Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Found in translation: an LLM approach to private and reproducible Workflow Management Systems

Author: Emanuele Simeone (2858949)

1st supervisor: Adam Belloum
2nd reader: Natallia Kokash

A thesis submitted in fulfillment of the requirements for the joint VU-UvA Master of Science degree in Computer Science

August 25, 2026

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Adam Belloum and my second reader Natallia Kokash for the guidance and constructive feedback throughout the development of this thesis. Your expertise and support were invaluable in shaping this work.

To Elena: thank you for your unconditional support, *patience*, encouragement, and for putting up with me during the more stressful moments of this project. Your presence made all the difference.

Finally, I am deeply grateful to my parents and siblings for their constant support and belief in me throughout my studies. This work would not have been possible without the foundation and values you have transmitted to me.

Emanuele Simeone
Amsterdam, August 2026

Abstract

Scientific workflows underpin modern data-driven research, yet composing them remains a task reserved for those with programming expertise. Domain scientists like clinicians, biologists, and policy analysts must all either learn a workflow-description language or rely on software engineers to express their analytical intent as an executable pipeline. With the recent emergence of large language models (LLMs), an opportunity to bridge this gap by translating natural-language descriptions of analytical intent into executable workflows arises. At the same time, relying on LLMs to automate this translation raises two practical concerns that are rarely addressed together: *privacy*, since scientific data is often sensitive and cannot be sent to a commercial cloud API, and *reproducibility*, since a system that generates a different workflow each time the same question is asked undermines the very goal of scientific computing.

This thesis explores whether these constraints can be satisfied simultaneously. It proposes a pipeline built entirely on locally deployable, open-weight models that translates natural-language descriptions of analytical intent into executable workflows for the Brane distributed computing framework. By keeping inference on-premises and caching verified translations by semantic equivalence, the system attempts to close the gap between the ease of natural-language expression and the precision that scientific workflow execution demands, without compromising on data governance or result consistency. The findings offer insight into where the real bottlenecks lie when bringing LLM-based automation into resource-constrained, privacy-sensitive scientific environments.

The pipeline combines retrieval-augmented generation, structured intent decomposition, and a semantic cache, evaluated across three open-weight model scales on a BraneScript test set. The results show that prompt and retrieval engineering outweigh model scale as the primary driver of translation quality, with the largest model reaching 93% execution rate after targeted grounding improvements. Intent decomposition provides consistent gains at every scale.

Supervised fine-tuning on a small training set revealed a memorisation failure that must be managed explicitly in any maintained deployment. The semantic cache achieves hit rates up to 99% on paraphrased inputs while exposing a fundamental limit of embedding-based similarity for parameter-sensitive scientific queries, a tension that points to open problems at the intersection of NLP and scientific reproducibility.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Scientific computing increasingly depends on complex, multi-step computational workflows: data must be fetched, cleaned, analysed, and interpreted, often across heterogeneous compute infrastructure. Workflow Management Systems (WMSs) such as Pegasus (1), Nextflow (2), and Snakemake (3) have become indispensable in fields like genomics, epidemiology, and climate science precisely because they automate resource allocation, fault recovery, and execution ordering. Yet, despite their maturity, these systems impose a heavy cognitive cost on scientists. To express even a straightforward intent, like "calculate the average age of patients in this dataset", a researcher must first understand the WMS's execution model, write a domain-specific language (DSL) script that explicitly enumerates every computational step, declare data dependencies as a directed acyclic graph (DAG), and then debug execution failures at the infrastructure level rather than the scientific level.

This mismatch between *scientific intent* and *workflow specification* has been recognised as a fundamental barrier to reproducible science. Scientists spend substantial effort debugging YAML configurations and DAG topologies instead of advancing research questions. The cognitive overhead compounds further in distributed settings, where data locality, containerisation, and security policies add additional layers of complexity to what should be a simple computational request.

Recent advances in Large Language Models (LLMs) open a new possibility: a researcher could describe a computational task in plain language, and an intelligent system could automatically translate that natural-language intent into a runnable workflow. This vision, *intent-based orchestration*, has attracted growing interest in the systems and HPC communities. However, existing approaches remain incomplete. Systems that accept natural

language either generate generic code that cannot be executed in a target WMS environment, or they lack the formal provenance guarantees required for reproducible science. Conversely, WMSs that provide strong reproducibility have no mechanism for accepting high-level declarative intent.

The gap is thus precise and actionable: no system currently bridges declarative, natural-language intent expression with the rigorous, provenance-tracked execution guarantees of a production WMS.

To add to this challenge, the scientific domains that would benefit most from intent-based orchestration, often work with data that cannot be freely centralised or shared with an external computing party. An intent-based system for these domains therefore faces additional constraints that a general-purpose coding assistant does not.

1.2 Problem Statement

In such distributed scientific computing infrastructures, the party executing a computation may not own the underlying data. The core problem this thesis addresses is how a researcher, working under this constraint, can describe a computational task in natural language and reliably obtain a workflow execution that is both correct and reproducible.

The problem decomposes into four interacting challenges.

Ambiguity of natural language. The same scientific intent can be phrased in many different ways, and different phrasings, or even the same phrasing repeated by different users, should ideally produce semantically equivalent workflows. Intents are also frequently underspecified: a request may omit which dataset, parameter, or output format is intended, in which case the system must either recover a reasonable default from context or explicitly ask for clarification, rather than silently guessing. Natural language also uses domain vocabulary (e.g., "heart-disease data") that does not correspond directly to any identifier in the target system; the translation system must therefore ground loosely specified terms in the concrete capabilities actually available to it, not merely in the words of the request.

Correctness with respect to a strongly typed, platform-specific DSL. Workflow specification languages of this kind are typically statically typed, with strict, platform-specific rules for invoking computational capabilities (functions, packages, or services) and for connecting inputs and outputs via user-defined workflows. Such languages are unlikely to be well represented in the data an LLM was pre-trained on, so a general-purpose model cannot be expected to produce valid programs in them purely from its prior knowledge;

it must be additionally grounded in the platform's actual capabilities, whether through retrieved documentation, fine-tuning on representative examples, or both.

Reproducibility. A scientifically useful translation system must ensure that the same intent, or an evident paraphrase of it, yields the same (or a demonstrably equivalent) workflow and execution trace across repeated invocations, across sessions, and potentially across different users. This requires more than a translator that happens to behave deterministically for a fixed input: because the underlying LLM component is stochastic and may be updated or replaced over time, reproducibility requires an explicit mechanism that relates a given intent to the workflow(s) it has previously produced, so that repeated or equivalent requests do not silently diverge, and so that computation is not repeated unnecessarily for a request that has already been answered.

Operating within a privacy boundary. Various scientific domains impose strict privacy constraints on data ownership and computation. Those constraints not only impact what the translation system is allowed to output, but where its components are allowed to run: an architecture that routes user intents through an externally hosted, centralised LLM service, or that stores any derived artefacts outside the data owner's boundary, is not a viable solution to the three challenges above, regardless of how accurate or reproducible its translations otherwise are.

Addressing these four challenges together requires not only a translation mechanism from natural language to a target workflow language, but also a retrieval mechanism to ground translation in the platform's available capabilities, a validation mechanism that distinguishes overt from silent errors, and a mechanism for relating a given intent to workflows it has previously produced, all without exposing data or derived artefacts beyond their owning organisation.

1.2.1 Scope of This Thesis

The four challenges above are general to any distributed, privacy-sensitive workflow platform with a strongly typed specification language; they do not, by themselves, depend on any particular choice of platform. This thesis studies them concretely through **Brane**, a distributed WMS whose federated execution model dispatches computation to the site holding the relevant data rather than moving the data to the computation, which makes it a natural instantiation of the privacy boundary described above. Brane and its workflow language, BraneScript, are introduced in detail in Chapter 2. From this point on, the thesis is stated concretely in terms of BraneScript: research questions, contributions, and evaluation, all target it directly, while the underlying translation, retrieval, validation, and

reuse techniques are not specific to BraneScript and are intended to generalise to other strongly typed workflow DSLs.

1.3 Research Questions

This thesis investigates the following research questions:

RQ1 To what extent can LLMs translate natural-language scientific intents into syntactically and semantically correct DSL workflows?

RQ2 How does the quality of retrieval-augmented context and prompt engineering affect translation accuracy, independent of model scale?

RQ3 How does supervised fine-tuning (SFT) on domain-specific Branescript examples affect translation accuracy and generalisation?

RQ4 Can semantic caching effectively serve as a reproducibility layer for intent-based orchestration, and what are its fundamental limitations for scientific computing?

1.4 Thesis Structure

The remainder of this thesis is organised as follows: Chapter 2 provides background on scientific workflow systems, provenance, intent-based orchestration, and LLMs in scientific computing. Chapter 3 presents a high-level architecture of the proposed system. Chapter 4 describes the design decisions. Chapter 5 details the implementation. Chapter 6 reports the empirical results. Chapter 7 interprets the results in the context of the research questions. Chapter 8 discusses threats to validity and Chapter 9 concludes with future directions.

2

Background

This chapter provides the conceptual foundations necessary to understand the remainder of the thesis. We cover scientific workflow systems, data provenance, FAIR principles for computational workflows, intent-based orchestration, and the use of LLMs in scientific computing. We conclude with the selection of a target platform for this work: Brane, the execution platform this thesis targets, and Dynamos, an earlier alternative considered but not adopted.

2.1 Scientific Workflow Management Systems

A *scientific workflow* is a structured composition of computational steps whose execution produces a reproducible scientific result. Workflow Management Systems (WMSs) automate the orchestration of these steps across potentially heterogeneous compute resources, handling scheduling, data transfer, error recovery, and resource allocation transparently from the scientist (1).

2.1.1 Representative Systems

Several WMS platforms have achieved broad adoption in the scientific community.

Pegasus (1) is a DAG-based system developed at USC/ISI that abstracts over heterogeneous grid and cloud resources. It supports four distinct lifecycle phases: workflow generation, workflow planning (site selection and data staging), workflow execution, and monitoring. Pegasus uses HTCondor as its underlying execution engine and emphasises fault tolerance through automatic retry and checkpointing.

Nextflow (2) adopts a dataflow programming model in which independent computational steps (processes) communicate via asynchronous channels. It has become dominant

in bioinformatics due to its native Docker and Singularity container support and its community registry of reusable pipelines (nf-core). Nextflow's DSL2 syntax is Groovy-based and explicitly describes data movement between processes.

Snakemake (3) follows a *make*-inspired declarative model: scientists define rules that describe how to produce output files from input files, and the system infers the necessary execution order. It is particularly popular in genomics and integrates natively with conda environments for software dependency management.

Galaxy (4) provides a web-based visual interface for workflow construction, lowering the barrier for non-programmer scientists. Workflows are composed graphically and can be shared through public repositories.

Common Workflow Language (CWL) (5) is a vendor-neutral specification rather than an execution engine. It defines a portable, YAML-based format for describing tool invocations and their data connections, with the goal of enabling workflow portability across WMS implementations.

Despite their diversity, all these systems share a common limitation from the perspective of this thesis: they require the scientist to specify workflows explicitly and at the computational level. The transition from a scientific intent to an executable workflow is left entirely to the researcher.

2.1.2 Workflow Lifecycle

A typical WMS workflow follows five phases (1): (1) *specification*, where the scientist describes the computational steps and their dependencies; (2) *planning*, where the WMS maps abstract steps to concrete computational resources; (3) *execution*, where steps run on target infrastructure; (4) *monitoring*, where execution progress and failures are tracked; and (5) *archiving*, where results and provenance are stored for future reference. This thesis focuses on two of these five phases. Its primary focus is the interface before phase 1: the translation of natural language into a formal workflow specification. Its secondary focus is phase 5: the translation system introduced here is required, by the reproducibility and privacy constraints motivated in Section 1.1, to record which intent produced which workflow, so that repeated or paraphrased requests can be related to prior executions rather than re-derived from scratch. This requirement motivates the treatment of data provenance (Section 2.2) as a main concern of this thesis.

2.2 Data Provenance

Reproducibility in scientific computing requires not only that results can be recomputed, but that the precise computational history that produced those results is recorded and queryable. This is the domain of *data provenance*, and it has been recognised as a first-class concern for scientific workflows for over a decade: workflow systems are expected to record what was run, on what inputs, and by whom, so that results can later be verified, audited, or reused (6). Provenance therefore serves at least three distinct purposes in a scientific workflow system: it lets a third party *verify* that a published result actually follows from the stated inputs and method; it lets a scientist *debug* a workflow by tracing an unexpected output back to the step or parameter that produced it; and it lets a data owner or auditor *attribute* responsibility for a computation to the specific request, user, and code version that caused it to run. This same attribution requirement becomes central once, as in this thesis, the request itself originates from natural language rather than from a hand-written script.

2.2.1 W3C PROV Data Model

The W3C PROV Data Model (PROV-DM) (7) provides a standard, domain-independent vocabulary for expressing provenance, developed as a W3C Recommendation to allow provenance produced by different, otherwise incompatible systems to be exchanged and reasoned about uniformly. Its core concepts are *entities* (data artefacts, e.g., a dataset or a generated program), *activities* (computational steps that act on or produce entities, e.g., a translation or an execution), and *agents* (the people, organisations, or software processes responsible for those activities). A small set of relationships connect these elements into a provenance graph that captures the causal history of any output: `used` (an activity consumed an entity), `wasGeneratedBy` (an entity was produced by an activity), `wasAssociatedWith` (an activity was carried out by an agent), `wasAttributedTo` (an entity is attributed to an agent), and `wasDerivedFrom` (an entity was transformed, or copied, from another entity without naming the intervening activity explicitly). Because PROV-DM is deliberately generic, it is not tied to any workflow language or execution engine; consequently, essentially every domain-specific provenance format in scientific computing, including the two discussed below, is defined as a specialisation of PROV rather than as an unrelated model.

Figure 2.1 illustrates this core graph structure for a generic scientific computation, using the standard PROV-O visual notation (yellow ellipses for entities, blue rectangles for ac-

tivities, and orange pentagons for agents). A *Computation* activity `used` an input *Dataset* entity and is `wasAssociatedWith` the *Researcher* agent who initiated it; the *Result* entity it produced `wasGeneratedBy` that activity and is, in turn, `wasAttributedTo` the same agent. This is the minimal graph that any of the domain-specific formats discussed in the remainder of this section extend or specialise.

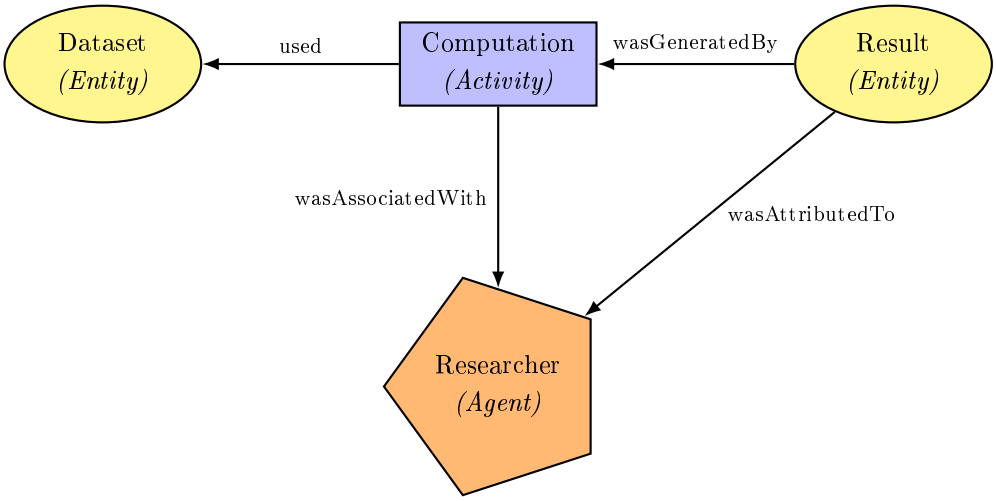

Figure 2.1: The core PROV-DM graph: entities (yellow ellipses), activities (blue rectangles), and agents (orange pentagons) connected by the `used`, `wasGeneratedBy`, `wasAssociatedWith`, and `wasAttributedTo` relations, following standard PROV-O visual notation.

2.2.2 Prospective vs. Retrospective Provenance

Provenance is commonly distinguished along two axes:

- *Prospective provenance* captures the workflow specification, which is the plan of what will be executed, and is equivalent to the workflow definition itself.
- *Retrospective provenance* captures what actually ran, including concrete inputs, parameter values, timestamps, and intermediate results.

Full reproducibility requires both: the specification must be re-runnable (prospective), and each execution must be uniquely identified (retrospective). **ProvONE** (8) makes this distinction explicit as a PROV extension purpose-built for scientific workflows: it adds workflow-specific entity and activity subtypes (e.g., distinguishing a *Program*, a static prospective artefact, from an *Execution*, its retrospective instantiation) so that the same PROV vocabulary can describe both the plan and the run without conflating them.

2.2.3 Workflow Provenance Formats

Several formats extend PROV-DM for workflows. **CWLProv** (9) records the full retrospective execution of a CWL workflow (every step invocation, concrete input and output file, and intermediate value) as a PROV-O graph packaged inside a Research Object (RO) bundle, so that a CWL run can be re-inspected or re-executed independently of the original environment. **RO-Crate** (10) is a more lightweight packaging format based on schema.org that bundles data with its metadata and provenance in a single, self-describing directory, enabling FAIR sharing of computational results without requiring the full PROV-O vocabulary. Both formats, however, are designed around *retrospective* execution provenance: they assume the input to be recorded is already a concrete, machine-executable workflow, and neither has a defined place for the natural-language request that led a human, or an automated translator, to construct that workflow in the first place.

Intent-based systems add a new provenance challenge: not only must the execution be recorded, but the *intent* that triggered the execution must also be captured. This is what in this thesis we term *intent provenance*: the mapping from a natural-language request to the workflow that was generated and run. Modelling intent within PROV-DM is possible thanks to the extensibility of the model: an intent can be represented as a specialised *entity* that is `used` by a translation *activity*, which in turn `wasGeneratedBy`-relates to the resulting workflow entity, mirroring how CWLProv and ProvONE specialise PROV's entities and activities for their own domains. What none of the existing provenance formats address is the semantics of that relationship when the translation activity is not deterministic: because the translation step in this thesis is performed by an LLM, the same intent entity can be `used` by two distinct translation activities that each `wasGeneratedBy`-produce a different workflow entity, so `wasGeneratedBy` alone cannot certify that a request has been answered reproducibly. This gap is one of the motivating design considerations of the semantic cache described in this thesis (Chapter 4), as it is the missing mechanism that lets a repeated or paraphrased intent be related, via `wasDerivedFrom`, to a workflow entity that was already generated, rather than triggering an independent, potentially divergent generation event.

2.3 FAIR Principles for Workflows

The FAIR principles (Findable, Accessible, Interoperable, and Reusable) were originally articulated for scientific data, but have since been extended to computational workflows (11).

FAIR workflows require that workflow specifications be discoverable in registries, accessible via stable identifiers, portable across execution environments, and documented with sufficient metadata to enable third-party reuse.

In practice, achieving FAIR compliance for LLM-generated workflows is non-trivial: outputs are generated dynamically, may not be stored in a registry, and may vary across sessions. The caching layer introduced in this thesis addresses the reusability dimension by ensuring that once a workflow is generated for a given intent, subsequent requests for semantically equivalent intents retrieve the same (or a semantically equivalent) workflow without additional LLM inference.

2.4 Intent-Based Orchestration

Intent-based networking (IBN) is a paradigm that originated in network management, where operators specify *what* a network should do rather than *how* to configure it (12). The system then autonomously translates the high-level intent into low-level configuration. The IBN lifecycle comprises four phases: *expression* (the operator states an intent), *translation* (the system converts the intent to a configuration), *activation* (the configuration is applied), and *assurance* (the system verifies that the intent is satisfied).

2.4.1 Application to Scientific Computing

The IBN paradigm generalises naturally to scientific computing. A scientist expresses an intent ("compute the GC content of this sequence"), the system translates it into a workflow, executes the workflow on the target infrastructure, and verifies that the output is correct. This framing shifts the cognitive burden from workflow engineering to intent expression, which is a task that scientists are more familiar with and can perform more reliably.

The MAPE-K (Monitor–Analyse–Plan–Execute with shared Knowledge) control loop from autonomic computing (13) provides a complementary model for adaptive orchestration that has also been applied to this work: the system monitors execution, analyses failures, re-plans the workflow, and re-executes, using a shared knowledge base to inform decisions. This model is relevant when LLM outputs fail to compile or execute, as the system can iterate on the translation rather than surfacing a raw error to the user.

2.5 Large Language Models in Scientific Computing

2.5.1 LLMs for Code and Workflow Generation

Recent LLMs have demonstrated substantial capability in code generation tasks. Models such as OpenAI Codex (14) and its successors have been evaluated on programming benchmarks including HumanEval (14) and MBPP (15), where they achieve competitive pass@k rates. More recent open-weight models (Qwen, Llama, Mistral families) (16, 17, 18) have closed much of the performance gap with proprietary systems, making local deployment on HPC infrastructure feasible.

In the workflow domain, Duque et al. (19) demonstrated that LLMs can generate workflows for computational scientific pipelines from natural-language descriptions, achieving promising accuracy on domain-specific tasks. This work established that LLMs can encode sufficient domain knowledge about workflow patterns to serve as initial translators, but noted that accuracy degrades significantly when the target WMS has a non-standard or less-common DSL, a challenge directly relevant to Brane used in this work.

2.5.2 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) (20) augments LLM inference by retrieving relevant context from an external knowledge base and prepending it to the input prompt. In the workflow translation setting, the knowledge base contains package documentation, such as function signatures, parameter descriptions, and usage examples, and the retrieval system selects the subset of packages most likely relevant to the current intent.

RAG addresses a critical limitation of pure LLM generation: models cannot know the exact API of a custom execution environment such as Brane without having seen it in training data. By retrieving package documentation at inference time, the system grounds generation in the actual available capabilities, reducing hallucinated function calls and incorrect parameter types.

2.5.3 ControlA and Agentic Orchestration

Gueroudji et al. (21) introduced ControlA, a system that uses LLMs as autonomous agents for scientific workflow management, allowing the model to iteratively refine a workflow in response to execution feedback, with the model itself reasoning about how to diagnose and repair failures. This agentic approach differs from the bounded, externally-driven retry

mechanism adopted in this thesis (Chapter 4), and represents a complementary direction for future work.

2.6 Gaps in the Literature

A systematic review of the scientific workflow, provenance, intent-based orchestration, and LLM literature reveals five persistent gaps:

G1 — Intent Provenance Model. No existing system defines a formal model for recording the intent that triggered a workflow execution alongside the standard retrospective execution provenance.

G2 — Hybrid Intent-to-WMS Bridge. No system simultaneously supports declarative natural-language intent expression, translation to a target WMS DSL, and WMS-grade reproducibility guarantees. Systems either provide natural-language interfaces without WMS integration, or WMS integration without natural-language interfaces.

G3 — Intent Reproducibility Definition. There is no agreed definition of what it means for an intent-based system to be reproducible: do two semantically equivalent intents need to produce the same workflow, or is it acceptable for them to produce different but equally valid ones?

G4 — Human-in-the-Loop Trust Checkpoints. Existing proposals for LLM-based workflow generation do not provide principled mechanisms for a scientist to inspect and approve a generated workflow before execution, particularly important in safety-sensitive or privacy-sensitive domains.

G5 — Cross-Paradigm Benchmarks. There are no publicly available benchmarks that evaluate intent-based systems against conventional user-defined WMS approaches on the same scientific tasks.

This thesis addresses **G2** directly: it implements a hybrid bridge between natural-language intent expression and the Brane WMS via LLM-based BraneScript generation. It also addresses **G1** by proposing an intent provenance model that records the mapping from natural-language intent to generated workflow alongside standard retrospective execution provenance (Chapter 4). Finally, it provides partial evidence towards **G3** through the semantic cache threshold analysis.

2.7 Target Platform Selection: Brane and Dynamos

2.7.1 Brane and BraneScript

Brane (22) is a distributed, containerised computing framework designed for federated scientific data analysis. It targets scenarios where data cannot leave a site due to privacy or regulatory constraints; (e.g., hospital patient records, genomics data under GDPR), but compute results can be returned to a central orchestrator. Brane wraps computational tools in OCI-compatible container images (*packages*), and provides a central registry from which packages can be invoked across participating sites.

Each Brane package exposes a set of typed functions. A function call specifies a package name, a function name, and typed argument values; Brane dispatches the call to a site that holds the relevant data and returns the result to the orchestrator. This model provides strong isolation and auditability, where every function invocation, data transfer, and result is logged.

This federated execution model is the specific reason Brane was selected as the target platform for this thesis. Unlike a workflow written for a generic Python or shell execution environment, a BraneScript workflow can be deployed and executed inside a third-party organisation's own infrastructure without that organisation ever having to share its raw data with the party that authored the workflow: the workflow is the entity that travels to the data and not the reverse. This property motivates a corresponding design constraint for the translation system built in this thesis: both the LLM used to generate BraneScript and the semantic cache introduced in Chapter 4 are designed to run locally, within the same trust boundary as the data, so that neither the natural-language intent nor the generated workflow needs to leave the organisation in order to be produced, validated, or reused.

BraneScript's static typing gives the translation pipeline a way to evaluate its correctness: the Brane compiler can reject a malformed generated workflow before any container is started or any data is touched, which is precisely the *overt*-error detection mechanism the problem statement in Section 1.1 requires, and it is a much weaker guarantee that a dynamically typed target language (e.g., a plain Python script) could offer, since there the same mistake would only surface, if at all, as a runtime failure deeper into execution. Finally, Brane's package model, composed of typed function signatures published through a central registry, maps directly onto the retrieval problem this thesis needs to solve: a package's signatures are themselves a natural, structured retrieval corpus for grounding LLM generation (the RAG design described in Section 4.2).

BraneScript is the DSL used to express Brane workflows. It is a statically typed, procedural language with Python-inspired syntax. Key constructs include:

- **Package imports**: `import` *`package_name`*`;` loads a package's function signatures into scope.
- **Function calls**: `let result :=` *`func`*`(`*`arg`*` :=` *`value`*`);` invokes a package function with named arguments.
- **Data commits**: `commit_result(`*`name`*`,` *`result`*`);` persists an intermediate result as a named dataset.
- **Variable declaration**: `let` *`var`*` :=` *`value`*`;` declares a variable and assigns a value to it.
- **Class instantiation**: `new` *`ClassName`*`{` *`field`*` :=` *`value`*` }` instantiates a built-in class; the two built-in classes are `Data` (a reference to a registered dataset, e.g. `new Data{ name := "heal_pa_2" }`) and `IntermediateResult` (a handle to a transient computation result produced by a package function and passed as input to a subsequent package call; unlike `Data`, an `IntermediateResult` is not a named dataset in the Brane registry but an in-memory result from the previous step in the workflow, e.g. the output of a filtering operation passed directly into an aggregation function).
- **Field access**: dot notation (*`obj.field`*) accesses fields on struct-typed return values from package functions.
- **Output**: `println(`*`expr`*`);` prints a value to standard output; string concatenation uses `+`.
- **Control flow**: Standard `if`/`else` and `for` constructs.
- **Parallel execution**: `parallel [`*`stmt1`*`,` *`stmt2`*`]` executes multiple statements concurrently.

The major challenge for LLM-based generation is that BraneScript is not represented in standard pre-training corpora. Models must infer its syntax from the few examples provided in the system prompt and from the RAG context. Supervised fine-tuning (Section 4.6) on a curated set of BraneScript examples is intended to internalise the syntactic patterns and package invocation conventions that the base model cannot reliably produce zero-shot; the specific packages and example set used to do so in this thesis are an evaluation testbed and are described in Chapter 6.

2.7.2 Alternative Considered: Dynamos

Before settling on Brane, an earlier iteration of this project targeted **Dynamos** (23, 24), a middleware developed at the University of Amsterdam for policy-driven, federated data exchange. Dynamos takes a microservice-based rather than a package/container-based approach: it abstracts common data-exchange patterns into reusable *archetypes*, and uses a MAPE-K control loop (the same autonomic-computing pattern introduced in Section 2.4) to dynamically compose and adapt microservice chains at runtime according to data-sharing agreements between participating institutions. Like Brane, it is built around the premise that data should stay under the control of the institution that owns it, and several exploratory translation experiments were carried out against it early in this project, targeting a small subset of intents.

Dynamos was ultimately not adopted as the platform for this thesis, for two concrete reasons observed during this early exploration. First, the set of available archetypes did not yet cover the breadth of workflow patterns, like typed function invocation, control flow, and parallel execution, that a general-purpose scientific translation system needs to be able to target, and its microservice composition model was still evolving in ways that would have made a stable target for LLM-based generation difficult to maintain over the course of the project. Second, the underlying Kubernetes-based execution runtime was, at the time, itself unreliable: exploratory test runs intermittently failed or behaved unpredictably at the container-orchestration level, in ways that were not always straightforward to diagnose, which was further evidence that the framework was not yet mature enough to build a production-oriented translation pipeline on top of. Brane's statically typed package model and comparatively stable runtime made it the more viable choice for the remainder of the thesis, though the underlying motivation, which is to enable privacy-preserving, federated distribution of scientific computation, is shared between the two systems, and the translation approach developed here does not depend on any feature unique to Brane over Dynamos beyond the availability of a stable, typed target language, compiler, and runtime.

In retrospect, one aspect of Dynamos's design remains an interesting fit for the intent-based framing pursued in this thesis. An archetype declares *which pattern* of data exchange should be instantiated, leaving the middleware to compose and configure the concrete microservices that realise it; this is arguably closer in spirit to a declarative statement of intent than a BraneScript workflow is, which remains an explicit, imperative sequence of typed function calls that must be fully specified before it can be executed. Had Dynamos

been mature enough to serve as the target platform, the natural translation target for an intent-based system built on top of it would plausibly have been archetype selection and parameterisation, rather than full workflow synthesis, a related but distinct problem from the one addressed in this thesis, and one that this work does not attempt to resolve.

3

Overview

This chapter presents the system built to address the problem stated in Chapter 1: translating a natural-language scientific intent into a correct, executable and reproducible BraneScript workflow within a set of privacy boundaries. We describe the system from the perspective of a researcher submitting an intent, and present the resulting architecture as it appears from that same vantage point; the design rationale behind these choices is developed in Chapter 4, and their implementation in Chapter 5.

3.1 User Perspective

The researcher uses the system by typing what they want done in plain English: *"I want to analyse all heart-disease data from the patients of the Amsterdam UMC"*. No knowledge about BraneScript's syntax, package's function signatures, or how Brane actually executes the workflow needs to be known beforehand or added to the request.

Entry points. Two entry points expose this interaction, aimed at different use cases:

- **Command-line tool**. Intended for local development, debugging, and batch experimentation, this entry point is invoked directly on a machine with the model loaded locally, either for a single intent or for a file of intents processed one after another. It runs synchronously to completion, submitting the generated script for execution once ready.
- **Web dashboard**. Intended for interactive day-to-day use. Its *Generate* page is where a researcher types an intent, picks a model and sees the generated script and its execution result. Generation is only one of the dashboard's roles, however:

Section 3.4 describes the further pages through which a researcher browses past intents, the package/dataset catalog, and model evaluation results.

Outcomes. Regardless of entry point, a submitted intent eventually resolves to one of three outcomes:

1. **Success.** A BraneScript workflow is generated, compiles, executes correctly on the Brane runtime, and its result (and, in the dashboard flow, the workflow itself) is returned to the researcher. The intent-workflow-result triple is also written to the semantic cache, so that a semantically equivalent future intent can be served immediately without invoking the model again.

2. **Corrective retry.** Generation succeeds syntactically but the script fails to compile or fails at runtime against Brane. The compiler or runtime error is fed back into the prompt as corrective context and generation is retried, up to a fixed number of attempts (three, in the current configuration), before the researcher is shown anything.

3. **Failure report.** All retries are exhausted without producing a working workflow. The researcher is shown the last generated script together with the error that caused it to fail, rather than a silent or generic error, so that the intent can be rephrased or the missing information (e.g., a dataset name) supplied explicitly.

3.2 Process Architecture

As both entry points are front doors onto the same underlying architecture, once an intent is submitted it goes through the same sequence of steps. This pipeline is organised around the six logical components that follow:

1. The **semantic cache** is consulted first, before any other work is done. It holds previously generated (intent, workflow, result) triples and returns it immediately if the incoming intent is close enough, in embedding space, to one already seen.

2. On a miss, **intent decomposition** breaks the (often compound) natural-language intent into a short list of concrete sub-tasks, so that later stages have a sharper, more explicit description of what needs to be done than the raw sentence alone provides.

3. The **knowledge base and retriever** takes the intent and its sub-tasks and selects the package and dataset documentation most relevant to it, including function signatures, parameter types and usage examples, out of everything registered within Brane.

4. **Prompt construction and LLM generation** combine the retrieved context, a static BraneScript syntax reference, and the original intent into a single prompt, and produce a candidate BraneScript workflow from a (possibly fine-tuned) language model.

5. The **Brane compiler and runtime** are the sole authority on whether the generated workflow is valid: a single `brane workflow run` invocation compiles the script and, if compilation succeeds, executes it against the real, federated data the workflow's functions address. The result is returned to the researcher.

6. A **feedback loop** connects the last component back to the fourth in case a compilation or runtime failure is captured. The error is fed into the next generation attempt as corrective context, for a bounded number of retries, before the researcher sees anything.

Figure 3.1 shows how these six components connect for a single request, independently of which entry point issued it. Physical placement is addressed next, in Section 3.3.

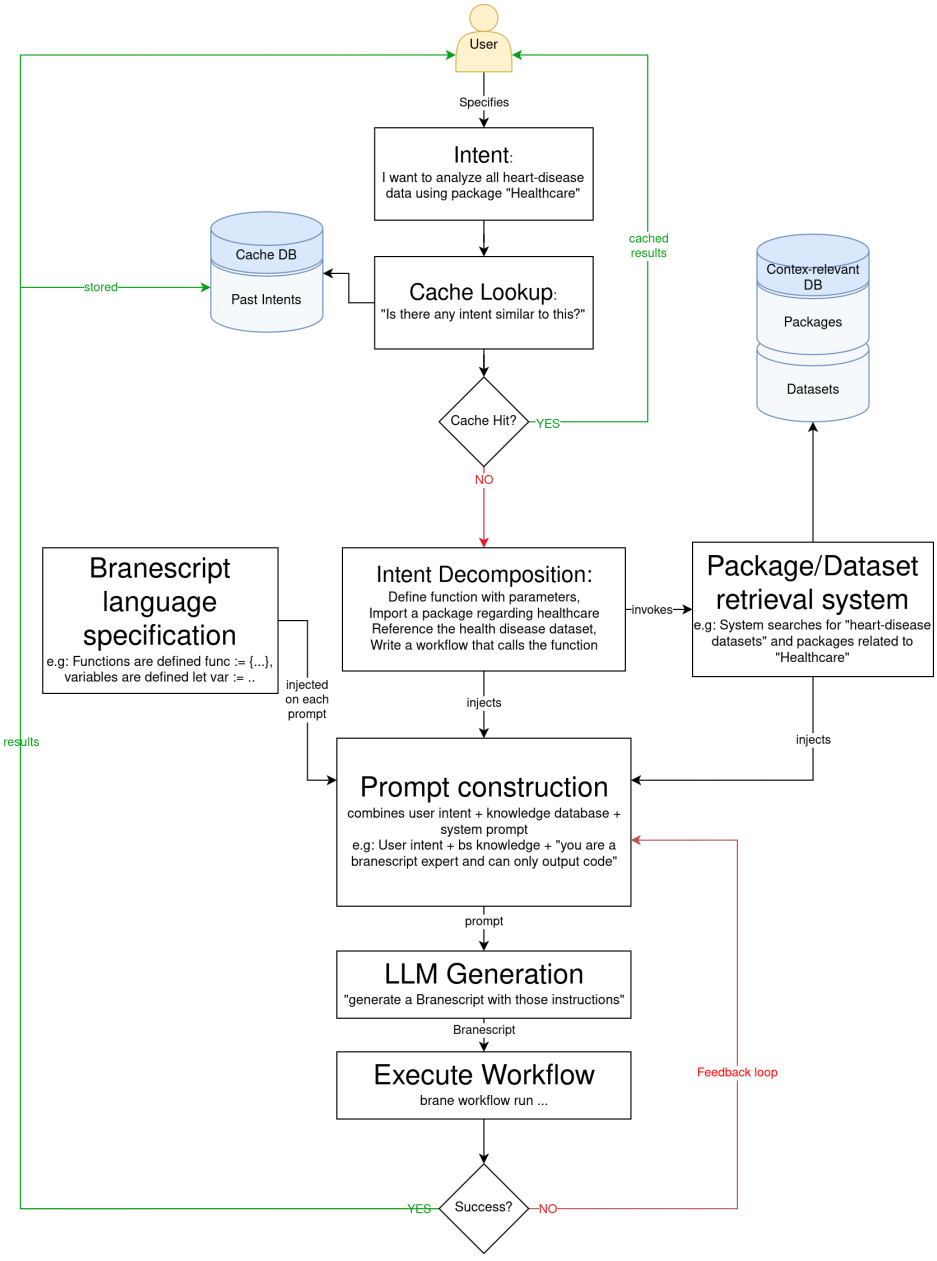

Figure 3.1: Conceptual architecture shared by both entry points. A cache hit returns a previous (intent, workflow, result) triple immediately; on a miss, the intent is first decomposed into a short list of su/b-tasks, which sharpen the query passed to the retrieval component, fetching package and dataset context from a single ChromaDB store; the BraneScript syntax reference is not retrieved but injected in full, statically, into every prompt. The Brane compiler and runtime, invoked as a single `brane workflow run` command, are the sole authority on whether the generated script is valid; a compilation or runtime failure feeds back into prompt construction as corrective context for a bounded number of retries, and a successful result is written to the semantic cache.

3.3 Infrastructure Architecture

This section describes the physical deployment of the six components introduced in Section 3.2: which machine each one runs on, and how that placement enforces the privacy motivation behind this work (Section 2).

The six logical components fall into three tiers of physical infrastructure. The **dashboard server** is the only tier a researcher's browser talks to directly as it hosts the frontend and local API process. On a cache miss, it submits a batch job over SSH to the **Snellius HPC cluster**, where the semantic cache, intent decomposition, package/dataset retrieval, and the fine-tuned LLM all run together as a single SLURM job. These four components are always co-located as the cache gates whether the (expensive) LLM is invoked at all.

Downstream of generation sits the **Brane control node**: the user translated intent is executed on this node via a `brane workflow run` call. The control node compiles the script and schedules each function's execution to whichever federated site actually holds the data that function needs and by following Brane's philosopy, it is not (always) a data-holding site itself. Figure 3.2 shows two such sites, Site A and Site B, as independent examples, where each runs its own Brane runtime worker against its own local data without exposing it to other entities.

This three-tier separation is what lets the LLM-facing part of the system run on whichever hardware is convenient (a laptop, a shared HPC cluster, eventually a cloud-hosted API) without that choice ever placing raw data outside the site that owns it

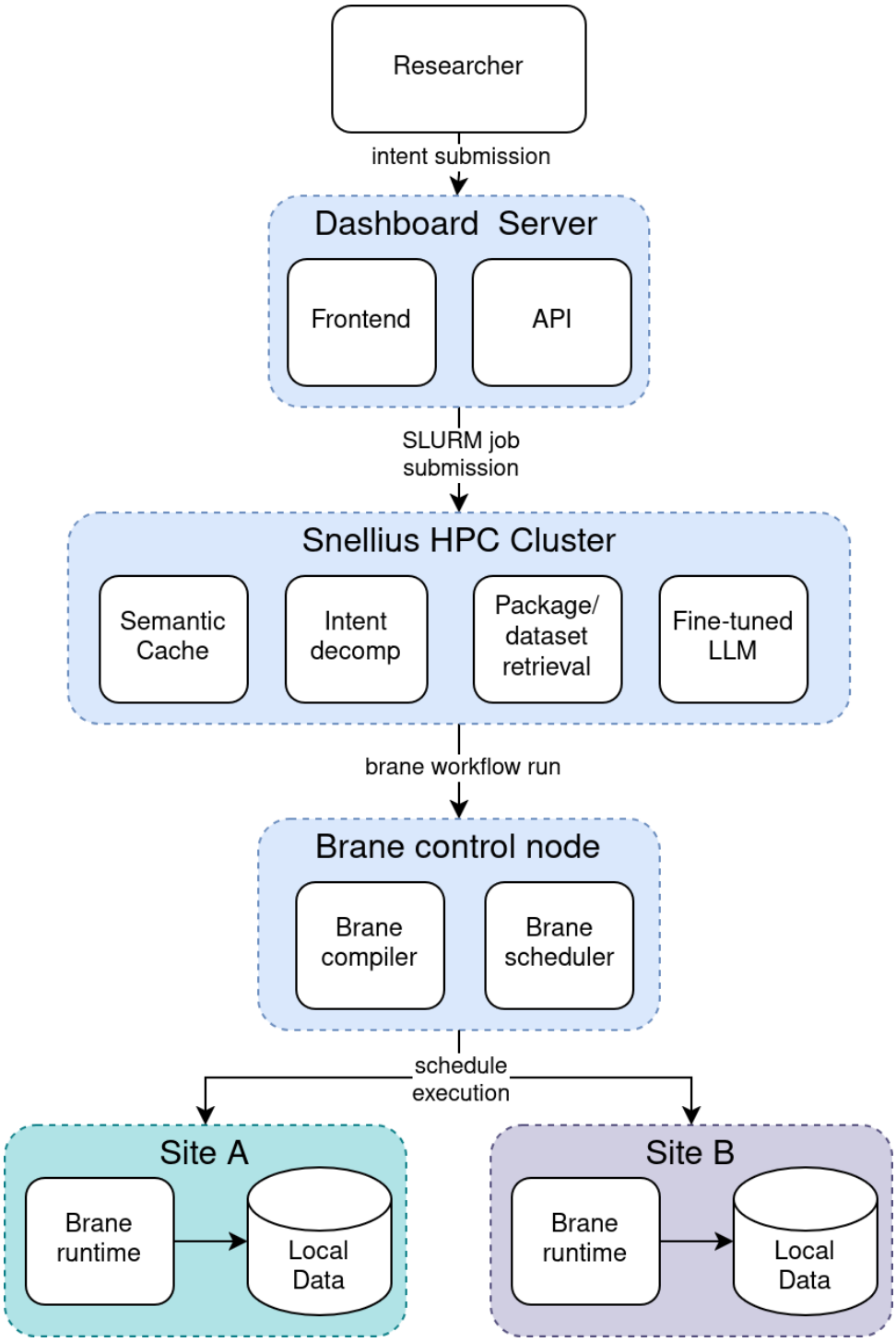

Figure 3.2: Physical deployment of the components shown in Figure 3.1

3.4 Dashboard: Monitoring and Catalog Browsing

The Generate page mentioned in (Section 3.1) is only one of few pages the dashboard exposes. The other five turn complete it as a dashboard that monitors the state of the whole system: what has been tried before, what capabilities are available to build a workflow from, and how the candidate models being fine-tuned actually compare. This supports for the intent-based framing of this thesis, specifically because a non-trained researcher, who does not have any prior knowledge of BraneScript, has no other way to inspect any of this information as packages, datasets, and model outputs are all hidden behind Brane CLI interface. The dashboard's other pages are:

- **Runs.** A searchable, sortable table of every intent previously submitted through the dashboard, together with the model used, the resulting verdict (pass/fail), the compiler or runtime error type on failure, the total duration, and the number of corrective-retry attempts it took. A stats bar above the table summarises the same information in aggregate (total runs, pass rate, average duration). This is the page a researcher returns to in order to see whether an intent they submitted earlier eventually succeeded, or to gauge, at a glance, how reliably the system handles a given class of intents before trying a new one.

- **Packages** and **Datasets.** Two browsable catalogs, read directly from the same package specifications and dataset manifests that back the retriever (Section 3.2). A researcher can browse what the system is capable of *before* writing an intent, rather than discovering an unsupported capability only after a generation attempt fails.

- **Exec Results.** A table of every real Brane execution produced by the offline evaluation pipeline (Chapter 6), distinct from the interactive Runs table above: each row records the intent, the source training file it came from, the exit code, and the execution time, letting a researcher inspect individual evaluation outcomes rather than only the aggregate metrics reported in the thesis.

- **Evaluation.** A model-comparison table summarising compile rate, execution rate, and output match rate (Chapter 4) across every evaluated model, paired with a per-intent browser that lets a researcher pick a single intent and see, side by side, the BraneScript every model generated for it and whether each one succeeded.

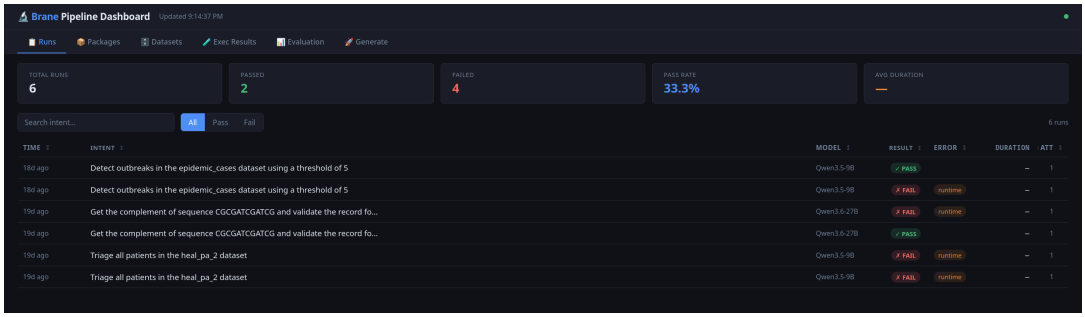

Figure 3.3: The Runs page, showing aggregate statistics and a searchable, sortable history of every intent previously submitted through the dashboard.

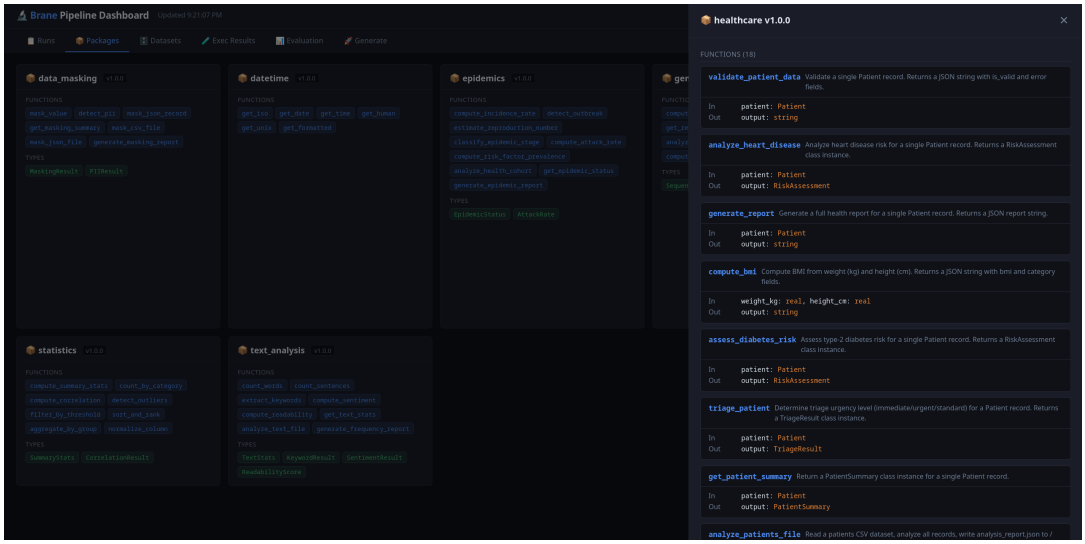

Figure 3.4: The Packages catalog, listing every package registered with Brane together with its exposed functions and parameter types; the Datasets catalog (not pictured) follows the same layout for registered datasets.

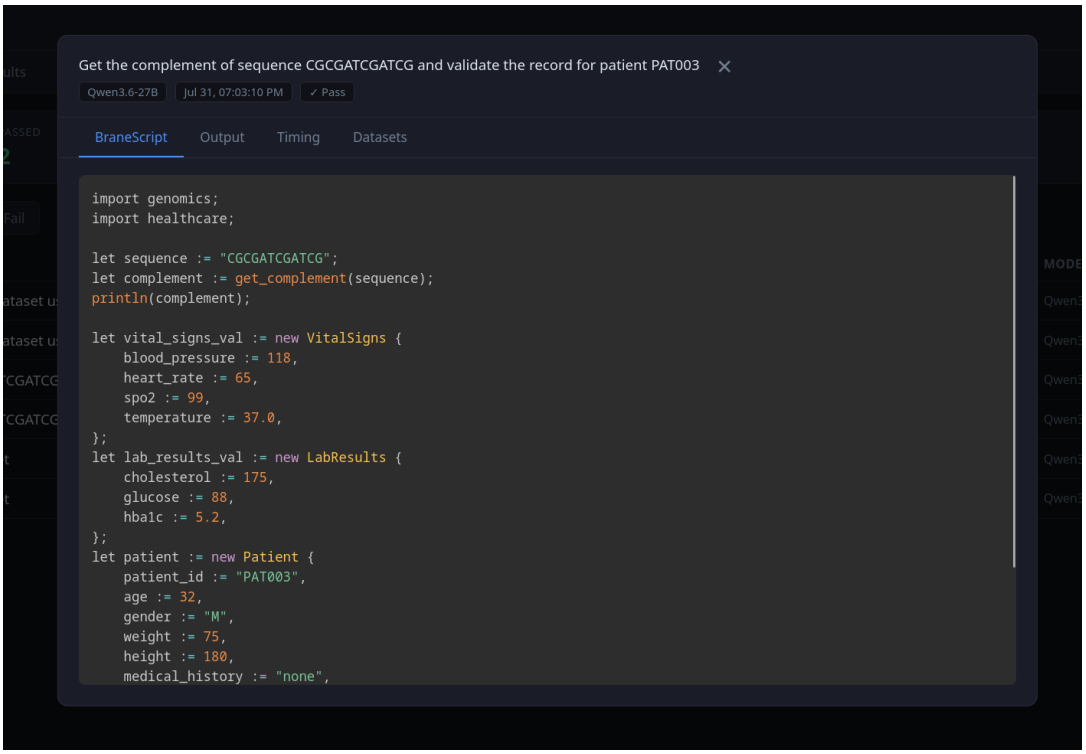

Figure 3.5: The Evaluation page, comparing compile rate, execution rate, and output match rate across evaluated models, with a per-intent browser for inspecting individual model outputs side by side.

Together, the preceding sections describe the same system from increasingly concrete angles: what a researcher submits and sees (Section 3.1), the logical components that turn an intent into an executed workflow (Section 3.2), where each of those components physically runs and why that placement keeps data at its owning site (Section 3.3), how they interact over a single request (Section 5.8), and what a researcher can see of the system's state beyond a single request (Section 3.4). Chapter 4 discusses the rationale behind the choices summarised in this chapter, and Chapter 5 describes how each component of Figures 3.1 and 5.1 is implemented.

4

Design

This chapter motivates the design decisions behind the architecture presented in Chapter 3. Each decision is presented together with the alternative it was chosen over and the evidence, where available, that justified the choice. The decisions collectively aim to serve **RQ1** by building a pipeline that gives the model the best possible grounding at generation time (**RQ2**), training it on domain-specific examples (**RQ3**), and caching verified outputs to avoid redundant generation ensuring reproducibility (**RQ4**).

4.1 Intent Decomposition

Before any retrieval or generation takes place, the pipeline runs the raw intent through a dedicated decomposition step: a short, separate prompt asks the same already-loaded generation model to break the intent into a numbered list of at most <N> concrete subtasks, phrased in plain English rather than code.

This step was introduced because retrieving directly against the raw intent tried to answer two different questions at once: understanding *what needs to be done* (which BraneScript constructs and package calls the request actually requires) and *how to express it* (what BraneScript syntax to use). A single embedding of a multi-part intent tends to average over all of its sub-goals rather than surfacing a chunk for each one. Stating explicitly, in English, that a request needs a package call, a loop over a dataset, and execution on a named node gives the retrieval stage (Section 4.2) several focused queries instead of one broad one; the per-subtask fallback search described there only exists because this decomposition happens first, sharpening the query before it is ever embedded, thereby directly shaping the retrieval-augmented context whose quality is examined in **RQ2**.

Two constraints shape what a sub-task is allowed to contain. First, each sub-task must correspond to one primitive BraneScript construct: an import, a function definition, a conditional, a parallel block, a return statement, and so on, all this deliberately scoped to what the *workflow* has to (and can) express, not what a package function computes internally: the decomposition prompt explicitly forbids sub-tasks such as "compute risk score" or "validate input", since those describe logic already implemented inside a package rather than something the generated BraneScript itself needs to construct from scratch. This keeps decomposition aligned with the retrieval it feeds: a sub-task describing internal package logic would only ever retrieve documentation about a function's *behaviour*, not the BraneScript needed to *call* it. Second, each sub-task is capped at 5–15 words and explicitly phrased as "a BraneScript manual search query" rather than a natural-language description, so the same text can be used, unmodified, as the query embedded for retrieval: decomposition and retrieval end up sharing one representation of a sub-task.

One consequence is worth naming explicitly: decomposition never produces code, as its rules explicitly ask for none, capping output at a bare numbered list of tasks.

4.2 Retrieval Design

The sub-tasks produced by decomposition (Section 4.1) are what this stage retrieves against. **G2** identified in Chapter 2.6 requires grounding natural-language translation in the concrete capabilities of the target WMS; how well this grounding works in practice, and whether richer context measurably improves translation accuracy, is what **RQ2** tries to answer. This capability surface has two very different shapes, and the design treats them differently rather than folding both into a single retrieval mechanism:

- **BraneScript syntax**, like control flows, function declarations, typed `class` declarations, the `parallel`, `on` constructs and more, are fixed by the language and identical for every intent regardless of which packages it invokes. The full reference has been synthesized to be small enough to be injected as static text into every generation prompt rather than retrieve. Because the reference does not vary with the intent, a retriever over it could only return a subset of a fixed, already-small document, and could end up saving at most a few hundred tokens per prompt, at the risk of omitting a construct the intent actually needs (e.g., dropping `parallel` on an intent that requires multi-site execution) if the retriever ranks it as irrelevant. This design choice was made to increased the reliability of the system, as always including the full reference removed that type of failure mode entirely.

- **Package and dataset context**, by contrast, grows without bound as more packages are registered with Brane, and only a small fraction of it is relevant to any single intent. The retrieval buys a real reduction in prompt size here without the correctness risk above. The **package/dataset retriever** indexes function signatures, parameter types, and usage examples for each package registered with Brane, plus metadata about registered datasets, and proceeds in two stages: it first scans the intent for a known package or dataset name (or one of its aliases, see below) and, on a match, retrieves *every* chunk for that package via a metadata filter, guaranteeing complete API coverage rather than a probabilistic top-k selection; only when no name is detected does it fall back to embedding similarity search, before deduplicating the combined result. In this thesis, this corpus is built from the seven packages used as the evaluation testbed (Chapter 6), but the retriever itself is not specific to those packages.

The retriever's index is built to grow one package at a time rather than being rebuilt whole. While treating package registration as a full rebuild would have been the simpler alternative, it was rejected because it ties the cost of adding a single package to the size of the entire catalogue, linearly increasing the time and compute required to register new packages as the system scales. For this reason, a newly registered package is embedded and inserted into the existing collection on its own, leaving every previously indexed package untouched while keeping the system scalable and responsive to new packages being added.

4.2.1 Chunking Granularity

An earlier design indexed each package's `container.yml` specification as a single chunk. This was found to frequently cause the model to select the wrong function within a package when a package exposed multiple similar functions. The design was revised to *per-function chunking*, where every function in every package's specification becomes its own retrievable chunk. This small change increases the precision of retrieval at the cost of a larger index, and was adopted because function-level ambiguity was the dominant source of retrieval-induced generation errors observed during development.

4.2.2 Package Aliasing and Hybrid Retrieval

Scientists describe their data using domain vocabulary ("heart disease data", "epidemic curve") rather than the internal package names (`healthcare`, `epidemics`) that BraneScript requires, so a name scan against package names alone would miss most intents. For this

reason, each package directory can define a `keywords.txt` file listing such domain terms, and the retriever loads these aliases from the collection's metadata at startup. The name-scan stage described above therefore matches on package/dataset names *or* their aliases before falling back to similarity search. This hybrid was chosen over pure similarity search because similarity search alone occasionally retrieved a semantically related but wrong package (e.g., `statistics` instead of `epidemics` for epidemic-curve intents), whereas exact name or alias matches, when available, are unambiguous.

4.3 Prompt and Feedback Design

The generation prompt combines the original intent, the retrieved package/dataset context, the static BraneScript syntax reference, and a system prompt encoding BraneScript-specific semantic rules not inheritable by the language syntax definition (e.g., that empty strings are invalid arguments, that `commit_result` must be called exactly once per named output, and that the model shall only return Branescript valid syntax). These rules were derived iteratively from observed failure modes rather than specified a priori, reflecting the practical reality that the invalid-output space of a fine-grained DSL is best characterised empirically. Both the system prompt and the user-message template are defined once and reused in the pipeline. Together with the retrieval design of Section 4.2, prompt construction is the other half of the context-quality question **RQ2** examines.

4.3.1 Failure Modes and Corrective Feedback

When a candidate workflow fails validation, the design reuses the same prompt-construction stage for a bounded number of retries (`MAX_RETRIES` $= 3$): the failing script and the error that caused it to fail are injected into the next prompt as context, rather than starting generation from scratch. This bounded retry loop is the system's concrete response to gap **G4**: rather than surfacing a raw failure to the user, it gives the model a structured opportunity to self-correct before the failure is reported. `MAX_RETRIES` is deliberately small and fixed rather than unbounded: if a model has not produced a valid script after three corrective attempts, the failure is treated as a genuine generation failure to be recorded and reported (Chapter 6), rather than something an arbitrarily long retry budget should paper over.

Compiler and runtime error messages from Brane itself are injected verbatim, rather than replaced with a generic "previous attempt failed" message, since Brane already reports which line and construct is at fault. The quality of this feedback matters directly for retry

success: a message naming the exact undefined variable or type mismatch gives the model a much stronger signal to correct against than a vague failure would, so the more descriptive Brane's own error output is, the more likely the next generation attempt is to succeed.

4.4 Typed Data Representation

BraneScript functions that return structured data represent it as tagged class instances (`["ClassName", {field: value, ...}]`) rather than as opaque JSON-encoded strings passed between functions. This design decision was adopted after observing that requiring the model to construct and parse ad hoc JSON strings inside BraneScript led to substantially more hallucinated or malformed intermediate values; typed class instances give the model a smaller, more constrained surface to generate correctly than free-form JSON would.

Letting the model serialise structured values as backslash-escaped JSON strings (e.g., `let p := "{\"iterations\": 100}";`) was the specific failure pattern the fine-tuned models kept reverting to, likely because it is the natural way to represent structured data in the general-purpose code the base models were originally trained on. The design responds to this on two levels: every structured value a package function accepts or returns is expressed as a declared `class`, giving a well-typed alternative wherever a JSON string might otherwise be used; and a dedicated check inspects each candidate script for this specific antipattern, triggering a retry whose corrective prompt shows the wrong pattern directly next to the correct typed-class equivalent.

The same typed representation is also why the design treats certain Brane values, like `Data` references to registered datasets and `IntermediateResult` values produced by package functions, as opaque to BraneScript itself: the model is specifically asked to *not* inspect or reconstruct their contents from within a script, only to pass them by reference between function calls and, when a result should be persisted, name it once via `commit_result`. Keeping these values opaque removes an entire class of intermediate outputs the model would otherwise have to represent correctly, at the cost of the model being unable to branch on their contents inside a script.

4.5 Dataset Design

The training and retrieval corpus is built around the seven packages described in Chapter 5. From those packages, a set of authored BraneScript examples is formulated, each

paired with a natural-language intent that describes the workflow it implements. Finally, the corpus is executed to capture the reference output for each example, so that the evaluation metrics of Section 4.5.1 can be computed against a known ground truth. The size and quality of this corpus is what **RQ3** ultimately depends on: lower coverage of BraneScript patterns result in the fine-tuned model to not be generalisable, and without verified reference outputs the evaluation of this generalisation becomes meaningless.

The resulting corpus accounted for a total of 584 synthetically generated examples through Claude Sonnet 5 and is then split into training and validation sets with a fixed 85/15 split and a fixed random seed, so that different fine-tuning runs compared in Section 6.2 are trained and validated on the exact same partition rather than one that varies run to run. The training set, used to update the model weights at training time, contains 497 examples, while the validation set, used to measure generalisation to unseen examples, contains 87 examples.

A more subtle problem surfaced specifically around the `datetime` package: nearly every one of its functions is, by design, a thin wrapper around the machine's current wall-clock time, so its output necessarily differs between the moment a training example's reference output was authored and any later moment the same BraneScript is re-executed. Early evaluation runs surfaced this directly: a generated script that was functionally identical to the reference would still be marked as an output mismatch, since the reference's stored timestamp could never match a timestamp produced at execution time, regardless of the model's actual competence, silently deflating the output-match rate for a reason entirely unrelated to translation quality. Since no BraneScript-level fix can make wall-clock time reproducible, the design instead replaced the affected examples from the corpus outright, alongside stripping incidentally non-deterministic timestamp fields returned by other packages (e.g. a generation timestamp inside the epidemics package's report output), so that the fine-tuning and output-match evaluation metrics reported in Section 6.2 only ever compare against genuinely reproducible ground truth.

4.5.1 Evaluation Metrics

Three key metrics are used throughout Chapter 6 to measure translation quality.

Compile rate is the fraction of generated scripts that pass the BraneScript compiler without error. BraneScript is statically typed, so any undefined variable, undefined function call, or syntax error is surfaced at compile time. A script that does not compile cannot be executed and is therefore of no practical use.

Execution rate is the fraction of generated scripts that execute to completion without a runtime error. A script may compile successfully but fail at runtime if it invokes a function with incorrect argument types, references a dataset name not present in the registry, or triggers an error inside a Brane package container.

Output match rate is the strictest metric: the fraction of all test examples for which the generated script's standard output is an exact string match of the reference output. Matching is case-sensitive and order-sensitive; outputs that contain the correct values but in a different order, or with different formatting, are marked incorrect. The metric is computed over all examples including those without a stored reference output, making it the most conservative measure of end-to-end correctness.

A complementary *lenient output match rate* is also reported for the 27B base model (Section 6.2.3). Under this criterion, a generated output is counted as correct if, after stripping any descriptive label prefixes from each output line or any additional formatting (e.g. `Mean: 52.27` → `52.27`), the remaining values match the reference exactly. This captures cases where the model produced the correct computation but formatted print statement calls with additional descriptive labels that the reference does not include.

Despite that, the strict metric remains the primary evaluation criterion: the lenient metric is reported as a sensitivity analysis to quantify how much the strict metric underestimates semantic correctness due to this formatting effect. Due to its nature, the lenient metric is computed only for a single model (Qwen3.6-27B) rather than for every model evaluated in Chapter 6 because of the additional manual effort required to strip labels from each output line.

4.6 Fine-Tuning Design

Even with RAG-injected documentation and carefully engineered prompts, base language models face a structural disadvantage when generating BraneScript as they were pretrained on large corpora of general-purpose code in which BraneScript does not appear. The model has no prior exposure to BraneScript's syntax, package call conventions, or the specific patterns that distinguish a valid workflow from a plausible-looking but incorrect one. RAG and prompt engineering can compensate for this to a degree by providing syntax examples and package documentation at inference time, but they cannot update the model's internal representations, forcing the model to still generalise from examples it has never been trained on, relying entirely on in-context learning.

Supervised fine-tuning (SFT) addresses this by continuing the pre-training process on a curated set of (intent, BraneScript workflow) pairs, directly adjusting the model's weights to associate the pipeline's input format with correct BraneScript output. This section describes the design intervention that tries to directly address **RQ3**.

Several design decisions shape the fine-tuning approach used to produce domain-adapted alternatives to the base LLMs.

4.6.1 Parameter-efficient fine-tuning (QLoRA)

Full fine-tuning on a 9 billion parameters (or a bigger model) is generally infeasible given the available HPC allocation, which was limited to a couple of hundred H100-GPU hours. This thesis instead uses 4-bit quantisation combined with Low-Rank Adaptation (LoRA), which updates a small number of adapter parameters while keeping the base weights frozen and quantised (exact quantisation and adapter hyperparameters are given in Chapter 5). This makes fine-tuning the second largest evaluated model (Qwen3.5-9B) tractable on the available GPU hardware, at the cost of a lower-rank approximation of the full fine-tuning update.

The adapter is attached to every linear projection in each transformer block: the four attention projections (query, key, value, output) and the three feed-forward projections (gate, up, down), rather than to attention alone, which is a common narrower default. As BraneScript is a small, rigid DSL, the difficulty of the base model centered around reproducing the exact syntax, keyword names, and package call signatures token by token. Restricting adaptation to attention only was rejected because it would have left the feed-forward pathway (the more likely source of the syntactic and package-signature errors this thesis specifically aims to reduce) frozen. The rank is kept the same across every attention and feed-forward matrix rather than allocated unevenly, and the LoRA scaling factor α is fixed at twice the rank throughout, following the widely used convention that keeps the effective adaptation strength comparable as rank is changed while avoiding introducing a second hyperparameter that would otherwise need its own justification and tuning budget alongside rank itself.

4.6.2 Training-inference prompt parity.

The fine-tuning corpus is built from the exact chat-formatted message list was previously constructed: a `system` turn carrying the BraneScript rules and syntax reference, a `user` turn carrying the intent and retrieved package/dataset context, and an `assistant` turn

holding the reference BraneScript itself, standing in for the reply the model is being trained to produce. The training script itself does no prompt construction of its own as it only applies the tokenizer's chat template to whatever messages the dataset already contains. This design decision was taken to teach the model to associate the reference BraneScript with the exact prompt text surrounding it. Routing training data through the identical function used by the pipeline and the evaluation script guarantees that whatever the model learns from is the same distribution of prompts it is later evaluated and used on, rather than a separately maintained approximation of it.

4.6.3 Response-only loss masking.

An early version of the training loop computed the language-modelling loss over the entire prompt, including the system prompt and retrieved context, not just the target BraneScript. This diluted the gradient signal for the stop token and caused generation to continue past the intended end of the script at inference time. The design was corrected to mask the loss so that it is computed only over the response tokens (the reference BraneScript), which is the standard approach for instruction-style fine-tuning and directly fixed the over-generation problem observed during development.

4.6.4 Training checkpointing

As HPC allocations are time-bounded and jobs can be pre-empted before training completes, the design checkpoints periodically and, on restart, resumes automatically from the latest checkpoint for that run rather than restarting from the base model (an explicit flag is available to force a clean restart when needed). These checkpoints are not only useful for resuming training in case a job is terminated due to time limits, but are also evaluated against the held-out validation split at the same interval, and the checkpoint kept at the end of a run is the one with the lowest validation loss and not necessarily the last one written. This decision guards against the case where additional epochs overfit the training split rather than continuing to improve.

4.7 Intent Provenance

In Section 2.2 we defined *intent provenance* the mapping from a natural-language request to the workflow it produced, alongside the standard retrospective record of how that workflow executed. As this type of provenance is identified as a gap, no existing provenance format addresses it (gap **G1**). For this reason, this thesis also aims to contribute to a design

that records this mapping for every submitted request. The pipeline, when run with data collection enabled, writes a per-request record containing the same fields: the original intent, the generated BraneScript, the model that produced it and which retry attempt it was, the pass/fail verdict, the specific error type on failure (a JSON-decoding antipattern, a Brane compilation failure, or a runtime failure), and the captured `stdout`/`stderr`/exit code for the command-line tool. Every generation event therefore leaves behind an explicit, machine-readable link between the intent entity, the generation activity, and the resulting workflow entity.

These per-request records are serialised as a PROV-DM document using a small vocabulary that specialises PROV-DM's existing concepts for this system, in the same way CWLProv and ProvONE specialise them for their own domains (Section 2.2). The mapping is straightforward: every object of interest becomes a `prov:Entity` and every action becomes a `prov:Activity`, connected by PROV-DM's standard relations. Concretely:

- The natural-language **intent** is an entity.
- A **TranslationActivity** uses that intent and generates a **BraneScriptWorkflow** entity, as it is associated with the model that produced the script.
- When the workflow is executed, an **ExecutionActivity** is informed by the translation (establishing causal order), uses the workflow entity, and generates an **ExecutionResult** entity carrying the results.

All entities and activities carry a qualified name derived from the originating request's identifier (e.g. `bt:workflow-<id>`), so that records from different requests remain unambiguous when combined.

Figure 4.1 shows this graph for a single request, together with the cache-hit extension described next.

In fact, a cache hit so far leaves no trace of its own: the pipeline returns the cached script directly without writing a record, so the intent-reproducibility relationship described before had nothing to serialise. To close this gap, the pipeline logs every cache hit, recording:

- the new intent that triggered the hit,
- the identifier of the run whose cache entry matched it, and
- the similarity score of the match.

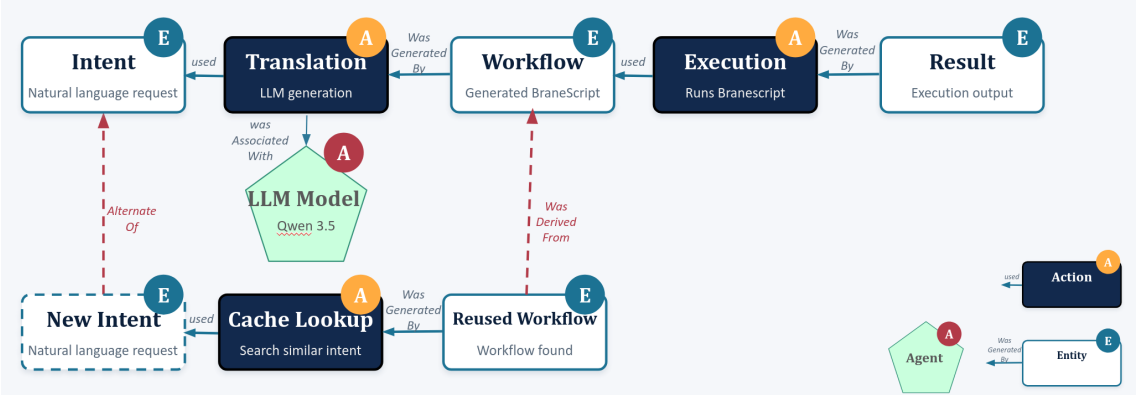

Figure 4.1: The PROV-DM graph produced for one request (solid, top row and agent) and for a cache hit against it (dashed, bottom row). The `Workflow` entity is typed `BraneScriptWorkflow` and the `Result` entity `ExecutionResult` in the exported document; both are shortened here for space. A cache hit does not run a new `TranslationActivity`: its intent is `alternateOf` the one that was already served, and the workflow it reuses `wasDerivedFrom` the original, rather than being generated independently.

The exporter then represents the cache hit as its own small provenance graph: the new intent is `alternateOf` the original one, and the workflow served in response `wasDerivedFrom` the workflow entity of the run that originally generated it, rather than from a fresh translation activity, closing the gap regarding the intent-reproducibility relationship from Section 2.2 identified as missing from every existing provenance format.

The export can finally be produced in PROV-N, PROV-JSON, PROV-XML, or a Graphviz rendering for visual inspection. What is not attempted in this thesis is a queryable provenance store: the export is a file produced on demand, not a graph database a researcher can query directly, and stable identifiers are scoped to a single run rather than persisting across a deployment's lifetime; both are left as future work (Chapter 7).

4.8 Semantic Cache Design

Recording intent provenance, as described above, is not by itself enough to make the system reproducible: the generation activity it records is performed by a non-deterministic LLM, so submitting the same intent again (or the same request paraphrased differently) could still produce a different workflow entity each time, with no relation recorded between the two events. The semantic cache is the mechanism that closes this gap (**G3**): the pipeline queries the cache before invoking the LLM at all, and if the same or a paraphrased intent has been served before, the cached BraneScript is returned deterministically instead of

triggering a fresh, potentially divergent generation event. Only on a cache miss does the pipeline fall back to LLM generation.

The design is based on a single fixed lookup-then-generate ordering, and not an adaptive or learning mechanism: the cache does not become more or less trusted over time, and no component of the pipeline tracks confidence across requests; a request that can be served from the cache simply never reaches the LLM, which also means it is never re-generated reducing the time it takes to serve the request and the associated compute cost. Whether this mechanism can genuinely serve as a reproducibility layer and where its fundamental limits lie is what **RQ4** asks.

An exact-string-match cache, based on looking up the intent verbatim rather than by embedding similarity, was considered but rejected as the primary mechanism. While this strategy avoids the false-positive risk discussed below, it only ever hits on the literal same sentence being submitted twice, defeating the purpose in a system whose stated premise (Chapter 3) is that a researcher can phrase a request however feels natural to them; on top of that, in a distributed and shared environment like what Brane aims to be, catastrophic divergence may occur, such as the case where two researchers describing the same analysis in different words would not share a cache entry under exact worded matching, resulting in different workflows being generated for the same underlying request.

4.8.1 Threshold Calibration and Cache Management

The design proposed above is a genuine trade-off: the same embedding space that enables useful generalisation across paraphrases also maps intents that differ only in a specific parameter (e.g., a numeric threshold or a dataset name) to very high similarity, since sentence embeddings capture propositional meaning rather than exact parameter values. For example, the intents *"Compute the average BMI for patients older than 40"* and *"Compute the average BMI for patients older than 65"* are propositionally near-identical, as they aim to perform the same action with a near identical structure, yet requiring different BraneScript: a cache hit here would silently return a workflow filtered at 40 in response to a request for 65, producing a wrong result without any compile or runtime error to signal the mismatch. The cache threshold is therefore an explicit precision–recall operating point that must be selected empirically: raising it reduces false positives at the cost of hit rate, while lowering it increases coverage at the cost of precision. The empirical characterisation in Chapter 6 identifies 0.92 as a reasonable default, but the trade-off itself cannot be eliminated through threshold tuning alone.

Characterising that operating point requires intents that are genuine paraphrases of an existing cache entry, as they should just be worded differently while still referring to the same underlying request. To avoid overfitting in the training phase, the initial curated dataset was meant to be varied and sparse rather than to contain near-duplicates intents. For this reason, to close this gap, every training intent is paraphrased 3 additional times while keeping its reference BraneScript unchanged. This accounted a total of 1,752 paraphrased intents (used only for cache evaluation) and its purpose is exclusively to sweep the similarity threshold and measure, for each candidate value, how often a paraphrase is matched to any cache entry at all and how often it is matched to the *correct* one giving the precision.

4.8.2 Cache Evaluation Metrics

Three metrics are used to characterise the cache threshold in Section 6.5.

Hit rate is the fraction of queries for which the cache returned any result, i.e. a cached entry whose similarity to the query exceeded the threshold.

Correct rate is the fraction of all queries for which the returned BraneScript matched the reference for that intent. A query that misses the cache counts as incorrect for the purpose of this metric.

False-positive rate is the fraction of queries where the cache returned a result but it did not match the reference, indicating that a semantically distinct intent was incorrectly retrieved above the threshold.

5

Implementation

This chapter describes how the system designed in Chapter 4 is realised in practice. Following the component overview of Chapter 3, the pipeline is implemented as a Python application with two entry points: a command-line interface and an interactive dashboard. Both entry points share the same orchestrator, retrieval stack, and semantic cache, while fine-tuning runs separately as a batch job on Snellius.

5.1 Pipeline Orchestrator

The orchestrator (`pipeline.py`) loops over the generation–validation–execution cycle described in Section 3 for at most three attempts per request. Before attempting generation, it queries the semantic cache (Section 5.5) for the intent and returns the cached script and result immediately on a hit; on a miss, it decomposes the intent into sub-tasks via `IntentDecomposer` before retrieval, as described in Section 4.1. This step reuses the generation model already loaded in memory rather than loading a second one, so its cost is one extra generation call rather than one extra model; a `-skip-decomposition` flag in the orchestrator disables the step outright, isolating its cost from the rest of the pipeline so its effect on the generated BraneScript can be assessed independently of retrieval and generation. A candidate workflow is checked for valid function and argument usage against the retrieved package context before `execute_workflow()` ever submits it to Brane, which compiles and executes it as a single command and returns `stdout`, `stderr`, and an exit code that distinguishes a compilation failure from a runtime failure. On success, the (intent, script, result) triple is written back to the semantic cache and also persisted to disk for later inspection and reuse. Intents the orchestrator judges too ambiguous to act on are

handled separately: rather than attempting generation, it returns a clarifying question to the user.

Setting up a pipeline run once at start-up rather than per-request loads the embedding model, builds the single package retriever shared across the run, and connects to the semantic cache's ChromaDB collection; if that connection fails, the failure is caught and logged and the run proceeds with caching disabled rather than failing outright, while the BraneScript syntax reference (`data/syntax_reference.md`) is loaded once as plain text and needs no embedding model at all: at roughly 450 lines (2,000 tokens) it is small enough to be included in full in every prompt without approaching context-length limits. A command-line flag allows the cache to be bypassed explicitly for a given invocation.

5.2 Knowledge Base and Retrieval

`src/knowledgeBase.py` builds and queries the single vector store introduced in Section 4.2, called `brane_pkg_db`, a ChromaDB collection persisted to disk and accessed through LangChain's `Chroma` and `HuggingFaceEmbeddings` wrappers, the same LangChain abstractions used throughout the retrieval stack. Package specifications (defined as `container.yml` files following Brane package definition) are parsed and exploded into per-function `Document` chunks, each annotated with the owning package name and a short usage hint, and embedded with `all-MiniLM-L6-v2`. On top of that, each package's `keywords.txt` file (Section 5.3) is loaded as its own short, unchunked document tagged with the owning package name; `PkgRetriever` reads these documents back out of the collection at startup and builds an in-memory alias table from them, so the hybrid retrieval strategy described in Chapter 4 never has to guess a package's domain vocabulary from its function signatures alone. BraneScript syntax is instead finally loaded via `data/syntax_reference.md`, a curated file of BraneScript constructs of interest to the generation model, which is small enough (approximately 2,000 tokens) to be included in full and formatted directly into the generation system prompt for every request.

As briefly mentioned in (Section 4.2), in order to support the continuous development of the systems, the knowledge base does not need to be rebuilt from scratch when new package or dataset content is added. `knowledgeBase.py` allows the addition of embeddings directly to the existing `brane_pkg_db` collection, so registering a new package only requires embedding that package's own documents and inserting them, leaving every previously indexed package untouched. This means that adding a new package, or editing an existing package's `container.yml`, `keywords.txt`, or `README.md`, only costs the time to embed the

changed content itself and not the whole catalogue. The current implementation of the pipeline doesn't detect a stale knowledge base automatically, so re-running the update is still a manual part of the package-authoring workflow rather than one triggered by pipeline startup.

5.3 Brane Packages

The seven packages implemented under `packages/` as the evaluation testbed for this thesis were purpose-built to provide a concrete end-to-end domain on which to evaluate the system.

Every package follows the same structure: a `container.yml` declaring the package's functions, their typed input parameters and return values, and any custom `class` types it exposes to BraneScript; a single Python source file implementing the actual logic; and a `keywords.txt` listing the domain vocabulary a scientist might use to refer to the package without naming it directly (Section 4.2). Additionally, the packages include a `README.md` documenting, in prose, the package's Brane dispatch conventions and the meaning of each exposed function and class, written from a human maintainer with the goal of extending the package documentation.

Each package follows a common convention: the container's entry point reads the invoked function name from `sys.argv[1]` (declared in the package's `command.args` specification), rather than relying on environment variables, and routes to the corresponding Python function. Structured return values are serialised as tagged class instances (`["ClassName", {...}]`), consistent with the typed data representation decision in Chapter 4. Every package is of Brane's `ecu` (*executable code unit*) kind, for which Brane itself generates the Docker container image from the `dependencies`, `install`, and `files` fields declared in `container.yml`.

5.4 Fine-Tuning Infrastructure

5.4.1 Training the models

`src/fine_tuning/train.py` runs fine-tuning with the base model supplied as a configuration value rather than hard-coded, so the same training code produced the Qwen3.5-9B and Qwen3.6-27B adapters compared in Chapter 6 without any per-model changes. Fine-tuning was limited to these two model sizes due to GPU memory and allocation constraints on Snellius: training a 27B model with 4-bit quantisation and LoRA already requires a full

H100 GPU (80 GB VRAM), leaving larger models (70B and above) outside the available resource envelope for this project. Default hyperparameters are LoRA rank 16 (alpha 32), 3 epochs, an effective batch size of 8, and a learning rate of 2×10^{-4} on a cosine schedule, using 4-bit quantisation through Unsloth where available and a plain Hugging Face/PEFT fallback otherwise; the response-only loss masking described in Chapter 4 is applied at this stage. Training runs on Snellius with a single H100 GPU and resumes automatically from the latest checkpoint if a job is pre-empted, rather than restarting from the base model. Every run's full hyperparameter set and training progress are logged alongside the resulting adapter so that multiple SFT runs against the same base model remain distinguishable later.

5.4.2 Tokenisation and Sequence Length

Training examples are formatted using the same prompt template as inference: a system prompt followed by the user intent, with the BraneScript workflow as the expected assistant response. The maximum sequence length is set to 2048 tokens, which accommodates the full prompt (system prompt + syntax reference + retrieved package context + intent) plus the expected BraneScript output without truncation for all but the longest multi-step workflows in the training set. Truncation would silently corrupt training examples by cutting off the BraneScript output mid-workflow (as experienced in some runs), producing malformed targets that the model would learn to reproduce; the 2048-token limit was chosen by inspecting the length distribution of the training set and confirming that all examples fit within it.

Response-only loss masking is applied so that the model is penalised only on the BraneScript output tokens and not on the prompt. Without this, the model would spend training capacity learning to reproduce the (fixed) system prompt and retrieved context, diluting the signal from the BraneScript generation task itself.

5.4.3 Adapter Merging

After training, the LoRA adapter weights are merged into the base model weights using `model.merge_and_unload()`, producing a single standalone model checkpoint that can be loaded for inference without the PEFT library. This avoids the runtime overhead of applying the adapter at each forward pass and simplifies deployment: the merged model is identical in interface to the base model and can be loaded with the standard Hugging Face `AutoModelForCausalLM` API. The unmerged adapter is retained alongside the merged

checkpoint so that further fine-tuning runs can resume from the adapter state rather than re-merging.

Evaluation is handled by `evaluate.py`, which computes the three metrics used throughout Chapter 6 and described in Subsection 4.5.1. Each evaluation run records which training configuration produced the model under test, so that the dashboard (Section 5.6) can distinguish between multiple SFT runs of the same base model without additional bookkeeping.

5.5 Semantic Cache

5.5.1 Embedding Model

Intent similarity is computed using `all-MiniLM-L6-v2`, a 22M-parameter sentence-embedding model that maps text to 384-dimensional unit vectors. Cosine similarity between the query and stored vectors is used as the retrieval score. The model was chosen for its inference speed and semantic quality on short English sentences, which matches the typical intent length of 10–30 words. Its known limitation, which revolved around encoding topic rather than precise parameter values, is the direct cause of the false-positive floor noticed in Section 6.5 and further discussed in Section 7.4.

5.5.2 Persistence

The cache collection is stored on disk and survives server restarts without needing to be rebuilt. The 584 seeded intents and any workflows added during interactive use are retained across process boundaries, meaning the first request for a known intent always hits the cache rather than requiring a prior successful generation to populate it. The cache collection is thought to be deployed to remote servers, but the current implementation runs it locally on the same machine as the dashboard and orchestrator, which is sufficient for the single-user research deployment.

5.5.3 Concurrency

The cache does not implement explicit application-level locking. The underlying storage engine serialises writes internally, so concurrent stores will not corrupt the collection, but a lookup that begins just before a concurrent store completes may miss the new entry. In the current deployment this race window is narrow, as the only runtime caller of store is the interactive entry point handling one request at a time. A multi-user production

deployment would require either an explicit lock or a migration to a client-server storage mode.

5.5.4 Cache Invalidation

The cache exposes an invalidation operation that removes all stored entries whose similarity to a supplied intent exceeds the current threshold. The intended use case is package maintenance: if a new version of a Brane package changes a function signature, any cached workflow calling the old signature will silently produce a compile failure on the next execution. Invalidating with a representative intent for the affected package forces fresh generation for subsequent requests. This is a manual operation in the current implementation; automatic invalidation on package version change is left as future work.

The cache is seeded with all 584 training intents at startup, so the first request for any known intent always hits the cache. The evaluation pipeline used throughout Section 6 deliberately bypasses the cache entirely, as its purpose is to measure raw model performance rather than cache behaviour.

5.6 Dashboard

The dashboard backend is implemented with Flask. The frontend is a single-page application served as static files by the same process, keeping the deployment to a single `python server.py` invocation on the researcher's machine.

Snellius job submission is handled over SSH via subprocess calls: the server connects to Snellius using a key-based SSH configuration read from environment variables and invokes the SLURM batch script remotely. The status polling works the same way: the server queries the SLURM job queue over SSH at each poll interval and maps the scheduler state to the dashboard's own Queued $\rightarrow$ Generating $\rightarrow$ Executing $\rightarrow$ Done progression. No persistent connection to Snellius is maintained and each status check is an independent SSH call.

Results are finally communicated back through a shared file-based job directory: the generation worker on Snellius writes its output to a known path, and the dashboard backend reads it on the next poll. This decoupling contributed to the design decision that the dashboard does not need to stay connected to Snellius while the job runs.

5.7 Job Watcher

The job watcher is a lightweight polling daemon that runs on whichever machine has network access to the Brane instance. In the current deployment, it is the researcher's own machine rather than the Snellius compute node that ran generation, since Snellius does not support Docker. This separation is intentional: generation (the LLM call and decomposition) happens on the HPC cluster, but Brane execution must happen where Brane is installed and reachable, which need not be the same machine.

The daemon monitors a shared job directory for BraneScript files deposited by the generation pipeline. For each pending script it invokes the Brane compiler and runtime directly, so the exit code and output are always Brane's own verdict rather than a re-implementation of its compilation or execution logic. The result, whether success or failure, standard output, and any error messages, is written back to a result file that the dashboard picks up and, on failure, feeds into the retry sequence shown in Figure 5.1. Keeping execution separate from the dashboard process limits the dashboard's responsibilities to submission, polling, and presentation, and makes it possible to run Brane on a different host from the server without changing any pipeline logic.

5.8 Interaction Sequence

Figure 3.1 shows the components involved in a single request but not the timing of the messages between them, in particular the asynchronous submit/poll/retry pattern of the interactive dashboard entry point. Figure 5.1 makes that timing explicit for the case a researcher exercises in day-to-day use: the researcher's browser, the dashboard backend, the semantic cache, the generation worker running as a batch job on Snellius, and the Brane runtime that ultimately compiles and executes the workflow.

The diagram distinguishes two branches. On a **cache hit** (top frame, steps 1–3), the dashboard looks up the submitted intent against the semantic cache and, on a similarity match, returns the cached script and its previously recorded execution result directly to the researcher; nothing on Snellius or Brane is touched, and no model is loaded. On a **cache miss** (bottom frame, steps 4–14), the dashboard instead submits a SLURM batch job to Snellius and immediately hands the researcher back a request identifier and a "submitted" status (steps 5–6), rather than blocking the HTTP request for however long generation and execution take; the researcher's browser polls a status endpoint for that identifier

until the job completes (step 7). On the Snellius side, the worker performs the process-architecture steps described in Section 3.2 in order – decomposing the intent, retrieving package/dataset context, and generating a candidate script (step 8) – and then submits that script to the Brane runtime with a single `brane workflow run` call (step 9), receiving back an exit code and captured stdout/stderr (step 10). The worker writes this result where the dashboard can find it (step 11); if execution failed and retries remain, the dashboard resubmits the same request with the failure fed back into the prompt as corrective context (step 12), returning to step 5. Once a script executes successfully, the result is written to the semantic cache (step 13) before the final script and outcome are returned to the researcher (step 14).

The command-line entry point (`pipeline.py`) follows the same overall shape – cache lookup, decomposition, retrieval, generation, cache store on success – but differs from the diagram in one respect: it has no equivalent of step 7's polling, since it runs synchronously to completion on the researcher's own machine instead of via a remote batch scheduler.

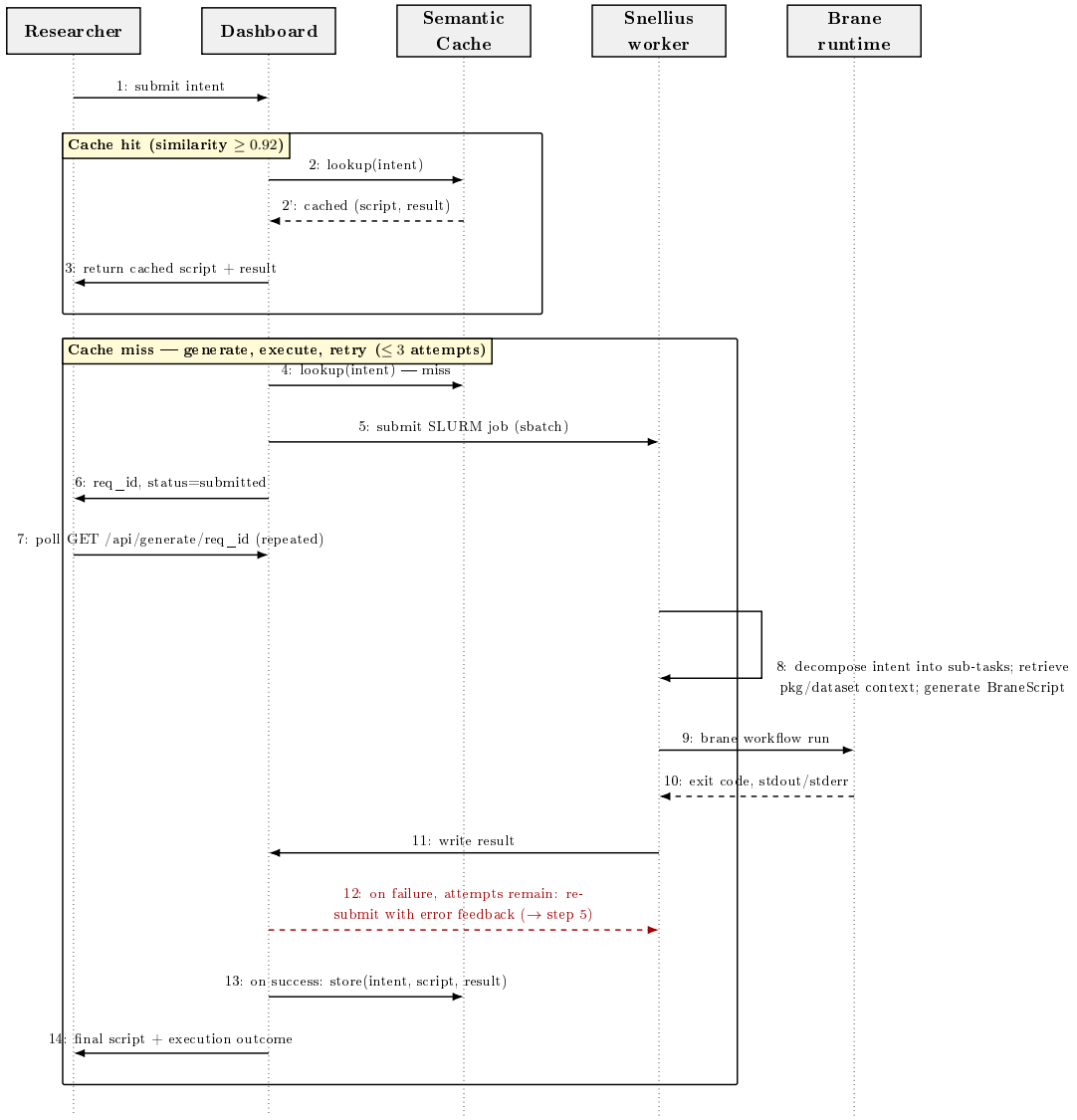

Figure 5.1: Sequence of interactions for the interactive (dashboard) entry point, discussed in the text above.

6

Evaluation

This chapter presents the experimental setup, evaluation runs, and results for all four research questions. Interpretation of the results and their implications are deferred to Chapter 7.

6.1 Experimental Setup

Dataset. The evaluation dataset consists of 584 unique intent–BraneScript pairs, which were then split into a training set of 497 examples and a held-out test set of 87 examples. The test set was not used during any stage of training or system development, as it was reserved exclusively for evaluation. The dataset covers the six Brane packages used in this project (`healthcare`, `genomics`, `epidemics`, `statistics`, `text_analysis`, `data_masking`, excluding `datetime`), with intent complexity ranging from single-package, single-function invocations to multi-package workflows with intermediate results passed across five or more steps. An additional 1752 paraphrases (three per intent, generated by Qwen3.5-4B) were produced for the semantic cache evaluation (Section 6.5).

Models. Three base models are evaluated as zero-shot baselines: Qwen3.5-4B, Qwen3.5-9B, and Qwen3.6-27B, all from the Qwen3 family to enable clean attribution of performance differences to model scale. Two supervised fine-tuning runs are also evaluated: a 9B model fine-tuned for three epochs and five epochs (`qwen3.5-9b-3ep` and `qwen3.5-9b-ep5`).

Pipeline. Unless stated otherwise, every evaluation run uses the complete three-stage pipeline: (1) the *intent decomposer* reformulates the user's natural-language intent into a structured set of sub-tasks with explicit package and function candidates; (2) the *RAG retriever* fetches the most relevant package documentation and example BraneScript fragments per sub-task from the retrieval index; and (3) the *generator* produces a BraneScript

workflow conditioned on the decomposed intent and retrieved context. The decomposer ablation in Section 6.3.2 is the only run that deviates from this configuration.

Metrics. The three primary metrics used throughout this chapter, compile rate, execution rate, and output match rate, were defined in Section 4.5.1. The cache-specific metrics (hit rate, correct rate, and false-positive rate) were defined in Section 4.8.2.

Hardware. All generation and fine-tuning runs execute on Snellius, the Dutch national HPC facility, on a single H100 GPU node allocated via SLURM. BraneScript execution and metric collection run on a local workstation with Docker-deployed Brane services.

6.2 RQ1: Translation Accuracy Across Model Scales

6.2.1 Main results

Figure 6.1 shows compile rate, execution rate, and output match rate for all three base models evaluated against the 497 test intents.

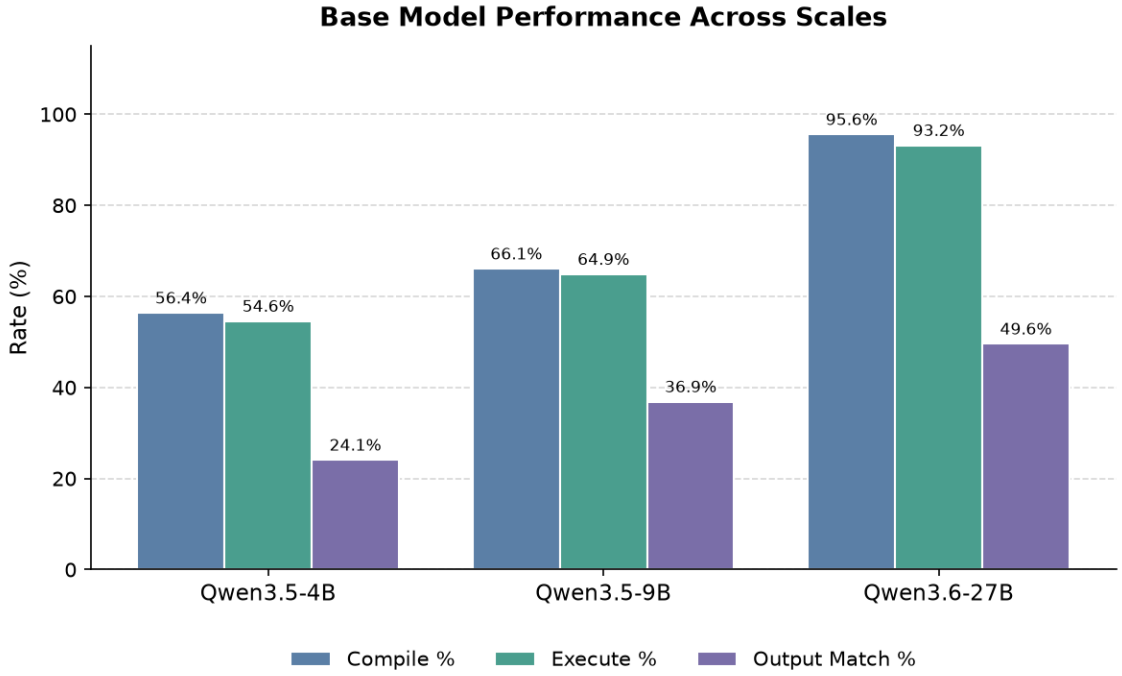

Figure 6.1: Compile rate, execution rate, and output match rate for the three base models. Each metric group is shown as a cluster of three bars, one per model ($n = 497$).

All three metrics improve monotonically with model scale. The 4B model achieves a compile rate of 56.4%, an execution rate of 54.6%, and an output match rate of 24.1%. At 9B parameters, compile rate rises to 66.1% and output match to 36.9%. The largest

jump occurs at the 9B-to-27B transition: compile rate reaches 95.6%, a gain of nearly 30 percentage points, while execution rate follows closely at 93.2%. Output match at 27B stands at 49.6%, an improvement of 12.7 percentage points over the 9B model.

The gap between execution rate and output match rate widens with scale. For the 4B model the gap is 30.5 percentage points; for the 27B model it is 43.6 percentage points. This indicates that as model scale increases, the dominant remaining failure mode shifts from the script failing to compile or execute to the script producing a syntactically and semantically valid workflow whose output does not exactly match the reference.

6.2.2 Error type breakdown

To understand *where* each model fails rather than merely *how often*, every failed evaluation was classified into one of five categories:

- *Compile: syntax error* — a parse or structural error detectable by the BraneScript grammar alone.
- *Compile: undefined variable* — a reference to a variable that was not previously bound in the script's scope.
- *Compile: undefined function* — an invocation of a function not exported by the imported packages.
- *Runtime error* — a script that compiled successfully but failed during Brane package execution.
- *Output mismatch* — a script that executed without error but produced output that did not match the reference.

Figure 6.2 shows the distribution of these categories for the three base models.

The 4B model shows the most compile-time failures overall. Undefined variable errors account for 24.1% of all 497 test examples, making it the single largest failure category. Syntax errors add a further 17.5%, and output mismatch accounts for 30.5%. Only 24.1% of scripts produce a correct result. Runtime errors and undefined function errors are minor contributors at 1.8% and 1.4% respectively.

The 9B model's profile shifts noticeably: undefined variable errors persist at 24.5% but syntax errors drop sharply to 4.8%, indicating that the larger model handles basic BraneScript grammar more reliably. Undefined function errors rise slightly to 3.8%. Output mismatch decreases to 28.1% and the correct fraction rises to 36.9%.

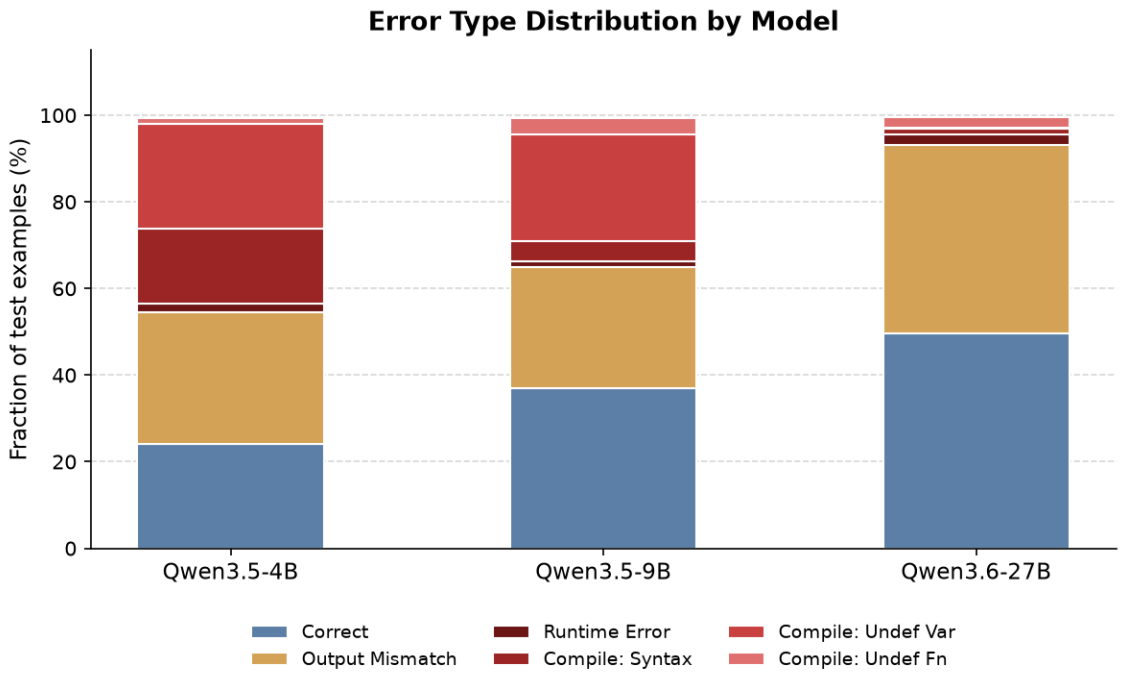

Figure 6.2: Error type breakdown for the three base models. Each bar shows the fraction of all 497 test examples in each outcome category. Shades of red distinguish the four compile-error sub-types; amber indicates output mismatch; blue indicates correct output.

The 27B model largely eliminates compile failures. Syntax errors fall to 1.4%, undefined variable errors to 0.2%, and undefined function errors to 2.4%. The dominant failure category becomes output mismatch at 43.6%: the model consistently produces scripts that compile and execute, but whose output diverges from the reference. Runtime errors account for 2.4%. The findings are summarized in the Table 6.1 below.

Table 6.1: Error type breakdown by percentage of all 497 test examples for each base model.

Error category	Qwen3.5-4B	Qwen3.5-9B	Qwen3.6-27B
Correct output	24.1%	36.9%	49.6%
Output mismatch	30.5%	28.1%	43.6%
Runtime error	1.8%	3.2%	2.4%
Compile: undefined var	24.1%	24.5%	0.2%
Compile: undefined func	1.4%	3.8%	2.4%
Compile: syntax error	17.5%	4.8%	1.4%

6.2.3 Output Match Sensitivity: Strict vs. Lenient Evaluation

The output match metric used throughout this evaluation is a strict exact-string comparison between the generated script's standard output and the reference output. While this is the most objective and reproducible criterion, manual inspection of the Qwen3.6-27B base model's mismatches revealed a recurring pattern: 26 of its 159 output mismatches are cases where the generated script produced the correct numerical or categorical values but wrapped them in descriptive labels absent from the reference. Two representative examples illustrate the pattern:

Reference	Generated
`5`	`Total PII matches:  5`
`true`	`PII found:  true`
`52.2667`	`Mean:  52.2667`
`14.5715`	`Std Dev:  14.5715`
`15`	`Count:  15`

Table 6.2: Two examples where the generated script produces correct values but adds descriptive labels not present in the reference output.

In both cases the underlying computation is correct and the difference lies in how the result was formatted by the generated `println` calls.

To quantify the effect of this formatting bias, a lenient matching criterion was applied to the 27B base model's results: an output is counted as correct if, after stripping any

`Label:` prefixes from each output line, the remaining values match the reference exactly. Under this criterion, 16.4% of the 159 failed output evaluations are reclassified as correct, raising the output match rate from 49.6% to 66.0%. Compile and execution rates are unaffected, since the scripts compiled and executed successfully in all cases.

This analysis is limited to the 27B base model, as it exhibited the highest overall translation quality and therefore the largest proportion of near-correct outputs. The pattern is worth noting because it results in the strict metric slightly underestimating the semantic correctness of the 27B model's outputs. All presented primary results in this chapter use the strict criterion, as it is the most reproducible and requires no heuristic label-detection logic.

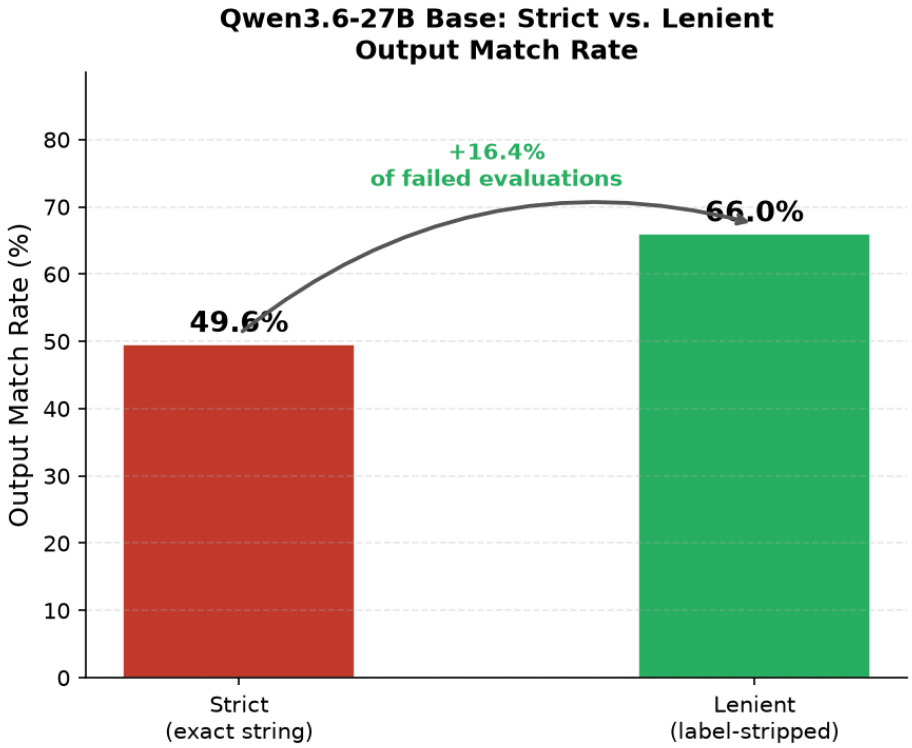

Figure 6.3: Qwen3.6-27B base model output match rate under strict (exact string) and lenient (label-stripped) evaluation criteria. 16.4% of the 159 failed output evaluations are reclassified as correct under the lenient criterion.

6.3 RQ2: Effect of Retrieval, Prompt Engineering, and Decomposition

6.3.1 Two evaluation rounds

An initial evaluation run was conducted with an early version of the system prompt and retrieval configuration. Several engineering issues were subsequently identified and corrected:

- The system prompt permitted empty-string function arguments and under-specified `commit_result` usage.
- RAG package aliases were insufficiently precise, causing the retriever to return documentation for the wrong package. Package keywords were also added to the retrieval index to improve matching precision.
- The post-processing step that strips stray markdown tokens from generation output was incomplete, causing otherwise valid scripts to fail at the BraneScript lexer. Additionally, a repetition penalty was added to generation to suppress degenerate repetition loops in which the model emitted sequences of backtick characters after otherwise valid BraneScript.

The same model weights were re-evaluated after all three fixes. Figure 6.4 shows a grouped bar chart with one pair of bars per model: the left bar (slate blue) represents the pre-fix output match rate, and the right bar (teal) represents the post-fix rate. For every model, the teal bar is substantially taller than the slate blue bar, indicating a consistent improvement across the board. The gap is largest for the 27B model, achieving the highest post-fix rate of the three, and smallest for the 4B model in absolute terms. Across all three models the output match rate increases substantially: from 3.6% to 24.1% for the 4B model, from 6.0% to 36.9% for the 9B model, and from 13.5% to 49.6% for the 27B model. The relative ordering of the three models is preserved between runs, with the 27B model consistently outperforming the smaller models on this metric.

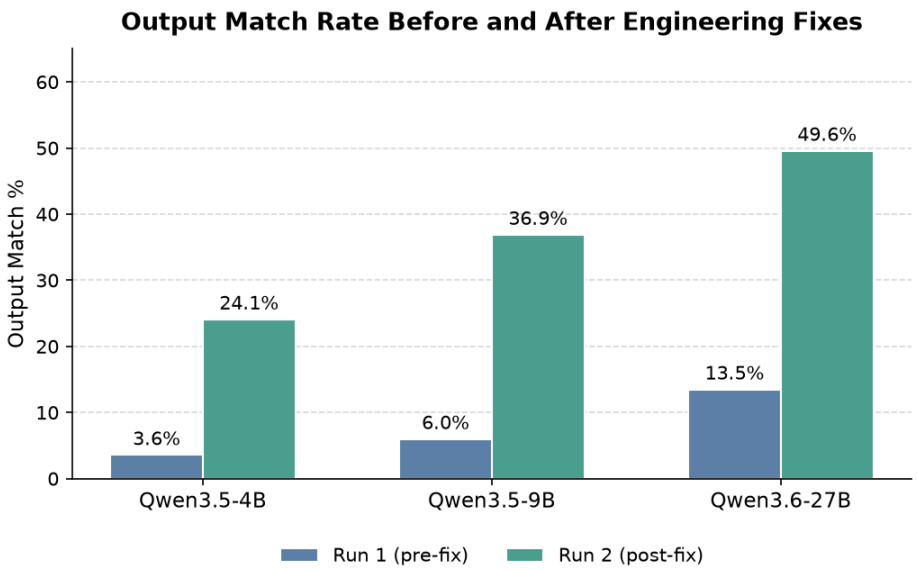

Figure 6.4: Output match rate before (Run 1, pre-fix) and after (Run 2, post-fix) the three engineering fixes ($n = 497$ in both runs). Identical model weights were used in both runs.

The 27B model's execution rate rises from 37.9% to 93.2% between the two runs using identical model weights. The 9B model's execution rate rises from 19.5% to 64.9%, and the 4B model's from 6.2% to 54.6%. Table 6.3 reports all three metrics across both rounds.

Model	Round	Compile (%)	Execution (%)	Output Match (%)
Qwen3.5-4B	Run 1 (pre-fix)	14.9	6.2	3.6
	Run 2 (post-fix)	75.9	54.6	24.1
Qwen3.5-9B	Run 1 (pre-fix)	34.5	19.5	6.0
	Run 2 (post-fix)	83.9	64.9	36.9
Qwen3.6-27B	Run 1 (pre-fix)	55.2	37.9	13.5
	Run 2 (post-fix)	95.6	93.2	49.6

Table 6.3: Compile, execution, and output match rates for all three models before (Run 1) and after (Run 2) the prompt and retrieval engineering fixes. Identical model weights were used in both runs ($n = 497$).

6.3.2 Intent decomposition ablation

The intent decomposer is the first stage of the pipeline. As described in Section 4.1, its goal is to reformulate the request into a structured set of sub-tasks, each annotated with explicit package and function candidates. This structured representation is then passed to

the RAG retriever and the generator. To isolate the contribution of this stage, an ablation run was conducted in which the decomposer was entirely disabled: the raw intent string was passed directly to the retriever and generator without any type of reformulation while keeping the other pipeline components identical to the full pipeline configuration. The experiment used Qwen3.5-9B as the generator.

Figure 6.5 presents the results as a grouped bar chart with two groups on the x-axis: full pipeline as the first one, and pipeline with no decomposition step as the second. For each run, the compile rate (slate blue), execution rate (teal), and output match rate (purple) is evaluated. Comparing the two groups, all three bars are shorter in the no-decomposition group, indicating a consistent drop across every metric when the decomposer is removed.

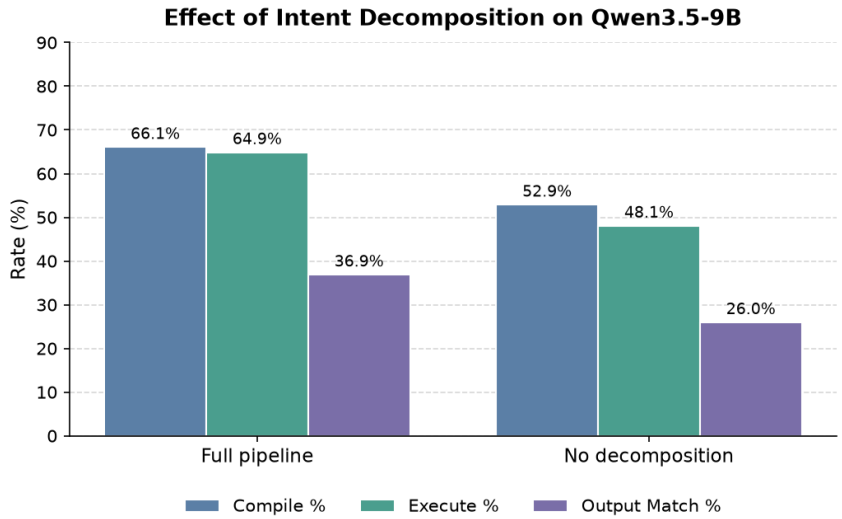

Figure 6.5: Effect of disabling the intent decomposition stage on Qwen3.5-9B ($n = 497$). Slate blue = compile rate; teal = execution rate; purple = output match rate.

Compile rate falls from 66.1% to 52.9%, a drop of 13.2 percentage points. Execution rate falls from 64.9% to 48.1%, a drop of 16.8 percentage points. Output match rate falls from 36.9% to 26.0%, a drop of 10.9 percentage points. The degradation is consistent across all three metrics, with execution rate showing the largest absolute drop and output match the smallest.

6.4 RQ3: Effect of Supervised Fine-Tuning

Figure 6.6 reports results for 2 SFT variants alongside the 9B base model, all evaluated on the 87-example evaluation test set. `qwen3.5-9b-ep5` was fine-tuned for five epochs on the deduplicated training set (497 examples) using the corrected system prompt, while `qwen3.5-9b-ep3` was fine-tuned for three epochs. Both training runs share the following hyperparameters:

- **Method:** QLoRA (quantised LoRA, 4-bit base model)
- **LoRA rank:** 16; **LoRA α:** 32
- **Learning rate:** 2×10^{-4} with cosine scheduler
- **Batch size:** 2 per device, gradient accumulation 8 steps (effective batch 16)
- **Max sequence length:** 2048 tokens
- **Optimiser:** AdamW with default betas

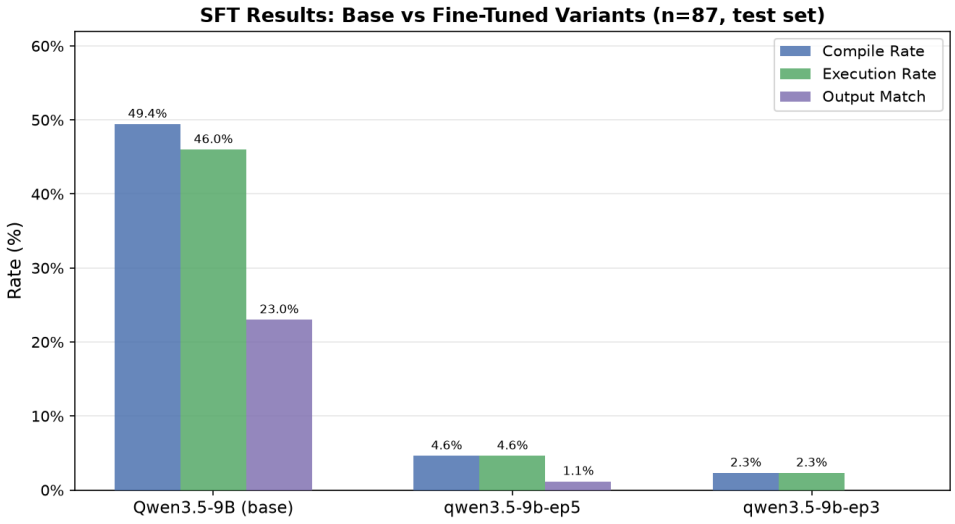

Figure 6.6: Compile, execution, and output match rates for the Qwen3.5-9B base model and both SFT variants ($n = 87$, test set).

Compared to the 9B base model (compile: 49.4%, execution: 46.0%, output match: 23.0%), both SFT variants show a near-total collapse across all three metrics. The 5ep model collapses to 4.6% compile, 4.6% execution, and 1.1% output match (1 correct script

out of 87). The ep3-new model performs similarly: 2.3% compile, 2.3% execution, and 0.0% output match.

Further inspection of the generated outputs reveals two recurring failure patterns. First, 95.4% of ep5 failures and 97.7% of ep3-new failures are compile errors with a `reached end-of-file unexpectedly` diagnostic, meaning the generated scripts are syntactically incomplete, as the model produces recognisable BraneScript structures but terminates before closing the final block, suggesting the generation is cut off at the output length limit. Generated scripts average 1,800–2,000 characters, consistent with hitting a token ceiling mid-script. Second, 17 of 87 scripts generated by ep5 and 15 of 87 generated by ep3-new contain at least one `import` statement referencing a non-existent package (e.g. `import intent`, `import risk_framework`, `import data`). These hallucinated imports would cause compile failures independently of the truncation issue. Third, many of the generated scripts contain extensive inline reasoning embedded as code comments, such as multi-line blocks explaining why a particular approach was chosen, contradicting themselves across successive comment lines, and in some cases repeating the same `commit_result` call multiple times with different arguments. This behaviour is absent from the base model's output.

6.5 RQ4: Semantic Cache as a Reproducibility Layer

The semantic cache stores previously generated BraneScript workflows indexed by the embedding of the intent that produced them. When a new intent arrives, its embedding is compared against all cached entries using cosine similarity. If the similarity to any cached entry exceeds a configurable threshold, the stored BraneScript is returned directly without invoking the LLM pipeline. This allows identical or near-identical intents to bypass generation entirely, guaranteeing reproducibility for repeated requests.

To evaluate the cache, it was seeded with all 584 intents paired with their reference BraneScript. The 1752 paraphrase intents described in Subsection 4.8.1 were then used as queries. Since each paraphrase corresponds to a known cached entry, the expected behaviour is a cache hit returning the correct BraneScript.

Three metrics are reported across eight similarity thresholds from 0.80 to 0.98, as defined in Section 4.8.2: hit rate, correct rate, and false-positive rate.

Table 6.4 and Figure 6.7 report the results across all eight thresholds.

Threshold	Hit rate	Correct rate	False-positive rate
0.80	100.0%	99.1%	0.9%
0.85	100.0%	99.1%	0.9%
0.88	99.9%	99.0%	0.9%
0.90	99.7%	98.9%	0.9%
0.92 (default)	99.1%	98.3%	0.8%
0.94	96.2%	95.4%	0.8%
0.96	88.1%	87.5%	0.6%
0.98	59.1%	59.1%	0.1%

Table 6.4: Semantic cache hit rate, correct rate, and false-positive rate over 1974 paraphrased intents at eight similarity thresholds.

Between thresholds 0.80 and 0.92 the hit rate remains near 100% and the correct rate stays above 98%, while the false-positive rate holds steady at approximately 0.9%. At the system default of 0.92, the hit rate is 99.1% and the correct rate is 98.3%, with a false-positive rate of 0.8%. Above 0.92, both the hit rate and correct rate decline sharply: at threshold 0.94 the hit rate drops to 96.2%, at 0.96 to 88.1%, and at 0.98 to 59.1%. The false-positive rate decreases in the same range but only marginally, reaching 0.1% at 0.98. Below 0.92, down to 0.80, the hit rate is at or near 100% but the false-positive rate does not decrease further, remaining at 0.9%.

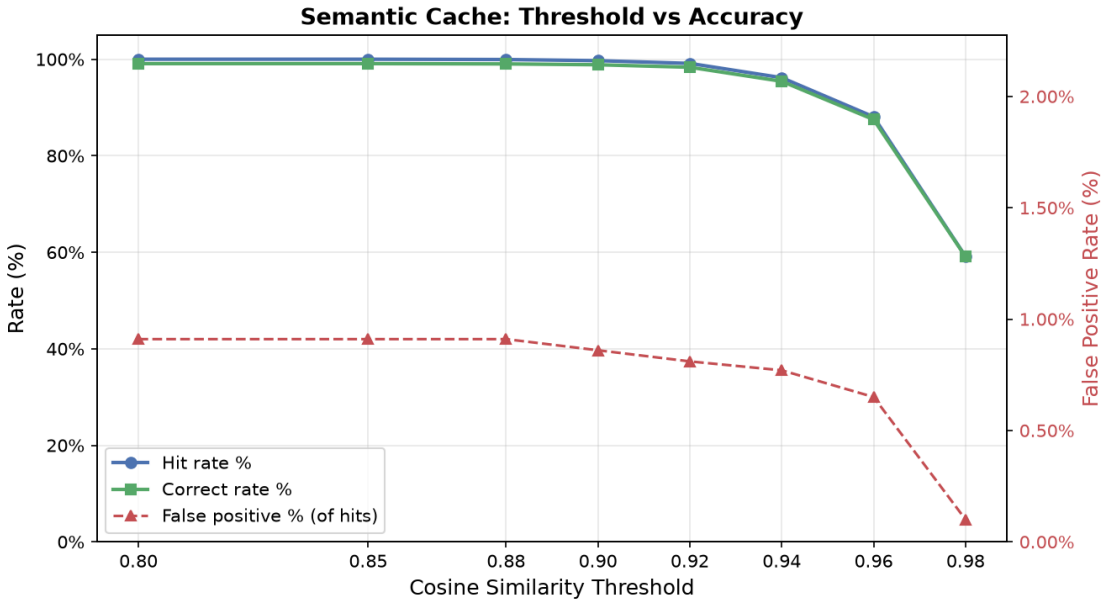

Figure 6.7: Cache hit rate and correctness rate as a function of the similarity threshold (left axis), with the false-positive rate on the magnified right axis. The system default of 0.92 is marked.

7

Discussion

This chapter interprets the results of Chapter 6 in light of the research questions and the motivation set out in Chapter 1.

7.1 RQ1 — Scale Shifts Failure Modes, Not Just Rates

7.1.1 The ceiling problem

The most striking aspect of the **RQ1** results is not the improvement in compile and execution rates across model scales, but the ceiling on output match rate. Even at 27B parameters, a model that compiles 95.6% of scripts and executes 93.2%, only 49.6% of generated workflows produce output that exactly matches the reference. This means that for every two scripts the 27B model generates and successfully executes, one produces a wrong answer.

The gap between execution rate (93.2%) and output match rate (49.6%) at 27B is 43.6 percentage points, substantially larger than the same gap at 4B (30.5 points) and 9B (35.9 points). As the model grows more capable at producing syntactically valid BraneScript, the bottleneck shifts: rather than failing to write code the compiler accepts, the model increasingly writes code that is structurally correct but logically wrong. This pattern is consistent with findings in code generation research more broadly, where larger models outperform smaller ones on syntax-heavy benchmarks far more readily than on semantic-correctness benchmarks requiring precise parameter passing and multi-step reasoning.

7.1.2 What the error breakdown reveals

The error type analysis in Section 6.2.2 provides a finer-grained picture of this shift. At 4B, the dominant failure category is compile-time undefined variable errors (24.1% of all

examples), which are straightforward structural failures: the model emits a reference to a variable it never declared. These errors suggest the model has not reliably internalised the BraneScript scoping rules, where every variable must be declared with `let` before use.

At 9B, undefined variable errors persist at nearly the same rate (24.5%), but syntax errors drop sharply from 17.5% to 4.8%. The 9B model appears to handle BraneScript grammar more reliably while still struggling with state tracking across multi-step workflows. The rise in undefined function errors (from 1.4% to 3.8%) suggests the model is becoming more ambitious in what it attempts, calling package functions it has not verified are available, even as its basic syntax improves.

At 27B, compile failures are almost entirely eliminated (syntax: 1.4%, undefined variables: 0.2%, undefined functions: 2.4%), but output mismatch rises to 43.6%. The model produces structurally valid workflows reliably and without error, yet returns incorrect output.

These are the hardest failures to detect and correct: unlike a compile error, which is immediately surfaced and detectable by a machine, a semantically wrong but syntactically valid BraneScript would silently return wrong results to a researcher relying on the pipeline. This class of failure based on confident, plausible-looking, but incorrect output, is arguably the most practically dangerous failure mode for a system used in scientific data analysis.

Considering the fact that the pipeline is aimed at domain scientists rather than software engineers, this shift in failure mode has practical implications for deployment: a system that compiles and executes successfully is not necessarily a system that produces correct results, and may end up misleading researchers if the output is not verified against a reference or ground truth.

7.1.3 Strict evaluation underestimates semantic correctness

All output match rates reported in this thesis use strict exact-string comparison between the generated script's standard output and the reference. A lenient analysis of the 27B base model's mismatches reveals that 16.4% of its 159 failed output comparisons are cases where the generated script produced the correct numerical or categorical values but wrapped them in descriptive labels absent from the reference — for instance, printing `Mean: 52.27` instead of `52.27`. This analysis was limited to the 27B model as the best-performing variant, where near-correct outputs are most frequent. The implication is that the strict metric moderately underestimates the semantic accuracy of the pipeline across all models: scripts that are logically correct but stylistically verbose in their output formatting are

penalised as failures. The true output correctness of the system is therefore closer to the lenient figures than the strict ones reported in Chapter 6.

7.1.4 Smaller models and their practical value

Despite the higher compilation failures seen in the 9B model compared to the 27B model, the results hide important details: the smaller model is still capable of producing correct workflows in half of the cases it manages to execute.

The fact that the 9B model is substantially cheaper to run than the 27B model (as it fits in a single Nvidia GTX 3060 GPU, a consumer-grade 5-year-old graphics card), suggests that for some use cases, where the cost of running a larger model is prohibitive, a smaller model may still be a viable option, provided that additional verification steps are in place to correct its compilation failures. The iterative nature of the pipeline proposed in this thesis, where the compilation step can be verified and corrected with a feedback loop, allows for a practical trade-off between model size, cost, and accuracy. Giving back the compilation error to the model for an iterative correction may allow the 9B model to reach similar execution rates as the 27B model, while still being cheaper to run, and therefore more accessible to researchers with limited computational resources.

7.2 RQ2 — Engineering Effort Dominates Model Scale

7.2.1 Prompt engineering as first investment

The single largest effect observed in this thesis came not from model scale but from correcting the system prompt and RAG configuration, which moved the 27B model's execution rate from 37.9% to 93.2% with no change to model weights as represented in Figure 6.4. A naive reading of the first evaluation round alone would have concluded that even a 27B model is unreliable at BraneScript generation, and that only a much larger commercial-grade model could close the gap. After the second round, it is clear that this conclusion would have been wrong, and that the actual bottleneck was grounding and instruction quality.

This has a direct implication for researchers deploying a local LLM-based DSL translators: investing in a precise retrieval and an unambiguous system prompt shall be the first priority, and only then model size or further fine-tuning shall be taken in consideration. Because direct edits to the base the prompt and to the retrieval configuration were all revised together between the two evaluation rounds, this finding should be read as evidence that retrieval-and-prompt engineering *as a bundle* outweighs model scale, not as an

isolated measurement of RAG's individual contribution against a no-RAG condition. It also validates gap **G2**, as the missing piece for practical intent-to-WMS translation is a carefully engineered bridge between the two and not a larger model alone.

7.2.2 The decomposition contribution

The ablation in Section 6.3.2 isolates one specific component of that bridge: the intent decomposer. Removing it reduces compile rate from 66.1% to 52.9%, execution rate from 64.9% to 48.1%, and output match from 36.9% to 26.0% on the 9B model. The consistent degradation across all three metrics indicates that the decomposer contributes positively at every stage of the pipeline, not just at one particular failure mode.

A closer look at the error breakdown reveals where decomposition's benefit is concentrated. Undefined function errors, where the model calls a function that does not exist in the package, occur at virtually the same rate with and without decomposition (28.5% vs. 28.8% of examples). Syntax errors, by contrast, fall sharply from 12.3% to 3.0% when decomposition is enabled. This suggests that the primary contribution of decomposition is not improved package or function selection (that is passed to the RAG), but rather that breaking the intent into explicit sub-tasks gives the model a cleaner, more constrained generation problem. Smaller, well-scoped sub-tasks appear to reduce the syntactic complexity the model must manage in a single generation pass, resulting in structurally more correct BraneScript. The function-level hallucinations (calling non-existent functions within the correct package) persist at the same rate regardless, suggesting they are driven by limitations in the model's knowledge of package APIs rather than by input ambiguity.

It is also notable that the output match rate under no-decomposition (26.0%) remains well above zero, meaning the generator can still produce correct workflows from unstructured input in roughly one in four cases. This suggests the base model can still absorb enough BraneScript knowledge from its inputs and produce correct outputs for single-package intents without decomposition.

Finally, a cross-model comparison further underlines decomposition's weight relative to model scale. The 4B base model with decomposition enabled achieves 54.6% execution rate, while the 9B model with decomposition disabled reaches only 48.1%. Decomposition applied to a smaller model outperforms a model one size tier larger running without it. This implies that for deployment scenarios where computational resources are constrained, investing in decomposition yields more return than upgrading to a larger model.

7.3 RQ3 — Fine-Tuning Degrades Performance Under Current Conditions

Both fine-tuned variants collapse to near-zero compile and execution rates on the held-out test set, far below the 9B base model they were derived from. The comparison between five epochs and three epochs rules out training duration as the cause: the results are nearly identical, confirming this is a systematic failure rather than an overfitting artefact.

The central argument of this section is that the root cause is memorisation driven by an insufficient training set, and that all observable failure modes in the generated outputs are downstream symptoms of that single constraint.

7.3.1 Why memorisation occurred

Three compounding factors drove the training into memorisation rather than generalisation.

Dataset too small relative to model capacity. Qwen3.5-9B is a model pre-trained on billion of tokens. Its LoRA adapter, even with a conservative rank of 16, still contains millions of trainable parameters. With only 497 training examples, there is vastly more model capacity than data. Under these conditions gradient descent finds the path of least resistance: memorising the 497 specific input–output mappings exactly instead of abstracting reusable patterns. Studies on fine-tuning small and mid-sized LLMs find that measurable generalisation requires at least 1,000–5,000 diverse, high-quality examples for code generation tasks (25, 26); 497 falls below the threshold at which any generalisation benefit over the base model can be expected.

Learning rate too aggressive. The SFT learning rate of 2×10^{-4} is still too high for a model of this size. A high learning rate causes large weight updates per step, making the adapter fit tightly to the training examples in very few steps. The training loss collapsed to near zero within 0.3 epochs after seeing roughly 150 examples for the first time confirming that memorisation was complete almost immediately. A lower learning rate (e.g. 1×10^{-5}) would have slowed this process, potentially allowing the model to build more generalisable associations before the adapter overfit.

No mechanism to stop memorisation. There was no early stopping based on validation loss, no weight decay, and no dropout on the adapter. Due to this, nothing penalised or interrupted memorisation once it began. The trainer continued updating weights for the full number of epochs even though loss had already reached zero, reinforcing the memorised mappings further.

7.3.2 Why memorisation causes test-set failure

The LoRA adapter encodes 497 specific context-window patterns: for each training example, it learned "when the system prompt <x> and the specific user intent is <y>, output <z> specific BraneScript". On the test set, the system prompt "<x>" is always the same but the user intent "<y>" is new. The adapter has no memorised answer for them.

For unseen inputs, the adapter does not simply stay inactive, as its weights are always applied on top of the base model, and they actively push the generation away from what the base model would naturally produce. The result is a generation that is neither the clean output of the untrained base model nor a valid BraneScript. This leads to the adapter disrupting the base model's competent zero-shot behaviour without replacing it with anything useful.

7.3.3 Symptoms of memorisation in the generated outputs

A deeper analysis of the generated outputs confirms this picture. Few patterns appear consistently across both fine-tuned variants.

Verbose inline reasoning and foreign syntax. Training examples contain only intent–BraneScript pairs with no comments or intermediate reasoning; while the average training output is 352 characters, generated outputs average 1,800–2,000 characters, more than five times longer. On unseen intents the adapter provides no memorised continuation, and the base model's pre-trained code generation behaviour floods back. From this, two distinct manifestations appear: verbose multi-line comment blocks with self-contradictory remarks, and, more critically, the model starts attempting to *implement algorithms from scratch* using constructs that do not exist in BraneScript. For example, for a normalisation intent the model generates Python-style `for` loops, manual array indexing with `patients.length`, and constructs such as `new array of real`, none of which are valid BraneScript. Rather than calling the appropriate package function (`normalize(data, "blood_pressure", "minmax")`), the model writes a general-purpose algorithmic implementation drawn from its pre-training on mainstream programming languages. This is

the clearest evidence that the model never learned the intent-package-function mapping and instead falls back on general code generation patterns when the memorised answer is absent.

Context contamination. Although the system and user tokens receive zero loss during training, the model still sees them in the context window for every training example. With loss collapsing immediately, the model effectively memorises each training example as a complete context-window pattern, conflating prompt content with assistant output. On unseen inputs it reproduces verbatim fragments of the system prompt as inline comments — phrases such as `# Do NOT attempt to re-implement the masking logic` or `# package function outputs`, drawn directly from the system prompt's constraint list. This is a consequence of full-context memorisation.

7.3.4 Implications and scope of the finding

The near-zero compile rates should be read as a consequence of insufficient training scale rather than as evidence that BraneScript is unlearnable via SFT. The most direct remedy is a larger, more diverse training set on the order of several thousand examples, covering the full range of package combinations and intent types. Complementary measures such as a lower learning rate, early stopping on validation loss, and data augmentation through intent paraphrasing would further reduce the risk of memorisation even with a modestly sized dataset. Additionally, fine-tuning was limited to the 9B model due to GPU allocation constraints on Snellius; larger models (27B, 70B) with stronger zero-shot instruction-following capabilities may require fewer examples to generalise, making them potentially more robust to the data scarcity observed here. All of these remain open directions for future work.

7.4 RQ4 — Semantic Caching as a Practical Reproducibility Layer

The evaluation of the semantic cache yields two sets of findings that must be read together. On one hand, the cache achieves a 99.1% hit rate and a 98.3% correct rate at the 0.92 threshold over the paraphrase test set, indicating that sentence-embedding similarity is an effective proxy for intent equivalence. On the other hand, the threshold sweep reveals an irreducible false-positive floor of approximately 0.9% for the same threshold value, attributable to the inability of general-purpose embeddings to distinguish intents that share the same topic but differ in specific parameter values.

The first finding addresses the core design goal of the cache: to return the same verified workflow for semantically equivalent intents without repeating the generation step ensuring reproducibility. The observed hit and correct rates confirm that this goal is met under the evaluation conditions. The second finding establishes a boundary on when the cache should be trusted: its error rate is hardly reducible to zero by threshold tuning alone. The two findings together suggest that the cache is best understood as a *performance optimisation with a bounded, known error rate* rather than a reproducibility guarantee in the strict scientific sense.

7.4.1 The false-positive floor

The cache threshold sweep reveals a false-positive rate that remains stable at approximately 0.9% across thresholds from 0.80 to 0.92, and only drops to 0.1% at a threshold of 0.98, at which point the hit rate collapses from 99.1% to 59.1%. This behaviour indicates that the false positives are not a function of the threshold being too low, but of the embedding model mixing structurally similar intents regardless of parameter values. Unlike the **RQ1**/**RQ2** findings above, this is not attributable to fixable engineering choices: a general-purpose sentence embedding model encodes what an intent is *about*, not the exact parameter values within it. Two intents that differ only in a threshold value, a column name, or a dataset identifier are, for most practical purposes, requests for different computations, yet they are nearly indistinguishable in embedding space. Threshold tuning is therefore not a completely viable path towards the elimination of false positives.

This is a direct, empirical instance of gap **G3**, where no agreed definition of intent reproducibility exists (Chapter 2): the semantic cache implicitly assumes that "reproducible" means "the same workflow for the same *propositional* intent", while for scientific correctness it must mean "the same workflow for the same intent *including all parameter values*".

A semantic cache that weights parameter values more heavily than topic similarity would be a more appropriate solution for scientific reproducibility, but it would require a different embedding model or a custom similarity function that is aware of the DSL's parameter semantics. This is a potential avenue left for future work.

7.4.2 The operating point and its practical meaning

The remaining 0.8% false-positive rate from the 0.92 threshold means that for every 125 cache hits, approximately one will return a workflow intended for a semantically similar but distinct intent. In a scientific computing context, this is not a negligible risk, as a

wrong workflow executed silently on a dataset could produce results that look plausible but are derived from the wrong analysis. For this reason, the cache should not be used as the sole quality gate.

It is worth noting, however, that the threshold can be treated as a confidence dial rather than a fixed parameter. Raising the threshold does not simply reduce false positives by a small margin, but it causes the cache to abstain on lower-confidence matches entirely, routing them to a fresh generation call instead. At 0.96, for example, the false-positive rate drops to 0.65% while the hit rate falls to 88.1%, meaning that roughly 12% of requests that would have been served from cache are instead regenerated. For deployments where correctness is critical and generation cost is acceptable, a higher threshold is therefore a principled strategy: uncertain matches will receive a new and independently generated workflow rather than a cached one of unknown validity. The threshold, in this sense, encodes a policy choice about the acceptable trade-off between latency and confidence, and the sweep data in Section 6.5 provides the empirical basis for making that choice explicitly.

7.5 Revisiting the Choice of Brane

The motivation for using Brane as the testbed for this work (Chapter 2) rested on two properties: its support for federated workflows in which the party executing a computation need not own the underlying data, and its enforcement, via typed, sandboxed packages of that separation at the platform level rather than by convention.

The results in this thesis evaluate translation accuracy and reproducibility, but the selection of an underlying architecture that makes evaluation possible under private conditions is itself evidence for a privacy-centered design. The translation problem studied here is therefore additive to Brane's existing privacy guarantees rather than a replacement for them: a correct BraneScript workflow inherits Brane's data-locality enforcement automatically, simply by virtue of being expressed in BraneScript rather than in a Python script with unrestricted file access.

The static typing of BraneScript has a secondary benefit that emerged empirically through this thesis: it makes evaluation cleaner than it would be for a dynamically typed target language. Because the compiler rejects undefined variables, undefined functions, and type mismatches at compile time, compile rate is a meaningful and automatically computable proxy for a large class of semantic errors, without requiring execution. This is not a property of most target languages used in comparable NL-to-code work (Python,

SQL, Bash), where many structural errors only surface at runtime or not at all. The strictness that makes BraneScript difficult to generate correctly is the same property that makes incorrect generation immediately detectable.

7.6 Summary

Taken together, the results suggest a layered picture of where accuracy comes from in intent-based scientific workflow generation.

Retrieval and prompt engineering account for the largest single improvement observed, and should be the first investment for anyone building a similar system for a low-resource DSL. The decomposer contributes a further, consistent improvement by converting vague multi-clause intents into structured retrieval queries.

Scaling the base model from 4B to 27B brings substantial compile and execution rate improvements, but the output match ceiling at 49.6% reveals that scale alone cannot solve the semantic accuracy problem. The dominant failure mode at 27B is no longer structural but semantic: the model generates plausible, executable workflows that return the wrong answer, a failure class that is harder to detect and more dangerous in practice than a compile error.

Semantic caching provides a strong and computationally cheap reproducibility mechanism for repeated or paraphrased intents, but has an irreducible false-positive floor imposed by the limitations of sentence embeddings as a proxy for computational equivalence. The threshold of 0.92 represents an acceptable operating point and not a principled solution to the parameter sensitivity problem. Administrators may choose to tune the threshold to a higher value if correctness is critical and generation cost is acceptable.

8

Threats To Validity

This chapter reports threats to the validity of the experiments in Chapter 6, following the classification proposed by Wohlin et al. (27).

8.1 Internal Validity

The difference between the two baseline evaluation in Subsection 6.3.1 presented in rounds differ in more than one variable simultaneously (system prompt, RAG aliases, and output post-processing all changed together), so the reported 13.5% → 49.6% improvement in output match rate for the Qwen3.6-27B model cannot be decomposed into the individual contribution of each fix. Additionally, gauging the quality of the first prompting round is a subjective judgement, and the choice of which prompts to include in the second round was informed by the first round's results, which may have introduced a more refined prompt that would not have been chosen in a blind evaluation.

A second internal validity threat concerns the supervised fine-tuning evaluation. The 87 test intents were drawn from the same 584-intent pool used to design the training set, meaning both splits share the same package set and the same operation types. If the fine tuning would have worked, the SFT model could have benefited from memorisation of structural patterns present in the training data rather than demonstrating genuine generalisation to unseen intents. A stricter evaluation would require test intents authored independently and targeting package or operation combinations not represented in training.

8.2 External Validity

All evaluated intents target one of six hand-built Brane packages designed specifically for this project, covering a curated set of scientific scenarios. It is not established that the

translation accuracy, RAG design, or cache behaviour reported in Section 6.5 generalise to (i) packages built by other teams with different documentation conventions, (ii) substantially larger package ecosystems, or (iii) intents drawn from real scientists' requests rather than machine-authored examples.

The pipeline is also tailored specifically to BraneScript: the RAG retrieval is designed around package documentation structure, the decomposition strategy targets BraneScript's function-call granularity, and the prompt templates encode BraneScript syntax conventions explicitly. Whether the approach generalises to other scientific workflow DSLs, such as CWL, Nextflow, or Snakemake, is untested; these languages have substantially different abstraction models and documentation styles that would require independent evaluation, but we expect the general approach to be adaptable given appropriate modifications to the pipeline.

All 584 intents in the dataset (and their relative paraphrases) were authored by a third party LLM with detailed knowledge of both the packages and BraneScript. This means the intents may unconsciously reflect what the system handles well, which is a natural precise phrasing with well-scoped requests. Real users with less familiarity with the underlying system may produce more ambiguous, colloquial, or domain-specific intents that the pipeline was not tested against. Additionally, all test intents were formalized in the English language; performance for non-native English speakers or non-English intents is unknown.

9

Conclusion

This thesis investigated whether large language models can bridge the gap between natural-language research intent and executable BraneScript workflows in the Brane privacy-preserving distributed computing framework. The motivation had practical ground: domain scientists working with sensitive cross-institutional health data should not need to learn a domain-specific workflow language to benefit from the privacy and reproducibility guarantees that Brane provides.

9.1 Summary of contributions and results

The thesis designed, implemented, and evaluated a pipeline consisting of an intent decomposer, a retrieval-augmented package context provider, a large language model code generator, and a semantic cache. Four research questions were addressed.

RQ1 — Translation accuracy scales with model size. Three base models from the Qwen3 family were evaluated on a dataset of 497 intent–BraneScript pairs spanning the full drafted Brane package library. All three metrics, compile rate, execution rate, and output match rate, improve monotonically with model scale. The dominant failure mode shifts with scale: small models fail primarily at compile time due to syntax and undefined-variable errors, while the 27B model compiles reliably but still misses roughly half of output values, indicating that semantic correctness remains an unsolved challenge even at this scale. The evaluation metric of output match rate is particularly stringent, and while it is the easiest to verify automatically, it does not account for semantically equivalent outputs that differ in formatting or ordering.

RQ2 — Engineering effort matters more than model scale. Prompt engineering and retrieval configuration had a larger absolute impact on output match than the difference between model sizes. Fixing the system prompt, improving RAG aliases, and correcting post-processing heavily lifted output match rates. Together with the intent decomposer, these findings indicate that a carefully engineered pipeline with a smaller model can outperform a larger model without engineering investment.

RQ3 — Supervised fine-tuning degrades performance under current conditions. Two QLoRA fine-tuning runs on the 9B model (3 and 5 epochs on 497 training examples) both collapsed to near-zero performance on the 87-example held-out test set: compile rates of 2.3% and 4.6% respectively, compared to the base model's 49.4%. The consistent failure across both runs rules out training duration as the cause. The root cause is memorisation: the dataset is too small relative to model capacity, and with no early stopping the LoRA adapter memorised the training examples rather than learning generalisable BraneScript patterns. On unseen intents the adapter disrupts the base model's competent zero-shot behaviour without providing a useful alternative — the fine-tuned model performs far worse than the base model it was derived from.

RQ4 — The semantic cache is effective but not a scientifically strict reproducibility guarantee. The semantic cache achieves a 99.1% hit rate and 98.3% correct rate at the default 0.92 cosine similarity threshold over a paraphrase test set, confirming that sentence-embedding similarity is a viable proxy for intent equivalence. However, the false-positive rate has an irreducible floor of approximately 0.9% that does not decrease meaningfully with threshold tuning until the threshold reaches 0.98, at which point the hit rate collapses to 59.1%. The floor arises because general-purpose embeddings cannot reliably distinguish intents that share a topic but differ only in parameter values. The cache therefore functions as a practical performance optimisation with a bounded known error rate, rather than a strict reproducibility guarantee in the scientific sense.

9.2 Limitations

BraneScript is an unknown language with no existing dataset. BraneScript is a domain-specific language developed exclusively within the Brane project, with no public corpora, no community presence, and no prior NL-to-BraneScript dataset from any research group. Every training and evaluation example used in this thesis was generated

from scratch, either manually or via LLM synthesis. This means the models had no BraneScript exposure during pre-training, the dataset size is constrained by the effort required to produce ground-truth scripts, and there is no external benchmark against which to validate results. All findings should be interpreted in this context.

Dataset size limits the strength of all findings. The full dataset of 497 training and 87 test examples is too small for NLP standards. Effect sizes reported throughout this thesis are point estimates without confidence intervals, and individual difficult examples disproportionately influence results. A larger, more diverse dataset would allow more reliable training and comparisons, and would directly enable the fine-tuning experiments that failed here due to memorisation.

Computational resources constrained the experimental scope. All experiments were run on a single Snellius GPU node with a fixed quota allocation. This prevented evaluation of models larger than 27B parameters, limited fine-tuning to the 9B model for a small number of epochs, and made the GRPO training loop too expensive to run.

Larger open-weight models were not evaluated. The largest model evaluated is 27B parameters. Models in the 35B–397B range, such as Qwen3.5-32B or Qwen3.5-397B were not evaluated, leaving open whether the gap to cloud-scale proprietary models (GPT-4o, Claude, Gemini) can be closed with open-weight models at intermediate sizes. A direct comparison on the same BraneScript task against state-of-the-art cloud LLMs that range from 70B to 1T parameters, and that have been trained on a much larger corpus, would provide a clearer picture of where local models stand and what the practical cost of privacy-preserving local execution is in terms of accuracy.

No iterative correction The pipeline evaluates single-shot generation with no feedback loop. An iterative correction step with the re-prompting of the model together with the compile error, is standard in code generation and could substantially improve compile rates, particularly for smaller models.

9.3 Future work

Several directions emerge directly from the findings of this thesis.

Scaling the training set for SFT. The most direct path to successful fine-tuning is a larger, more diverse training dataset. A dataset of 2,000–10,000 intent–BraneScript pairs covering the full range of package combinations and intent types, combined with a lower learning rate and early stopping on validation loss, would address the memorisation failure identified in **RQ3**.

Reinforcement learning from execution feedback (GRPO). The training infrastructure developed in this thesis includes a full GRPO (Group Relative Policy Optimisation) implementation that rewards scripts based on actual execution against a live Brane instance. This approach bypasses the need for reference BraneScript entirely: the model learns from whether its generated scripts compile and produce correct results.

Larger model fine-tuning. Fine-tuning the 9B model, which already demonstrates strong zero-shot BraneScript generation, could yield a specialised model that combines the 27B base model's compilation reliability with improved semantic accuracy on domain-specific intents. The models with 70B+ scale remains entirely unexplored.

Parameter-aware semantic cache. The current semantic cache uses general-purpose sentence embeddings that encode what an intent is *about* rather than the exact parameter values within it. Two intents that differ only in a column name, threshold, or dataset identifier are nearly indistinguishable in embedding space, producing a false-positive floor that cannot be eliminated by threshold tuning alone. A semantic cache that weights parameter values more heavily than topic similarity would be a more appropriate solution for scientific reproducibility, but would require a different embedding model or a custom similarity function aware of the DSL's parameter semantics. This is a concrete avenue for future work that would directly address the irreducible false-positive rate identified in **RQ4**.

Extension to other domains and frameworks. Adapting the pipeline to other privacy-preserving workflow frameworks would establish whether the findings generalise beyond the healthcare domain and the specific BraneScript language studied here.

References

[1] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. **Pegasus, a Workflow Management System for Science Automation**. In *Future Generation Computer Systems*, **46**, pages 17–35, 2015. 1, 5, 6

[2] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. **Nextflow enables reproducible computational workflows**. *Nature Biotechnology*, **35**(4):316–319, 2017. 1, 5

[3] Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O. Twardziok, Alexander Kanitz, Andreas Wilm, Manuel Holtgrewe, Sven Rahmann, Sven Nahnsen, and Johannes Köster. **Sustainable data analysis with Snakemake**. *F1000Research*, **10**:33, 2021. 1, 6

[4] Enis Afgan, Dannon Baker, Bérénice Batut, Marius van den Beek, Dave Bouvier, Martin Cech, John Chilton, Dave Clements, Nate Coraor, Björn A. Grüning, Aysam Guerler, Jennifer Hillman-Jackson, Saskia Hiltemann, Vahid Jalili, Helena Rasche, Nicola Soranzo, Jeremy Goecks, James Taylor, Anton Nekrutenko, and Daniel Blankenberg. **The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update**. *Nucleic Acids Research*, **46**(W1):W537–W544, 2018. 6

[5] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan

Gavrin, and Carole Goble. **Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language**. *Communications of the ACM*, **65**(6):54–63, 2022. 6

[6] Susan B. Davidson and Juliana Freire. **Provenance and Scientific Workflows: Challenges and Opportunities**. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1345–1350, 2008. 7

[7] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. **PROV-DM: The PROV Data Model**. Technical report, World Wide Web Consortium (W3C), 2013. W3C Recommendation, 30 April 2013. 7

[8] Víctor Cuevas-Vicenttín, Bertram Ludäscher, Paolo Missier, Khalid Belhajjame, Fernando Chirigati, Yaxing Wei, Saumen Dey, Parisa Kianmajd, David Koop, Shawn Bowers, Ilkay Altintas, Christopher Jones, Matthew B. Jones, Lauren Walker, Peter Slaughter, Ben Leinfelder, and Yang Cao. **The ProvONE Data Model for Scientific Workflow Provenance**. *DataONE Cyberinfrastructure Working Group, Draft*, 2016. 8

[9] Farah Zaib Khan, Stian Soiland-Reyes, Richard O. Sinnott, Andrew Lonie, Carole Goble, and Michael R. Crusoe. **Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv**. *GigaScience*, **8**(11):giz095, 2019. 9

[10] Stian Soiland-Reyes, Peter Sefton, Mercè Crosas, Leyla Jael Castro, Frederik Coppens, José M. Fernández, Daniel Garijo, Björn Grüning, Marco La Rosa, Simone Leo, et al. **Packaging research artefacts with RO-Crate**. *Data Science*, **5**(2):97–138, 2022. 9

[11] Carole Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R. Crusoe, Kristian Peters, and Daniel Schober. **FAIR Computational Workflows**. *Data Intelligence*, **2**(1–2):108–121, 2020. 9

[12] IETF IRTF. **Intent-Based Networking—Concepts and Definitions**, 2022. RFC 9315. 10

[13] Jeffrey O. Kephart and David M. Chess. **The Vision of Autonomic Computing**. *IEEE Computer*, **36**(1):41–50, 2003. 10

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. **Evaluating large language models trained on code**. *arXiv preprint arXiv:2107.03374*, 2021. 11

[15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. **Program Synthesis with Large Language Models**. *arXiv preprint arXiv:2108.07732*, 2021. 11

[16] Qwen Team. **Qwen2.5 Technical Report**. *arXiv preprint arXiv:2412.15115*, 2025. 11

[17] Abhimanyu Dubey et al. **The Llama 3 Herd of Models**. *arXiv preprint arXiv:2407.21783*, 2024. 11

[18] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. **Mistral 7B**. *arXiv preprint arXiv:2310.06825*, 2023. 11

[19] Jonas Duque et al. **Leveraging Large Language Models to Build and Execute Computational Workflows**. *arXiv preprint arXiv:2312.07711*, 2023. 11

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. In *Advances in Neural Information Processing Systems*, **33**, pages 9459–9474, 2020. 11

[21] Amal Gueroudji, Tanvir Mallick, Renan Souza, et al. **ControlA: Agentic Workflow Control Mechanisms for Reliable Science**. In *Proceedings of the IEEE International Conference on e-Science (eScience)*, 2025. 11

[22] Tim Mulder, Peter Vos, Gijs Schot, and Alexandru Iosup. **Brane: A Framework for Distributed Scientific Computing**. https://github.com/epi-project/brane, 2022. 13

[23] Jorrit Stutterheim, Aleandro Mifsud, and Ana Oprescu. **DYNAMOS: Dynamic Microservice Composition for Data-Exchange Systems, Lessons Learned**. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*, 2024. 15

[24] Jorrit Stutterheim. *DYNAMOS: Dynamically Adaptive Microservice-based OS — A Middleware for Data Exchange Systems*. Master's thesis, University of Amsterdam, 2024. 15

[25] Rui Guo et al. **Unveiling the Secret Recipe: A Guide for Supervised Fine-Tuning Small LLMs**. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. 66

[26] Zhuang Chen et al. **Massive Supervised Fine-Tuning Experiments Reveal How Data, Layer, and Training Factors Shape LLM Alignment Quality**. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2025. 66

[27] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012. 72