimensions, a quarterly budget fact table, and a purchase request fact table. These tables are supplemented by per session schemas and views created by the pipeline.

4.3.1 IT device configuration (dim_device_config)

Type: Relational dimension table (`dim_device_config`, PostgreSQL).
Access: Loaded from a static sponsor provided extract into a controlled schema.
Use: Primary device inventory for all experiments. A fixed snapshot is loaded before each run.

Fields used (mirror schema). The device table holds 28 devices that cover three broad tiers: basic office machines, mainstream business laptops, and higher performance developer or data roles. Key fields include:

- **Identifiers:** `device_id`.
- **Vendor and model:** `pc_brand`, `model_name`.
- **Hardware specification:** `cpu_model`, `gpu_model`, `ram_gb`, `ssd_size_gb`.
- **Commercials:** `device_price_nzd` (list price for the device in NZD).

These fields are later mirrored into a per session `inventory_candidates` table that adds derived features such as a performance tier label and an auto approve ok flag used in the rule engine.

Snapshot and preprocessing. The sponsor extract is treated as read only. Text fields are normalised, numeric fields are cast to appropriate types, and each row is assigned a coarse performance tier that distinguishes integrated GPU office devices from higher performance models. A short narrative description is also constructed from brand, model, CPU, GPU, and RAM, and is used when building the text corpus for vector retrieval in Experiments 1 and 3.

4.3.2 Employee and department dimensions

Type: Relational dimensions (`dim_department`, `dim_employee`, PostgreSQL).
Access: Loaded from sponsor supplied extracts into the same warehouse.
Use: Provide department and role context that drives budget allocation and device rules.

Departments. The `dim_department` table contains three departments that are typical for the sponsor's environment:

- DEP_IT (IT)
- DEP_SALES (Sales)
- DEP_FRONT (Front Desk)

Each department has a stable identifier, a human readable name, and a short description of its typical device needs. These attributes are used when defining department level budget rules and auto approval limits.

Employees. The `dim_employee` table stores employee attributes used by the decision system. Each row links an employee to a department and role and records a seniority level (for example, junior, mid, senior). These fields support role-based budget allocation and device eligibility rules. These attributes provide contextual features that allow the model and rule engine to distinguish IT, Sales, and Front Desk use cases and, if needed, to prioritise more critical technical roles when allocating limited device budgets.

4.3.3 Quarterly budget fact table (fact_budget_quarterly)

Type: Relational fact table (`fact_budget_quarterly`, PostgreSQL).
Access: Loaded from a sponsor supplied extract.
Use: Encodes quarterly IT procurement budgets and overspend behaviour per department. This table drives the numeric budget limit that is applied to each new request.

Business assumption. The warehouse records a total IT procurement budget of 9,000 NZD per quarter. This is allocated across three departments:

- IT: 4,000 NZD per quarter
- Sales: 3,000 NZD per quarter
- Front Desk: 2,000 NZD per quarter

The sample data covers one calendar year with one record per department and quarter. Some rows intentionally show overspends, which allows the evaluation to test rules such as departments with repeated budget overruns must not auto approve high cost devices.

Fields used. Key fields in `fact_budget_quarterly` are:

- **Identifiers:** `budget_id`, `department_id`, `year`, `quarter`.
- **Budget amounts:** `quarter_budget_nzd`, `planned_spend_nzd`, `actual_spend_nzd`.
- **Overrun behaviour:** `budget_overrun_flag`, used to mark when actual spend exceeds the allocated budget.

For each new device request, the pipeline joins the intake answers to the relevant budget row and derives a per request budget limit (`request_budget_nzd`) and an impact ratio that reflect department history and current quarter capacity.

4.3.4 Purchase request fact table (fact_purchase_request)

Type: Relational fact table (`fact_purchase_request`, PostgreSQL).
Access: Populated by the workflow during testing; read only for analysis.
Use: Records device requests, the candidate set returned by each agent, and the final decision outcome.

The `fact_purchase_request` table captures each user case used in the experiments. Fields include:

- **Context:** `request_id`, `employee_id`, `department_id`, `year`, `quarter`, and whether the request is a replacement or a net new device.
- **Budget snapshots:** the `quarter_budget_nzd` and projected spend used when the request was evaluated.
- **Decision outputs:** the chosen device identifier, a numeric `decision_score`, and an `auto_approve_flag` or equivalent label indicating whether the request can be auto approved or should be routed to manual review.

During the controlled experiments the ten user cases A to J are submitted through each pipeline. The resulting request records and decision flags are used later in Chapters 4 and 5 to compute correctness and efficiency metrics.

4.3.5 Usage in experiments

All three experiments consume the same device, employee, department, and budget snapshot. Only the retrieval and query path differs.

- **Experiment 1 (RAG only):** builds a text corpus from the device catalogue, budget table, and simple policy sentences. A Qdrant index is used to retrieve relevant devices and rules from this corpus given the intake answers. The agent then applies budget and auto approval rules in the model chain and returns a ranked list of devices.

- **Experiment 2 (SQL only):** constructs a per session `inventory_candidates` table by joining `dim_device_config` with the appropriate `fact_budget_quarterly` row. A single validated `SELECT` applies the price cap and department rules inside the warehouse and returns the candidate pool and auto approvable subset without using vector retrieval.

- **Experiment 3 (RAG + KG + DW):** uses the same warehouse snapshot as Experiment 2 but augments it with vector retrieval over the text corpus and a small rule oriented knowledge base. Retrieval is used to surface relevant policy statements and budget history, while the final filter and ranking are enforced in the warehouse view so that only devices that satisfy both budget and auto approval rules appear in the final list.

Table 4.6. *Dataset summary including source, data type, key fields, and how each dataset is used in the experiments.*

Source	Type / Access	Key fields used	Use in experiments
`dim_device_config`	Relational dimension (sponsor provided, read only)	Brand and model; CPU and GPU models; RAM and SSD size; device_price_nzd (list price)	Core device pool. All three experiments draw candidates from this table via a per session `inventory_candidates` slice.
`dim_department` and `dim_employee`	Relational dimensions (sponsor provided, read only)	Department identifiers and names; employee role and seniority; seniority level; entitlement tier	Provide department and role context for each request. Used to apply different device rules for IT, Sales, and Front Desk and to interpret results per employee profile.
`fact_budget_quarterly`	Relational fact table (sponsor provided, read only)	Year and quarter; per department budget; planned and actual spend; budget_overrun_flag	Supplies the quarterly budget limit and overrun history used to compute per request budget caps and to decide when auto approval is allowed or blocked.
`fact_purchase_request`	Relational fact table (populated during tests)	Request identifiers; employee and department context; device chosen; budget snapshot; decision score; auto approval flag	Logs the ten user cases and the decision outcome for each pipeline. Used in later chapters to measure correctness, policy compliance, latency, and token usage.

Licensing and Ethics

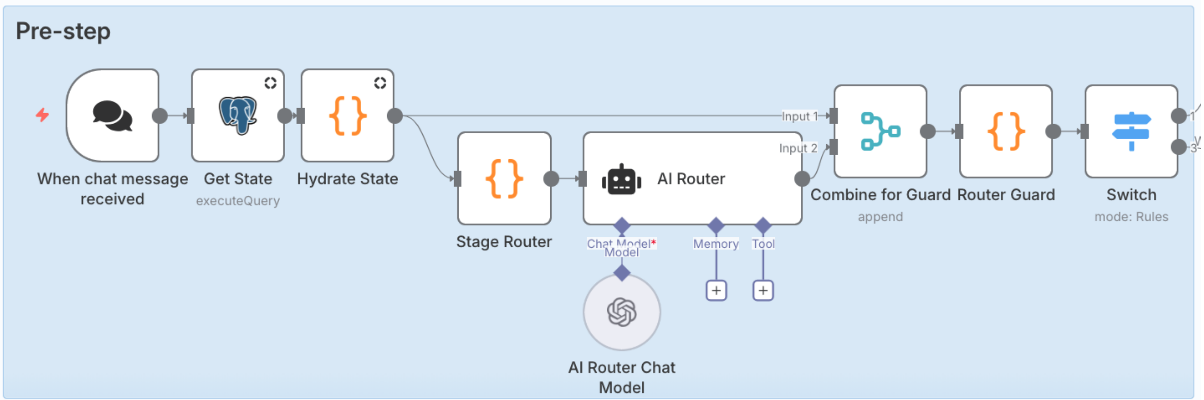
All base tables and extracts used in this thesis were provided by the industry sponsor from a de-identified internal environment. Additional derived views and per session tables were created only for experimental control. No external commercial systems are queried at run time, and no identifiable personal or customer data are present in the warehouse. Session specific schemas and logs are stored in a controlled PostgreSQL instance and can be dropped after analysis.

4.4 Common Assumptions and Notation

4.4.1 Pre-step - Trigger, Hydrate, Route, Guard (used by all experiments)

This pre step runs before any experiment specific process. It handles the session trigger, restores prior state, and decides whether to run **Phase 1 - Spending Questions** or move directly to device browsing and retrieval in **Phase 2 - Device Suggestions**. It also extracts any early hints about department, role, quarter, and budget from the user message so that later stages start with consistent context.

Figure 4.1. *Shared pre step workflow used by all experiments*



Notes. The chain triggers on a chat message, hydrates session state, routes the message to the selected stage, and applies a guard node before any retrieval, SQL, or knowledge base components run.

Implemented by (n8n nodes)

`When chat message received` → `Get State (executeQuery)` → `Hydrate State` → `Stage Router` → `AI Router Chat Model` + `AI Router` → `Combine for Guard` → `Router Guard` → `Switch`.

This chain is identical across all experiments. It ensures consistent entry behaviour and routing logic before any retrieval, SQL, or knowledge graph components run.

Inputs

- User chat message (free text or structured UI action).
- Session identifier `sessionId`.
- Previously saved `state_json` and `ui_json` from the database.
- Hydrated runtime object {`chatInput`, `sessionId`, `userId`, `state`, `ui`} passed into the Stage Router and AI Router.

Process

Each stage lists its main **Input** and **Output**.

1. **Trigger & Get State**
   - **Input:** chat message and `sessionId`.
   - **Output:** matching row from `public.chat_state` containing any prior state and UI data.
2. **Hydrate State** Merge the saved state with the new message to build a runtime object.
   - **Input:** database row and trigger payload.
   - **Output:** consolidated JSON object {`chatInput`, `sessionId`, `userId`, `state`, `ui`, `persist`}.
3. **Stage Router** Apply pattern rules to detect whether the message is a fresh intake question or a request to browse devices. For example, existing state that already

contains a completed capacity block is treated as a signal to skip back to device suggestions, while simple greetings or vague questions stay in Phase 1.

- **Input:** hydrated state and message text.
- **Output:** preliminary `route` suggestion (for example `phase1` or `phase2`) that is passed along with the hydrated object.

4. **AI Router (LLM)** Refine the route decision using the language model and extract structured assumptions about the request. The system prompt defines the router as an IT device procurement assistant and restricts routes to a small set of subflows.

   - **Input:** hydrated JSON and route hint from Stage Router {`chatInput`, `sessionId`, `userId`, `state`, `ui`, `route`}, plus model configuration from `AI Router Chat Model` (model `gpt-4.1-mini`, low temperature, JSON output).
   - **Output:** structured JSON {`ok`, `route`, `assumptions`, `reason`, `confidence`} where `assumptions.capacity` may include early values for `role`, `year`, `quarter`, `request_type`, `department_id`, and `request_budget_nzd`.

5. **Combine for Guard & Router Guard** Merge the hydrated state with the LLM router output and apply a small set of guard rules before branching. The guard prefers an explicit browse intent such as "show devices" or "list laptops" when deciding to skip Phase 1, and otherwise falls back to the intake stage. Existing state can also pin the route, for example when a previous step has already set the stage to device suggestions.

   - **Input:** pattern based router output and LLM result.
   - **Output:** final router object `router.{route, flags}` attached to the runtime JSON.

6. **Switch (branch)** Route the workflow according to `router.route`: if `phase1` → run Phase 1 - Spending Questions; if `phase2` → proceed directly to the experiment's device retrieval and ranking stage.

Outputs

- `router.route` ∈ {`phase1`, `phase2`} determining the next workflow branch.

- Optional `assumptions.capacity` block that captures early values such as department, role, quarter, request type, and an initial budget estimate for later phases.
- Updated runtime state persisted in memory until Phase 1 or device retrieval completes.

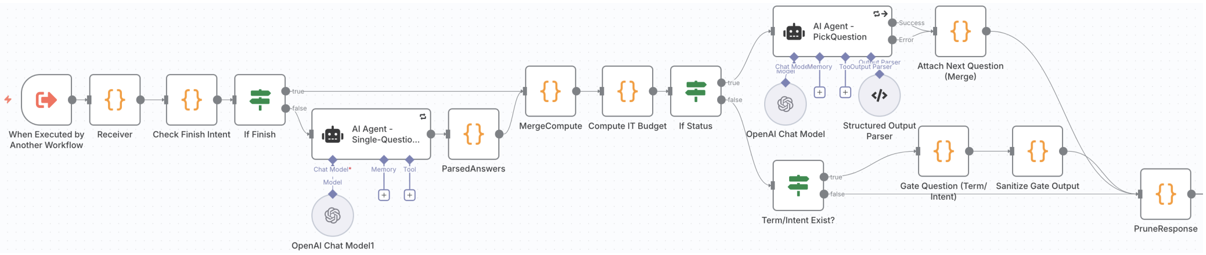
Notes

- The routing and guard logic is identical across Experiments 1 to 3.
- Guard rules are conservative: only explicit browse intent and known device terms promote a message to Phase 2; other messages fall back to Phase 1 for clarification.
- All preprocessing completes before any RAG, SQL, or knowledge base component executes.

4.4.2 Phase 1 - Intake & Capacity (used by all experiments)

This phase collects a small set of categorical answers about the request context and computes a per request budget limit for devices. Instead of modelling household income and expenses, it asks which department the user is in, what their role is, which quarter the purchase is for, and whether the request is a replacement or a net new device. Once all fields are present, the workflow derives a numeric `request_budget_nzd` from departmental quarterly budgets and overrun history and returns a validated capacity block for downstream ranking and filtering.

Figure 4.2. *Phase 1 intake and capacity workflow used by all experiments*



Notes. The loop asks one intake question at a time (department, role, quarter, replacement vs new), merges answers into `state.it_request`, and once all are present computes a per request budget from the department's quarterly allocation and overspend pattern. The same capacity block is then reused by all three experiment pipelines.

Implemented by (n8n nodes)

`When Executed by Another Workflow` → `Receiver` → `Check Finish Intent` → `If Finish` ⇒ (`MergeCompute` *or* `AI Agent - Single-Question Answer Extractor` + `ParsedAnswers` → `MergeCompute`) → `Compute IT Budget` → `If Status` ⇒ (`AI Agent - PickQuestion` + `Structured Output Parser` → `Attach Next Question (Merge)` *or* `PruneResponse`) → `PruneResponse`.

This chain is identical across all experiments and ensures that each pipeline receives the same `it_request` and capacity state before any retrieval or SQL processing.

Inputs (user/session state)

- Raw answers to intake questions about department, role, quarter, replacement vs new, and an optional short usage intent (free text).
- Session state object with `state.it_request` (may be empty on first turn) and optional `state.phase2.capacity`.
- UI shell with an optional `ui.next_question` and `ui.hint`.

Process

Each step lists its **Input** and **Output**. The loop asks one question at a time and recomputes capacity only when all required categorical answers are present.

1. **Start / Receiver** (*When Executed by Another Workflow* + *Receiver*) Hydrate the incoming payload for this session.
   - **Input:** {`user_message`, `state`, `ui`, `persist`}.
   - **Output:** normalised runtime object for downstream nodes.
2. **Finish gate** (*Check Finish Intent* + *If Finish*) Detect "finish" or "ok, proceed" intent before asking another question.
   - **Input:** runtime object and chat text.
   - **Output:** branch:
     - *finish=true* ⇒ go directly to **MergeCompute**.

- *finish=false* ⇒ go to **AI Agent - Single-Question Answer Extractor**.

3. **Ask one question** (*AI Agent - Single-Question Answer Extractor* with *OpenAI Chat Model1*) Pick the current prompt in `ui.next_question` and parse the user's reply for exactly one categorical answer. The agent maps:
   - department names to canonical codes (`DEP_IT`, `DEP_SALES`, `DEP_FRONT`),
   - free text job titles to a short `role` string,
   - quarter phrases to a canonical `YYYY-Qn` label,
   - replacement phrases to `replacement` or `net_new`,
   - and an optional short usage description to `user_intent`.
   - **Input:** current `ui.next_question` and `user_message`.
   - **Output:** structured item {`key`, `answer/text`, `status`} or a re ask with `status=false`.
4. **Parse and validate** (*ParsedAnswers*) Confirm that there is a single clear answer for the current key or emit a re ask.
   - **Input:** extractor output.
   - **Output:**
     - `status=true` ⇒ `answers[]` with canonicalised key and value;
     - `status=false` ⇒ a single `key+question` to re ask, with updated `ui.next_question`.
5. **Merge intake answers** (*MergeCompute*) Upsert the new categorical answers into `state.it_request` and mark readiness.
   - **Input:** prior `state` and new `answers[]`.
   - **Output:** updated
     - `state.it_request.`{`department_id, role, year, quarter, request_type, user_intent`},

- `calc` block with placeholders including `department_id`, `year`, `quarter`, `request_type`,
- `ok` flag set to true when department, role, quarter, and request type are all present, and `reason` set to `ready` or `need_inputs`.

6. **Compute IT budget** (*Compute IT Budget*) Once the required intake fields are present, compute a per request budget based on department allocation and past overspend behaviour.

   - **Input:** `state.it_request` and `calc` from *MergeCompute*.
   - **Output:**
     - `calc.request_budget_nzd` as a fraction of the quarterly budget for the department,
     - updated `state.phase2.capacity` with `department_id`, `role`, `year`, `quarter`, `request_type`, and `request_budget_nzd`.

7. **Status check and next question** (*If Status + AI Agent - PickQuestion* with *Structured Output Parser + Attach Next Question (Merge)*) Decide whether to continue the intake loop.

   - **Input:** `ok`, `reason`, and current `state.it_request`.
   - **Output:** either
     - a new `ui.next_question` for the next missing key (department, then role, then quarter, then request type), or
     - no further question when all fields are present, leaving `ui.next_question = null`.

8. **Prune response** (*PruneResponse*) Shape the final payload for the caller and suppress stale finish flags when a question is still pending.

   - **Input:** combined state and UI from the previous nodes.
   - **Output:** final turn payload {`ui.next_question` or none, `calc`, `state`, `finish`}.

Methods

- **Categorical mapping.** Intake answers are normalised to canonical codes and formats. Department names and numeric options are mapped to `DEP_IT`, `DEP_SALES`, or `DEP_FRONT`. Quarter phrases are converted to the format `YYYY-Qn`. Replacement phrases are mapped to `replacement` or `net_new`. Role and user intent fields are trimmed and shortened to keep them as concise labels.
- **Budget computation.** The *Compute IT Budget* node applies simple rules over the quarterly budget for each department and its observed overrun count. IT, Sales, and Front Desk start with different budget ratios, departments with repeated overspends receive a tighter cap, and replacement requests receive a small bonus that is clipped at 100 percent of the quarterly budget.
- **One question loop.** Each turn either confirms one categorical answer or emits a re ask, then reruns the readiness and budget checks. Once all required fields are present, the loop stops and the capacity block is passed to Phase 2.

Outputs

- `state.it_request` with canonical fields `department_id`, `role`, `year`, `quarter`, `request_type`, and optional `user_intent`.
- `state.phase2.capacity` with the same fields plus `request_budget_nzd` for use by downstream SQL and retrieval components.
- `calc.request_budget_nzd` and supporting metadata such as `calc.department_id`, `calc.year`, `calc.quarter`, and `calc.request_type`.
- `ok/reason` flags that indicate whether Phase 1 is complete or still needs further answers.

4.4.3 Common Formulae

This subsection collects the shared derived fields and evaluation rules that are used across all three experiments in the IT device procurement setting.

Per request budget cap. Phase 1 writes a single row into the `capacity` table for each session with the fields `department_id`, `year`, `quarter`, `request_type`, and `request_budget_nzd`. The numeric budget cap `request_budget_nzd` is computed from the department's quarterly budget `budget_quarter_nzd` and its recent overrun history.

Let B_q be the quarterly budget for the department in the requested year and quarter, and let `overruns_4q` count how many of the last four quarters were overspent. We first set a base ratio r_{base}:

$$r_{\text{base}} = \begin{cases} 0.90 & \text{if department is Front Desk,} \\ 0.60 & \text{if } \texttt{overruns\_4q} \geq 2, \\ 0.70 & \text{if } \texttt{overruns\_4q} = 1, \\ 0.80 & \text{otherwise.} \end{cases}$$

Replacement requests receive a small bonus, capped at the full quarterly budget:

$$r_{\text{eff}} = \begin{cases} \min(1.0, 1.10 \cdot r_{\text{base}}) & \text{if } \texttt{request\_type} = \text{replacement,} \\ r_{\text{base}} & \text{otherwise.} \end{cases}$$

The per request budget cap is then

$$\texttt{request\_budget\_nzd} = r_{\text{eff}} \times B_q.$$

This value is stored in `capacity.request_budget_nzd` and used consistently by all three experiments as the hard price limit for that session.

Budget impact ratio. For each device in the per session candidate table `inventory_candidates` we compute a simple budget impact measure that describes how much of the quarterly budget a single device would consume:

$$\texttt{budget\_impact\_ratio} = \begin{cases} \dfrac{\texttt{price\_nzd}}{\texttt{budget\_quarter\_nzd}} & \text{if } \texttt{budget\_quarter\_nzd} > 0, \\ \text{NULL} & \text{otherwise.} \end{cases}$$

This field is used for explanation and for knowledge graph enrichment, not as a hard constraint.

Device slice and expected sets. The global IT device catalogue `dim_device_config` is materialised per session into `user_dw_<sessionId>.inventory_candidates`. Each row joins the static device attributes (brand, model, CPU/GPU, RAM, SSD size, `price_nzd`, `performance_tier`) with session specific budget context:

- `department_id`, `year`, `quarter`, `budget_quarter_nzd`,
- `budget_impact_ratio`,
- `auto_approve_ok`, computed from departmental rules and the per request budget.

The departmental rules for `auto_approve_ok` follow the sponsor's policy:

- IT can auto approve mid, high, and workstation tier devices within the request budget.
- Sales can auto approve basic and mid tier devices within the request budget.
- Front Desk can auto approve basic tier devices within the request budget.

In all cases a device is only marked `auto_approve_ok = true` when `price_nzd` is less than or equal to `request_budget_nzd` for that session.

For evaluation we define two deterministic expected sets for each user case, computed once using SQL over the per session warehouse:

- **Q1 (budget only).** All devices where `price_nzd` $\leq$ `request_budget_nzd`, ignoring `auto_approve_ok`.
- **Q2 (budget + policy).** All devices where `price_nzd` $\leq$ `request_budget_nzd` and `auto_approve_ok` = true.

These expected sets are used as the ground truth for Experiments 1–3 when checking correctness.

Ranking and determinism. Whenever a pipeline returns a list of devices it must produce a stable, reproducible order. For the SQL based pipeline (Experiment 2) and for the hybrid pipeline when it delegates ranking to SQL, we use the following ordering:

```
ORDER BY
  price_nzd ASC,
  performance_tier DESC,
  device_id ASC
```

Here `performance_tier` is treated as an ordinal category, with `basic` → `mid` → `high` → `workstation`. The final `device_id` key breaks any remaining ties to keep the ordering deterministic.

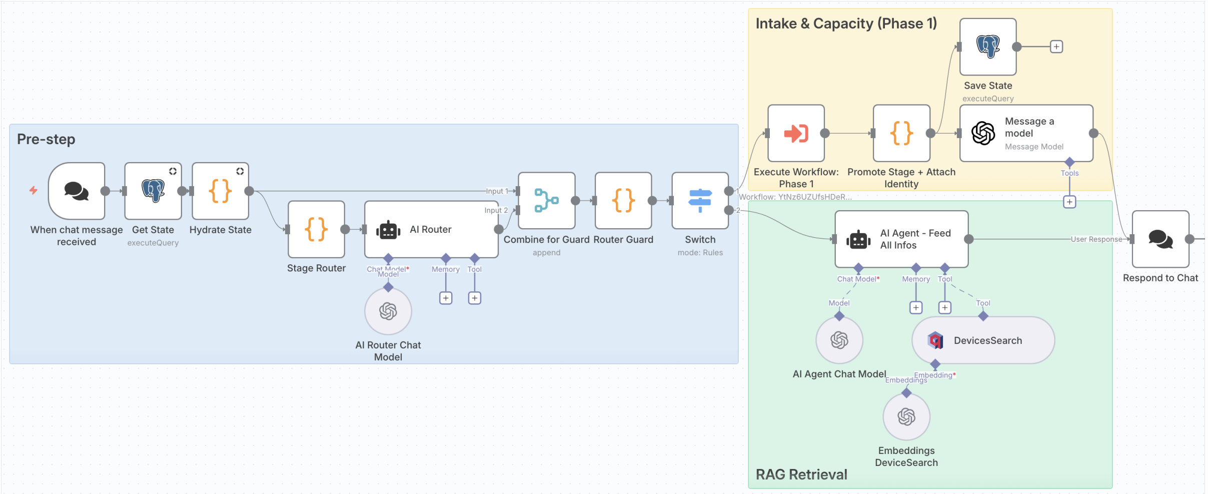
Metrics and reporting. The experiments share a common set of correctness and efficiency measures:

- **Top 1 budget compliance.** Whether the first returned device satisfies `price_nzd` ≤ `request_budget_nzd` for that session.
- **Matches expected set.** Order agnostic equality between the device identifiers returned by the pipeline and the SQL expected set for Q1 or Q2.
- **Latency.** End to end time from receiving the user message at the n8n trigger to emitting the final answer.
- **Token usage.** Total prompt plus completion tokens per run, as reported by the model provider.
- **Tukey summaries and IQR.** For latency and token usage we report the five point summary (minimum, Q_1, median, Q_3, maximum) using Tukey's hinges, and the interquartile range $IQR = Q_3 - Q_1$.

4.5 Experiment 1 – All Information, RAG to the Model

This experiment tests whether a RAG only agent can recommend suitable IT devices when it is given the employee context and per request budget up front. The agent uses vector retrieval over a semantic index of devices and approval rules and does not issue SQL. All budget logic is applied inside the model chain using the capacity block from Phase 1 and the shared rules in Section 4.4.3.

Figure 4.3. *n8n workflow for Experiment 1 (RAG only)*



Notes. After the shared pre step and Phase 1 intake and capacity, the workflow promotes the per request budget and IT context into `state.phase2.capacity`, then calls a RAG agent. The agent uses semantic search over the device catalogue and policy text and returns a ranked list of devices and a short explanation to the chat interface.

Scope

- Uses only the vector retrieval layer (`devices_main` collection in Qdrant) via the `DevicesSearch` tool.
- No SQL execution or schema aware reasoning inside the warehouse in this experiment.
- Capacity and departmental budget logic come from Phase 1 and the derived `request_budget_nzd` field (Section 4.4.2 and Section 4.4.3).
- The same retrieval and ranking prompt is used for all ten user cases (A–J).

Inputs

- From Phase 1 (see Section 4.4.2): `state.phase2.capacity.`{`department_id, role, year, quarter, request_type, request_budget_nzd`}.
- Device and policy fields from the vector index `devices_main`: identifiers such as `device_id`; hardware attributes (`pc_brand`, `model_name`, `cpu_model`, `gpu_model`, `ram_gb`, `ssd_size_gb`); commercial fields (`price_nzd` or `device_price_nzd`); and policy metadata such as `performance_tier` and `auto_approve_ok`.

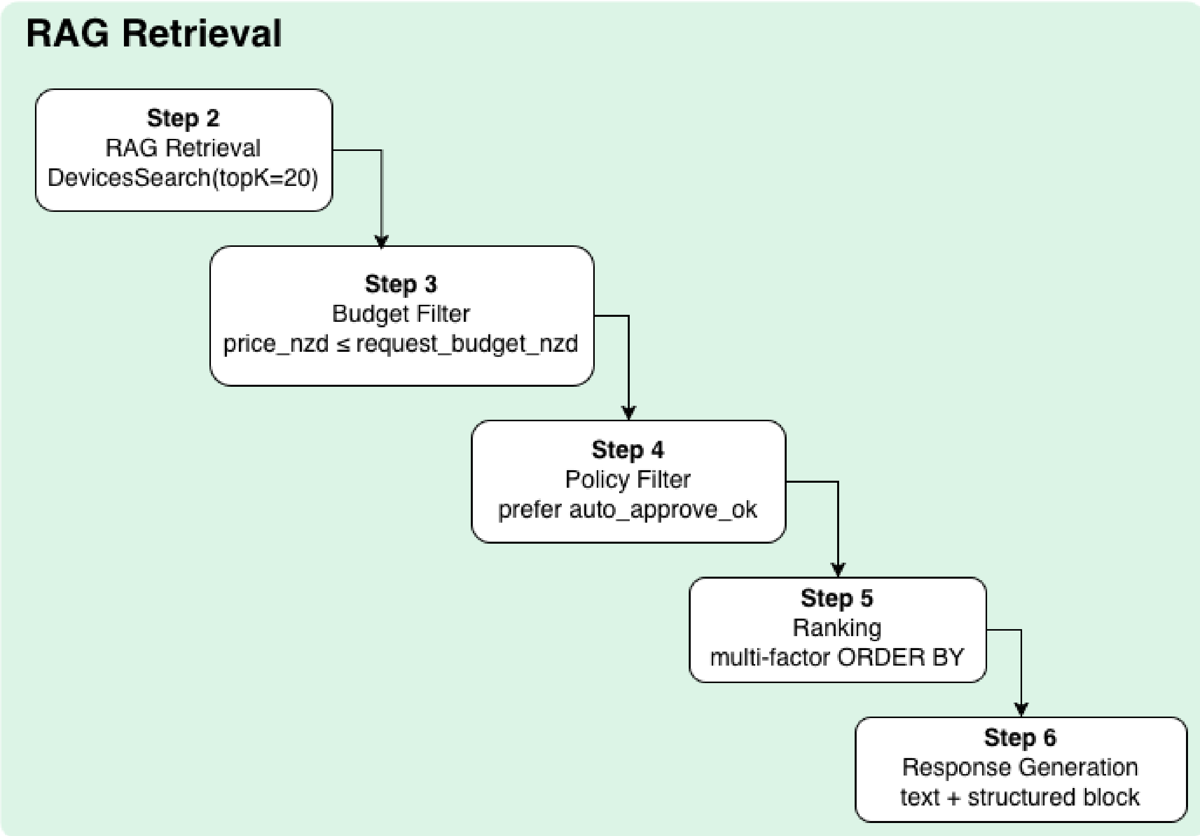
- User natural language query or intent (for example "basic front desk PC under budget" or "high performance laptop 32 GB RAM for data work").

Process

Each process step is described with its **Input** and **Output**.

1. **Pre step** (**Trigger, Hydrate, Route, Guard**) Reuse the shared workflow defined in Section 4.4.1.
   - **Input:** user chat message; `sessionId`; previously saved `state_json/ui_json`.
   - **Output:** `router.route` $\in$ {`phase1`, `phase2`}; if `route = phase1` the workflow calls Phase 1, otherwise it proceeds directly to the RAG agent.
2. **Intake & Capacity (Phase 1)** Reuse the shared workflow defined in Section 4.4.2.
   - **Input:** user chat message and previous session state.
   - **Output:** canonical `state.it_request` and `state.phase2.capacity` with `department_id`, `role`, `year`, `quarter`, `request_type`, and `request_budget_nzd`. The helper node *Promote Stage + Attach Identity* also sets `state.stage = phase2` and attaches hints such as "show devices" to the UI.

Figure 4.4. *Internal RAG pipeline for Experiment 1 (vector retrieval and model side filtering)*



Notes. The agent receives the last user message and a TOOL_ARGS object containing the department, role, quarter, request type, and request budget. It builds a focused search query, calls `DevicesSearch` once, filters candidates by price and approval rules, ranks them, and emits a numbered list of devices with brief justifications.

3. **RAG retrieval and planning (Phase 2)** The node *AI Agent - Feed All Infos* implements the RAG agent for Experiment 1. It uses *AI Agent Chat Model* (GPT 4.1 mini) together with the `DevicesSearch` tool over Qdrant.

   - **Input:**
     - last user query `chatInput`,
     - capacity block `state.phase2.capacity`,
     - `DevicesSearch` tool bound to `devices_main` with `topK = 20`.
   - **Process:**
     - Build a JSON `TOOL_ARGS` object containing `request_budget_nzd`,

department_id, role, year, quarter, request_type, and limit.

  - Compose a short natural language search query that combines the employee role, department, quarter, request type, and any explicit device preferences from the chat message.
  - Call DevicesSearch once with that query and TOOL_ARGS, which performs semantic retrieval over device configs, prices, and approval rule snippets in Qdrant.

- **Output:** a list of candidate devices and associated metadata from the semantic index.

4. **Budget and policy filtering (inside the agent)** The system prompt instructs the agent to treat request_budget_nzd as a hard cap and to respect department specific auto approval rules.
   - **Input:** candidate devices from DevicesSearch and TOOL_ARGS.
   - **Output:** a filtered subset where:
     - price_nzd ≤ request_budget_nzd when a budget is provided, and
     - devices violating the sponsor's auto approval rules for the department are either dropped or deprioritised (for example front desk should get basic tier devices, IT can receive higher performance tiers when within budget).

5. **Ranking** Ranking is also handled inside the agent according to the shared guidelines:
   - First, enforce the hard price cap.
   - Second, prefer devices with auto_approve_ok = true for the current department and quarter.
   - Third, sort by device price ascending.
   - Finally, break ties by role appropriate performance tier (higher tiers for technical roles, mid range for non technical roles) and device identifier.

   The agent then selects up to limit devices (20 in this experiment) to present.

6. **Response generation** The agent formats the shortlist into a numbered list of

devices, each line starting with `device_id` followed by a human readable description and price. It may also include a brief explanation of how the devices match the department, role, and budget. The result is passed to the *Respond to Chat* node, which sends the message back to the client.

- **Input:** ranked shortlist with metadata.
- **Output:** natural language message plus structured data in the workflow state for logging and evaluation.

Methods and Assumptions

- Retrieval uses Qdrant with OpenAI embeddings. The `Embeddings DeviceSearch` node computes vector embeddings for the text fields that describe devices and approval rules, and `DevicesSearch` performs cosine similarity search over the `devices_main` collection.
- The agent uses the same `gpt-4.1-mini` chat model for planning, retrieval, and response generation; all budget and policy logic is expressed through the prompt and the tool outputs rather than SQL.
- Phase 1 capacity results are held only in the runtime session state and in the shared `chat_state` table; Experiment 1 does not create a per session warehouse slice and does not write to `inventory_candidates`.
- Ranking parameters (`topK = 20` retrieval limit and role sensitive tier preferences) are kept constant across all user cases for this experiment.

Outputs

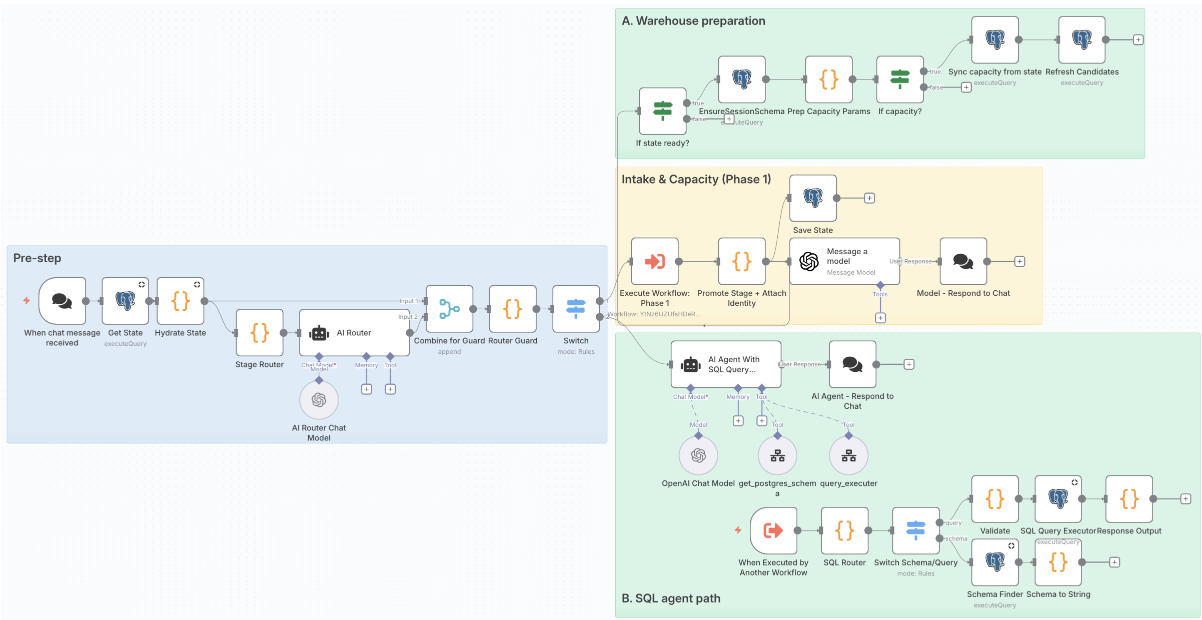
- A ranked shortlist of IT devices including identifier, brand, model, key specs (CPU, RAM, SSD, performance tier), price, and whether the device appears auto approvable for the current department and quarter.
- A summary line or brief explanation that reflects the applied budget cap `request_budget_nzd` and the user's department and role.
- Logged metadata per run, including retrieval latency, number of candidates before and after budget filtering, and final list size, which are used in Chapter 5 when

comparing correctness and efficiency against Experiments 2 and 3.

4.6 Experiment 2 - Model Queries the DW (SQL)

Experiment 2 tests a warehouse-first agent that never reads vector-indexed text at answer time. Instead of RAG over unstructured documents, the agent issues a single validated `SELECT` against a per-session schema `user_dw_<sessionId>` that exposes the request budget and a slice of sponsor-provided device inventory. The goal is to see whether an SQL-only agent can recover the same budget-eligible device sets as the RAG and Hybrid pipelines, and whether it can respect departmental auto-approval rules using only structured data.

Figure 4.5. *n8n workflow for Experiment 2 (SQL only)*



Notes. After the shared pre step and Phase 1 intake, the workflow prepares a per-session data warehouse schema, writes the request budget and context into `capacity(id=1)`, refreshes `inventory_candidates` from the sponsor's device view, then lets an SQL agent discover the schema, compose one safe `SELECT`, execute it, and return the result to the chat.

Scope

- Per-session warehouse schema: `user_dw_<sessionId>`.
- Two core tables for this experiment:

- capacity(id=1) with IT request fields (request_budget_nzd, department_id, year, quarter, request_type).
- inventory_candidates holding the device surface for the session (device_id, brand/model, CPU/GPU, RAM/SSD, price_nzd, performance_tier, budget_impact_ratio, auto_approve_ok, timestamps).[1]

- The agent path always produces a single read-only SELECT over the session schema, validates it against safety rules, enforces a LIMIT, and never performs DDL or DML itself.
- Only Experiment 2 performs warehouse preparation (schema creation and per-session INSERT/DELETE) before the agent runs.
- No vector retrieval, no unstructured RAG context, and no knowledge graph are used in this experiment.

Inputs

- **From Phase 1 (shared):** IT intake output stored in state.phase2.capacity, including request_budget_nzd, department_id, year, quarter, and request_type, plus any user preferences (state.prefs).
- **Warehouse sources:** sponsor-provided view public.vw_device_candidates (device specifications and performance tiers) and fact table public.fact_budget_quarterly (per-department quarterly budgets).
- **User query:** natural-language description of the request and any filters (for example "high performance IT laptop", "basic front desk PC").
- **Agent tools:** get_postgres_schema (schema discovery for the per-session DW) and query_executer (safe SQL execution).[2]

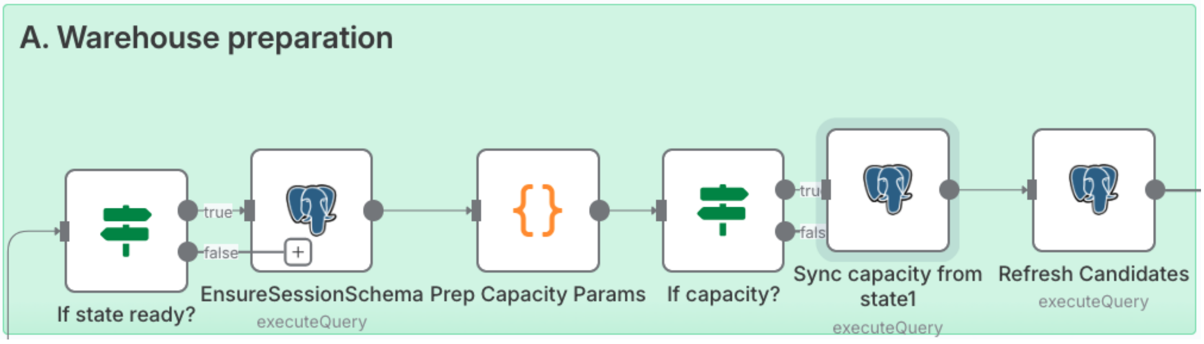
[1] The table structure mirrors the sponsor-provided view vw_device_candidates and budget table fact_budget_quarterly.

[2] Both tools are exposed as n8n sub-workflows that enforce a single SELECT statement with a mandatory LIMIT.

Process

Before Step 1, the shared pre step (Section 4.4.1) runs. If it routes to Phase 1, the intake and capacity workflow (Section 4.4.2) completes the IT request budget and then promotes the session to the device stage.

Figure 4.6. *Warehouse preparation steps for Experiment 2*



Notes. The workflow ensures the per-session schema and tables exist, extracts the IT request budget and context from Phase 1, writes them into `capacity(id=1)`, and refreshes `inventory_candidates` with devices from the sponsor's view that fit the request budget and departmental policy attributes.

1. **EnsureSessionSchema**
   - **Input:** `sessionId`.
   - **Process:** Create schema `user_dw_<sessionId>` if missing and ensure two tables exist: `capacity` and `inventory_candidates`. The DDL reflects the IT fields described above.
   - **Output:** Guaranteed per-session schema with empty or existing tables.
2. **Prep Capacity Params**
   - **Input:** Phase 1 state including `state.phase2.capacity` and `state.it_request`.
   - **Process:** Extract and normalise `request_budget_nzd`, `department_id`, `year`, `quarter`, and `request_type`.
   - **Output:** a compact payload for the DW: {`sessionId`, `request_budget_nzd`, `department_id`, `year`, `quarter`, `request_type`}.
3. **If capacity? (gate)**
   - **Input:** capacity payload from Step 2.

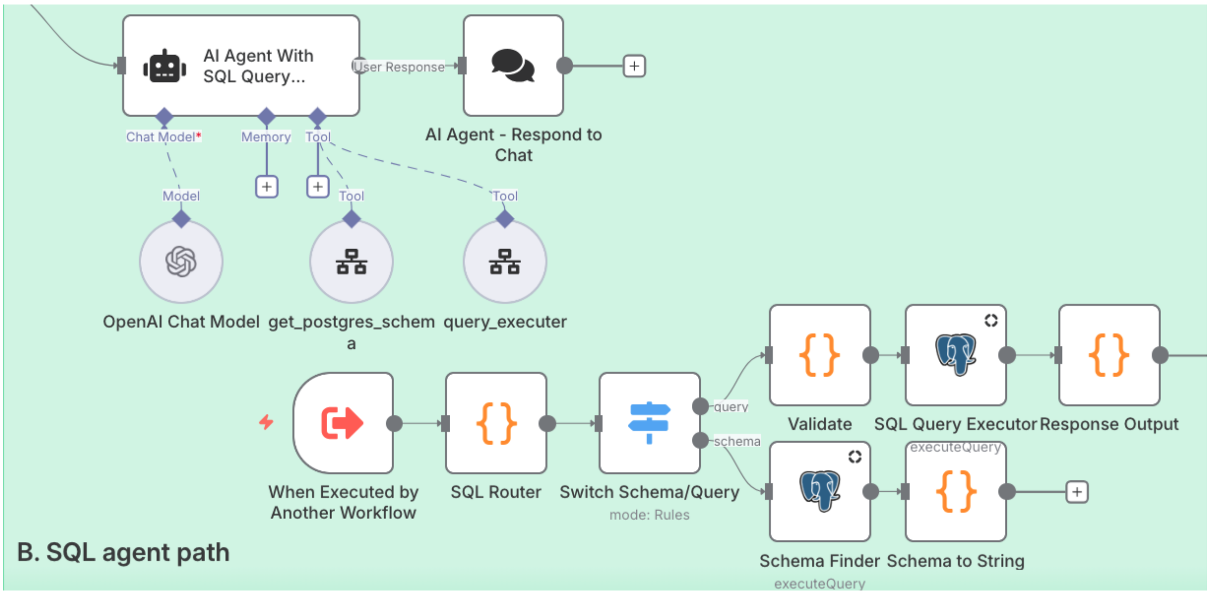
- **Process:** Check that `request_budget_nzd` is non-null.
- **Output:** if a budget is available, continue to Step 4; otherwise skip DW sync and allow the agent to run with an empty candidate table.

4. **Sync capacity from state**
   - **Input:** `sessionId` and IT capacity fields.
   - **Process:** Upsert row `id = 1` in `capacity`, setting `request_budget_nzd`, `department_id`, `year`, `quarter`, `request_type`, and `updated_at`.
   - **Output:** per-session capacity table containing the current request budget.

5. **Refresh Candidates**
   - **Input:** `capacity(id=1)`, `vw_device_candidates`, and `fact_budget_quarterly`.
   - **Process:**
     (a) Read `department_id`, `year`, `quarter`, and `request_budget_nzd` from `capacity`.
     (b) Look up the corresponding quarterly budget in `fact_budget_quarterly`; if missing, fall back to the request budget.
     (c) Clear any previous rows from `inventory_candidates` for this session.
     (d) Insert devices from `vw_device_candidates` with `price_nzd` less than or equal to the request budget, computing:
       - `budget_quarter_nzd` (the reference budget for that department and quarter),
       - `budget_impact_ratio` (device price divided by quarter budget),
       - `auto_approve_ok`, a Boolean flag derived from department-specific performance tiers (for example IT allowed mid, high, and workstation tiers within budget, while Sales and Front Desk have stricter rules).[3]
   - **Output:** populated `inventory_candidates` table containing all devices that fit the request budget, enriched with quarter budget and auto-approval flags.

[3] The auto-approval flag is precomputed in SQL using the sponsor's rules so the agent can filter on a single column.

Figure 4.7. *SQL agent path for Experiment 2*



Notes. The SQL agent discovers the per-session schema, composes exactly one read-only `SELECT` over `inventory_candidates` with hard budget constraints, calls the query tool to validate and execute the SQL, then forwards the result rows to a response agent that formats the shortlist for the user.

6. **AI Agent With SQL Query Prompt** (LLM orchestrator)

- **Input:** user intent (`chatInput`); IT capacity fields from Phase 1; optional user preferences; tool handles for `get_postgres_schema` and `query_executer`.
- **Process:**
  (a) Call `get_postgres_schema` to locate the user schema whose name starts with `user_dw_` and to receive a concise description of `capacity` and `inventory_candidates`.
  (b) Compose a single read-only `SELECT` against `"user_dw_<sessionId>".inventory_candidates` aliased as `ic`, applying at least:
    – `ic.price_nzd IS NOT NULL`,
    – `ic.price_nzd <= request_budget_nzd`.

  When the user question is about auto-approval (Q2), the agent also enforces `ic.auto_approve_ok = true`; otherwise it must not filter on this column.

(c) Project device fields needed for ranking and explanation, such as `device_id`, `pc_brand`, `model_name`, `cpu_model`, `gpu_model`, `ram_gb`, `ssd_size_gb`, `price_nzd`, `performance_tier`, and `auto_approve_ok`.

(d) Attach an `ORDER BY` clause (see below) and a fixed `LIMIT` (default 20).

(e) Call `query_executer`, which routes through a validation node to strip trailing semicolons, block non-`SELECT` verbs and multiple statements, enforce a `LIMIT`, and then execute the query against PostgreSQL.

- **Output:** validated SQL statement kept for traceability, plus an execution preview containing result rows and a short text summary for the downstream response agent.

7. **AI Agent - Respond to Chat**

   - **Input:** result rows and summary from the SQL agent, plus the request budget and department context.
   - **Process:** render a ranked shortlist of devices and a one-line explanation stating how many devices were found within the budget. The agent lists every row from the SQL result and includes the original `device_id` for auditability.
   - **Output:** natural-language response and a structured block for the UI containing the shortlist.

Methods and Assumptions

- **Budget in SQL.** Every query must enforce the hard constraint `ic.price_nzd <= request_budget_nzd` derived from Phase 1 and recorded in `capacity(id=1)`; the agent is not allowed to relax this when answering Q1 or Q2.
- **Auto approval.** For Q2-style questions (devices that are "auto approved" or "safe, standard within policy"), the agent sets a flag `require_auto_approve` and must include `AND ic.auto_approve_ok = true` in the `WHERE` clause. For Q1 (budget-only) questions it must not filter on `auto_approve_ok`.
- **Projection.** Typical projection includes

```
SELECT
```

```
  ic.device_id,
  ic.pc_brand, ic.model_name,
  ic.cpu_model, ic.gpu_model,
  ic.ram_gb, ic.ssd_size_gb,
  ic.price_nzd,
  ic.performance_tier,
  ic.auto_approve_ok
FROM "<USER_SCHEMA>".inventory_candidates AS ic
```

- **Ranking.** Devices are ordered by ascending price within the budget, with missing prices treated as worst. A typical pattern is:

```
ORDER BY
  ic.price_nzd ASC NULLS LAST
LIMIT 20;
```

- **Safety.** A validation node ensures that only read-only `SELECT` statements are executed, that no internal semicolons or DDL/DML verbs are present, and that a `LIMIT` is always applied.

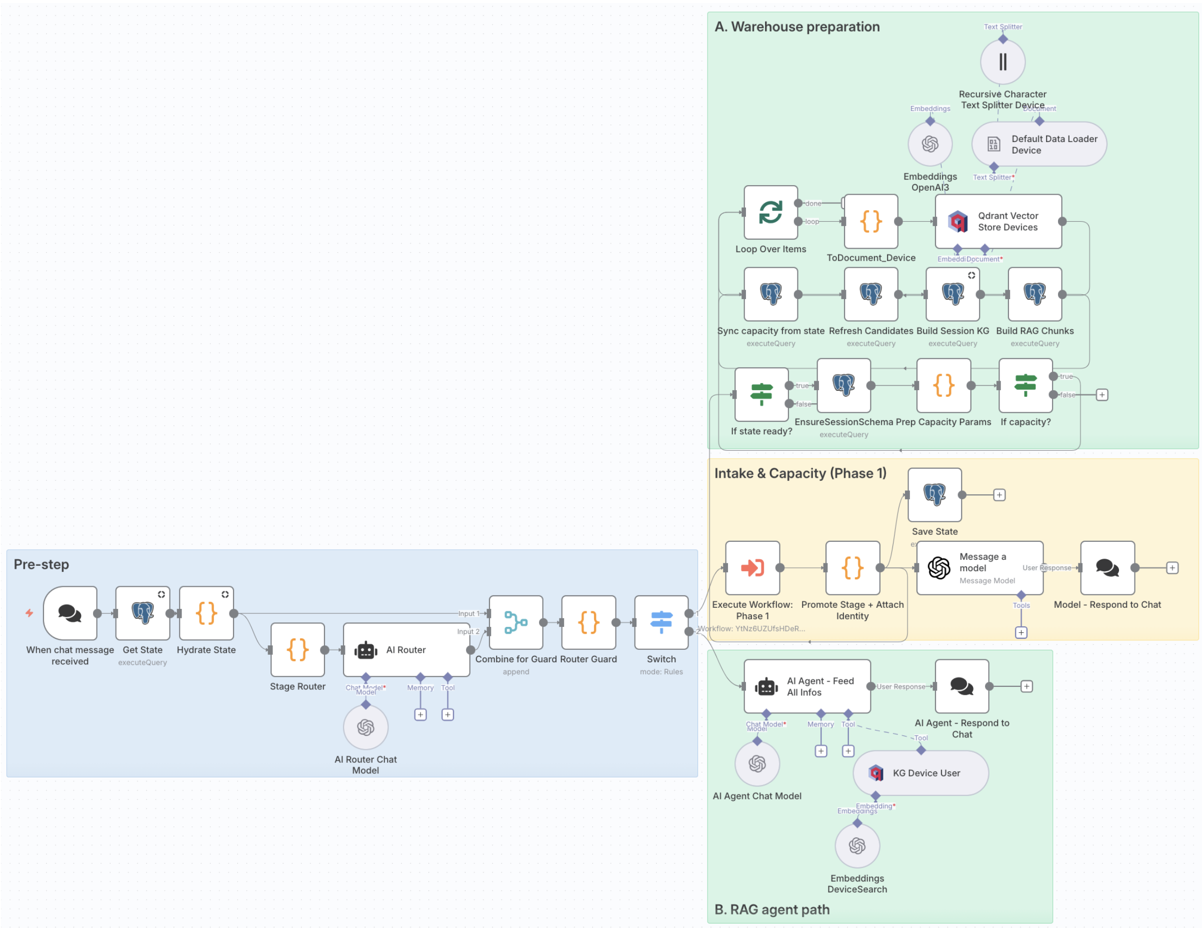
Outputs

- Ranked shortlist of budget-eligible devices including `device_id`, brand, model, key specifications, price in NZD, performance tier, and the `auto_approve_ok` flag when present.
- One-line summary stating how many devices were found within the budget and, for Q2-style questions, how many are auto-approvable under the current rules.
- Logged metrics for evaluation, including warehouse preparation latency, schema discovery time, SQL execution latency, and row counts before and after applying hard constraints and optional auto-approval filters.

4.7 Experiment 3 – RAG with Knowledge Graph and DW

Experiment 3 adds a per session data warehouse and knowledge graph on top of the IT device index, together with semantic retrieval over RAG text. After Phase 1 has captured the department, quarter, and per request budget, the workflow materialises a session scoped snapshot of eligible device candidates, builds knowledge graph nodes and edges that relate the user and budget to each candidate, and writes compact device descriptions into a `kg_text` table. These rows are embedded into a shared Qdrant collection and queried via a session filter so that each session has an isolated logical index. At answer time, the agent embeds the user request, calls the device retrieval tool over this index, filters candidates to respect the request budget and any auto approval requirement, and formats a ranked shortlist and budget summary. No SQL is executed at retrieval time; all SQL runs in the warehouse preparation stage. The underlying n8n workflow instantiates these steps using explicit nodes for schema setup, capacity sync, candidate refresh, knowledge graph construction, RAG chunking, embedding, and device retrieval.

Figure 4.8. *n8n workflow for Experiment 3 (Hybrid RAG + knowledge graph + DW)*



Notes. After the shared pre step and Phase 1 intake and capacity, stage A prepares the per session warehouse and knowledge graph and writes RAG chunks to the vector index; stage B runs the device retrieval agent that queries the index, applies budget or auto approval constraints, and responds to the chat.

Scope

- Per session schema: `user_dw_<sessionId>` with tables `capacity`, `inventory_candidates`, `kg_node`, `kg_edge`, and `kg_text`.
- Logical session index in Qdrant: collection `kg_devices`, filtered at query time on `metadata.session_id` and `kind='candidate'`.
- Retrieval at answer time uses only semantic search over the embedded RAG texts and metadata filtering; there is no direct SQL issued by the agent.
- The same hybrid pipeline serves both Q1 (budget only) and Q2 (budget plus auto approval policy) queries, controlled by a flag in the tool arguments.

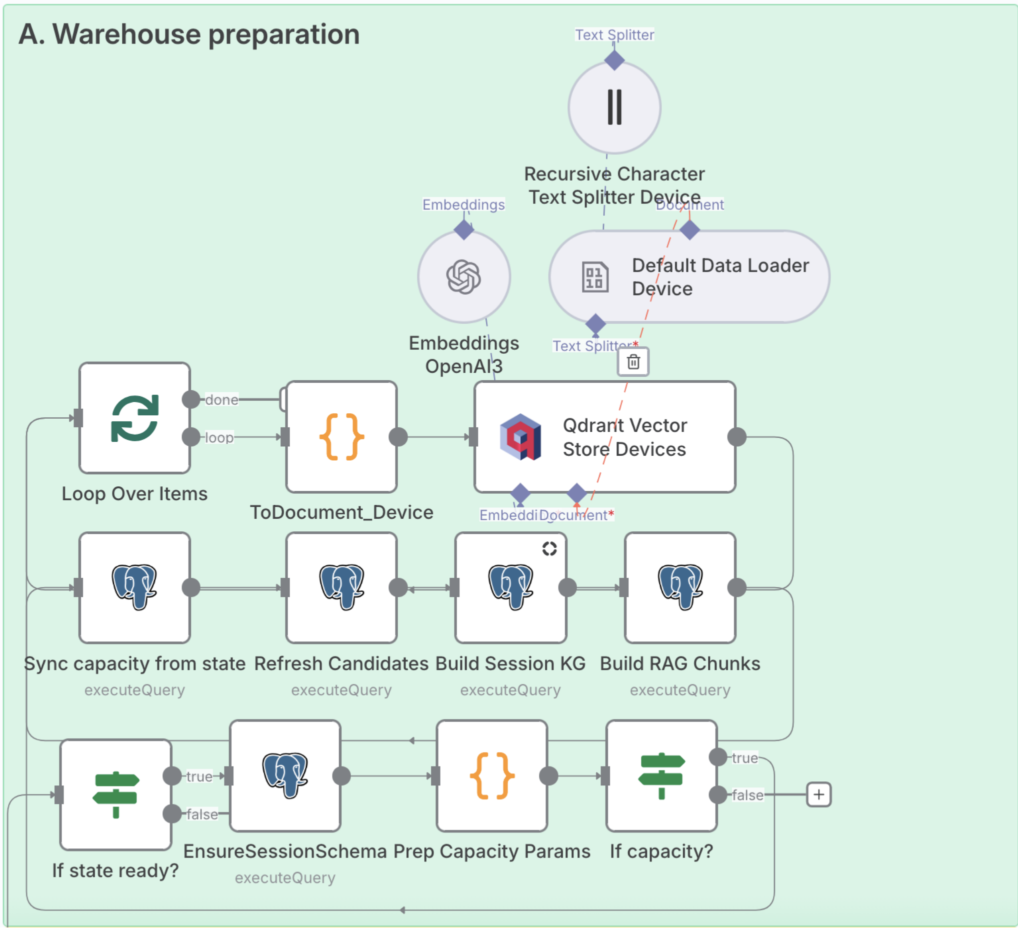
Inputs

- **From Phase 1 (shared, Sec. 4.4.2):** final per request budget `request_budget_nzd`, department identifier, year, quarter, and request type.
- **DW sources:** sponsor view `public.vw_device_candidates` and fact table `public.fact_budget_quarterly`, plus the per session `capacity(id=1)` row.
- **User query:** free text query describing desired devices, performance preferences, or a policy constrained request (for example "safe standard choices within policy for IT support").

Process

Shared pre step and Phase 1. Before any experiment specific logic runs, the shared pre step (Sec. 4.4.1) hydrates session state, routes the message, and applies the confidence guard. If routing selects Phase 1, the workflow calls the shared intake and capacity workflow (Sec. 4.4.2) to collect and validate IT request details and to compute the per request budget. Once Phase 1 reports that intake is complete and the budget is ready, the system promotes the session to the device stage and triggers the warehouse preparation steps below.

Figure 4.9. *Warehouse and knowledge graph preparation steps for Experiment 3*



Notes. The pipeline ensures the per session schema and core tables exist, syncs capacity parameters, refreshes the session device candidates based on the sponsor view and quarterly budgets, builds knowledge graph nodes and edges, constructs RAG chunks in `kg_text`, and embeds them into the shared `kg_devices` collection with per session metadata.

Stage A: DW and KG preparation. Each step lists its main **Input** and **Output**.

1. **EnsureSessionSchema** (DW bootstrap)
   - **Input:** `sessionId`.
   - **Process:** create schema `user_dw_<sessionId>` if missing; ensure `capacity`, `inventory_candidates`, `kg_node`, `kg_edge`, and `kg_text` tables exist with the required columns and indexes; add uniqueness constraints on knowledge graph keys and edges.
   - **Output:** per session schema with DW and KG tables ready.
2. **Prep Capacity Params**

- **Input:** runtime state after Phase 1 (IT intake) containing `state.phase2.capacity` and `state.it_request`.
- **Process:** extract and normalise `request_budget_nzd`, department, year, quarter, and request type; coerce numeric values and tidy quarter labels.
- **Output:** compact parameter object {`sessionId`, `request_budget_nzd`, `department_id`, `year`, `quarter`, `request_type`}.

3. **If capacity?** (gate)
   - **Input:** capacity parameters.
   - **Process:** check that `request_budget_nzd` is present and non null.
   - **Output:** if budget is missing, skip the DW sync; otherwise continue to update `capacity` and candidates.
4. **Sync capacity from state**
   - **Input:** capacity parameters.
   - **Process:** upsert row `capacity(id=1)` with the per request budget, department, year, quarter, and request type for this session.
   - **Output:** deterministic capacity row representing the current IT request.
5. **Refresh Candidates**
   - **Input:** per session capacity row `capacity(id=1)`, sponsor view `public.vw_device_candidates`, and quarterly budget facts.
   - **Process:**
     (a) Look up the department and quarter level budget in `fact_budget_quarterly` for the relevant department, year, and quarter, falling back to the per request budget if no quarter record is found.
     (b) Delete any existing rows from `inventory_candidates` for this session.
     (c) Insert devices from `vw_device_candidates` where `price_nzd` is less than or equal to `request_budget_nzd`.

(d) For each device, attach the department, year, quarter, quarter budget, and `budget_impact_ratio` (price divided by quarter budget), performance tier, and an `auto_approve_ok` flag that encodes the sponsor's auto approval rules for IT, Sales, and Front Desk tiers.

- **Output:** `inventory_candidates` snapshot holding only devices within the per request budget together with department specific policy metadata.

6. **Build Session KG**

- **Input:** `capacity(id=1)`, `inventory_candidates`, and intake context (user role, department, request type).
- **Process:**

(a) Upsert three core nodes in `kg_node`: *User*, *Budget*, and *Assumption*, keyed by `user`, `budget`, and `assumption`, with JSON properties describing the request and budget.

(b) For each device candidate, upsert a *CandidateDevice* node (label chosen for compatibility with the previous car study) with key `cand:<device_id>` and properties taken from `inventory_candidates`.

(c) Insert or update *within_budget* edges from the *Budget* node to each candidate node, with a weight of 1 when the device price is at or below the budget and 0 otherwise; clean up edges that point to non existent candidates.

- **Output:** populated `kg_node` and `kg_edge` tables that encode the relationship between the request budget and each candidate device.

7. **Build RAG Chunks**

- **Input:** session `capacity` and `inventory_candidates`.
- **Process:**

(a) Delete any prior `kg_text` rows for this session with `kind` in {`candidate`, `summary`}.

(b) For each device candidate, insert a one line text description (`kind='candidate'`) summarising device id, brand, model, CPU, RAM, SSD size, price, performance tier, and whether it is auto approvable; attach rich metadata including device id, performance tier, auto approval flag, department, year, quarter, quarter budget, budget impact ratio, and request budget.

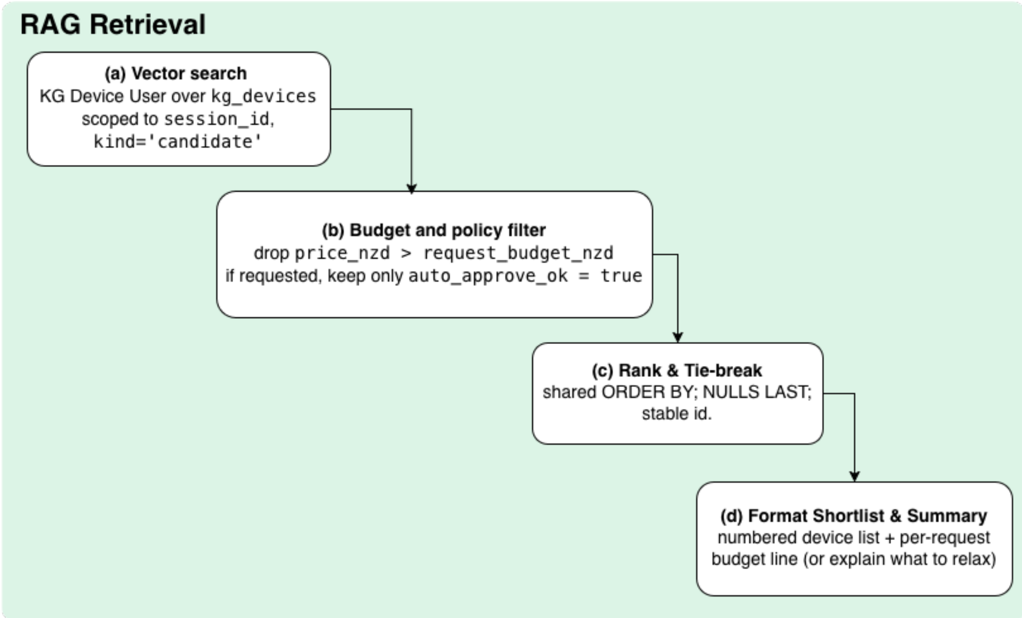
    (c) Insert a single kind='summary' row describing the department, quarter, request type, and budget in natural language for use as high level context.

- **Output:** per session RAG documents in kg_text, tagged with session_id and kind.

8. **Vectorise and store** (session index)

    - **Input:** kg_text rows for this session.
    - **Process:** convert each row to a LangChain document with page content equal to the text and metadata carrying session_id, doc_id, kind, key, and the device level attributes; apply optional text splitting; embed documents using the OpenAI embeddings model; insert them into Qdrant collection kg_devices with a composite id <sessionId>:<doc_id>.
    - **Output:** embedded RAG documents in Qdrant, filterable by session id and kind.

Figure 4.10. *Internal RAG and knowledge graph based retrieval pipeline for Experiment 3*



Notes. The agent embeds the user query, calls the `KG Device User` tool to retrieve this session's device candidates, filters and ranks the results according to budget and policy settings, and formats a numbered shortlist plus a one line budget summary.

Stage B: RAG agent path. After Stage A has prepared the per session warehouse, knowledge graph, and vector index, the routing logic directs device queries to the hybrid RAG agent.

1. **Vector search with tool arguments**

    - **Input:** user message (`chatInput`), Phase 1 capacity fields (request budget, department, year, quarter, request type), and any saved device preferences.

    - **Process:** the `AI Agent - Feed All Infos` node constructs a `TOOL_ARGS` object that includes the budget and contextual fields plus a boolean `require_auto_approve` flag. The flag is set when the message contains phrases that explicitly or implicitly request auto approved or "safe standard" options within policy, and is false for budget only queries. The agent then calls the `KG Device User` tool, which performs semantic search over `kg_devices` with a filter on `session_id` and `kind='candidate'` and returns up to 20 candidate documents with their metadata.

    - **Output:** list of candidate devices with associated metadata (device id, brand, model, price, performance tier, auto approval flag, budget impact ratio, and request budget).

2. **Filter for budget and auto approval**
   - **Input:** retrieved candidates and `TOOL_ARGS`.
   - **Process:** within the agent, apply the following rules:
     - Always discard candidates whose `price_nzd` exceeds `request_budget_nzd` when that value is present in `TOOL_ARGS`.
     - If `require_auto_approve` is true, keep only candidates with `auto_approve_ok = true`; do not include devices that would require manual approval.
     - If `require_auto_approve` is false, retain both auto approvable and non auto approvable devices that meet the budget, while still surfacing the auto approval flag in the description.
   - **Output:** filtered candidate set consistent with the requested budget and policy mode (Q1 or Q2).

3. **Rank and tie break**
   - **Input:** filtered candidate devices.
   - **Process:** sort the final list by `price_nzd` in ascending order so that cheaper options appear first, and within a given price prefer higher performance tiers over lower ones. The agent keeps up to a fixed upper bound on the number of devices to display (for example 10 to 50).
   - **Output:** ranked shortlist for presentation.

4. **Format shortlist and summary**
   - **Input:** ranked shortlist and capacity context.
   - **Process:** render each device as a numbered line starting with `device_id`, followed by brand, model, key specifications, price in NZD, and performance tier, explicitly noting whether each option is auto approvable. Add a one sentence summary describing how the list relates to the request budget and,

when applicable, the auto approval policy. If no devices remain after filtering, explain which constraint made the result empty (budget too low or auto approval too strict) and ask whether the user wishes to increase the budget, allow non auto approved devices, or relax performance preferences.

- **Output:** formatted text plus a structured block that the UI can render as a list of device options.

5. **Respond to chat**

   - **Input:** formatted message and structured device list.
   - **Process:** send the generated response back through the chat node, preserving the device identifiers so later turns can refer to them.
   - **Output:** final user facing reply for this turn.

Methods and Assumptions

- **Budget enforcement.** Devices are filtered by price at the DW layer when populating `inventory_candidates` and again inside the retrieval agent, which discards any candidate whose price exceeds the per request budget.
- **Auto approval semantics.** The `auto_approve_ok` flag is computed when refreshing candidates from the sponsor view, using department specific tier rules and the per request budget. The `require_auto_approve` flag in the tool arguments switches between Q1 (budget only) and Q2 (budget plus policy) modes.
- **Session isolation.** Every RAG document carries `session_id` in its metadata; `KG Device User` always applies a filter on `session_id` and `kind='candidate'`, so one session cannot see another session's devices or budget context.
- **Ranking.** Final device lists are ordered by `price_nzd` ascending and, within a price band, by performance tier from higher to lower capability; this keeps the cheapest suitable devices first while still favouring more capable models when prices are equal.

Outputs

- Shortlist of budget-eligible IT devices, each with device id, brand, model, key specifications, price in NZD, performance tier, and auto approval status.
- One line budget summary indicating the per request budget and, when applicable, that all listed devices satisfy the auto approval rules for the user's department.
- Logged metrics for later evaluation: DW and KG preparation time, number of candidate devices inserted and embedded for the session, retrieval latency, and final shortlist size per run.

In summary, this chapter defined a controlled and repeatable experimental protocol: a fixed dataset slice, consistent assumptions, ten user cases spanning departments and request types, and three pipeline configurations that differ only in retrieval and execution behaviour. These design choices provide a fair basis for comparing correctness, constraint compliance, and efficiency across approaches. The next chapter reports the quantitative results and interprets what they imply for grounding, policy enforcement, and operational cost in governed procurement workflows.

Chapter 5

Results and Discussion

This chapter reports and interprets the experimental findings for the three research questions. It first presents the quantitative results for the two tasks used to evaluate RQ1 and RQ2, comparing device-pool correctness, Top-1 budget compliance, and policy-consistent auto-approval behaviour across the evaluated pipelines. It then analyses efficiency outcomes for RQ3 using latency and token-usage summaries and per-case deltas to show the operational cost trade-offs of each design. The chapter concludes by discussing what these results imply for deploying NL2SQL and RAG in governed environments, including observed error modes, auditability, and where constraint enforcement is most reliable.

5.1 Results

This chapter reports the outcomes of the three experiment designs introduced in Chapter 4. All experiments answer the same two questions for each user case: (Q1) "Show all devices that fit my department budget for this request" and (Q2) "From those options, which devices are standard choices that procurement policy would normally auto-approve for my role and department?"

The experiments share the same warehouse slice described in Section 4.3. Each session uses the intake flow from Chapter 4 and the same per-session schema with `capacity`, `inventory_candidates`, `dim_department`, `dim_employee`, `fact_budget_quarterly`, and `fact_purchase_request`. The ten user cases (A–J) reuse rows from `dim_employee` and cover IT, Sales, and Front Desk roles across different quarters and replacement/new requests.

What differs across experiments is how the agent accesses and filters the `inventory_candidates` table:

- **Experiment 1 (RAG only).** The model receives the user profile, request details, and high level device information, then calls the vector index directly. It does not issue SQL.
- **Experiment 2 (SQL only).** The model issues a single validated `SELECT` over the per-session warehouse. The query template enforces the budget and joins only allowed tables and columns.
- **Experiment 3 (Hybrid: RAG + KG + DW).** The system builds a per-session knowledge graph and text index over the warehouse slice. Retrieval runs on the session index with payload filters and then maps the results back to the same warehouse tables for display.

Across all three designs the hard budget rule is identical: only devices with `price_nzd` less than or equal to the per-case request budget `request_budget_nzd` are treated as members of the Q1 budget-eligible set. Where SQL is used, ranking follows the shared `ORDER BY` template in Section 4.4. Evaluation metrics reuse the definitions from Chapter 2: expected-set comparison against a deterministic SQL baseline, Top-1 budget compliance, and latency and token statistics.

For Q1 there are 65 budget-eligible devices in total across the ten user cases (ground truth from the baseline SQL query). Experiment 2 (SQL only) and Experiment 3 (Hybrid) both return all 65 devices, with the correct Q1 count for every case (10 of 10 cases match the expected pool size). The RAG-only pipeline in Experiment 1 is more conservative: it returns 49 budget-eligible devices in total, missing between 1 and 4 devices in eight of the ten cases (mean absolute shortfall 1.6 devices per case). It usually stays within budget at the top of the list, but in two cases it leads with a device slightly above the department budget. Top-1 budget compliance is therefore 8 of 10 cases for the RAG-only pipeline and 10 of 10 cases for the SQL and Hybrid pipelines.

Q2 is only evaluated for the SQL and Hybrid pipelines, because the auto approval rules live in the warehouse and cannot be checked reliably from text alone. For each case these two experiments produce a small shortlist of standard devices that are both within budget and marked as auto approvable under the current departmental rules. Detailed Q1 correctness results are reported in Section 5.1.1, and Q2 behaviour is examined in Section 5.1.3. Section 5.2 then compares efficiency in terms of latency and token usage.

5.1.1 Device Suggestion Accuracy

Table 5.1 summarises Q1 correctness across the three experiments. The deterministic SQL baseline expects 65 devices in total across cases A–J. Experiment 1 (RAG only) returns 49 devices, while Experiments 2 and 3 each return all 65.

Table 5.1. *Q1 correctness metrics for all three experiments.*

Metric	**Exp. 1 (RAG)**	**Exp. 2 (SQL)**	**Exp. 3 (Hybrid)**
Exact-match rate (cases)[a]	2/10	10/10	10/10
Mean absolute count error[b]	1.6	0.0	0.0
Over-/under-inclusion rate[c]	80%	0%	0%
Top-1 budget compliance[d]	8/10	10/10	10/10

Note. Q1 asks for all devices that fit the budget for the current request.

[a] Order-agnostic equality of the returned device set with the deterministic SQL baseline.

[b] Average absolute difference in device counts compared with the baseline.

[c] Proportion of cases where the returned set size differs from the expected set.

[d] First device price less than or equal to the per-case request budget `request_budget_nzd`.

The SQL-only and Hybrid pipelines reproduce the expected Q1 device pool for every case. This is consistent with their design: both operate directly over the structured warehouse slice and apply the same budget predicate as the baseline. Any difference in their behaviour comes later, through formatting and policy explanations, not through the devices that are returned.

The RAG-only pipeline is weaker on Q1. It exactly matches the expected device set in only two of the ten cases. In the remaining eight cases the shortlist is a strict subset of the baseline pool, missing between one and four devices per case. The mean absolute error of 1.6 devices shows that vector retrieval over text descriptions tends to drop edge cases near the budget boundary or less frequently mentioned devices, even when the prompt explicitly asks for all options within budget.

Top-1 budget compliance follows a similar pattern. Experiments 2 and 3 always return a first device whose price is within the request budget. Experiment 1 does so in eight of the ten cases. In the two failures (Cases E and H) the first recommendation is a mid-range laptop priced at 1,899 NZD while the department budget for that request is 1,800 NZD. The overshoot is small but it violates the hard constraint that the other pipelines always respect.

Overall, the warehouse-backed designs give exact and fully auditable Q1 device sets, while the RAG-only design behaves more like a heuristic recommender. It often finds sensible devices and usually stays within budget at the top of the list, but it does not reliably enumerate the full budget-eligible pool for the department and request.

Per case request budgets (A–J) are listed in Table 5.2.

Table 5.2. *Per case request budgets and expected in-budget device count (Q1)*

Case	Request budget cap (NZD)	Expected devices
A	$2,640	10
B	$2,400	9
C	$2,640	10
D	$1,800	3
E	$1,800	3
F	$1,980	4
G	$1,800	3
H	$1,800	3
I	$2,640	10
J	$2,640	10

Note. "Request budget cap" is the per case `request_budget_nzd` derived from `fact_budget_quarterly` for that department, role, quarter, and request type. Expected devices = ground truth in-budget device count for each case (A–J), computed by the deterministic SQL baseline (Q1) under that case's budget constraint.

Table 5.2 lists, for each user case A–J, the request budget cap and the expected number of budget-eligible devices used for Q1 evaluation. Budgets range from 1,800 NZD to 2,640 NZD, and the expected set sizes range from 3 to 10 devices. These per case values act as hard budget filters in all experiments and define the expected budget-eligible set against which results are compared.

Table 5.3. *Q1 results for Experiment 1, with all information given to the model and answers generated using RAG only.*

Q1: Candidate devices within request budget limits

Experiment 1: All information - RAG to the model

User A (E003, Carol Wang, DEP_IT, Data Scientist, Senior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: high performance workstation for ML within sensible department budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 7 devices (Success in 8.999s / 7,594 Tokens).

1. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD - $1,899 NZD - Tier: basic
2. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,999 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,799 NZD - Tier: basic
4. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD - $1,699 NZD - Tier: basic
5. D002 - Dell Latitude 5540 - Intel i7-1365U 16 GB RAM 512 GB SSD - $2,199 NZD - Tier: mid
6. D012 - HP EliteBook 840 G10 - Intel i7-1355U 16 GB RAM 512 GB SSD - $2,299 NZD - Tier: mid
7. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U 16 GB RAM 512 GB SSD - $2,599 NZD - Tier: mid (just over budget)

User B (E009, Iris Gu, DEP_IT, Junior Developer, Junior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: new laptop for coding and light debugging on a reasonable budget

– Request budget cap: $2,400

– **Match Expected: 9 devices (≤$2,400)**

Result: 9 devices (Success in 7.057s / 7,506 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,799 NZD - Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 1: All information - RAG to the model

4. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,999 NZD - Tier: basic
6. D002 - Dell Latitude 5540 - Intel i7-1365U 16 GB RAM 512 GB SSD - $2,199 NZD - Tier: mid
7. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P 16 GB RAM 512 GB SSD - $2,099 NZD - Tier: basic
8. D015 - Apple MacBook Air 13 - Apple M2 16 GB RAM 512 GB SSD - $2,299 NZD - Tier: mid
9. D012 - HP EliteBook 840 G10 - Intel i7-1355U 16 GB RAM 512 GB SSD - $2,299 NZD - Tier: mid

User C (E010, Jack Wu, DEP_IT, DevOps Engineer, Mid)
– **Quarter / request type: 2024-Q4 / Replacement**
– Preference summary: powerful laptop for CI/CD and container workloads, ok with higher spend
– Request budget cap: $2,640
– **Match Expected: 10 devices ($\leq$$2,640)**

Result: 7 devices (Success in 6.339s / 7,488 Tokens).

1. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $1,899 NZD - Tier: basic
2. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $1,799 NZD - Tier: basic
3. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $1,699 NZD - Tier: basic
4. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $1,999 NZD - Tier: basic
5. D002 - Dell Latitude 5540 - Intel i7-1365U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $2,199 NZD - Tier: mid
6. D012 - HP EliteBook 840 G10 - Intel i7-1355U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe - $2,299 NZD - Tier: mid
7. D015 - Apple MacBook Air 13 - Apple M2 16 GB RAM 512 GB SSD Integrated - $2,299 NZD - Tier: mid

User D (E005, Emma Liu, DEP_SALES, Sales Manager, Senior)
– **Quarter / request type: 2024-Q3 / New**
– Preference summary: portable, professional looking laptop for presentations and meetings
– Request budget cap: $1,800
– **Match Expected: 3 devices ($\leq$$1,800)**

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 1: All information - RAG to the model

Result: 2 devices (Success in 4.461s / 7,321 Tokens).

1. MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
2. Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic

User E (E008, Harry Zhou, DEP_SALES, Account Executive, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: reliable business laptop for travel and client demos, mid range budget

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 2 devices (Success in 4.595s / 7,328 Tokens).

1. D001 - Dell Latitude 5440 - Intel i5 16 GB RAM 512 GB SSD - $1,899 NZD - Tier: basic
2. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5 16 GB RAM 512 GB SSD - $1,999 NZD - Tier: basic

User F (E004, David Zhang, DEP_SALES, Sales Representative, Junior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: light laptop for travel, email, and slides with moderate budget

– Request budget cap: $1,980

– **Match Expected: 4 devices (≤$1,980)**

Result: 3 devices (Success in 5.009s / 7,431 Tokens).

1. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,799 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,699 NZD - Tier: basic
3. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,899 NZD - Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 1: All information - RAG to the model

User G (E006 Fiona Hu, DEP_FRONT, Front Desk Officer, Junior)
– **Quarter / request type: 2024-Q1 / New**
– Preference summary: basic front desk office laptop for browser, system, and printer
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Result: 2 devices (Success in 5.929s / 7,377 Tokens).

1. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD - $1,699 NZD - Tier: basic
2. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,799 NZD - Tier: basic

User H (E007, George Sun, DEP_FRONT, Front Desk Supervisor, Mid)
– **Quarter / request type: 2024-Q2 / New**
– Preference summary: slightly nicer laptop than basic front desk devices, still office focused
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 6.747s / 7,456 Tokens).

1. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD - $1,899 NZD - Tier: basic
2. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,999 NZD - Tier: basic
3. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD - $1,699 NZD - Tier: basic

User I (E001, Alice Chen, DEP_IT, IT Support, Mid)
– **Quarter / request type: 2024-Q1 / Replacement**
– Preference summary: reliable mid range business laptop, no gaming GPU needed
– Request budget cap: $2,640
– **Match Expected: 10 devices (≤$2,640)**

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 1: All information - RAG to the model

Result: 8 devices (Success in 3.696s/ 7,478 Tokens).

1. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,799 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,699 NZD - Tier: basic
3. D026 - Acer Swift 3 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,599 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $1,999 NZD - Tier: basic
6. D002 - Dell Latitude 5540 - Intel i7-1365U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $2,199 NZD - Tier: mid
7. D012 - HP EliteBook 840 G10 - Intel i7-1355U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $2,299 NZD - Tier: mid
8. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U 16 GB RAM 512 GB SSD Integrated Intel Iris Xe GPU - $2,599 NZD - Tier: mid

User J (E002, Bob Li, DEP_IT, System Administrator, Senior)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: mobile workstation with some GPU performance for virtualisation and admin tools, within sensible budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 6 devices (Success in 3.27s / 7,506 Tokens).

1. D001 - Dell Latitude 5440 - Intel i5-1340P 16 GB RAM 512 GB SSD - $1,899 NZD - Tier: basic
2. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,999 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U 16 GB RAM 512 GB SSD - $1,799 NZD - Tier: basic
4. D023 - MSI Modern 14 - Intel i5-1235U 16 GB RAM 512 GB SSD - $1,699 NZD - Tier: basic
5. D002 - Dell Latitude 5540 - Intel i7-1365U 16 GB RAM 512 GB SSD - $2,199 NZD - Tier: mid
6. D012 - HP EliteBook 840 G10 - Intel i7-1355U 16 GB RAM 512 GB SSD - $2,299 NZD - Tier: mid

Note. Prices are in NZD. "Match" is the expected device count from the deterministic SQL baseline for that case. The request budget cap `request_budget_nzd` is derived from `fact_budget_quarterly` for the corresponding department, role, quarter, and request type.

Table 5.3 reports the Q1 results for Experiment 1 (RAG only). For each user case A–J, the upper block summarises the employee profile (department, role, quarter, and whether the request is a replacement or a new device), the per case request budget cap `request_budget_nzd`, any budget overrun flag for that department, and the ground truth *Match* count (expected in-budget device count) from the deterministic SQL baseline summarised in Table 5.2. Beneath the profile, the *Result* block lists the devices returned by the RAG only pipeline in rank order, with device identifier, brand, model, key specifications (for example CPU, RAM, SSD), and `price_nzd`. These lists show where the text only configuration omits valid budget-eligible devices or, in a few cases, suggests devices that exceed the budget relative to the expected budget-eligible set defined by the SQL baseline.

Table 5.4. *Q1 results for Experiment 2, where the system creates a per session data warehouse for the user and answers using SQL only.*

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

User A (E003, Carol Wang, DEP_IT, Data Scientist, Senior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: high performance workstation for ML within sensible department budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 13.635s / 8,417 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1999 NZD, Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2099 NZD, Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

User B (E009, Iris Gu, DEP_IT, Junior Developer, Junior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: new laptop for coding and light debugging on a reasonable budget

– Request budget cap: $2,400

– **Match Expected: 9 devices (≤$2,400)**

Result: 9 devices (Success in 11.006s / 8,297 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1999 NZD, Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2099 NZD, Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: 2299 NZD, Tier: mid

User C (E010, Jack Wu, DEP_IT, DevOps Engineer, Mid)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: powerful laptop for CI/CD and container workloads, ok with higher spend

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 11.273s / 8,531 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1999 NZD, Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2099 NZD, Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid

User D (E005, Emma Liu, DEP_SALES, Sales Manager, Senior)
– **Quarter / request type: 2024-Q3 / New**
– Preference summary: portable, professional looking laptop for presentations and meetings
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 7.347s / 7,204 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic

User E (E008, Harry Zhou, DEP_SALES, Account Executive, Mid)
– **Quarter / request type: 2024-Q2 / New**
– Preference summary: reliable business laptop for travel and client demos, mid range budget
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 8.644s / 7,190 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic

User F (E004, David Zhang, DEP_SALES, Sales Representative, Junior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: light laptop for travel, email, and slides with moderate budget

– Request budget cap: $1,980

– **Match Expected: 4 devices (≤$1,980)**

Result: 4 devices (Success in 10.067s / 7,479 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic

User G (E006 Fiona Hu, DEP_FRONT, Front Desk Officer, Junior)

– **Quarter / request type: 2024-Q1 / New**

– Preference summary: basic front desk office laptop for browser, system, and printer

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 9.603s / 7,209 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

User H (E007, George Sun, DEP_FRONT, Front Desk Supervisor, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: slightly nicer laptop than basic front desk devices, still office focused

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 6.238s / 7,229 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic

User I (E001, Alice Chen, DEP_IT, IT Support, Mid)

– **Quarter / request type: 2024-Q1 / Replacement**

– Preference summary: reliable mid range business laptop, no gaming GPU needed

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 11.833s / 8,510 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1999 NZD, Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2099 NZD, Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 2: Create DW for the user (SQL only)

10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid

User J (E002, Bob Li, DEP_IT, System Administrator, Senior)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: mobile workstation with some GPU performance for virtualisation and admin tools, within sensible budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 12.263s / 8,564 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1899 NZD, Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1999 NZD, Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2099 NZD, Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid
10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid

Note. Prices are in NZD. "Match" is the expected device count from the deterministic SQL baseline for that case. The request budget cap `request_budget_nzd` is derived from `fact_budget_quarterly` for the corresponding department, role, quarter, and request type.

Table 5.4 reports the same Q1 cases for Experiment 2 (SQL only). The layout matches Table 5.3: for each user case A–J, the upper block records the employee profile (department, role, quarter, and replacement or new), the per case request budget cap `request_budget_nzd`, any budget overrun flag for that department, and the expected *Match* count from the deterministic baseline summarised in Table 5.2. The *Result* block now shows the devices returned by a single read only `SELECT` over the per session warehouse that applies the budget cap as a hard filter and orders the surviving rows by the shared ranking rule. The row counts in these lists match the expected in-budget device count in Table 5.2 for all cases and therefore act as the ground truth for Q1.

Table 5.5. *Q1 results for Experiment 3, where the system creates a per session data warehouse for the user and answers using RAG over a knowledge graph and session text index built from that warehouse.*

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

User A (E003, Carol Wang, DEP_IT, Data Scientist, Senior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: high performance workstation for ML within sensible department budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 9.014s / 4,946 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,999 NZD - Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, 16 GB RAM, 512 GB SSD - $2,099 NZD - Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid

User B (E009, Iris Gu, DEP_IT, Junior Developer, Junior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: new laptop for coding and light debugging on a reasonable budget

– Request budget cap: $2,400

– **Match Expected: 9 devices (≤$2,400)**

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

Result: 9 devices (Success in 11.594s / 4,523 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,999 NZD - Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, 16 GB RAM, 512 GB SSD - $2,099 NZD - Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid

User C (E010, Jack Wu, DEP_IT, DevOps Engineer, Mid)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: powerful laptop for CI/CD and container workloads, ok with higher spend

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 7.29s / 4,905 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,999 NZD - Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, 16 GB RAM, 512 GB SSD - $2,099 NZD - Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid
8. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid

9.D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid

User D (E005, Emma Liu, DEP_SALES, Sales Manager, Senior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: portable, professional looking laptop for presentations and meetings

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 6.218s / 2,594 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic

User E (E008, Harry Zhou, DEP_SALES, Account Executive, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: reliable business laptop for travel and client demos, mid range budget

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 5.95s / 2,608 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

User F (E004, David Zhang, DEP_SALES, Sales Representative, Junior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: light laptop for travel, email, and slides with moderate budget

– Request budget cap: $1,980

– **Match Expected: 4 devices (≤$1,980)**

Result: 4 devices (Success in 6.246s / 2,918 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic

---

User G (E006 Fiona Hu, DEP_FRONT, Front Desk Officer, Junior)

– **Quarter / request type: 2024-Q1 / New**

– Preference summary: basic front desk office laptop for browser, system, and printer

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 4.595s / 2,586 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic

---

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

User H (E007, George Sun, DEP_FRONT, Front Desk Supervisor, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: slightly nicer laptop than basic front desk devices, still office focused

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Result: 3 devices (Success in 6.47s / 2,566 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic

User I (E001, Alice Chen, DEP_IT, IT Support, Mid)

– **Quarter / request type: 2024-Q1 / Replacement**

– Preference summary: reliable mid range business laptop, no gaming GPU needed

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 7.464s / 4,900 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic
5. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,999 NZD - Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, 16 GB RAM, 512 GB SSD - $2,099 NZD - Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid
8. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
9. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid

Continued on next page

Continued from previous page

Q1: Candidate devices within request budget limits

Experiment 3: Create DW for the user (Hybrid)

10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid

User J (E002, Bob Li, DEP_IT, System Administrator, Senior)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: mobile workstation with some GPU performance for virtualisation and admin tools, within sensible budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Result: 10 devices (Success in 9.916s / 4,874 Tokens).

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic
4. D006 - Lenovo ThinkPad T14 Gen 4 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,999 NZD - Tier: basic
5. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic
6. D019 - ASUS Zenbook 14 OLED - Intel i7-1360P, 16 GB RAM, 512 GB SSD - $2,099 NZD - Tier: basic
7. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid
8. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
9. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid
10. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid

Note. Prices are in NZD. "Match" is the expected device count from the deterministic SQL baseline for that case. The request budget cap `request_budget_nzd` is derived from `fact_budget_quarterly` for the corresponding department, role, quarter, and request type.

Table 5.5 presents the Q1 results for Experiment 3 (Hybrid). Each user case A–J uses the same profile block as in Table 5.3 and Table 5.4: the employee profile (department, role, quarter, and whether the request is a replacement or a new device), the per case request budget cap `request_budget_nzd`, any budget overrun flag for that department, and the expected *Match* count from the deterministic SQL baseline summarised in Table 5.2. Beneath the profile, the *Result* block lists the devices returned by the hybrid pipeline in rank order, with device identifier, brand, model, key specifications (for example CPU, RAM, SSD), and `price_nzd`. In this experiment the lists come from the schema-aware hybrid design: the per session warehouse slice is used to build a knowledge graph and a `kg_text` surface, the session index in Qdrant is queried with payload filters based on the budget and policy mode, and the resulting hits are mapped back to the same warehouse tables for display. In our runs the surviving devices matched the SQL-only baseline for all ten cases, consistent with the exact match metrics in Table 5.1, but they are reached through this retrieval-guided hybrid route rather than a direct SQL-only path.

5.1.2 Auto-approval rule enforcement

Q2 asks a policy question: given the budget-eligible device pool for a request, which options are standard choices that procurement policy would normally auto-approve for the employee's department and role. In the warehouse this is represented by a boolean auto-approval flag derived from the departmental rules and the historical budget behaviour in `fact_budget_quarterly`. For evaluation we treat as ground truth a deterministic SQL query over the session slice that selects devices within the Q1 budget and marked `auto_approve_ok = true`.

Only the SQL-backed pipelines are evaluated on Q2. Experiment 1 (RAG only) has no direct access to the warehouse policy flags and cannot reliably enforce auto-approval rules from text alone, so it is scored on Q1 correctness and top-1 budget compliance but not on Q2. Experiments 2 and 3 both run their Q2 logic entirely inside the governed warehouse slice: they start from the Q1 budget-eligible set, join the departmental rules, and keep only devices that are flagged as auto approvable for the current department, role, quarter, and request type.

Table 5.6. *Q2 auto-approval rules: enforcement metrics for all three experiments.*

Metric	Exp. 1 (RAG)	Exp. 2 (SQL)	Exp. 3 (Hybrid)
Cases with Q2 evaluated	0/10	10/10	10/10
Q2 set equals baseline auto-approved pool[a]	–	10/10	10/10
Devices outside budget or not auto approved[b]	–	0%	0%

Note. Q2 asks for standard devices that would normally be auto approved for the employee's department and role.

[a] Order-agnostic equality between the devices returned by the experiment and a deterministic SQL baseline that filters the Q1 budget-eligible device pool on `auto_approve_ok = true`.

[b] Share of Q2 devices that either exceed the per-case `request_budget_nzd` or are not marked as auto approvable in the warehouse.

Across the ten user cases there are 65 budget-eligible devices in total and 8 unique devices that appear in the auto-approval baseline. Experiments 2 and 3 always select a small shortlist of 3 or 4 devices per case from this baseline pool. The Q2 lists from the SQL-only and Hybrid pipelines are identical case by case and are strict subsets of the corresponding Q1 budget-eligible sets, so no device in Q2 ever violates the budget or the auto-approval rules. Any differences between these two experiments appear only in how the results are explained to the user, not in which devices are recommended.

Table 5.7. *Q2 auto-approval outputs for Experiment 2, listing devices within budget and their auto-approval status*

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 2 (Model queries the DW, SQL only)

User A (E003, Carol Wang, DEP_IT, Data Scientist, Senior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: high performance workstation for ML within sensible department budget

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Found 4 devices within the budget.

1. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, Integrated GPU, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid - Auto approve: Yes

User B (E009, Iris Gu, DEP_IT, Junior Developer, Junior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: new laptop for coding and light debugging on a reasonable budget

– Request budget cap: $2,400

– **Match Expected: 9 devices (≤$2,400)**

Found 3 devices within the budget.

1. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, Integrated GPU, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 2 (Model queries the DW, SQL only)

User C (E010, Jack Wu, DEP_IT, DevOps Engineer, Mid)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: powerful laptop for CI/CD and container workloads, ok with higher spend

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

Found 4 devices within the budget.

1. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2199 NZD, Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: $2299 NZD, Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $2599 NZD, Tier: mid - Auto approve: Yes

---

User D (E005, Emma Liu, DEP_SALES, Sales Manager, Senior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: portable, professional looking laptop for presentations and meetings

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Found 3 devices within the budget.

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 2 (Model queries the DW, SQL only)

User E (E008, Harry Zhou, DEP_SALES, Account Executive, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: reliable business laptop for travel and client demos, mid range budget

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

Found 3 devices within the budget.

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1599 NZD, Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1699 NZD, Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: $1799 NZD, Tier: basic - Auto approve: Yes

User F (E004, David Zhang, DEP_SALES, Sales Representative, Junior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: light laptop for travel, email, and slides with moderate budget

– Request budget cap: $1,980

– **Match Expected: 4 devices (≤$1,980)**

Found 4 devices within the budget.

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1599 NZD, Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1699 NZD, Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1799 NZD, Tier: basic - Auto approve: Yes
4. D001 - Dell Latitude 5440 - Intel i5-1340P, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1899 NZD, Tier: basic - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 2 (Model queries the DW, SQL only)

User G (E006 Fiona Hu, DEP_FRONT, Front Desk Officer, Junior)
– **Quarter / request type: 2024-Q1 / New**
– Preference summary: basic front desk office laptop for browser, system, and printer
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Found 3 devices within the budget.

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1599 NZD, Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1699 NZD, Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1799 NZD, Tier: basic - Auto approve: Yes

User H (E007, George Sun, DEP_FRONT, Front Desk Supervisor, Mid)
– **Quarter / request type: 2024-Q2 / New**
– Preference summary: slightly nicer laptop than basic front desk devices, still office focused
– Request budget cap: $1,800
– **Match Expected: 3 devices (≤$1,800)**

Found 3 devices within the budget.

1. D026 - Acer Swift 3 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1599 NZD, Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1699 NZD, Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 1799 NZD, Tier: basic - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 2 (Model queries the DW, SQL only)

User I (E001, Alice Chen, DEP_IT, IT Support, Mid)
– **Quarter / request type: 2024-Q1 / Replacement**
– Preference summary: reliable mid range business laptop, no gaming GPU needed
– Request budget cap: $2,640
– **Match Expected: 10 devices (≤$2,640)**

Found 4 devices within the budget.

1. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2199 NZD, Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2299 NZD, Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: 2299 NZD, Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2599 NZD, Tier: mid - Auto approve: Yes

User J (E002, Bob Li, DEP_IT, System Administrator, Senior)
– **Quarter / request type: 2024-Q4 / Replacement**
– Preference summary: mobile workstation with some GPU performance for virtualisation and admin tools, within sensible budget
– Request budget cap: $2,640
– **Match Expected: 10 devices (≤$2,640)**

Found 4 devices within the budget.

1. D002 - Dell Latitude 5540 - Intel i7-1365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2199 NZD, Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2299 NZD, Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, Integrated, 16 GB RAM, 512 GB SSD, Price: 2299 NZD, Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, Integrated Intel Iris Xe, 16 GB RAM, 512 GB SSD, Price: 2599 NZD, Tier: mid - Auto approve: Yes

Table 5.7 presents the Q2 auto-approval outputs for Experiment 2 (SQL only). For each user case A–J, the table repeats the same profile block as in the Q1 tables: the employee profile (department, role, quarter, and whether the request is a replacement or a new device), the per case request budget cap `request_budget_nzd`, any budget overrun flag for that department, and the expected *Match* count for Q2 from the deterministic SQL baseline. The *Result* block then lists all devices returned by the SQL-only pipeline that satisfy both the budget constraint and the warehouse auto-approval rule for that department and role. In this configuration the `SELECT` query filters on `price_nzd` ≤ `request_budget_nzd` and `auto_approve_ok = true`, and orders the surviving rows using the shared ranking rule. The row counts match the expected auto-approved pools for all ten cases, showing that the SQL-only pipeline can enforce departmental approval policy exactly once the rules are expressed in the warehouse.

Table 5.8. *Q2 auto-approval outputs for Experiment 3, listing devices within budget and their auto-approval status*

Q2 (auto approval): devices within budget; auto-approval status shown **Experiment 3 (Hybrid: RAG + KG + DW)**
User A (E003, Carol Wang, DEP_IT, Data Scientist, Senior) – **Quarter / request type: 2024-Q3 / Replacement** – Preference summary: high performance workstation for ML within sensible department budget – Request budget cap: $2,640 – **Match Expected: 10 devices (≤$2,640)**
1. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid - Auto approve: Yes 2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes 3. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes 4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid - Auto approve: Yes (Note: slightly exceeds budget by $59 NZD but close) All options listed are auto approved and fit within or near the department budget of $2,640 NZD for this request, making them safe standard device choices for your role and department.
User B (E009, Iris Gu, DEP_IT, Junior Developer, Junior) – **Quarter / request type: 2024-Q3 / New** – Preference summary: new laptop for coding and light debugging on a reasonable budget – Request budget cap: $2,400 – **Match Expected: 9 devices (≤$2,400)**
1. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 3 (Hybrid: RAG + KG + DW)

2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes

These devices are auto-approved safe choices that fit within your $2,400 NZD budget and comply with your department's standard policy for your role.

User C (E010, Jack Wu, DEP_IT, DevOps Engineer, Mid)

– **Quarter / request type: 2024-Q4 / Replacement**

– Preference summary: powerful laptop for CI/CD and container workloads, ok with higher spend

– Request budget cap: $2,640

– **Match Expected: 10 devices (≤$2,640)**

1. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid - Auto approve: Yes

All options listed are within the department budget of $2,640 NZD for this replacement request and are safe standard choices that comply with your normal policy for your role and department.

User D (E005, Emma Liu, DEP_SALES, Sales Manager, Senior)

– **Quarter / request type: 2024-Q3 / New**

– Preference summary: portable, professional looking laptop for presentations and meetings

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic - Auto approve: Yes

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 3 (Hybrid: RAG + KG + DW)

All options listed are within the department budget of $1,800 NZD for this request and are marked as safe standard choices with auto approval suitable for your role and department policy.

User E (E008, Harry Zhou, DEP_SALES, Account Executive, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: reliable business laptop for travel and client demos, mid range budget

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic - Auto approve: Yes

All options listed are within the department budget of $1,800 NZD for this request and meet the auto-approval policy for your role and department. These are safe, standard choices suitable for procurement and manager approval.

User F (E004, David Zhang, DEP_SALES, Sales Representative, Junior)

– **Quarter / request type: 2024-Q3 / Replacement**

– Preference summary: light laptop for travel, email, and slides with moderate budget

– Request budget cap: $1,980

– **Match Expected: 4 devices (≤$1,980)**

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic - Auto approve: Yes
4. D001 - Dell Latitude 5440 - Intel i5-1340P, 16 GB RAM, 512 GB SSD - $1,899 NZD - Tier: basic - Auto approve: Yes

All options listed are within the department budget of $1,980 NZD for this request and meet the auto approval policy for your role and department.

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 3 (Hybrid: RAG + KG + DW)

User G (E006 Fiona Hu, DEP_FRONT, Front Desk Officer, Junior)

– **Quarter / request type: 2024-Q1 / New**

– Preference summary: basic front desk office laptop for browser, system, and printer

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic - Auto approve: Yes

All options listed are within the budget of $1,800 NZD and are auto-approvable for your department and role. These are safe, standard choices that procurement and your manager should be comfortable with.

User H (E007, George Sun, DEP_FRONT, Front Desk Supervisor, Mid)

– **Quarter / request type: 2024-Q2 / New**

– Preference summary: slightly nicer laptop than basic front desk devices, still office focused

– Request budget cap: $1,800

– **Match Expected: 3 devices (≤$1,800)**

1. D026 - Acer Swift 3 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,599 NZD - Tier: basic - Auto approve: Yes
2. D023 - MSI Modern 14 - Intel i5-1235U, 16 GB RAM, 512 GB SSD - $1,699 NZD - Tier: basic - Auto approve: Yes
3. D011 - HP ProBook 440 G10 - Intel i5-1335U, 16 GB RAM, 512 GB SSD - $1,799 NZD - Tier: basic - Auto approve: Yes

All options listed are within the department budget of $1,800 NZD for this request and meet the auto approval criteria for your role and department.

Continued on next page

Continued from previous page

Q2 (auto approval): devices within budget; auto-approval status shown

Experiment 3 (Hybrid: RAG + KG + DW)

User I (E001, Alice Chen, DEP_IT, IT Support, Mid)
– **Quarter / request type: 2024-Q1 / Replacement**
– Preference summary: reliable mid range business laptop, no gaming GPU needed
– Request budget cap: $2,640
– **Match Expected: 10 devices (≤$2,640)**

1. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid - Auto approve: Yes
2. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
3. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid - Auto approve: Yes

All listed devices are safe standard choices that fit the department budget of $2,640 NZD for this replacement request and comply with the auto-approval policy for your role and department.

User J (E002, Bob Li, DEP_IT, System Administrator, Senior)
– **Quarter / request type: 2024-Q4 / Replacement**
– Preference summary: mobile workstation with some GPU performance for virtualisation and admin tools, within sensible budget
– Request budget cap: $2,640
– **Match Expected: 10 devices (≤$2,640)**

1. D002 - Dell Latitude 5540 - Intel i7-1365U, 16 GB RAM, 512 GB SSD - $2,199 NZD - Tier: mid - Auto approve: Yes
2. D012 - HP EliteBook 840 G10 - Intel i7-1355U, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
3. D015 - Apple MacBook Air 13 - Apple M2, 16 GB RAM, 512 GB SSD - $2,299 NZD - Tier: mid - Auto approve: Yes
4. D007 - Lenovo ThinkPad X1 Carbon Gen 11 - Intel i7-12899365U, 16 GB RAM, 512 GB SSD - $2,599 NZD - Tier: mid - Auto approve: Yes

All options listed are auto approved and fit within the department budget of $2,640 NZD for this request.

Table 5.8 presents the Q2 auto-approval outputs for Experiment 3 (Hybrid). The same user profiles A–J are used as in the Q1 and Q2 tables for Experiments 1 and 2: each block shows the employee profile (department, role, quarter, and whether the request is a replacement or a new device), the per case request budget cap `request_budget_nzd`, any budget overrun flag for that department, and the expected *Match* count for Q2 from the deterministic SQL baseline in Table 5.2. The *Result* block then lists the devices returned by the hybrid pipeline that satisfy both the budget constraint and the auto-approval rule (`auto_approve_ok = true`) for the current department and role.

In this experiment the shortlist is produced via the hybrid path: the per-session warehouse slice is first converted into a small knowledge graph and a `kg_text` surface, the session vector index in Qdrant is queried with payload filters based on `request_budget_nzd` and policy mode, and the resulting hits are mapped back to the same warehouse tables for display. For all ten cases the Q2 device lists in Experiment 3 match the auto-approved pools from Experiment 2, so no device in this table exceeds the budget or violates the departmental auto-approval rules, even though the results are reached through a retrieval guided hybrid route rather than a direct SQL-only query.

5.1.3 Efficiency Metrics

We report latency and token usage for Experiment 2 (SQL only) and Experiment 3 (Hybrid) on the Q2 auto approval task.[1]Experiment 1 (RAG only) was not instrumented for time or tokens in this study. Each configuration was run once per user case (A–J), so N=10 per experiment. Quartiles use Tukey's hinges (Q1 and Q3 as the medians of the lower and upper halves), and medians follow the N=10 sample definition.

Latency.

Table 5.9. *Latency summary in seconds for Q2, using Tukey five number summaries with N=10 runs per configuration.*

Experiment	Min	Q1	Median	Q3	Max	IQR
Exp. 2 (SQL only)	6.489	7.321	9.452	10.162	12.929	2.841
Exp. 3 (Hybrid)	4.491	5.209	5.770	6.025	6.125	0.816

Notes. Q1 and Q3 are Tukey's hinges; IQR = Q3 − Q1. With N=10, the empirical p95 coincides with the sample maximum.

Table 5.9 summarises end-to-end Q2 latency for the SQL and Hybrid pipelines. The Hybrid design returns answers faster in every user case, with a median of 5.77 s compared with 9.45 s for SQL. Its spread is also much tighter (IQR 0.82 s versus 2.84 s). The five-number summaries are 6.489–7.321–9.452–10.162–12.929 s for SQL and 4.491–5.209–5.770–6.025–6.125 s for Hybrid.

Token usage. Under the same Q2 prompts, Hybrid reduces total tokens (input plus output) by about half compared with SQL (Table 5.10).

Table 5.10. *Token usage summary for Q2 (input plus output tokens) using Tukey five number summaries with N=10 runs per configuration.*

Experiment	Min	Q1	Median	Q3	Max	IQR
Exp. 2 (SQL only)	7,314	7,324	7,457	7,580	7,719	256
Exp. 3 (Hybrid)	2,645	2,688	3,656	4,696	4,802	2,008

Notes. Tokens are input plus output. Q1 and Q3 are Tukey's hinges; IQR = Q3 − Q1.

[1]All latency timings were recorded inside the `n8n` container for consistency across runs.

Table 5.10 shows that Hybrid uses far fewer tokens at every summary statistic. At the median, total tokens fall from 7,457 for SQL to 3,656 for Hybrid, a reduction of 51.0%. The Hybrid IQR is larger (2,008 tokens compared with 256 for SQL), which reflects greater variability in response length, including cases where the Hybrid agent returns shorter structured shortlists versus longer policy explanations (for example, when few or no auto-approvable devices exist for the current role/quarter).

Table 5.11. *Q2 efficiency by user case, comparing Experiment 2 and Experiment 3 in time, tokens, speedup, and token reduction.*

Case	Exp. 2 time (s)	Exp. 2 tokens	Exp. 3 time (s)	Exp. 3 tokens	Speedup	Token Δ (%)
A	10.236	7,719	4.854	4,747	2.11	38.5
B	6.987	7,315	5.818	4,327	1.20	40.8
C	12.929	7,580	5.722	4,696	2.26	38.0
D	9.683	7,324	5.608	2,683	1.73	63.4
E	7.321	7,314	6.025	2,713	1.22	62.9
F	9.221	7,606	6.103	2,985	1.51	60.8
G	10.162	7,326	5.958	2,688	1.71	63.3
H	6.489	7,351	4.491	2,645	1.44	64.0
I	9.762	7,577	5.209	4,802	1.87	36.6
J	8.380	7,562	6.125	4,656	1.37	38.4

Notes. Times and token counts are taken from the Q2 logs for each user case. Speedup $= \frac{\text{Exp. 2 time}}{\text{Exp. 3 time}}$. Token $\Delta(\%) = \frac{\text{Exp. 2 tokens} - \text{Exp. 3 tokens}}{\text{Exp. 2 tokens}} \times 100$.

Table 5.11 breaks efficiency down by case A to J. Hybrid is faster in every case, with a median speedup of about 1.61× (mean 1.64×). Token usage also drops in every case. The median token reduction is about 50.8% (mean 50.7%), with per-case savings ranging from 36.6% (Case I) to 64.0% (Case H). In other words, the Hybrid agent delivers the same auto-approval behaviour as the SQL-only design for Q2, but with consistently lower latency and roughly half the token cost.

Baseline limitations. The comparative results should be interpreted in light of known limitations of the baseline designs. The RAG-only baseline operates without direct execution over the governed warehouse and therefore cannot enforce budget predicates, join constraints, or policy rules at query time. As a result, it may return fluent but incomplete or non-compliant answer sets, and its failures are often silent rather than explicit. In addition, because retrieval and generation are not bound to an executable surface, the RAG-only configuration was not instrumented for reliable latency or token-cost measurement, limiting its suitability for operational efficiency comparison. The SQL-only baseline, by contrast, provides deterministic correctness and policy compliance when queries are well-formed, but depends heavily on accurate schema specification and fixed query templates. It lacks the semantic flexibility to adapt retrieval context dynamically, which can increase prompt size, reduce robustness to variation in user phrasing, and limit opportunities for efficiency gains. These baseline characteristics motivate the hybrid design evaluated in this chapter, which aims to combine the strengths of both approaches while mitigating their respective weaknesses.

5.2 Discussion

Operational interpretation of improvements. From an operational perspective, improvements are considered meaningful when they change system behaviour in ways that reduce risk or cost, rather than merely shifting aggregate metrics. For correctness, a meaningful improvement corresponds to fewer policy violations, fewer mismatches against expected device sets, and a reduced incidence of silent failure modes that require analyst intervention. For latency, improvements must be consistent across user cases and support interactive use, not only lower median values under ideal conditions. For token cost, a meaningful improvement implies a material reduction in per-query usage that enables the system to scale economically under repeated or concurrent requests. In this context, the observed reductions in latency and token usage for the Hybrid pipeline, while preserving exact-set correctness and rule compliance, represent operationally significant gains rather than marginal metric variation.

This section discusses the proposed schema-aware hybrid pipeline in relation to the experimental results presented in Chapter 5. The study evaluated three designs: a RAG only baseline, a SQL only baseline, and a hybrid pipeline that combines semantic retrieval, a per-session knowledge slice, and a constrained warehouse query. These were tested against three research questions focused on accuracy (RQ1), constraint enforcement (RQ2), and efficiency (RQ3).

Across the ten controlled user cases (A–J) in the IT-device procurement domain, only the schema-aware designs, SQL-only (Exp. 2) and Hybrid (Exp. 3), recovered the exact Q1 budget-eligible device pool for every case (10/10 exact-set matches) and maintained perfect Top-1 budget compliance (10/10). The RAG-only baseline (Exp. 1) behaved more conservatively: it matched the expected set in only 2/10 cases, returned 49 of 65 budget-eligible devices in total (mean absolute shortfall 1.6 devices per case), and violated Top-1 budget compliance in two cases (8/10). For Q2, the evaluation focuses on Exp. 2 and Exp. 3 because the auto-approval rules are represented as structured policy fields in the warehouse and cannot be verified reliably from text alone. Under these conditions, both schema-aware pipelines applied the Q2 policy correctly (100% inference with 0% violations in our runs). Within the schema-aware pair, the Hybrid pipeline preserved correctness and policy enforcement while improving efficiency: for Q2 it reduced median latency from 9.45 s to 5.77 s and reduced median token usage from 7,456.5 to 3,656, largely by scoping each session to a narrower slice of schema and policy context and reusing retrieved evidence. The following sections revisit each research question in turn, interpret the key quantitative results, and provide a critical comparison of the three designs, before consolidating the main contributions, limitations, and implications for future work.

5.2.1 Finding 1: Schema-aware retrieval vs. RAG-only/SQL-only

Research Question. Does schema-aware retrieval improve evidence recall and grounding compared with RAG-only or SQL-only approaches in the IT-device procurement task?

Across the ten controlled user cases (A–J), SQL-only (Exp. 2) and Hybrid (Exp. 3) each returned the exact budget-eligible device set in all cases (10/10), matching the deterministic SQL baseline used as ground truth. In contrast, RAG-only (Exp. 1) matched only 2/10 cases and introduced systematic under-inclusion, with a mean absolute shortfall of 1.6 devices and set-size deviations in 80% of runs. Top-1 budget compliance was perfect for Exp. 2 and Exp. 3 (10/10) and lower for Exp. 1 (8/10), where the first-ranked device exceeded the per-case request budget in two cases (Tables 5.1–5.5). These outcomes indicate that exact-set recovery is tightly coupled to execution on the warehouse under explicit schema and budget constraints. The Hybrid design constrains the model to a session-bounded slice of tables, attributes, and budget rules, and then produces outputs grounded in that governed slice. This reduces common prompt-only failure modes such as omitting eligible devices near the budget boundary, returning only a partial pool, or drifting from the defined budget constraint rule. The SQL-only baseline attains the same correctness when the full budget constraint predicate is encoded directly into a single validated `SELECT`. By contrast, the RAG-only pipeline depends on semantic retrieval over device descriptions and therefore behaves more like a heuristic recommender: it can surface sensible options, but it does not reliably enumerate the full budget-eligible pool required by Q1.

Contribution. The experiments show that schema-aware designs (Exp. 2 and Exp. 3) can meet a strict, order-agnostic correctness target in a governed procurement task: 10/10 exact budget-eligible set equality and 10/10 Top-1 budget compliance against a deterministic baseline. This provides a concrete, auditable standard for evaluating agents over budget-scoped catalogue selection, and supports the claim that schema and policy integration, not prompt engineering alone, is necessary for reliable enumeration behaviour.

5.2.2 Finding 2: Constraint-respecting SQL and final answers

Research Question. Does the hybrid pipeline produce a higher rate of executable, constraint-respecting decisions and more accurate final answers than both baselines?

The Q2 experiments tested each method's ability to recognise and enforce the procurement policy requirement for *standard, auto-approvable* devices, given the Q1 budget-eligible device pool for the same department, role, quarter, and request type. Both SQL-only (Exp. 2) and Hybrid (Exp. 3) applied the policy correctly in every case and returned no violations in our runs (0% devices outside budget and 0% devices failing the auto-approval rule; Table 5.6). RAG-only (Exp. 1) was not evaluated for Q2: its Q1 runs already failed to produce reliable budget sets, so using those outputs as the basis for a follow-up query would not give a meaningful or fair comparison. Although Exp. 2 and Exp. 3 reach the same Q2 outcome (Tables 5.7–5.8), they do so through different mechanisms. SQL-only encodes the budget and policy predicates directly in a single read-only `SELECT`, yielding deterministic filtering from the governed slice. The Hybrid pipeline applies defence-in-depth: it scopes retrieval to the session slice and policy context, filters candidates using payload constraints aligned with the Q2 rule, and then re-validates the final shortlist against warehouse-derived attributes before responding. This design makes policy compliance auditable at both the evidence stage (what was retrieved) and the decision stage (what was finally returned), and it remains stable under the multi-turn flow where Q2 is asked after Q1.

Contribution. The results show that schema-aware pipelines can enforce hard organisational rules reliably when those rules are represented as governed warehouse attributes. The Hybrid design provides a reusable pattern for policy enforcement in multi-turn settings: detect the rule, filter at retrieval, re-check against governed fields before output, and treat missing required policy fields conservatively. This pattern generalises to other procurement constraints (for example, warranty requirements, preferred-vendor lists, or role-based device tiers) without relying on ad hoc prompt instructions.

5.2.3 Finding 3: Latency and token-cost envelope

Research Question. What latency and token-cost differences arise when using the hybrid pipeline compared with a baseline NL2SQL approach?

Efficiency was measured across ten user cases (N=10) on the Q2 auto-approval task using end-to-end latency and total token counts (input + output) for SQL-only (Exp. 2) and Hybrid (Exp. 3). The Hybrid method showed consistent performance gains: median latency decreased from 9.452 s (Exp. 2) to 5.770 s (Exp. 3), and the interquartile range tightened from 2.841 s to 0.816 s (Table 5.9). Median token usage dropped from 7,457 to 3,656 (–51.0%) under the same Q2 prompts (Table 5.10). Per-case comparisons (Table 5.11) confirm that Hybrid is faster in every case, with a median speedup of 1.61× (mean 1.64×), and reduces tokens in every case, with a median reduction of 50.8% (mean 50.7%). Experiment 1 (RAG-only) was not included in efficiency analysis because it was not instrumented for consistent timing and token logging in this study. These results show that adding session-scoped retrieval around a constrained warehouse execution path can improve efficiency rather than adding overhead. By narrowing the visible context to a per-session slice and avoiding repeated schema re-description at answer time, the Hybrid pipeline reduces prompt size and model work while preserving the policy-correct behaviour required for Q2.

Contribution. The Hybrid pipeline achieves a better latency and cost profile while maintaining full correctness and policy enforcement for the governed IT procurement task. The use of Tukey five-number summaries for latency and token analysis provides a transparent benchmarking format for small-N evaluations of structured agents. Together, these results demonstrate that grounding the model in session-scoped schema and policy context can reduce both token overhead and response time, which is important for cost-sensitive and interactive deployments.

5.2.4 Other Findings

Beyond the three research questions, the experiments surfaced several cross-cutting observations about how schema-aware agents behave in practice in the IT procurement setting.

Shift in error modes and trust profile

The three configurations differed not only in accuracy but also in how they failed. In this study the RAG-only baseline tended to be *conservative* on Q1: it often returned a plausible shortlist but omitted valid in-budget devices, and in a small number of cases its first recommendation exceeded the request budget. These are governance-relevant failures because they are not always obvious to an end user when the response is fluent and the device descriptions look reasonable. By contrast, the schema-aware methods either returned the exact budget-eligible device pool (Q1) and policy-consistent shortlists (Q2), or failed in more transparent ways when constraints became too restrictive (for example, by returning no auto-approvable options under the current departmental rules and budget). This shift in error mode matters for procurement workflows: it is easier to detect and justify "no result under these rules" than a plausible-looking device list that silently violates budget or policy predicates.

Observability and instrumentation as design constraints

The experiments also highlighted how observability choices constrain what can be evaluated and governed. The SQL-only and Hybrid pipelines naturally expose auditable artefacts: the scoped schema, the executed predicates, the row counts, and the final shortlist derived from governed fields. They also support consistent timing and token logging at the workflow boundary, which enables end-to-end efficiency reporting. In contrast, the RAG-only baseline did not provide a comparable trace for policy decisions such as auto-approval, and it was not included in Q2 and Q3 analysis because the key rule checks live in structured warehouse attributes rather than text. This is a practical finding for enterprise deployment: if an agent is expected to be benchmarked and governed, tracing and evaluation hooks need to be designed into the workflow and the data surface from the outset, not added later. In practice, pipeline design choices determine what can be measured, explained, and audited.

Support for interactive constraint refinement

Another cross-cutting observation concerns multi-turn behaviour. The IT procurement task is naturally staged: users first ask for the full in-budget device pool (Q1), then refine the decision by asking which options are standard and auto-approvable for their role and department (Q2). In the schema-aware pipelines this "refine previous set" interaction is stable because the same session slice, budget cap, and governed policy fields are reused across turns. The Q2 predicate is applied deterministically on top of the Q1 budget-eligibility logic, which preserves consistency and reduces conversational drift. This pattern generalises to other interactive refinements in procurement, such as tightening requirements on RAM/SSD, requesting a discrete GPU for a role, or prioritising replacement requests during budget-overrun quarters, while still expecting the same underlying decision logic to hold across turns. A text-only configuration would need to reconstruct the effective policy state from prompts and retrieved snippets each time, increasing the risk of inconsistency.

Reusable pattern for governable deployments

Finally, the experiments illustrate that the proposed hybrid design functions as a reusable pattern for governable deployments, not just a one-off prototype. Binding constraints into the warehouse slice (through explicit predicates over sponsored, operationally aligned tables) and aligning retrieval to that same governed surface (through session-scoped indexing and filterable metadata) yields a pipeline where the same rules shape retrieval, decision, and final answers. This treats schema, policy encoding, and retrieval scope as a single governance surface. While the present study uses a small controlled case set, it demonstrates that the pattern can be implemented with current orchestration and storage tooling and can transfer to other organisational decision settings where rule traceability is as important as accuracy.

Together, these observations extend the main findings by showing how schema-awareness reshapes failure modes, observability, interaction patterns, and governance, beyond the specific metrics used to answer RQ1-RQ3.

5.2.5 Limitations

Ground truth and case coverage

The RQ1 analysis anchors on a single ground-truth construction: a deterministic SQL baseline over a fixed sponsor-provided device catalogue and $N = 10$ controlled user cases. We evaluate strict set equality for the Q1 budget-eligible device pool and budget compliance for the first ranked suggestion, but we do not assess alternative "near-correct" rankings, user-perceived usefulness, or satisfaction with the returned device mix. The case set covers three departments and multiple quarters, but it remains small and intentionally structured. Broader retrieval and ranking metrics (for example, Recall@k or nDCG for recommendation-style outputs) and a larger, more diverse set of employee profiles, preferences, and procurement contexts are needed before generalising the accuracy findings beyond this sponsor-aligned scenario.

Constraint scope and policy saturation

The Q2 analysis focuses on one policy dimension: whether a device is a standard, auto-approvable choice for the requesting role and department under the current budget and overrun conditions. This captures a realistic governance decision point, but it does not cover the full policy space that an operational procurement system may require (for example, mandatory security baselines, approved vendors, lifecycle constraints, exceptions for specialised roles, or fleet standardisation targets). In addition, the auto-approval outcome may be driven by the sponsor's rule thresholds and the available device tiers in the catalogue, so some observed behaviours can reflect policy design and catalogue composition rather than method superiority. Future studies should test additional rule types and conflicting constraints (for example, "GPU required" with a tight budget, or replacement requests during a flagged overrun quarter) to better assess how the enforcement patterns generalise.

Local efficiency measurements and scaling

Efficiency measurements for RQ3 were taken from a local execution environment using API-hosted models and workflow-level timers. Latency and token behaviour may differ under concurrent load, different model deployments, network conditions, or alternative orchestration layouts. With $N = 10$, tail estimates such as p95 or p99 coincide with the sample maximum, so additional runs are required for robust scaling analysis. Future studies should also report cost-per-query and, where feasible, system-level resource metrics (for example, CPU time per workflow step and database query time) to characterise operational efficiency under production-like conditions.

5.2.6 Summary of Discussion

This study addressed three research questions on accuracy (RQ1), constraint enforcement (RQ2), and efficiency (RQ3) using a controlled, reproducible setup aligned to an IT procurement domain. The main contributions are:

- **Schema-aware hybrid pipeline.** The thesis implements and evaluates a Hybrid design that limits context to a session-bounded slice of a sponsor-aligned operational warehouse, applies semantic retrieval with filterable metadata, and uses governed warehouse fields to produce auditable device recommendations. In the IT procurement task, this pipeline supports exact Q1 budget filtering and policy-consistent Q2 shortlists under department and role rules.
- **Deterministic evaluation protocol.** The work defines a clear evaluation frame for structured NL2SQL-style agents in procurement: order-agnostic set equality and Top-1 budget compliance for Q1; rule-consistent auto-approval behaviour for Q2; and Tukey five-number summaries plus per-case deltas for latency and tokens for Q3. These measures make correctness and efficiency transparent and repeatable.
- **Defence-in-depth policy enforcement.** The study demonstrates a reusable pattern for hard predicates in governed decision flows: detect the rule from the request context, apply it at the earliest feasible stage (candidate slicing and retrieval), and re-check

it when forming the final output. In this dataset, the schema-aware pipelines avoid returning devices that exceed the request budget and support auditable application of the auto-approval rule using warehouse attributes rather than prompt-only heuristics.

- **Efficiency at parity of correctness.** Under identical prompts and ground-truth targets, the Hybrid pipeline reduces latency and token usage relative to the SQL-only baseline while preserving exact-set correctness for Q1 and consistent rule behaviour for Q2. This shows that schema scoping and retrieval-guided prompting can improve the cost–latency profile rather than adding overhead.

- **Reproducible case design and artefacts.** Ten clearly specified user cases (A–J), grounded in sponsor-provided employee and budget context, together with per-run logs and deterministic baselines, provide a concrete template that other practitioners can replicate or extend for similar governed procurement domains.

Taken together, these contributions offer a practical pattern for deploying NL2SQL+RAG systems in governed organisational settings where budgets, approval rules, and auditability matter. The design favours strict schema integration and conservative rule application to avoid incorrect inclusions, while acknowledging that catalogue coverage and policy configuration can shape observed outcomes. These trade-offs are carried forward into the recommendations for future work.

In summary, the results show that reliable procurement decisions require binding the model to governed schema and policy surfaces rather than relying on text-only retrieval or prompt-only constraint enforcement. The schema-aware designs (Experiments 2 and 3) achieve exact in-budget device-set recovery and consistent rule enforcement, and the Hybrid configuration improves efficiency compared with the SQL-only pipeline while preserving the same policy-compliant decision behaviour on Q2. These findings set up the concluding chapter, which consolidates the thesis contributions, reflects on limitations, and outlines practical directions for future work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This study designed, implemented, and evaluated a schema-aware hybrid pipeline for NL2SQL and retrieval-augmented generation over a governed IT device procurement warehouse. The pipeline constrains each interaction to a session-bounded slice of the warehouse, ties candidate retrieval to budget and policy metadata, and produces answers that are traceable to structured evidence. When query execution is required, the agent is limited to a single validated, read-only `SELECT`, preserving warehouse governance and preventing uncontrolled data access.

6.1.1 Thesis contributions

This thesis makes five main contributions to deploying NL2SQL and RAG in governed procurement settings. First, it implements and evaluates a schema-aware Hybrid design that limits context to a session-bounded slice of a sponsor-aligned operational warehouse, uses semantic retrieval with filterable metadata, and relies on governed warehouse fields to produce auditable device recommendations. Second, it defines a deterministic evaluation protocol for structured agents in procurement, using order-agnostic set equality and Top-1 budget compliance for Q1, rule-consistent auto-approval behaviour for Q2, and Tukey five-number summaries with per-case deltas for latency and token usage for Q3. Third, it demonstrates a defence-in-depth enforcement pattern in which hard predicates are derived from request context, applied at the earliest feasible stages (candidate slicing and retrieval), and verified again at output time, reducing leakage compared with prompt-only safeguards. Fourth, it shows that retrieval-guided prompting around a constrained execution path can improve efficiency: under identical prompts and targets, the Hybrid pipeline reduces latency and token usage relative to the SQL-only baseline while preserving exact-set correctness for Q1 and consistent rule behaviour for Q2. Finally, it provides a reproducible case design and artefacts, including ten specified user cases (A–J) grounded in sponsor-provided employee and budget context, together with per-run logs and deterministic baselines, forming a template that others can replicate or extend for similar governed procurement domains.

The experiments were designed to test whether the pipeline can deliver grounded, policy-respecting, and efficient answers in a realistic budget-governed procurement scenario. The results answer the three research questions as follows.

RQ1 (retrieval and grounding). Reliability depends on grounding against the governed warehouse definition of budget eligibility rather than relying on unstructured retrieval alone. Across user cases A–J, both the SQL-only pipeline (Experiment 2) and the Hybrid pipeline (Experiment 3) recovered the full expected budget-eligible device pool (65 devices in total across the ten cases) and matched the per-case pool sizes for all cases. In contrast, the RAG-only pipeline (Experiment 1) behaved conservatively, returning a strict subset of the expected pool (49 devices total), matching the full set in only two cases and missing between one and four devices in most cases. Top-1 budget compliance was perfect for Experiments 2 and 3, while Experiment 1 produced Top-1 budget compliance violations in two cases. Overall, strict set recovery for Q1 was achieved only when the pipeline was constrained by schema-aware execution over the warehouse slice.

RQ2 (constraint-respecting answers under policy). The schema-aware designs enforced procurement policy consistently when the policy is represented as structured rules in the warehouse. Q2 asks for standard devices that procurement policy would normally auto-approve for the role and department, under the same per-case request budget constraint used in Q1. Because these rules depend on governed attributes (for example, tiering and auto-approvable flags), Q2 was evaluated only for the SQL-only and Hybrid pipelines. In all ten cases, both pipelines applied the policy correctly and returned only devices that were within budget and marked as auto-approvable under the current departmental rules. This shows that, in the procurement setting, rule-respecting answers are obtained by binding the model to governed rule surfaces rather than inferring approval policy from free text.

RQ3 (latency and token cost). Adding a lean retrieval layer around a schema-scoped execution path improves efficiency rather than degrading it on the Q2 auto-approval task. Under matched Q2 prompts, the Hybrid pipeline reduced median latency from 9.452 s to 5.770 s and cut median token usage from 7,457 to 3,656 (a 51.0% reduction). It was faster in every user case, with a median speedup of 1.61×, and reduced tokens in every case, with a median reduction of 50.8%, while preserving the same policy-compliant Q2 behaviour as the SQL-only design.

6.1.2 Other findings and limitations

Beyond the three research questions, the experiments also revealed broader patterns in how the pipelines behave and where their reliability is strongest.

Error modes and trust profile. The RAG-only baseline often failed silently, returning fluent but incomplete budget-eligible device pools and occasionally placing an over-budget device first. In contrast, the schema-aware SQL-only and Hybrid pipelines either returned the correct set or failed in a more transparent way (for example, by yielding an empty result when no devices satisfied the encoded budget and policy predicates). This shift from plausible-looking errors to explicit "no result under these rules" outcomes matters for governance because it makes mistakes easier to detect and explain.

Observability and instrumentation. Observability and instrumentation emerged as design constraints rather than afterthoughts. The RAG-only configuration was not carried forward into efficiency analysis because its logging and enforcement surface was not comparable to the schema-aware pipelines. By contrast, the SQL-only and Hybrid pipelines were instrumented for end-to-end timing and token counts, enabling a consistent efficiency comparison. This reinforces that pipeline architecture and logging strategy determine what can be measured and audited, not just the choice of model.

Stable interactive refinement. The schema-aware designs supported stable interactive refinement across turns. The workflow naturally supports a "refine the Q1 pool into a Q2 shortlist" interaction, where the same session slice and budget definition are reused and the approval rule is applied deterministically on top. This behaviour is well suited to procurement workflows, where users expect consistent decision logic across a multi-turn request.

Reusable pattern for governable deployments. The Hybrid configuration illustrates a reusable pattern for governable deployments beyond this specific inventory. Encoding constraints in the warehouse representation and applying them at the retrieval boundary yields a pipeline where the same rules shape retrieval, ranking, and final answers. This aligns with a defence-in-depth posture: treat schema, rules, and retrieval policy as a single governance surface rather than separate concerns.

These findings sit within clear limits, summarised in Section 5.2.5. Empirically, the study is based on a single inventory snapshot with $N = 10$ user cases and focuses on strict set correctness (Q1) and rule compliance (Q2) under the current warehouse policy fields. Efficiency results were collected in a local, low-contention environment, so absolute latencies and costs may differ at scale. The pipeline adopts a conservative stance on missing or ambiguous fields by treating them as non-qualifying for policy-critical outputs, which supports governance but can reduce recall for incomplete records. Together, these boundaries define the conditions under which the proposed pipeline is most dependable and motivate broader evaluations and robustness tests outlined in Section 6.2.

Collectively, these findings show that the proposed schema-aware hybrid pipeline answers the thesis questions in the IT procurement domain: it grounds answers in governed warehouse data, respects departmental budget and approval policy, and does so with improved operational efficiency.

6.1.3 Scope and Generalizability

This thesis evaluates a schema-aware hybrid RAG-NL2SQL pipeline using a single enterprise IT procurement case study. The findings therefore support conclusions about the proposed design under the constraints, data characteristics, and governance practices represented in this scenario. To clarify what is likely to generalise beyond the case study, the conclusions are separated into transferable and context-dependent elements.

Transferable conclusions. Several findings are likely to transfer to other governed enterprise querying settings where a structured warehouse schema and policy constraints define valid queries and answers. First, session-scoped retrieval of schema and policy context can reduce irrelevant context and align generation with the active query surface. Second, grounding query construction to explicit schema elements and policy text supports auditability and reduces the risk of hallucinated tables, columns, or unsupported conditions. Third, applying constraint checks before execution (for example, validating join paths, permitted attributes, and rule compliance) provides a practical safeguard for read-only querying in controlled environments. Finally, combining retrieval with constrained SQL generation can improve reliability while managing latency and token usage by limiting the amount of context the model must process.

Context-dependent conclusions. Other findings depend on the specific procurement dataset and governance configuration used in this study. Results related to budget caps, auto-approval rules, catalogue attributes (e.g., device type, price bands, stock status), and department-specific policies are tied to this schema and may not directly transfer to domains with different constraint structures. Observed error patterns and performance trade-offs are also influenced by catalogue size and quality, schema design choices (e.g., degree of normalisation), and the distribution of question types in the evaluation set. If operational targets or thresholds are used, these reflect local requirements and infrastructure assumptions and should be re-calibrated for other deployments.

Overall, the study provides evidence that schema-aware grounding and governance-aligned safeguards can improve NL2SQL/RAG behaviour in a controlled enterprise setting. Applying the approach to other domains should preserve the design principles above, while re-mapping domain rules, schema constraints, and evaluation criteria to the target context.

6.2 Future Work

Building on these contributions and within the boundaries identified above, several directions emerge for extending and hardening the proposed pipeline:

1. **Larger and richer evaluations.** Future work should go beyond a single-snapshot, ten-case setting by including multiple departments, time-varying device catalogues, and a wider range of procurement constraints (for example, minimum RAM, storage, OS requirements, warranty terms, vendor restrictions, and security compliance). Incorporating ranking metrics (e.g., Recall@k, nDCG) and user-centred measures would make it possible to assess not only exact-set correctness, but also the quality and usefulness of near-correct recommendations.

2. **Operational benchmarking.** The current evaluation uses a controlled environment with limited contention. A natural next step is to test the pipeline under production-like conditions with concurrent queries, caching, rate limits, and failover. Measuring throughput, p95/p99 latency, cost-per-query, and error recovery behaviour would provide a clearer view of how the hybrid design behaves at scale.

3. **Richer rule sources and policy versioning.** The approval decision surface can be expanded beyond a single warehouse flag by ingesting rules from auditable sources such as policy documents, maintained rule repositories, or IT standards catalogues. Future work should explore policy versioning, change detection, and conflict handling so that rule updates remain transparent and governance-grade over time.

4. **Session continuity and multi-turn procurement flows.** While this study demonstrates session-consistent follow-ups for short interactions, real procurement often involves longer conversations with changing constraints, substitutions, and approvals. Systematic stress tests over multi-turn dialogues would verify that enforcement remains stable, earlier guarantees are not silently broken, and each answer can be traced back to its evidence and (when used) executed SQL.

5. **Multilingual and heterogeneous inputs.** In many organisations, users query in different languages and policies are stored in heterogeneous formats (tickets, PDFs, wikis). Extending the pipeline to support multilingual inputs and mixed-format

policy evidence while preserving schema alignment and rule enforcement is an important next step, including quantifying how extraction errors propagate into retrieval and selection.

6. **Confidence gating and safe fallbacks.** Future work should include explicit confidence signals, such as coverage checks over the expected candidate pool, validator scores, and rule-application traces, to determine when the system should proceed with the hybrid path and when it should fall back to deterministic SQL templates, simplified flows, or human review. This would help control rare but high-impact failures in ambiguous or low-evidence requests.

7. **Evidence trace and reviewer tooling.** Finally, there is scope to deepen the evidence trail that underpins each answer. Structured logging of retrieved items, interpreted rules, and final outputs (with timestamps and signatures) would support audit, debugging, and compliance review. Designing these traces and associated reviewer tools is key to turning the proposed pipeline into a practical component of governed procurement analytics platforms.

These future directions are aimed at maturing the schema-aware hybrid pipeline from a validated prototype into a more flexible, robust, and governable pattern for AI-assisted querying over structured procurement data. By broadening the evaluation scope, tightening operational guarantees, and strengthening rule and evidence management, this line of work can support systems that remain reliable in governed enterprise environments and transferable to other regulated domains where correctness, transparency, and trust are essential.

References

Asaad, C. (2023). Towards leveraging artificial intelligence for nosql data modeling, querying and quality characterization. In *2023 acm/ieee international conference on model driven engineering languages and systems companion (models-c)* (p. 192-198). doi: 10.1109/MODELS-C59198.2023.00047

Attawar, A., Vora, S., Narechania, P., Sawant, V., & Vora, H. (2023). Nlsql: Generating and executing sql queries via natural language using large language models. In *2023 international conference on advanced computing technologies and applications (icacta)* (p. 1-6). doi: 10.1109/ICACTA58201.2023.10392861

Bahaa, A., Eldemerdash, S. R., & Fahmy, H. (2021, Nov). A systematic literature review for implementing data ops in the data warehouse lifecycle during the etl phase. *Journal of Computer Science*, *17*(11), 1011-1030. doi: 10.3844/jcssp.2021.1011.1030

Bambhaniya, A., Wu, H., Subramanian, S., Srinivasan, S., Kundu, S., Yazdanbakhsh, A., ... Krishna, T. (2025). *Understanding and optimizing multi-stage ai inference pipelines.* doi: 10.48550/arXiv.2504.09775

Bappy, M. A. H., Mustafa, H. A., Saha, P., & Salehat, R. (2025). *Case study: Fine-tuning small language models for accurate and private cwe detection in python code.* doi: 10.48550/arXiv.2504.16584

Bouaziz, S., Nabli, A., & Gargouri, F. (2019). Design a data warehouse schema from document-oriented database. *Procedia Computer Science*, *159*, 221-230. (Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 23rd International Conference KES2019) doi: 10.1016/j.procs.2019.09.177

Chen, L., Zaharia, M., & Zou, J. (2023). *Frugalgpt: How to use large language models while reducing cost and improving performance.* doi: 10.48550/arXiv.2305.05176

Chen, Y., Sun, P., Li, X., & Chu, X. (2025, 04). *Mrd-rag: Enhancing medical diagnosis* *with multi-round retrieval-augmented generation.* doi: 10.48550/arXiv.2504.07724

Deeksha, D., Krishnan, A., & Nambiar, M. (2025). Emu-llm: Emulators for performance evaluation of llm-based applications. In *Companion of the 16th acm/spec international conference on performance engineering* (p. 130–135). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3680256.3721312

Dehdouh, K., Boussaid, O., & Bentayeb, F. (2020). Big data warehouse: Building columnar NoSQL OLAP cubes. *International Journal of Decision Support System Technology (IJDSST)*, *12*(1), 1–24. doi: 10.4018/IJDSST.2020010101

Deng, X., Awadallah, A. H., Meek, C., Polozov, O., Sun, H., & Richardson, M. (2021, June). Structure-grounded pretraining for text-to-SQL. In K. Toutanova et al. (Eds.), *Proceedings of the 2021 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 1337–1350). Online: Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.105

Dhaouadi, A., Bousselmi, K., Gammoudi, M. M., Monnet, S., & Hammoudi, S. (2022). Data warehousing process modeling from classical approaches to new trends: Main features and comparisons. *Data*, *7*(8). doi: 10.3390/data7080113

Ekeh, A. H., Apeh, C. E., Odionu, C. S., & Austin-Gabriel, B. (2025, February). Advanced data warehousing and predictive analytics for economic insights: A holistic framework for stock market trends and GDP analysis. *International Journal of Engineering Research and Development*, *21*(2), 175–190.

Fan, Y., Ren, T., Huang, C., He, Z., & Wang, X. (2024, 11). *Grounding natural language to sql translation with data-based self-explanations.* doi: 10.48550/arXiv.2411.02948

Filice, S., Horowitz, G., Carmel, D., Karnin, Z., Lewin-Eytan, L., & Maarek, Y. (2025). *Generating diverse q&a benchmarks for rag evaluation with datamorgana.* doi: 10.48550/arXiv.2501.12789

Freitas, N., Rocha, A. D., & Barata, J. (2025, February). Data management in industry: concepts, systematic review and future directions. *Journal of Intelligent Manufacturing*, 1-29. doi: 10.1007/s10845-025-02570-z

Gadde, H. (2020). Ai-enhanced data warehousing: Optimizing etl processes for real-time analytics. *Revista de Inteligencia Artificial en Medicina*, *11*(1), 300–318.

Gadde, H. (2022). Integrating ai into sql query processing: Challenges and opportunities. *International Journal of Advanced Engineering Technologies and Innovations*, *1*(3), 194–219.

Gadde, H. (2023). *Leveraging ai for scalable query processing in big data environments* (Vol. 1) (No. 2).

Gadde, H., & Raza, F. (2024, 11). *Ai-augmented database management systems for real-time data analytics.*

Galitsky, B., Ilvovsky, D., & Morkovkin, A. (2025). Enhancing rag and knowledge graphs with discourse. In *Computational linguistics and intellectual technologies: Proceedings of the international conference "dialogue 2025"* (Vol. 23).

Hule, K., & Ranawat, R. (2023, Oct.). Analysis of different orm tools for data access object tier generation: A brief study. *International Journal of Membrane Science and Technology*, *10*(1), 1277–1291. doi: 10.15379/ijmst.v10i1.2842

Inmon, W. H. (2005). *Building the data warehouse* (4th ed.). John Wiley & Sons.

James, F. (2024). *The integration of conversational AI with data lakes and warehouses.* ResearchGate preprint. (Article published October 2024)

Jeong, C. (2023). A study on the implementation of generative ai services using an enterprise data-based llm application architecture. *Advances in Artificial Intelligence and Machine Learning*, *03*(04), 1588–1618. doi: 10.54364/aaiml.2023.1191

Johnson, J., Douze, M., & Jégou, H. (2017). *Billion-scale similarity search with gpus.* arXiv preprint arXiv:1702.08734. (FAISS library; widely used for ANN indexing and vector search) doi: 10.48550/arXiv.1702.08734

Kaldeich, C., & Sá, J. O. e. (2004). Data warehouse methodology: A process driven approach. In A. Persson & J. Stirna (Eds.), *Advanced information systems engineering* (pp. 536–549). Berlin, Heidelberg: Springer Berlin Heidelberg.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., ... Yih, W.-t. (2020, November). Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 conference on empirical methods in natural language processing (emnlp)* (pp. 6769–6781). Online: Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550

Kim, J., & Mahajan, D. (2025). *Vectorliterag: Latency-aware and fine-grained resource partitioning for efficient rag.* doi: 10.48550/arXiv.2504.08930

Kimball, R., & Ross, M. (2013). *The data warehouse toolkit: The definitive guide to dimensional modeling* (3rd ed.). John Wiley & Sons.

Li, G., Zhou, X., & Cao, L. (2021, July). Machine learning for databases. *Proc. VLDB Endow.*, *14*(12), 3190–3193. doi: 10.14778/3476311.3476405

Liakopoulos, D., Hu, T., Sinha, P., & Yadwadkar, N. J. (2025). *iserve: An intent-based serving system for llms.* doi: 10.48550/arXiv.2501.13111

Liu, Y., Huang, L., Li, S., Chen, S., Zhou, H., Meng, F., ... Sun, X. (2023). *Recall: A benchmark for llms robustness against external counterfactual knowledge.*

Liu, Y., Pena, E. H. M., Santos, A., Wu, E., & Freire, J. (2025). Magneto: Combining small and large language models for schema matching. *Proceedings of the VLDB Endowment*, *18*(8), 2681–2694. doi: 10.14778/3742728.3742757

Lyu, Y., Li, Z., Niu, S., Xiong, F., Tang, B., Wang, W., ... Chen, E. (2024). *Crud-rag: A comprehensive chinese benchmark for retrieval-augmented generation of large language models.*

Ma, L., Pu, K., & Zhu, Y. (2024). *Evaluating llms for text-to-sql generation with complex sql workload.* doi: 10.48550/arXiv.2407.19517

Machireddy, J. R. (2022, november). Scalable machine learning workflows in data warehousing: Automating model training and deployment with ai. *Australian Journal of Machine Learning Research & Applications*, *2*(2).

Machireddy, J. R. (2023, January). Enhancing predictive analytics with ai-powered rpa in cloud data warehousing: A comparative study of traditional and modern approaches. *Journal of Deep Learning in Genomic Data Analysis*, *3*(1), 74–100.

Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*(4), 824-836. doi: 10.1109/TPAMI.2018.2889473

Mandal, K. P., Mukheriee, P., & Chakraborty, B. (2022). Natural language query in bengali to sql generation using named entity recognition. In *2022 ieee conference on interdisciplinary approaches in technology and management for social innovation (iatmsi)* (p. 1-6). doi: 10.1109/IATMSI56455.2022.10119243

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval.* Cambridge: Cambridge University Press. (See Chapter 6: Scoring, term weighting and the vector space model)

Mengmeng, S., Zhibin, L., Qingwei, W., Man, H., & Feiyang, X. (2024). An effective retrieval method to improve rag performance. In *2024 7th international conference on data science and information technology (dsit)* (p. 1-5). doi: 10.1109/DSIT61374.2024.10881380

Naderi, A. (2024). Data warehouse design and implementation according to data vault 2.0 methodology [Bachelor's thesis]. Jyväskylä, Finland.

Ngo, V. M., Le-Khac, N.-A., & Kechadi, M.-T. (2019). Designing and implementing data warehouse for agricultural big data. In K. Chen, S. Seshadri, & L.-J. Zhang (Eds.), *Big data – bigdata 2019* (pp. 1–17). Cham: Springer International Publishing. doi: 10.1007/978-3-030-23551-2_1

Noor, U., & Abbas, G. (2024, 09). *Explainable ai meets distributed databases: Leveraging reinforcement learning and generative models for advanced predictive scaling and query optimization.* doi: 10.13140/RG.2.2.10076.40323

Olujimi, P. A., & Ade-Ibijola, A. (2023). Nlp techniques for automating responses to customer queries: A systematic review. *Discover Artificial Intelligence*, *3*, 20. doi: 10.1007/s44163-023-00065-5

Pan, Z., Pan, G., & Monti, A. (2022). Semantic-similarity-based schema matching for management of building energy data. *Energies*, *15*(23), 8894. doi: 10.3390/en15238894

Panwar, V. (2024, March). Ai-driven query optimization: Revolutionizing database performance and efficiency. *International Journal of Computer Trends and Technology (IJCTT)*, *72*(3), 18–26. doi: 10.14445/22312803/IJCTT-V72I3P103

Pasimeni, F. (2019). Sql query to increase data accuracy and completeness in patstat. *World Patent Information*, *57*, 1-7. doi: 10.1016/j.wpi.2019.02.001

Paul, D., Cao, J., Li, F., & Srikumar, V. (2021, December). Database workload characterization with query plan encoders. *Proc. VLDB Endow.*, *15*(4), 923–935. doi: 10.14778/3503585.3503600

Peukert, E., Maßmann, S., & König, K. (2010, September). Comparing similarity combination methods for schema matching. In *Gi-workshop: Informationsintegration in service-architekturen (iisa)* (p. 692-701).

Qu, G., Li, J., Li, B., Qin, B., Huo, N., Ma, C., & Cheng, R. (2024, August). Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-SQL generation. In L.-W. Ku, A. Martins, & V. Srikumar (Eds.), *Findings* *of the association for computational linguistics: Acl 2024* (pp. 5456–5471). Bangkok, Thailand: Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.324

Quinn, D., Nouri, M., Patel, N., Salihu, J., Salemi, A., Lee, S., ... Alian, M. (2025). Accelerating retrieval-augmented generation. In *Proceedings of the 30th acm international conference on architectural support for programming languages and operating systems, volume 1* (p. 15–32). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3669940.3707264

Rachakatla, S. K., Ravichandran, P., & Machireddy, J. R. (2021). The role of machine learning in data warehousing: Enhancing data integration and query optimization. *Journal of Bioinformatics and Artificial Intelligence*, *1*(1), 74–102.

Reimers, N., & Gurevych, I. (2019, November). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In K. Inui, J. Jiang, V. Ng, & X. Wan (Eds.), *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (emnlp-ijcnlp)* (pp. 3982–3992). Hong Kong, China: Association for Computational Linguistics. doi: 10.18653/v1/D19-1410

Ren, H., Galkin, M., Cochez, M., Zhu, Z., & Leskovec, J. (2023). *Neural graph reasoning: Complex logical query answering meets graph databases.* doi: 10.48550/arXiv.2303.14617

Rezaei, M. R., & Dieng, A. B. (2025). *Vendi-rag: Adaptively trading-off diversity and quality significantly improves retrieval augmented generation with llms.* doi: 10.48550/arXiv.2502.11228

Rodier, S., & Carter, D. (2020, May). Online near-duplicate detection of news articles. In N. Calzolari et al. (Eds.), *Proceedings of the twelfth language resources and evaluation conference* (pp. 1242–1249). Marseille, France: European Language Resources Association.

Rogers, W. P., Kahraman, M. M., & Dessureault, S. (2017, November). Exploring the value of using data: a case study of continuous improvement through data warehousing. *International Journal of Mining, Reclamation and Environment*, *33*(4), 286–296. doi: 10.1080/17480930.2017.1405473

Russell-Rose, T., & Shokraneh, F. (2020). Designing the structured search experience:

Rethinking the query-builder paradigm. *Weave: Journal of Library User Experience*, *3*(1). doi: 10.3998/weave.12535642.0003.102

Sahara, C. R., & Aamer, A. M. (2021). Real-time data integration of an internet-of-things-based smart warehouse: a case study. *International Journal of Pervasive Computing and Communications*, *18*(5), 622-644. doi: 10.1108/IJPCC-08-2020-0113

Sauchuk, A., Thorne, J., Halevy, A., Tonellotto, N., & Silvestri, F. (2022). On the role of relevance in natural language processing tasks. In *Proceedings of the 45th international acm sigir conference on research and development in information retrieval* (p. 1785–1789). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3477495.3532034

Seethala, S. C. (2020, December). Ai-enabled data pipelines: Modernizing data warehouses in healthcare for real-time analytics. *International Research Journal of Innovations in Engineering and Technology*, *4*(12), 43–45. doi: 10.47001/IRJIET/2020.412007

Shen, M., Umar, M., Maeng, K., Suh, G. E., & Gupta, U. (2024). *Towards understanding systems trade-offs in retrieval-augmented generation model inference.* doi: 10.48550/arXiv.2412.11854

Silva, N., Barros, J., Santos, M. Y., Costa, C., Cortez, P., Carvalho, M. S., & Gonçalves, J. N. C. (2021). Advancing logistics 4.0 with the implementation of a big data warehouse: A demonstration case for the automotive industry. *Electronics*, *10*(18). doi: 10.3390/electronics10182221

Singh, S. P. (2024). Vector search in the era of semantic understanding: A comprehensive review of applications and implementations. *International Journal of Computer Engineering and Technology (IJCET)*, *15*(6), 1794–1805. doi: 10.34218/IJCET_15_06_153

Srivastava, A., Nalluri, M., Lata, T., Ramadas, G., Sreekanth, N., & Vanjari, H. B. (2023). Scaling ai-driven solutions for semantic search. In *2023 international conference on power energy, environment & intelligent control (peeic)* (p. 1581-1586). doi: 10.1109/PEEIC59336.2023.10451301

Steve Jeffrey, T. F. (2024, September). *Natural language query engine for relational databases using generative ai.* doi: 10.48550/arXiv.2410.07144

Stolfo, A. (2024, June). Groundedness in retrieval-augmented long-form generation: An empirical study. In K. Duh, H. Gomez, & S. Bethard (Eds.), *Findings of the* *association for computational linguistics: Naacl 2024* (pp. 1537–1552). Mexico City, Mexico: Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.100

Suryavanshi, C., Dyreson, C., & Adams, J. (2019). Renovating database queries with query autoawesome. In *2019 ieee 20th international conference on information reuse and integration for data science (iri)* (p. 253-260). doi: 10.1109/IRI.2019.00048

Svarre, T., & Russell-Rose, T. (2025). Think outside the search box: A comparative study of visual and form-based query builders. *Journal of Information Science*, *51*(2), 354–367. doi: 10.1177/01655515221138536

Thorne, J., Yazdani, M., Saeidi, M., Silvestri, F., Riedel, S., & Halevy, A. (2021, February). From natural language processing to neural databases. *Proc. VLDB Endow.*, *14*(6), 1033–1039. doi: 10.14778/3447689.3447706

Tian, Y., Zhang, Z., Ning, Z., Li, T. J.-J., Kummerfeld, J. K., & Zhang, T. (2024). *Interactive text-to-sql generation via editable step-by-step explanations.* doi: 10.48550/arXiv.2305.07372

Trappolini, G., Santilli, A., Rodolà, E., Halevy, A., & Silvestri, F. (2023). Multimodal neural databases. In *Proceedings of the 46th international acm sigir conference on research and development in information retrieval* (p. 2619–2628). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3539618.3591930

Troy, C., Sturley, S., Alcaraz-Calero, J. M., & Wang, Q. (2023). Enabling generative ai to produce sql statements: A framework for the auto- generation of knowledge based on ebnf context-free grammars. *IEEE Access*, *11*, 123543-123564. doi: 10.1109/ACCESS.2023.3329071

Winter, C., Giceva, J., Neumann, T., & Kemper, A. (2022, July). On-demand state separation for cloud data warehousing. *Proc. VLDB Endow.*, *15*(11), 2966–2979. doi: 10.14778/3551793.3551845

Xie, X., Xu, G., Zhao, L., & Guo, R. (2025). *Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment.* doi: 10.48550/arXiv.2502.14913

Yaghmazadeh, N., Wang, Y., Dillig, I., & Dillig, T. (2017, October). Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, *1*(OOPSLA). doi: 10.1145/3133887

Yang, W., Fu, R., Amin, M. B., & Kang, B. (2025). The impact of modern ai in metadata management. *Human-Centric Intelligent Systems*, *5*, 323–350. doi: 10.1007/s44230-025-00106-5

Yu, B., Wang, B., & Zhang, Y. (2024). Application of artificial intelligence in coal mine ultra-deep roadway engineering: A review. *Artificial Intelligence Review*, *57*, 262. doi: 10.1007/s10462-024-10898-w

Yu, H., Gan, A., Zhang, K., Tong, S., Liu, Q., & Liu, Z. (2025). Evaluation of retrieval-augmented generation: A survey. In *Big data* (pp. 102–120). Singapore: Springer Nature Singapore. doi: 10.1007/978-981-96-1024-2_8

Yu, W., Liao, N., Luo, S., & Liu, J. (2025). *Ragdoll: Efficient offloading-based online rag system on a single gpu.* doi: 10.48550/arXiv.2504.15302

Zhao, P., Zhang, H., Yu, Q., Wang, Z., Geng, Y., Fu, F., ... Cui, B. (2024). *Retrieval-augmented generation for ai-generated content: A survey.*

Zhou, X., Chai, C., Li, G., & Sun, J. (2022). Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, *34*(3), 1096-1116. doi: 10.1109/TKDE.2020.2994641

Appendix A

SQL Details for Experiment 2

A.1 Methods and Assumptions

- **Budget in SQL.** Base `WHERE` clause enforces `ic.price_nzd IS NOT NULL` and `ic.price_nzd` ≤ `request_budget_nzd` (passed via `TOOL_ARGS` and consistent with `capacity(id=1)`). *This budget predicate is mandatory and must never be relaxed silently.*
- **Session tables.** Each run queries the per-session schema `user_dw_<sessionId>` over: `capacity(id=1)` containing `request_budget_nzd`, `request_type`, `role`, `department_id`; and `inventory_candidates` as the device search surface.
- **Projection.** `device_id, pc_brand, model_name, cpu_model, gpu_model, ram_gb, ssd_size_gb, price_nzd, performance_tier, auto_approve_ok`.
- **User filters.** Optional constraints are mapped to simple predicates over the candidate surface (examples: brand/model contains, CPU/GPU contains, `ram_gb` minimum, `ssd_size_gb` minimum, `performance_tier` selection). For Q2, auto approval is controlled by `TOOL_ARGS.require_auto_approve`.
- **Auto approval (Q2).** If `TOOL_ARGS.require_auto_approve=true`, the query must additionally enforce `ic.auto_approve_ok = true`. If false or missing (Q1 budget-only), the query must not filter on `auto_approve_ok`.
- **Ranking.** Default ordering is by `price_nzd` ascending with `NULLS LAST`; ties break

deterministically on `device_id`.

- **Safety.** Exactly one read-only `SELECT`; no DDL/DML; no multiple statements; trailing semicolons removed; and a `LIMIT` enforced if missing.

A.2 Ranking and NULLs (SQL)

We use PostgreSQL `ORDER BY` semantics with explicit `NULLS LAST` to avoid promoting incomplete pricing rows. Ties break deterministically on `device_id`.

```
ORDER BY
  ic.price_nzd ASC NULLS LAST,
  ic.device_id ASC
```

Rationale:

1. **Budget-first ordering.** Lower priced devices are surfaced first within the request budget.
2. **Robustness to missing values.** Rows with NULL `price_nzd` are excluded by the base filter; `NULLS LAST` protects ordering if optional expressions are added later.
3. **Stability.** `device_id ASC` ensures reproducible ordering across runs.

A.3 Experiment 2 Query Template (single SELECT)

The agent emits a single, read-only `SELECT` scoped to the session schema, with an enforced `LIMIT`. Optional user filters are appended as shown.

```
-- <USER_SCHEMA> is the per-session schema (user_dw_<sessionId>)
SELECT
  ic.device_id,
  ic.pc_brand,
  ic.model_name,
  ic.cpu_model,
  ic.gpu_model,
  ic.ram_gb,
```

```
  ic.ssd_size_gb,
  ic.price_nzd,
  ic.performance_tier,
  ic.auto_approve_ok
FROM "<USER_SCHEMA>".inventory_candidates AS ic
WHERE ic.price_nzd IS NOT NULL
  AND ic.price_nzd <= <request_budget_nzd>

  -- Q2 only (auto approval question):
  -- AND ic.auto_approve_ok = true

  -- Optional user constraints (examples):
  -- AND LOWER(ic.pc_brand)     LIKE LOWER('%dell%')
  -- AND LOWER(ic.model_name)   LIKE LOWER('%latitude%')
  -- AND LOWER(ic.cpu_model)    LIKE LOWER('%i7%')
  -- AND ic.ram_gb              >= 16
  -- AND ic.ssd_size_gb         >= 512
  -- AND ic.performance_tier    IN ('mid','high','workstation')

ORDER BY
  ic.price_nzd ASC NULLS LAST,
  ic.device_id ASC
LIMIT 20
```