Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Policy-Driven Federated Workflow Generation using the Brane Framework

Author: Aditya Pradeep Gangadharan (2851278)

1st supervisor:	Dr. Adam Belloum
daily supervisor:	Dr. Natallia Kokash
2nd reader:	Dr. Yixian Shen

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

August 17, 2026

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Federated data processing allows organisations to collaborate while retaining data within their respective administrative domains. However, governance-oriented project information collected through BraneHub cannot be executed directly by Brane, which requires concrete packages, dataset references, worker locations, and BraneScript workflows. This thesis designs and evaluates the Brane Integrator, a system that translates BraneHub project configurations into reviewable and executable artefacts for federated workflow execution through Brane. The Integrator consists of a Package Manager, Node Provisioner, and Workflow Handler, and supports package authoring, participant and coordinator environment provisioning, governance interpretation, deterministic and LLM-assisted BraneScript generation, structural and registry validation, traceability reporting, review, and execution. The prototype was evaluated in a locally simulated Brane federation using governance scenarios, injected workflow faults, generator comparisons, package-generation experiments, execution tests, and numerical baselines. All 324 populated in-scope governance-field instances across eight scenarios were accounted for through generated constructs, processing outcomes, or explicit flags. The nine-rule validator accepted all eight expected-valid workflows, rejected all ten deliberately invalid workflows, and detected both tested locality fault types. Runtime enforcement remained partial because placement annotations were retained during execution, while governance tags were removed because of a Brane/eFLINT incompatibility. All five deterministic workflows executed successfully, compared with four of five LLM-generated workflows. The package-generation experiments produced consistently schema-valid manifests, but also revealed unsupported function declarations, imperfect governance extraction, and numerical deviations in generated Python. The results demonstrate the feasibility of a bounded and auditable bridge between governance configuration and federated execution. Deterministic generation was more reliable within the supported workflow pattern, while

LLM assistance provided additional flexibility but required authoritative interface information, deterministic validation, execution-based testing, and human review. The Integrator does not establish legal compliance, privacy, scientific correctness, or general support for arbitrary federated workflows.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Research Context

Research collaborations can depend on data held by multiple organisations (1). Centralising such data is often undesirable or infeasible because of privacy concerns, legal restrictions, institutional policies, and data-sovereignty requirements. Federated data processing addresses this problem by coordinating computation across independent locations while allowing data to remain within the administrative domain of its provider (1). However, keeping data distributed does not by itself resolve the technical and governance challenges involved in deciding what computation may run, where it may run, and how the resulting workflow should be reviewed.

Brane is a framework for programmable orchestration of multi-site applications. It packages computational functionality into portable containers and uses BraneScript to describe how these functions are combined and at which locations they are executed (2). This separation between computational functionality and distributed orchestration makes Brane suitable for federated workflows, but researchers must still provide valid packages, dataset references, execution locations, and BraneScript.

BraneHub complements this execution environment by supporting the configuration and governance of federated projects. It records project objectives, participant information, data descriptions, and legal or organisational requirements as part of a broader compliance-aware federated-processing lifecycle (1). A practical gap nevertheless remains between the governance-oriented information collected in BraneHub and the executable artefacts required by Brane.

1.2 Problem Statement and Research Gap

Translating a BraneHub project configuration into an executable Brane workflow requires several related tasks. Participant datasets and execution nodes must be associated correctly, computational functions must be packaged and described through valid manifests, and the intended computation must be expressed as BraneScript. Governance information must also be considered without assuming that every collected field has a corresponding executable construct.

Performing this translation manually places a substantial burden on researchers. It requires familiarity with BraneScript, package manifests, distributed execution, and the interpretation of participant requirements. Manual translation can also make it difficult to determine whether a governance field was represented in the workflow, left for human review, or omitted unintentionally. Conversely, generating plausible-looking policy syntax for unsupported requirements would create the appearance of enforcement without establishing its meaning or runtime effect.

Previous work establishes Brane as a programmable multi-site orchestration framework (2) and describes a wider architecture in which BraneHub and Brane support compliance-aware federated data processing (1). What remains to be investigated is a concrete and evaluated mechanism that translates a BraneHub project configuration into reviewable Brane packages and workflows, explicitly accounts for unsupported governance fields, validates the generated artefacts, and preserves traceability between the source configuration and the resulting workflow.

1.3 Research Objective

The objective of this thesis is to design and evaluate the *Brane Integrator*, a system that connects governance-oriented project configuration in BraneHub to federated workflow execution in Brane. The Integrator supports package authoring, participant and coordinator Brane environment provisioning, governance interpretation, deterministic and LLM-assisted BraneScript generation, structural validation, traceability, review, and execution.

1.4 Research Questions

This thesis addresses the following research questions:

RQ1 How should federated project configurations and participant policies be represented so that every governance obligation is either translated into an enforceable workflow construct or explicitly accounted for?

RQ2 Which BraneScript constructs correctly enforce a given set of participant policies, and to what extent can compliance be verified statically before execution?

RQ3 How do template-based and LLM-assisted BraneScript generation strategies compare in correctness, policy compliance, execution success, and cost?

RQ4 How reliably can an LLM produce and interpret the artefacts that bridge governance specifications to executable federated packages, and is the generated computation numerically correct?

In answering these questions, the thesis distinguishes between collecting a governance value, representing it in an artefact, validating its presence or structure, enforcing it during execution, and establishing broader compliance. In particular, "policy compliance" in the evaluation refers only to the explicitly defined and implemented checks. It does not denote proof of legal or organisational compliance.

1.5 Contributions

This thesis contributes an operational architecture connecting BraneHub configuration to Brane packages, infrastructure, and workflows; a construct-or-flag transformation that maps supported governance fields to reviewable workflow constructs while exposing unsupported fields; a traceability and validation process for generated artefacts; and an empirical comparison of deterministic and LLM-assisted generation across governance representation, structural correctness, execution, numerical agreement, generation reliability, and cost. It also documents the boundary between governance metadata included in the reviewed workflow and the constraints that are retained during execution.

1.6 Scope

The implemented system is a research prototype evaluated using a locally simulated Brane federation. It supports a bounded single-pass federated aggregation pattern, with limited structural exploration of a multi-round workflow. Participant placement annotations and named dataset references remain part of the executable workflow. Governance tag and

workflow-tag annotations are retained in the reviewed artefact and traceability report but are removed before execution because of the identified Brane/eFLINT incompatibility. They are therefore not demonstrated runtime controls. The thesis does not claim to prove package-level privacy, scientific validity, legal compliance, institutional eligibility, consent, retention, or post-execution governance.

2

Background

2.1 Federated Data Processing

Federated data processing (FDP) enables data from multiple independent sources to be accessed and processed without physically moving or consolidating the raw datasets in a single location (1). Computation can instead be executed at or near the environments in which the data are maintained. This approach is relevant when centralisation is restricted by privacy concerns, legal requirements, institutional policies, or data-sovereignty considerations.

A federated project may involve several independently administered participant sites and a coordinating component. Computational tasks are executed at participant sites, after which their outputs may contribute to a shared result. The information that leaves a participant environment nevertheless depends on the implemented computation and the controls applied to it. Federated execution therefore supports data locality, but does not by itself guarantee privacy, security, or compliance (1).

Technical execution is only one part of a federated collaboration. Participating organisations must also agree on matters such as eligible data, permitted purposes, participant responsibilities, access conditions, and the handling of results. These requirements may apply before execution, while the computation is running, or after its completion. The compliance-aware FDP lifecycle described by (1) consequently distinguishes between pre-exchange, exchange, and post-exchange activities.

Governance also extends beyond the initial decision to permit access. Usage-control research distinguishes policy specification from the mechanisms used to evaluate and enforce policies throughout the use of data or resources (3). Effective decentralised usage control may require policy decision points, policy enforcement points, monitoring, and auditing

across the participating sites. Consequently, recording a governance requirement or attaching it to a workflow does not by itself establish that the requirement is enforced during or after execution.

2.2 The Brane Framework

Brane is a framework for the programmable orchestration of applications across multiple computing sites (2). It separates computational functionality from distributed orchestration. Software engineers package reusable computational functions, while workflow authors compose those functions and specify where they should execute. This separation allows a workflow to coordinate computation across independently managed environments without requiring its author to implement the underlying distributed-execution mechanisms directly.

Computational functionality in Brane is distributed through packages. A package contains application code, its software dependencies, and a manifest describing the functions exposed to workflows. The manifest defines information such as the package name and version, callable functions, input parameters, and return types. Brane packages are built as containerised artefacts, allowing their functionality and dependencies to be distributed consistently across execution sites (2).

Workflows are expressed using BraneScript, Brane's domain-specific workflow language. A BraneScript program can import packages, create or refer to its inputs, invoke package functions, combine intermediate values, and return a result. Function calls can also be restricted to named execution locations. In the Brane version used in this thesis, an annotation such as `#[on("participant-1")]` associates the following operation with the specified node.

Datasets are represented through named references rather than by embedding their contents in the workflow. A workflow can therefore refer to a dataset available within the federation while Brane resolves that reference as part of workflow planning and execution. The presence of a data reference does not inspect or verify the dataset's contents. It identifies the data asset expected by the corresponding computation.

A Brane deployment contains central coordination components and worker components. The central environment receives and plans a workflow, resolves the required packages and named locations, and coordinates the execution. Worker nodes provide the environments in which assigned package functions are run. In an institutional deployment, workers can represent independently administered computing sites. In the experimental environment

used by this thesis, the coordinator and participants are represented by separate logical nodes within a local container-based deployment.

Brane therefore supplies the mechanisms required to package computations, identify datasets, constrain execution locations, and orchestrate a multi-site workflow. It does not, by itself, determine how a particular BraneHub project configuration should be translated into the package interfaces, participant references, workflow structure, and governance artefacts required by that project. This translation is the responsibility addressed by the Brane Integrator.

2.3 BraneHub and Compliance-Aware Project Configuration

BraneHub is a project-facing platform for configuring and coordinating federated data-processing collaborations. It supports activities such as describing a proposed project, recruiting participants, collecting information about available data and institutional conditions, and reviewing participation requests. The resulting project configuration provides context about the project, its participants, their datasets, and the conditions under which the collaboration is intended to operate (1).

Participant onboarding is important because organisations in a federation may operate under different technical, legal, and institutional conditions. BraneHub can collect this information through structured questionnaire responses and supporting free-text explanations. A researcher can then review the submitted information and decide whether a participant should be accepted into the project. The collected answers may describe matters such as applicable jurisdictions, available data, identifiability, legal conditions, security arrangements, agreements, and operational responsibilities.

The compliance-aware architecture presented by (1) assigns different responsibilities to BraneHub and Brane. BraneHub represents the collaboration and governance context, whereas Brane provides the programmable environment in which distributed computations are executed. This division allows project setup and participant onboarding to remain separate from the lower-level concerns of package deployment and workflow execution.

The same architecture considers governance across the full FDP lifecycle. During the pre-exchange stage, a project is defined, participants are recruited, applicable requirements are identified, and the necessary agreements, policies, and infrastructure are prepared. The exchange stage concerns controlled access to data and the execution of the distributed computation. The post-exchange stage includes matters such as result handling, retention, auditing, and other continuing responsibilities (1).

The information collected in BraneHub is not immediately executable by Brane. BraneHub represents project and participant information through configuration fields, questionnaire answers, dataset descriptions, and free text. Brane, by contrast, requires concrete technical artefacts such as package manifests, callable functions, dataset references, named execution locations, and BraneScript.

Furthermore, not every governance field has a direct BraneScript representation or an available runtime enforcement mechanism. A translation layer must therefore distinguish between requirements that can be represented using supported workflow constructs and requirements that must remain visible for review or external handling. The Brane Integrator introduced in this thesis provides this layer between BraneHub project configuration and Brane workflow execution.

3

Overview

3.1 System Context

The Integrator connects BraneHub's research and governance layer with the Brane execution environment. BraneHub stores the project configuration, participant information, data-processing requirements, and governance-related fields collected during project setup. Brane provides the infrastructure for packaging computations and executing distributed workflows. The Integrator translates the information maintained by BraneHub into the artefacts required to configure and operate a Brane-based federated study.

The implemented system consists of three main components. The Package Manager assists with creating a Brane computation package, either by generating the computation from a study description or by preparing a package descriptor for researcher-supplied Python code. The Node Provisioner configures the coordinator and participant Brane environments required by the study. The Workflow Handler retrieves the current project configuration, generates and validates a BraneScript workflow, produces a traceability report, and manages its submission for execution when it receives a workflow run request carrying the researcher's approval decision. The detailed responsibilities and interactions of these components are presented in the design chapter.

The prototype focuses on map-reduce-style studies involving a coordinator and multiple participant nodes. Computation is performed locally at the participant nodes, while partial outputs are combined at the coordinator. The researcher interacts with the process through BraneHub and remains responsible for reviewing the generated package, workflow, and identified governance limitations before execution.

3.2 End-to-End Operation

The process begins when a researcher defines a study in BraneHub and participants join the project. The resulting project configuration identifies the participating Brane nodes, the datasets used by the study, and the associated governance information. The computation package is then prepared through one of two supported routes: the Integrator can generate the Python computation and its package descriptor from a natural-language study description, or it can infer a descriptor for Python code supplied by the researcher. After researcher review, the resulting package is built.

The Node Provisioner prepares the Brane environments required by the project. Before registry-dependent validation and again before execution, the Integrator attempts to publish the built package to the Brane package registry. When workflow generation is requested, the Workflow Handler retrieves the latest configuration from BraneHub and interprets its structured and free-text governance fields. Each populated governance field included in the Handler's internal configuration is either mapped to a corresponding workflow construct or recorded explicitly in the traceability report when it cannot be represented at the workflow level.

The workflow is generated using either the deterministic template-based strategy or the optional LLM-assisted strategy. The generated BraneScript undergoes selected textual checks covering package references, node placements, dataset declarations, generated governance annotations, and combine-call form, together with a live registry check of package and function availability. The workflow and its traceability report are then returned to BraneHub for researcher review and approval.

Following a workflow run request carrying the researcher's approval decision, an executable version of the workflow is submitted to the Brane environment. Local computation tasks are dispatched to the relevant participant nodes, and their partial outputs are returned to the coordinator for aggregation. The final result is subsequently returned to BraneHub.

The system distinguishes between representing governance information and enforcing it during execution. Node-placement constraints and dataset references are retained as executable BraneScript constructs. Other governance annotations and traceability entries support review and accountability but are not demonstrated runtime policy controls. The detailed mappings, validation process, and limitations of the evaluated Brane–eFLINT execution path are discussed in the design chapter.

4

Design

This chapter presents the design of the Brane Integrator, which connects governance-oriented project management in BraneHub to federated workflow execution in Brane. Brane supplies the mechanisms for packaging computations, referring to data, selecting execution locations, and orchestrating multi-site applications (2). It does not, by itself, translate a BraneHub project configuration into the package, infrastructure references, and BraneScript required for execution. The Integrator addresses this gap.

The design problem is broader than producing syntactically plausible BraneScript. A generated workflow must refer to functions that exist in an approved package, datasets registered for the participating locations, and workers known to Brane. It must also account for the governance information collected for the project. Some of that information has a supported workflow-level representation, while other requirements depend on institutional procedures, package behaviour, or runtime mechanisms that are outside the demonstrated system. The design must therefore expose both what it translates and what it cannot enforce.

The Integrator is organised into three modules. The *Package Manager* prepares the executable computation and its declared interface. The *Node Provisioner* prepares the participant and coordinator locations referenced by a workflow. The *Workflow Handler* combines these artefacts with the BraneHub configuration to interpret governance information, generate and validate BraneScript, produce traceability, obtain an external review decision, and submit an accepted workflow to Brane. These modules are not equally central to the research contribution: package and node preparation establish the executable context, whereas the Workflow Handler implements the policy-driven generation process itself.

A central distinction used throughout the design is that a governance requirement may be *collected*, *interpreted*, *represented*, or *enforced*. These properties are not interchangeable. Receiving a field from BraneHub does not mean that the Integrator has a supported interpretation for it. Likewise, generating an annotation does not establish that the deployed runtime acts on that annotation. Preserving these distinctions is necessary to describe the system as policy-driven without presenting it as a guarantee of end-to-end legal compliance.

4.1 Design Principles and Scope

The architecture follows five principles.

Separate governance management from workflow execution. BraneHub remains the project-facing authority for membership, questionnaire answers, workflow cycles, and researcher decisions. Brane remains responsible for planning and executing distributed workflows. The Integrator translates between these two systems rather than duplicating either of them. This boundary follows the broader BraneHub–Brane compliance architecture proposed for federated data processing (1), but the allocation of responsibilities described here is specific to the implemented Integrator.

Operate on references rather than participant files during generation. The Workflow Handler uses participant identifiers, worker names, dataset identifiers, package metadata, and governance fields. It does not load participant datasets in order to generate BraneScript. The generated workflow instead contains Brane `Data` references and schedules the local computation at the participant worker. This design preserves the intended separation between workflow coordination and institution-controlled data. It does not, however, prove that package output is privacy-preserving. That depends on the computation implemented by the package.

Prefer explicit transformations over probabilistic generation. When the requested workflow follows the supported single-pass aggregation pattern, its structure can be derived directly from typed configuration. The deterministic template is therefore the default. LLMs are used when the input is semantic or less structured, such as for package authoring, manifest inference, free-text extraction, optional explanations, and an optional LLM-assisted BraneScript generation strategy. This division avoids using a probabilistic model for transformations that are already specified by explicit rules. It is particularly relevant for BraneScript because generation for low-resource and domain-specific languages

is complicated by specialised syntax, semantics, and limited task-specific data (4). Empirical studies also show that generated code can contain requirement conflicts, hallucinated interfaces, and defects that are not evident from surface plausibility alone (5, 6, 7).

Make unsupported requirements visible. The Integrator uses a *construct-or-flag* rule. A governance value produces a workflow construct only when an explicit mapping exists. Otherwise, the value is retained as a flagged requirement for review. The interpreter does not ask an LLM to invent policy syntax for an unfamiliar field because syntactic plausibility would not establish the meaning or enforcement point of the resulting construct. This conservative registry also makes policy mappings maintainable: extending support requires an explicit syntax, intended semantics, validation rule, and enforcement mechanism. This is consistent with evidence that policy-as-code artefacts remain subject to maintenance: observed changes may be substantial, and behavioural changes often make policies stricter (8).

Place checks and review between generation and execution. Generated artefacts are not submitted directly for execution. Package artefacts are exposed through an approval interface before building, and generated workflows are structurally validated before being sent to BraneHub for review. Machine-checkable constraints and test feedback have been shown to improve LLM-based code generation in other settings (9, 10, 11). In this system, however, validation is deliberately described according to what the implemented checks establish. It is not treated as a proof of scientific correctness, privacy, or legal compliance.

The privacy-aware processing lifecycle described in (1) distinguishes pre-exchange, exchange, and post-exchange obligations. The Integrator participates mainly in preparing and initiating the exchange phase. Consent, institutional approval, agreements, and participant eligibility originate outside it. Result sharing, retention, audit review, and data-subject procedures also remain external. Related usage-control research shows that an initial access decision may be followed by controls or obligations during and after use (3). The Integrator therefore records requirements outside its supported execution model instead of implying that workflow generation discharges them.

4.2 Overall Architecture

BraneHub supplies project information and lifecycle decisions. The Package Manager and Node Provisioner establish the functions and locations that may be referenced by a work-

flow. The Workflow Handler then performs the governance-to-workflow transformation and delegates distributed execution to Brane.

The separation reflects the dependencies of an executable workflow. First, the computation must be represented by a package with a known interface. Second, the participant and coordinator locations must be registered and associated with stable identifiers. Only then can generated BraneScript refer to concrete functions, datasets, and workers. Separating these responsibilities also confines change: package-authoring methods can evolve without changing infrastructure preparation, while alternative workflow-generation strategies can reuse the same approved package and location mappings.

The end-to-end control flow is:

1. A package is generated from a computation description or prepared from researcher-supplied Python code.
2. The package artefacts are exposed for review; an approval request triggers the Brane package build and coordinator provisioning.
3. Approved participants are associated with Brane workers and registered dataset identifiers.
4. For a project cycle, the Workflow Handler retrieves and normalises the current BraneHub configuration.
5. Governance fields are interpreted as supported constructs or explicit flags, after optional extraction from designated free-text fields.
6. A template or LLM strategy generates BraneScript, after which structural and registry checks are applied.
7. The annotated script and traceability report are sent to BraneHub for review.
8. Following a run event, the executable script is submitted to Brane and the resulting status or value is returned to BraneHub.

This sequence contains two distinct approval boundaries. The package interface is established before workflow generation, and the generated workflow is reviewed before execution. The Integrator exposes the relevant artefacts at both boundaries, but its API cannot itself prove that a particular person performed the review. It acts on the approval decision received from BraneHub or another authorised caller.

4.3 Package Manager

BraneScript invokes functions exposed through a Brane package. The Package Manager therefore precedes workflow generation: it establishes the package name, callable functions, and manifest against which a workflow can be generated and checked. Containerised packages are part of Brane's separation between application functionality and multi-site orchestration (2).

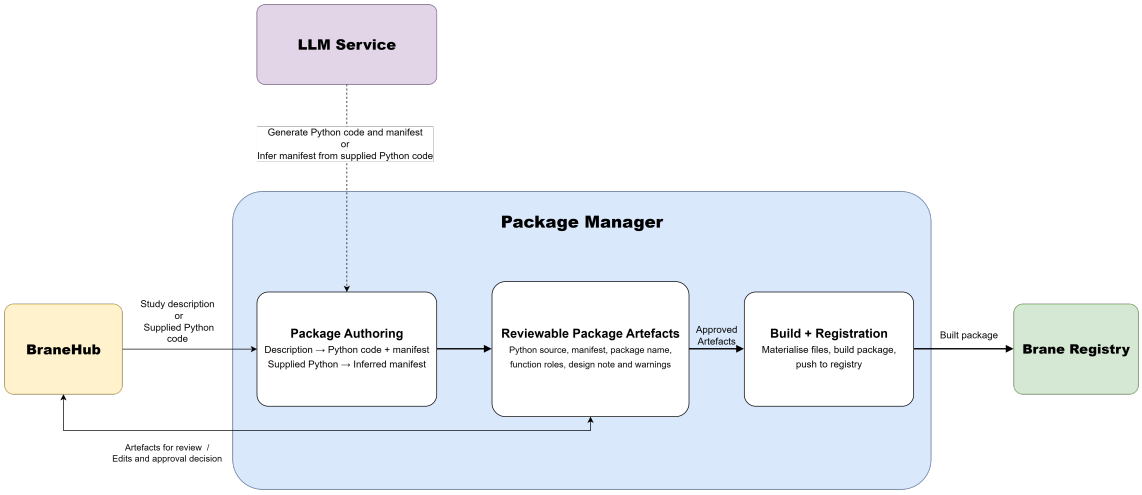

Figure 4.1: Package Manager authoring and build flow.

4.3.1 Authoring Paths and Common Interface

The module supports two authoring paths because researchers may begin with different artefacts.

In the *generated-package path*, the researcher supplies a study objective, a computation description, and optional data-type information. The LLM returns Python source, a `container.yml` manifest, and a short design note. The implementation fixes the package roles to `compute_local`, `combine_results`, and `finalize`. Fixing these roles reduces the number of function-interface choices left to the LLM and gives the deterministic workflow template a stable interface to call.

In the *uploaded-code path*, the researcher supplies Python source. An LLM generates the `container.yml` manifest from the code, after which the Integrator derives the local, combine, and optional finalisation roles from the manifest's action signatures. A separate focused LLM assessment asks whether the local function appears capable of returning

individual-level information. Any detected concern is presented as a warning. This assessment is advisory: it is not a formal privacy analysis, it is not applied as a second pass to generated packages, and an assessment that returns no result is treated as finding no concern. These limitations are stated explicitly because the presence of an assessment must not be interpreted as a privacy guarantee.

Both paths converge on the same package representation: Python source, a manifest, the package name, and the local, combine, and optional finalisation functions. This convergence is important because downstream workflow generation should depend on an explicit executable interface rather than on how the package was authored. Supplying concrete function-interface information has also been used to ground code generation for specialised control-code libraries (12). In the Integrator, the manifest plays the narrower role of declaring the package interface presented to the generator and checked against Brane's registry.

4.3.2 Constrained Computation Pattern

The supported package structure follows a local–combine–finalise pattern. The local function receives a participant dataset and produces an intermediate result. The combine function accepts two intermediate results and returns another value of the same role. An optional finalisation function converts the accumulated intermediate value into the workflow result.

This constraint exists because participant count is not fixed when the package is authored. A binary combine function can be applied as a left fold for any project with two or more participants, while a single-participant workflow can use its local result directly. The stable roles also make placement decisions explicit: local functions belong at participant workers, while combination and finalisation belong at the project coordinator.

The constraint improves predictability but limits expressiveness. It directly supports single-pass federated computations whose participant outputs can be accumulated through the declared binary interface. It does not by itself express adaptive convergence, arbitrary branching, asynchronous coordination, or general multi-package workflows. The Package Manager also does not prove that the Python implementation obeys the intended accumulator semantics or produces a scientifically correct result. The manifest describes the interface, not the full behaviour of the code.

4.3.3 Review, Build, and Registry Boundary

Before building, the current source, manifest, design note, and any advisory warnings are available through the package interface. The source and manifest may be edited, while the design note and warnings are exposed for review. An approval request materialises the source and manifest as a Brane package and invokes the Brane package builder. After a successful build, coordinator provisioning is initiated and the package can be pushed to Brane's registry.

This ordering prevents workflow generation from relying only on a natural-language description of the computation. The Workflow Handler generates calls using the package and function names stored for the project. During final validation, it also queries Brane's live package catalogue to confirm that the package and referenced functions are currently registered. The design therefore separates three questions: whether an artefact was generated, whether it was approved for building, and whether Brane currently exposes the package and functions referenced by the workflow. None of these questions alone establishes numerical correctness or privacy.

4.4 Node Provisioner

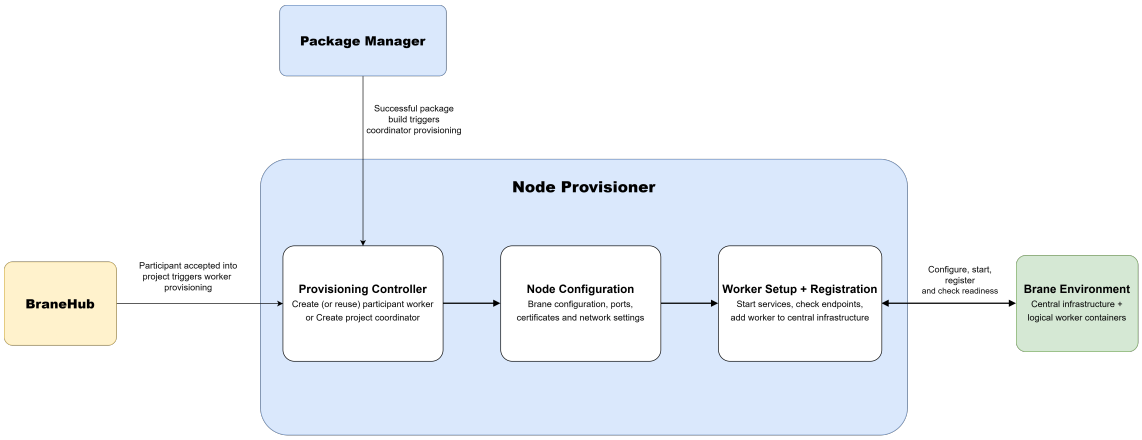

Figure 4.2: Node provisioning for participant workers and project coordinators.

The Node Provisioner prepares the locations referenced by generated workflows. It connects BraneHub participants and projects to stable Brane worker names and makes the participant–coordinator topology available to the Workflow Handler.

A participant worker represents the location at which a participant dataset is registered and the local package function is executed. A project coordinator hosts the binary combi-

nation steps and optional finalisation. This division makes the intended data flow explicit: dataset references are consumed at participant workers, while intermediate results are sent to the coordinator. It does not guarantee that intermediate results are non-identifying. That property depends on the package implementation.

The provisioner creates names derived from the BraneHub participant and project identifiers and records the corresponding mappings. A participant worker can be reused when the same participant joins more than one project, while each project receives its own coordinator. Before a location is marked ready, the local prototype creates its Brane configuration, starts the required services, checks the registry and job endpoints, and adds the worker to the central infrastructure description. The low-level allocation of ports, files, certificates, and container networks is an implementation mechanism and does not affect the logical contract exposed to workflow generation.

The demonstrated environment simulates the topology with Docker containers on one physical machine. Development certificates, local dataset registration, and optional network forwarding are scaffolding for that simulation, not evidence of a deployed multi-institution federation. In a production setting, institutions would normally administer their own workers, credentials, network endpoints, and datasets. The architectural requirement retained by the Integrator is the availability of an authenticated worker identity and dataset reference, not local ownership of every worker.

4.5 Workflow Handler

The Workflow Handler implements the central transformation studied in this thesis. It converts a BraneHub project configuration and the executable context prepared by the other modules into an annotated, validated, and reviewable BraneScript workflow. The pipeline is intentionally staged so that external data, probabilistic interpretations, explicit mappings, generated code, validation evidence, and runtime behaviour remain distinguishable.

4.5.1 Configuration Retrieval and Normalisation

For each workflow cycle, the handler retrieves the current project configuration from BraneHub. The external payload contains project-level governance fields, accepted participants, participant onboarding answers, dataset identifiers, and data-format answers. Package and coordinator information comes from the Integrator's stored project configuration.

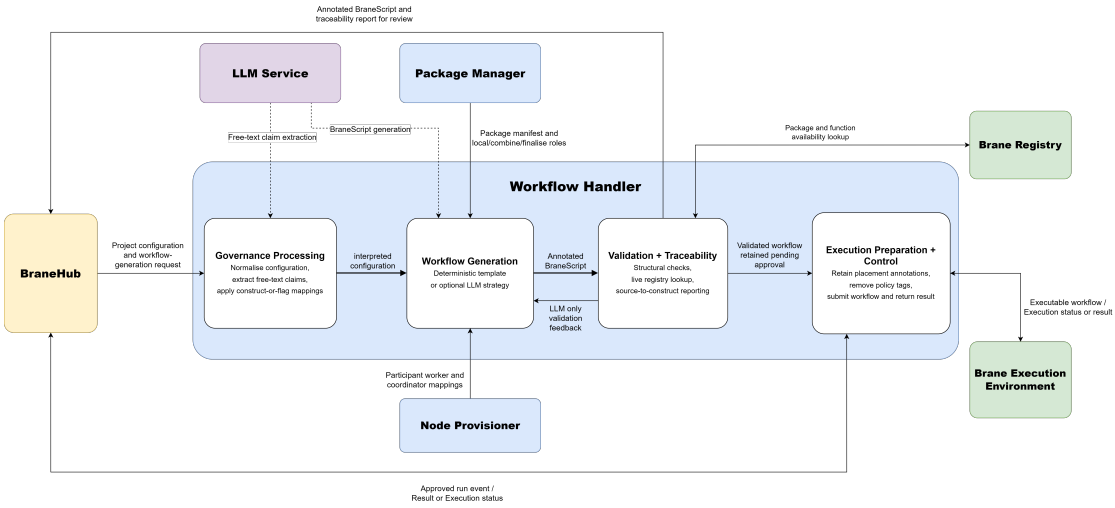

Figure 4.3: Workflow generation, validation, review, and execution flow.

The payload is converted into a typed internal configuration before interpretation. This normalisation has two purposes. First, it decouples policy interpretation and workflow generation from BraneHub's questionnaire identifiers. Changes to an external field mapping can be handled at the boundary rather than being repeated throughout the pipeline. Second, it ensures that workflow generation stops before unchecked values are inserted into BraneScript when a required mapping is missing, such as when an accepted participant has no known Brane worker. Typed models cannot establish that a governance answer is legally or semantically correct, but they provide a defined structural input to the remaining stages.

Only active accepted participants are included. For each participant, the internal configuration contains the mapped Brane worker, dataset identifier, structured onboarding fields, and three designated free-text fields: privacy and legal notes, data provenance, and source of truth. The Workflow Handler operates on these references and descriptions. It does not retrieve the participant's dataset contents during generation.

4.5.2 Free-Text Interpretation

Some governance information is supplied as prose rather than as a value from a fixed schema. Ignoring this text would silently discard information, while copying it into BraneScript would not create a meaningful execution control. The Free-Text Extractor therefore uses an LLM to propose structured claims from the designated fields.

Each proposed claim records its source field, a candidate policy field, a candidate value,

and a confidence category. A structured-output schema constrains the response format. Claims labelled as low confidence are removed before policy interpretation so that weak model guesses do not generate annotations. Medium- and high-confidence claims remain candidates rather than authoritative decisions: confidence is produced by the same model and is not a correctness oracle.

The design retains the original source field in traceability regardless of whether a claim is extracted. This choice distinguishes three cases that would otherwise appear identical: no text was supplied, text was processed but yielded no retained claim, and text produced a claim that was subsequently mapped or flagged. It also allows a reviewer to examine the original statement rather than relying only on the model's reformulation.

4.5.3 Construct-or-Flag Interpretation

After normalisation and free-text extraction, the Policy Interpreter applies explicit registries. Table 4.1 lists the supported mappings and their runtime status in the implemented prototype.

Brane's `tag` and `wf_tag` annotations provide a generic mechanism for attaching namespaced metadata to a task or to the workflow as a whole. They do not define a built-in vocabulary for governance concepts. The Integrator therefore defines a limited, project-specific category-value convention for selected BraneHub fields. Identifiability is associated with a participant's local computation, while project data sensitivity, declared legal basis, and the set of participating jurisdictions are associated with the workflow as a whole. These fields were selected as a bounded prototype subset because they are governance claims with a direct participant-level or workflow-level scope. Their values are taken from the corresponding BraneHub answers, with retained free-text identifiability claims processed through the same mapping. The resulting annotations record declared governance context rather than verifying the claims or enforcing their consequences.

The supported registry is deliberately small. It includes the operational information required to address data and workers, together with a small project-defined set of governance metadata labels expressed through Brane's generic tag syntax. Fields concerning consent, institutional approval, identifiers, audit requirements, connectivity, certifications, agreements, model updates, per-round approval, unlearning, security measures, responsibilities, and result sharing are not converted into invented workflow syntax. They are retained as flags. Claims extracted from free text follow the same rule: a supported participant field can produce a construct, while any other retained claim is flagged.

Source information	Generated representation	Status in the implemented prototype
Participant worker	`#[on("node")]`	Retained in the executable workflow and used by Brane as a placement constraint.
Dataset identifier	Brane `Data` declaration	Retained in the executable workflow and resolved through the Brane data infrastructure.
Identifiability	Participant `#[tag(...)]`	Present in the reviewed script and traceability report, but removed from the executable copy because of the eFLINT integration failure described below.
Data sensitivity and legal basis	Workflow `#![wf_tag(...)]`	Present in the reviewed script and traceability report, but removed from the executable copy.
Participant jurisdictions	Deduplicated workflow tags	Present in the reviewed script and traceability report, but removed from the executable copy.
Other populated governance fields	Explicit traceability flag	No BraneScript construct is generated; institutional, package-level, or other enforcement remains external.

Table 4.1: Supported governance-to-workflow mappings and their runtime status.

This design favours an incomplete but auditable mapping over an apparently comprehensive translation with undefined semantics. A new mapping to the reviewed representation is justified only when its target syntax, intended interpretation, and validation treatment are defined. Claiming the resulting construct as a runtime control additionally requires an implemented enforcement point, corresponding policy rules, and evidence that those rules affect execution as intended. The resulting traceability report can therefore show the boundary of the implemented policy model instead of silently treating every collected field as enforced.

4.5.4 Workflow-Generation Strategies

The interpreted configuration can be passed to one of two BraneScript generators.

4.5.4.1 Deterministic template

The default generator implements a single-pass map-reduce workflow. For each participant, it declares the participant's dataset and places the local function call at the mapped participant worker. It then combines the intermediate values at the coordinator through a binary left fold. For intermediate results $r_1, \ldots, r_N$, the aggregation is

$$a_1 = combine(r_1, r_2), \qquad a_i = combine(a_{i-1}, r_{i+1}) \quad for \quad 2 \leq i \leq N-1.$$

If there is one participant, its local result is used without a combine call. If a finalisation function is configured, it is called once at the coordinator before the result is returned.

Template generation is the default because package roles, participants, datasets, and placements are already represented as typed values. For this supported pattern, an LLM would add cost and nondeterminism without adding necessary information. Determinism does not make the complete workflow semantically correct: the package may implement the wrong computation, the supplied configuration may be wrong, and the validator may not express every desired invariant. It does make the transformation from a valid configuration to the supported workflow structure explicit and repeatable.

The template's limitation is structural. It supports one package and one local-combine-finalise pass. New patterns require new templates and corresponding validation rules. This boundary is preferable to implying that a fixed generator supports arbitrary federated algorithms.

4.5.4.2 LLM-assisted generation

The alternative generator asks an LLM to arrange the supplied components into complete BraneScript. Its prompt includes the exact package name, configured functions, package manifest, participant workers, dataset references, generated policy annotations, coordinator, and optional finalisation function. The model is therefore given the current executable context rather than being asked to recall Brane or project-specific interfaces from general training data. Research on specialised code generation likewise shows the value of supplying external library information, although such grounding does not eliminate generation errors (6, 12).

The LLM strategy provides flexibility for exploring structures beyond the fixed template, but it is not the authority for governance semantics or infrastructure state. Participant selection comes from BraneHub, worker mappings come from the provisioner, package interfaces come from the stored manifest and live registry, and policy constructs come from the explicit interpreter. The strategy should therefore be understood as an alternative arrangement mechanism operating within supplied constraints, not as a general policy reasoner or unrestricted BraneScript synthesiser.

4.5.5 Structural and Registry Validation

Both generated forms pass through the same final validator. The validator checks four categories of properties:

- the expected package import and a return statement are present;
- participant placements, dataset declarations, workflow tags, and participant tags produced by the interpreter occur in the script;
- detected combine calls use two arguments and are associated with the coordinator placement; and
- the package, configured local and combine functions, and additional call-like names detected in the script are available through the live Brane registry.

The live registry check is important because a function name may be syntactically valid while being absent from the approved package. Final validation is therefore performed after an attempt to publish the built package. A workflow that fails the implemented checks is recorded as failed and is not sent to BraneHub for approval.

The script checks use textual patterns and regular expressions. They are not a complete BraneScript parser or compiler. They establish the presence and form of the constructs they inspect, but they do not prove the complete data flow of the program. In particular, they do not prove that every participant's local result contributes to the returned value, that a combine call exists whenever multiple participants are present, or that an optional finalisation function is invoked. They also do not establish numerical correctness, privacy, or the legal sufficiency of the represented governance information.

For the LLM strategy, a failed initial validation is returned to the model once as structured feedback. The resulting script still undergoes the Workflow Handler's final validation. The retry is bounded because validator feedback can correct an identifiable structural defect, whereas repeated self-refinement adds cost and does not guarantee monotonic improvement. Regression errors during iterative code refinement have been reported empirically (13). One retry is an engineering bound for this prototype, not a claim that one iteration is universally optimal.

4.5.6 Traceability and Review

For a successfully validated workflow, the Integrator produces a traceability report alongside the BraneScript. Every relevant non-empty project or participant field is assigned one of the following outcomes:

- a generated construct and the line on which its text occurs;
- a free-text processing outcome and any child claims derived from that source;
- an explicit flag explaining that no supported BraneScript mapping exists; or
- a note describing a known difference between the reviewed and executable scripts.

The report reuses the interpreter's mapping registries. This avoids maintaining one policy mapping for generation and a separate, potentially inconsistent mapping for explanation. Construct line numbers are found through textual matching, so they locate a representation but do not prove semantic equivalence between a source requirement and program behaviour.

Traceability is retained because a reviewer needs to determine not only what appears in the generated script, but also what happened to the supplied governance information. Work on process compliance similarly emphasises connecting requirements to executable processes and explaining why a requirement is or is not enforced (14). Workflow Run

RO-Crate demonstrates the broader role of machine-actionable provenance for workflow plans, executions, inputs, and outputs (15). The Integrator's report is narrower: it records governance-to-script transformation rather than complete retrospective workflow provenance.

The validated script and traceability report are sent to BraneHub for external review. An optional LLM-generated plain-language note may accompany them, but this note is best-effort and does not influence validation. Keeping it outside the trusted decision path prevents an explanatory model response from replacing the structured report or validator result.

4.5.7 Execution and Enforcement Boundary

Execution begins only after BraneHub issues a run event for the generated workflow. The Workflow Handler prepares a temporary executable script, invokes Brane's remote workflow command, and returns the resulting status or value to BraneHub. Brane remains responsible for parsing, planning, resolving named locations, and dispatching package functions across the configured workers (2).

The reviewed and executable scripts are not identical in the current prototype. Placement annotations of the form `#[on("node")]` remain in the executable script, and Brane uses them to schedule the associated computation at the named worker. By contrast, submitting participant and workflow policy tags to the evaluated Brane nightly version caused policy deliberation to fail because the serialised tag assertion was incompatible with the corresponding eFLINT fact representation. The Workflow Handler therefore removes `#[tag(...)]` and `#![wf_tag(...)]` lines from the temporary script submitted to Brane.

The full annotated script remains stored and is sent to BraneHub. Directly mapped participant tags and workflow tags are marked in the traceability report as stripped before execution, while all tag lines, including those produced from extracted free-text claims, are removed from the executable copy. The tags are therefore *represented and reviewable*, but they are not runtime-enforced in the implemented environment. Enabling tag-based enforcement would require a defined vocabulary and source-ownership convention, corresponding eFLINT policy rules, a compatible Brane/eFLINT representation, and allow and deny tests demonstrating that policy deliberation affects workflow execution as intended.

Placement enforcement also has a defined limit. It controls where Brane schedules a package function. It does not prove that the function itself applies a required transformation or that its output cannot reveal individual information. Aggregation, anonymisation,

or other data transformations are effective only when correctly implemented by the approved package. Likewise, the Integrator cannot establish the legal validity of consent, enforce institutional retention procedures, or govern results after they leave the demonstrated environment.

4.6 Assurance and Responsibility Boundaries

Table 4.2 summarises the division of responsibility produced by the design.

Actor	Primary responsibility	Boundary
Institutions and BraneHub	Project membership, governance answers, participant eligibility, approvals, dataset descriptions, and post-execution decisions.	The Integrator consumes these decisions but does not establish their legal validity or replace institutional governance.
Brane Integrator	Package preparation, worker mapping, explicit governance interpretation, workflow generation, implemented validation, traceability, and submission after a run event.	It provides evidence of the transformation it performs. It does not prove package semantics, privacy, or complete compliance.
Brane and package runtime	Package registration, workflow planning, location resolution, and execution of package functions.	Active placement constraints affect scheduling. Stripped policy tags and external organisational obligations are not enforced by the demonstrated runtime.

Table 4.2: Responsibility boundaries in the implemented architecture.

Within these boundaries, the design provides a concrete chain from BraneHub configuration to a reviewable workflow. It makes package interfaces and execution locations explicit, applies deterministic generation where the supported structure is known, constrains probabilistic steps with schemas and registries, checks selected structural and registry properties, and records how governance fields were handled.

The same boundaries identify what the design does not provide. A traceability entry is not proof that the represented requirement is satisfied. A passed validator result is not proof of scientific or numerical correctness. A placement annotation constrains location but does not inspect package behaviour. An advisory privacy assessment does not establish privacy. Finally, policy tags that are removed from the executable script cannot be claimed as runtime controls. These qualifications are properties of the design, not merely

deployment caveats, because they determine the strength of the claims that can be made about the resulting workflow.

4.7 Design Summary

The Brane Integrator separates the preparation of computation packages, execution locations, and policy-driven workflows. The Package Manager establishes an approved executable interface. The Node Provisioner maps participants and projects to the worker topology required by Brane. The Workflow Handler retrieves and normalises governance information, extracts candidate claims from designated free-text fields, applies explicit construct-or-flag mappings, generates BraneScript through a deterministic or LLM-assisted strategy, validates selected structural and registry properties, and produces traceability for external review before execution.

The principal design decision is to preserve the difference between information that was supplied, information that was interpreted, information that was represented, and information that was enforced. In the implemented prototype, worker placement remains an active execution constraint. Policy tags are retained for review and traceability but removed before Brane execution because of the eFLINT integration failure. Requirements without a supported workflow representation remain explicit flags. The resulting system therefore demonstrates a bounded and auditable form of policy-driven federated workflow generation without claiming automatic end-to-end compliance.

5

Evaluation

This chapter evaluates whether the implemented system can transform governance requirements into traceable and executable federated workflows. The evaluation addresses governance representation (RQ1), static and runtime enforcement (RQ2), comparison of the BraneScript generators (RQ3), and reliability of the LLM-generated bridge artefacts (RQ4).

The chapter reports the experimental method and results. Their broader interpretation is reserved for the Discussion chapter, while limitations of the experimental design are considered in the Threats to Validity chapter.

5.1 Evaluation Method

5.1.1 Evaluation Principles

Functional measures such as *pass@k* are widely used to evaluate LLM-generated code (16). However, general-purpose benchmarks do not fully measure correctness in domain-specific languages, where generated code must follow specialised syntax, semantics, and interfaces (4). BraneScript has these requirements because a valid workflow must use Brane-specific package functions, annotations, locations, and data flows.

Syntactic validity also does not establish reliability or deployability. Generated code may execute while misusing real APIs (17), and valid infrastructure configurations may still fail during deployment (18). The present evaluation therefore applies a sequence of increasingly strong correctness checks:

1. validation of the generated file's schema and syntax;
2. static validation of governance and workflow rules;

3. verification that generated calls match the declared package interfaces;
4. container build and registry push;
5. execution of the workflow on Brane; and
6. comparison of the execution result with a centralised numerical baseline.

An LLM is not used as the final correctness judge because LLM judges can misclassify generated programs (19). Correctness is instead determined using deterministic validators, manually specified gold entries, package interfaces, execution outcomes, and numerical comparisons.

5.1.2 System and Evaluation Sets

Brane provides the multi-site execution environment. It encapsulates computational functionality in containers and orchestrates it using a domain-specific language (2). The evaluated implementation operationalises a focused subset of the compliance-aware federated-processing architecture proposed in (1), specifically the translation of structured and free-text governance information into traceable Brane workflow artefacts, followed by validation and federated execution.

The evaluation uses:

- eight governance scenarios and their expected-valid BraneScript workflows;
- ten deliberately invalid BraneScript files;
- five representative federated packages;
- one four-participant workflow;
- one two-round federated k-means task; and
- 18 manually specified governance entries.

The eight governance scenarios and the 18-entry extraction gold set serve different purposes. The scenario set evaluates end-to-end representation of all populated in-scope configuration fields, whereas the gold set evaluates the accuracy of converting individual free-text inputs into structured governance claims. Each gold entry was evaluated in five independent LLM calls, producing 90 entry–run observations and 90 extraction calls.

The packages are `fed_mean`, `fed_logreg`, `fed_histogram`, `fed_variance`, and `fed_single`. They cover different function signatures, return types, local computations, and aggregation requirements. The first four permit comparison with a centralised baseline. The fifth has no coordinator-side combination action and therefore tests whether a generator invents unsupported package functionality.

Two BraneScript generation strategies are compared. The template strategy uses a deterministic workflow structure encoded in the implementation. The LLM strategy generates BraneScript from the supplied project, participant, governance, and package information.

Package generation is evaluated through two additional paths. Path A asks the LLM to generate both a Python implementation and its `container.yml` manifest. Path B provides an existing Python implementation and asks the LLM to generate only its `container.yml`, thereby evaluating whether the model can correctly identify the functions, parameters, and types present in the existing code.

5.1.3 Governance and Compliance Measures

Not every governance statement has a direct executable representation. RQ1 therefore measures whether each source field is accounted for, rather than only counting the fields converted into BraneScript.

A field is counted as represented when it produces a generated BraneScript construct, is consumed by another component, or is explicitly documented as non-enforceable. Construct-generating traceability entries also contain source-line references so that each generated construct can be connected to its location in the reviewed script. Connecting workflow artefacts with their inputs and execution context supports traceability and reproducibility (15).

RQ2 separates static verification from runtime enforcement. Metrics M2.1–M2.3 therefore evaluate the custom validator, while M2.4 separately evaluates which policy constructs reach the Brane and eFLINT runtime layers.

5.1.4 Generated-Code Measures

The LLM receives the actual package manifest as part of its BraneScript generation context. Providing real library information is important when the target environment contains specialised functions not reliably represented in the model's general knowledge (12).

The custom validator expresses nine workflow requirements as machine-checkable rules. Explicit software constraints provide a stronger generation and evaluation target than natural-language requirements alone (9).

Execution and numerical tests are used wherever an executable oracle is available. Test-based evaluation makes intended behaviour explicit (10), while outcome-based evaluation distinguishes functional behaviour from surface-level correctness (20).

5.1.5 Repetition, Consistency, and Cost

Each principal LLM experiment was repeated five times. Consistency is measured using Jaccard similarity:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}. \tag{5.1}$$

BraneScript is compared using normalised source-line sets. Extraction output is compared using sets of (*field*, *value*) claims. Manifests are compared using (*action*, *parameter*, *type*) triples. Generated Python is compared using source-line sets and sets of normalised AST tokens, with identifier names replaced by positional placeholders. Because the AST comparison applies Jaccard similarity to token sets, token ordering and multiplicity are not preserved; it is therefore a structural consistency measure rather than a functional-equivalence oracle.

Regeneration attempts, latency, token use, and monetary cost are recorded. Regeneration is not assumed to be beneficial because additional refinement can introduce regressions and increase inference cost (13).

All experiments used `gpt-4o`. Temperature was not explicitly set and therefore followed the API default. The production prompts are located in `app/application/utils/prompts.py`. Structured output is used for extraction and package artefacts where implemented. BraneScript is returned as free-form text and is subsequently post-processed and validated.

Extraction precision, recall, and F_1 are calculated against 18 manually specified gold entries. The reported F_1 is the macro-average of the per-entry, per-run values. It is not calculated from the final aggregate precision and recall. This explains why $P = 0.822$ and $R = 0.933$ accompany a macro-F_1 of 0.841.

The experiments primarily use descriptive statistics. Several metrics achieved a result of 1.0, meaning that every item in the specified test set passed its corresponding correctness check. However, the test set was relatively small and was not designed as a comprehensive adversarial benchmark. These perfect scores may therefore reflect a ceiling effect and

should not be interpreted as evidence of correctness across all possible projects, governance policies, workflows, or package configurations.

The recorded evaluation ran on one physical machine under WSL2 and Docker Desktop, using Python 3.12.3, a Brane 3.0.0 nightly build, and `gpt-4o` through the Chat Completions API. The environment comprised Brane's central services and three logical worker nodes: `participant-1`, `participant-3`, and `coordinator-1`. For reproducibility, the exact software revision, test inputs, package descriptions, datasets, prompts and configurations, evaluation scripts, reference outputs, rerun commands, and known limitations are collected in the public evaluation guide.[1]

5.2 Results

Table 5.1 defines every metric used in the evaluation while grouping related measures by experimental purpose.

Table 5.1: Consolidated evaluation results with explicit metric identifiers.

RQ	Metric group	*N*	Metrics and results	Note
RQ1	Governance representation (M1.1–M1.3)	324 in-scope configuration fields in 8 scenarios	**M1.1** field coverage = 1.0 (324/324); **M1.2** construct-to-flag ratio = 0.37; **M1.3** construct source-line verifiability = 1.0	Every populated in-scope configuration field was represented as a construct, processing outcome, or explicit flag. Line verifiability applies to construct-generating entries.
RQ2	Custom static validation (M2.1–M2.3)	8 valid workflows; 10 invalid workflows; 2 locality fault types	**M2.1** valid-workflow acceptance = 8/8; **M2.2** locality faults detected = 2/2; **M2.3** invalid workflows detected = 10/10	The nine-rule validator accepted the valid set and rejected every injected fault.
RQ2	Runtime enforcement (M2.4)	Qualitative integration test	**M2.4** runtime enforcement = partial	Location annotations reached runtime, but eFLINT policy tags could not be serialised.

Continued on next page

[1] `https://github.com/aditya-130/brane-integrator/blob/evaluation/evaluation/README.md`

RQ	Metric group	N	Metrics and results	Note
RQ3	Main generator comparison (M3.1–M3.3)	8 template and 40 LLM outputs; 5 executions per strategy	**M3.1** first-pass validity: both 1.0; **M3.2** execution: template 5/5, LLM 4/5; **M3.3** validation-failure distribution: no failures	The LLM execution failure called an unsupported combination action in `fed_single`.
RQ3	Stability and efficiency (M3.4–M3.6, M3.8, M3.10)	40 LLM calls; 80 output pairs	**M3.4** regenerations = 0; **M3.5** calls per output = 1.0; **M3.6** latency $\approx$ 3.3 seconds; **M3.8** line-set $J = 0.80$; **M3.10** cost = USD 0.298 total	Template generation was effectively immediate and incurred no model cost.
RQ3	Scale and extensibility (M3.7, M3.9)	1 four-party scenario; 1 template and 3 LLM k-means outputs	**M3.7** four-party workflow: both generators passed; **M3.9** iterative structure: template 0/3, LLM 3/3	The k-means result has $N = 3$, was assessed structurally, and was not executed as a Brane package.
RQ4	Build and execution (M4.1, M4.2, M4.4)	5 packages; 8 numerical executions; 20 local outputs	**M4.1** numerical equivalence = 8/8; **M4.2** accumulator-shape contract = 20/20; **M4.4** build/push success = 5/5	Applicable federated executions matched their centralised baselines. One representative generated variant of each package was built and pushed.
RQ4	Path A package artefacts (M4.5, M4.6, M4.14)	25 outputs	**M4.5** type accuracy = 1.0, hallucination rate = 0.16; **M4.6** schema validity = 25/25; **M4.14** manifest $J = 0.921$	Four outputs invented unsupported combine or finalise actions for `fed_single`.
RQ4	Path B manifests (M4.7)	25 outputs	**M4.7** schema validity = 25/25, structural $J = 1.0$	All 25 manifests were schema-valid, and repeated outputs for the same package had identical action–parameter–type structures.

Continued on next page

RQ	Metric group	N	Metrics and results	Note
RQ4	Governance extraction (M4.8–M4.11)	18 gold entries × 5 runs = 90 entry–run observations and 90 LLM calls; 95 returned claims	**M4.8** precision = 0.822; **M4.9** recall = 0.933, macro-F_1 = 0.841; **M4.10** extraction $J = 0.944$; **M4.11** confidence: 92 high, 3 medium	Confidence was model-produced metadata and was not used as a correctness oracle.
RQ4	Generated Python (M4.3, M4.12, M4.13)	25 outputs; 20 numerically applicable	**M4.3** privacy correctness: unverified; **M4.12** raw accuracy = 14/20, adjusted = 17/20; **M4.13** line $J = 0.457$, AST-token $J = 0.941$	The adjusted result treats three baseline-rounding mismatches as measurement artefacts and retains three genuine deviations.

5.2.1 RQ1: Governance Representation

The independent enumeration procedure identified 324 populated in-scope configuration-field instances across the eight scenarios. All 324 were represented in the traceability output, giving the M1.1 field-coverage score of 1.0. Every construct-generating traceability entry also contained a verifiable source-line reference, giving an M1.3 score of 1.0.

The average M1.2 construct-to-flag ratio was 0.37. This means that construct-generating traceability entries were less frequent than explicitly flagged entries. The ratio should not be interpreted as the percentage of source fields that became executable constructs. The coverage result therefore shows that no populated field disappeared silently. It does not mean that every field was technically enforced.

5.2.2 RQ2: Static and Runtime Enforcement

For M2.1, all eight valid workflows passed the nine-rule custom validator. All ten fault-injection workflows were rejected under M2.3, while M2.2 detected both tested locality fault types.

Runtime enforcement was more limited. Brane location annotations controlled whether computations were assigned to participant or coordinator locations. However, the `#[tag()]` and `#![wf_tag()]` annotations caused a serializer failure and were removed before execution. M2.4 therefore reports only partial runtime enforcement.

M2.1–M2.3 demonstrate the behaviour of the custom validator. They do not demonstrate

that eFLINT enforced the same rules at runtime. RQ2 is consequently answered with complete detection within the static test set and partial runtime enforcement.

5.2.3 RQ3: Comparison of BraneScript Generators

Under M3.1, both generators achieved first-pass syntactic validity of 1.0. The M3.3 rule-failure distribution was empty because none of the 48 main-scenario workflows failed the custom validator. Correspondingly, M3.4 recorded zero regenerations and M3.5 recorded one LLM call per output.

Execution exposed a difference not visible in the static results. M3.2 recorded successful execution for all five template workflows, compared with four of the five LLM workflows. The failed LLM workflow called a coordinator combination action that was not declared by `fed_single`. Invented objects and unsupported attributes are recognised failure patterns in LLM-generated code (5).

For M3.7, both generators successfully constructed the four-participant aggregation chain. In the two-round k-means experiment used for M3.9, the fixed template satisfied none of the three required structural properties, whereas all three LLM outputs satisfied them. This result has $N = 3$ and was assessed structurally rather than through Brane execution.

The M3.8 pairwise line-set similarity of the LLM BraneScripts was 0.80. M3.6 recorded a mean generation latency of approximately 3.3 seconds. Together with M3.5, this corresponds to one call and zero regenerations per output. Under M3.10, the 40 BraneScript-generation calls cost USD 0.298 in total, or approximately USD 0.0075 per output. Template generation was effectively immediate and incurred no model cost.

5.2.4 RQ4: Reliability of the Bridge Artefacts

For M4.4, one representative generated variant of each of the five packages was built and pushed successfully. Under M4.1, all eight applicable federated executions matched their centralised baselines.

All 20 applicable local outputs also satisfied the M4.2 accumulator-shape contract: the output of the combination action had the same top-level key set, or the same output type, as the local result. This shape check does not by itself establish semantic correctness, which is evaluated separately through numerical comparison.

For Path A, M4.6 found all 25 generated manifests to be schema-valid. M4.5 found that all expected reference types were present, giving type accuracy of 1.0. Nevertheless, four outputs invented unsupported combine or finalise actions for `fed_single`, producing a hallucination rate of 0.16. M4.14 measured Path A manifest consistency at $J = 0.921$.

For Path B, all 25 manifests evaluated by M4.7 were schema-valid and structurally identical, producing $J = 1.0$. This experiment generated a manifest from an existing

Python implementation and did not require the model to generate the computation itself.

For governance extraction, M4.8 measured precision at 0.822, while M4.9 measured recall at 0.933 and macro-F_1 at 0.841. M4.10 measured extraction consistency at 0.944. Of the 95 returned claims examined by M4.11, 92 were marked high confidence and three medium confidence. Confidence was model-produced metadata and was not treated as evidence of correctness.

For M4.12, 14 of the 20 numerically applicable generated Python outputs passed the raw comparison. Three additional mismatches were caused by rounding in the baseline measurement rather than errors in the generated computation. After treating these three baseline-rounding mismatches as measurement artefacts rather than LLM errors, 17 of the 20 outputs were counted as correct, giving an adjusted accuracy of 0.85. The three remaining deviations involved an incorrect logistic-regression sign, mishandling of a CSV header, and use of an unintended Bessel correction.

M4.13 measured raw line-set similarity at 0.457 and normalised AST-token-set similarity at 0.941. The gap is consistent with substantial variation in formatting and identifier selection. However, because the AST comparison uses token sets rather than an executable functional-equivalence oracle, numerical testing was still required to identify behavioural differences.

Privacy correctness was not measured. Location annotations control where a function executes but do not determine whether its returned value exposes sensitive data. Functionally correct generated code may still contain security or privacy defects (20, 21). M4.3 is therefore reported as unverified.

5.2.5 Overall LLM Cost

Across the four cost-accounted LLM experiments, 180 calls were made at a total cost of USD 1.171. These comprised 90 governance-extraction calls (one for each of 18 gold entries across five runs), 40 BraneScript-generation calls, 25 Path A package-generation calls, and 25 Path B manifest-generation calls. Governance extraction accounted for USD 0.288, BraneScript generation for USD 0.298, Path A package generation for USD 0.448, and Path B manifest generation for USD 0.137.

The BraneScript portion of these measurements constitutes M3.10. Path A was the most expensive operation because it generated Python code in addition to the package interface. These figures measure the four evaluated model-usage categories only. The extraction subtotal covers the controlled 18-entry benchmark and not separate LLM use during end-to-end traceability preparation. The figures also exclude infrastructure operation, human review, and the separate Path B privacy-review call.

5.3 Evaluation Summary

For RQ1, M1.1 showed that every populated in-scope configuration field was represented in the traceability output, while M1.2 produced an average construct-to-flag ratio of 0.37 and M1.3 verified the source line of every construct-generating entry. For RQ2, M2.1–M2.3 showed complete detection within the static test set, whereas M2.4 showed that runtime policy enforcement remained partial.

For RQ3, the generators tied on first-pass validity, but the template achieved complete execution success while the LLM produced one unsupported function call. The LLM nevertheless expressed the tested iterative structure that the fixed template could not represent. For RQ4, the generated bridge artefacts were consistently schema-valid, but interface hallucinations, numerical deviations, and the unverified privacy property remained detectable only through grounded and execution-based oracles.

These results provide the empirical basis for the following Discussion chapter.

6

Discussion

The evaluation supports a bounded conclusion about the Brane Integrator. Within the implemented scope, the system can transform selected BraneHub project and participant information into a reviewable BraneScript workflow, execute the supported federated computation pattern, and record how the supplied governance information was handled. The evidence does not support describing the Integrator as a general compliance compiler. It does not establish legal compliance, privacy, scientific correctness, or support for arbitrary federated algorithms. The main result is therefore the feasibility of a traceable and policy-driven bridge between governance configuration and Brane execution, together with an explicit account of where that bridge currently stops.

6.1 RQ1: Governance Representation

Across the eight governance scenarios, all 324 populated in-scope configuration-field instances were represented in the traceability output. Every construct-generating entry had a source-line reference, while the average construct-to-flag ratio was 0.37. This ratio does not mean that 37% of the source fields were enforced. It shows that construct-generating entries were less common than explicit flags. The significant result for RQ1 is therefore that no populated in-scope field disappeared silently in the evaluated scenarios, not that every governance statement became executable policy.

Accordingly, the implemented answer to RQ1 is a layered representation rather than a simple code-or-flag distinction in which every generated construct is necessarily enforced. The representation retains the source field and value, records whether the field produced a construct, a processing outcome, or a flag, gives a reason when no supported mapping exists, and records the location of a construct when one was generated. It also records known differences between the reviewed and executable scripts. This distinction is consistent with the pre-exchange, exchange, and post-exchange stages described in (1). Retaining

the source and transformation outcome also supports the traceability and accountability goals associated with workflow provenance (15).

The representation result must nevertheless be interpreted narrowly. A source-line reference is obtained by textual matching and shows where a generated string occurs. It does not prove that the construct implements the legal or organisational meaning of its source field. Similarly, complete field coverage applies to the eight evaluated scenarios and the fields classified as in scope. It is evidence that the implemented reporting process accounted for those inputs, but it is not evidence that all possible governance requirements can be represented or enforced.

6.2 RQ2: Static and Runtime Enforcement

The custom validator accepted all eight expected-valid workflows, rejected all ten deliberately invalid workflows, and detected both tested locality fault types. These results show that the nine implemented rules distinguished the valid and injected-invalid cases in the evaluation set. The live registry query also grounds package and function references in Brane's current catalogue. However, the remaining checks use textual patterns and regular expressions rather than a complete BraneScript parser or compiler. The validator is therefore a useful pre-submission guard for the properties it expresses, but a passing result is not a proof of complete program correctness or compliance.

Runtime evidence was more limited. The `#[on("node")]` annotations remained in the executable workflow and affected where Brane assigned participant and coordinator computations. By contrast, the generated participant and workflow metadata annotations caused policy deliberation to fail in the evaluated Brane and eFLINT integration because the serialised tag assertion was incompatible with the corresponding eFLINT fact representation. The Workflow Handler consequently removed `#[tag(...)]` and `#![wf_tag(...)]` lines from the temporary executable script. These tags remained visible in the reviewed script and traceability report, but they were not runtime controls. RQ2 is therefore answered with complete detection within the specified static test set and partial runtime enforcement, with node placement as the demonstrated active governance-related constraint.

This boundary matters because policy enforcement can apply before, during, and after access (3). The current Integrator performs selected checks before submission and relies on Brane for runtime placement. It does not continuously monitor every policy condition, enforce institutional procedures, or govern the use of results after execution. Fields without a supported runtime mechanism are therefore more accurately treated as review obligations than as automatically enforced policies.

6.3 RQ3: BraneScript Generation Strategies

In the eight governance-scenario generation experiment, both strategies achieved first-pass validity under the evaluated checks, and none of the 48 generated workflows triggered a validator-based retry. The separate five-package execution experiment exposed a difference that the scenario-level checks did not establish. All five template workflows executed, while four of the five LLM-generated package workflows executed. For `fed_single`, the LLM called an unsupported `combine_results` function even though the package exposed only its local action. This is consistent with the project-context conflicts reported for LLM-generated code, where generated code can contradict interfaces available in the target project (6).

The iterative experiment demonstrated the opposite side of the trade-off. The fixed template satisfied none of the three tested structural properties of the two-round federated k-means task, while all three LLM outputs satisfied them. This result shows that the LLM could arrange the supplied functions into the tested iterative structure that the current template could not express. It does not show that these outputs were executable or numerically correct, because the hypothetical package was not registered and the outputs were evaluated structurally with $N = 3$.

These findings justify retaining the deterministic template as the default for the supported single-pass local-combine-finalise pattern while treating LLM generation as an alternative for structures outside that template. Template generation was effectively immediate and incurred no model cost. The 40 LLM generations required one call per output, averaged approximately 3.3 seconds, had a line-set consistency of approximately 0.80, and cost USD 0.298 in total. Cost was modest in this experiment, but determinism, repeatability, and execution reliability are stronger reasons for the default. The broader difficulty of generating code for specialised domain-specific languages supports this caution (4). Supplying the package manifest and explicit constraints is still justified by work using function specifications and design constraints to ground generation (9, 12), but the `fed_single` result shows that grounding does not eliminate interface errors when the supplied instructions and actual package interface disagree.

6.4 RQ4: Reliability of the Bridge Artefacts

The package experiments show that structural validity and semantic correctness must be separated. In Path A, all 25 generated manifests were schema-valid and contained the expected reference types, but four outputs invented unsupported combination or finalisation actions for `fed_single`, giving a hallucination rate of 0.16. The implementation assigns the generated-package path three fixed roles and its prompt requires all three functions. The `fed_single` finding should therefore be attributed to the complete generation path,

including this fixed prompt assumption, rather than to the model in isolation. In Path B, all 25 manifests generated from existing Python code were schema-valid, and repeated outputs for each package had identical action-parameter-type structures. Within this evaluation, the narrower task of deriving an interface from supplied code was therefore more structurally consistent than jointly generating the computation and its manifest.

Execution-based results further illustrate why schema checks are insufficient. One representative generated variant of each package was built and pushed successfully. In the workflow execution experiment, all eight numerically applicable template and LLM executions matched their centralised baselines. The separate generated-Python experiment produced 14 strict matches among 20 numerically applicable outputs. Three additional mismatches resulted from rounding in the baseline measurement, giving an adjusted result of 17/20. The three remaining deviations involved an incorrect logistic-regression sign, mishandling of a CSV header, and an unintended Bessel correction. A valid manifest, successful build, or compatible accumulator shape therefore does not establish that the generated computation implements the intended mathematics. Numerical comparison remains necessary when an executable oracle is available.

Free-text governance extraction also produced useful but imperfect results. Across 90 entry-run observations, precision was 0.822, recall was 0.933, macro-F_1 was 0.841, and extraction consistency was 0.944. The model labelled 92 of the 95 returned claims as high confidence and three as medium confidence, but this confidence was produced by the same model and was not used as a correctness oracle. These results support using the extractor to propose candidate claims while retaining the original source text and subjecting the resulting mappings to review. They do not support treating extracted claims as authoritative policy interpretations.

Privacy correctness remained unverified. Running a function at a participant worker constrains its location, but it does not determine whether the function returns an aggregate, an individual record, or another sensitive value. The focused privacy assessment is advisory, applies only to the uploaded-code path, and treats a failed assessment call as finding no concern. Functional correctness and security or privacy correctness are distinct evaluation dimensions, as also observed in outcome-based evaluations of generated code (20). The answer to RQ4 is therefore mixed: the bridge artefacts were consistently schema-valid in the evaluated cases, and many produced correct executable results, but interface hallucinations, extraction errors, numerical deviations, and unverified privacy prevent a general reliability claim.

6.5 Implications and Future Work

The Integrator's contribution is not the automation of every governance decision. It connects governance configuration to a constrained federated workflow while distinguishing information that is collected, interpreted, represented, and enforced. In this thesis, "policy-driven" therefore means that governance information influences placement, metadata annotations, validation, traceability, and review. It does not mean that every supplied policy is automatically enforced by the demonstrated runtime.

Future work should first formalise the current project-defined metadata vocabulary as a versioned schema specifying supported fields, value types, scope, and intended interpretation. This schema should distinguish requirements checked before execution, conditions monitored during execution, and obligations concerning outputs after execution, reflecting the wider policy lifecycle described in (1, 3). The Brane and eFLINT metadata representations must also be made compatible and tested with both permitted and prohibited cases before the generated annotations can be treated as runtime controls. Brane's task-status and audit mechanisms could provide inputs for continuous monitoring, but logging alone would not establish enforcement.

The supported workflow structures should also be broadened beyond the current single-pass local-combine-optional-finalise pattern. Recurring federated workflow patterns could be identified and implemented as additional deterministic templates or through a typed intermediate representation. Package roles should be grounded in the authoritative manifest, particularly for AI-generated packages. Retrieval-augmented generation could additionally provide version-matched Brane documentation, manifests, schemas, and approved examples to the LLM (12). These extensions should be accompanied by parser or compiler integration, stronger interface and data-flow checks, and execution-based evaluation. Compilation would provide stronger evidence of language-level correctness, but would still not prove scientific correctness, privacy, or legal compliance.

A production study should evaluate the Integrator across infrastructure controlled by multiple institutions rather than separate Brane environments on one host. This would require secure node enrolment, institution-controlled provisioning, package distribution, service discovery, and recovery from network or node failures. The evaluation should also be repeated by independent users using held-out packages, additional workflow patterns, varied governance scenarios, and operational failure cases. Privacy and security require dedicated outcome-based tests because correct placement and numerical results do not establish that sensitive information was protected.

Overall, the results demonstrate a bounded and auditable approach to policy-driven federated workflow generation. Within the evaluated scenarios, in-scope governance fields were accounted for, selected workflow properties were checked before execution, and node placement was enforced through Brane. Deterministic generation was more reliable within

the implemented pattern, while LLM generation demonstrated additional structural flexibility and risk. Extending the approach therefore requires stronger interface grounding, validation, runtime evidence, and independent evaluation rather than broader claims of automatic compliance.

7

Threats to Validity

This chapter considers threats affecting the interpretation and generalisation of the evaluation. The discussion distinguishes internal, external, construct, and conclusion validity. The mitigations described below reduce particular risks, but do not eliminate them.

7.1 Internal Validity

The principal internal threat is that the Integrator, its prompts, evaluation scenarios, expected outputs, and tests were developed within the same project. This creates a risk that the selected cases and oracles reflect assumptions made during implementation. The evaluation reduces this risk by combining several forms of evidence: explicit validator rules, deliberately injected faults, live package-interface queries, container builds, Brane execution, and comparison with centralised numerical baselines. Raw and adjusted Python accuracy are also reported separately, making the treatment of three baseline-rounding mismatches visible. Nevertheless, the evaluation was not performed independently, and confirmation bias cannot be excluded. Transient API, Docker, and Brane conditions may additionally affect latency and execution measurements, so these measurements should be interpreted as observations from the demonstrated environment rather than stable performance benchmarks.

7.2 External Validity

The evaluation covers eight governance scenarios, ten invalid workflows, five federated packages, one four-participant workflow, one structurally assessed iterative workflow, an 18-entry extraction gold set evaluated across 90 entry-run observations, 25 Path A package-generation outputs, and 25 Path B manifest-generation outputs. These cases exercise multiple interfaces and failure modes, but they do not represent every possible package,

governance requirement, or federated computation. The experiments also use one LLM, `gpt-4o`, one evaluated Brane nightly version, and participant and coordinator environments deployed on the same host. A real federation would introduce independent institutional control, wider network conditions, and additional operational failures. The findings are therefore limited to the evaluated system configuration and should not be generalised to other models, Brane versions, workflow patterns, or legal environments without further testing.

7.3 Construct Validity

Several measures intentionally capture narrower properties than their names might otherwise suggest. Governance representation means that a populated in-scope field was accounted for through a generated construct, use by another component, a recorded processing outcome, or an explicit flag. It does not mean that the field was enforced. Similarly, a source-line reference establishes that generated text occurs in the reviewed script, but not that the construct implements the legal or organisational meaning of its source. The nine-rule validator checks selected structural and placement properties using textual patterns. Passing these checks does not establish complete BraneScript correctness, scientific validity, privacy, security, or legal compliance. The iterative workflow experiment evaluates three structural properties and does not establish execution or numerical correctness. Jaccard and normalised AST-token-set similarities measure textual or structural consistency rather than functional equivalence because token ordering and multiplicity are not preserved. Where executable or interface-level oracles were available, these limitations were reduced through container builds, live package-interface checks, Brane execution, and numerical comparison. Privacy correctness remained unverified because correct placement and numerical output do not show whether a package reveals sensitive information.

7.4 Conclusion Validity

The evaluation primarily uses descriptive statistics over bounded test sets. Several measures achieved a score of 1.0, but these results show complete success only within the corresponding evaluated cases and may reflect a ceiling effect. Repeating the principal LLM experiments five times reduces dependence on a single generated output, while the iterative experiment includes three LLM outputs. These repetition counts remain too small to characterise the full distribution of model behaviour or support broad statistical conclusions. The absence of an established Brane or BraneScript benchmark also prevents comparison against a standard reference set. Consequently, the evidence supports the reported conclusions about the implemented scenarios, including the observed relia-

bility difference between the deterministic and LLM generators, but it does not establish universal reliability, general policy enforcement, or end-to-end compliance.

8

Related Work

Existing research overlaps with this thesis in three main areas: compliance-aware federated data processing, machine-actionable policy representation and verification, and the controlled generation of executable artefacts using large language models. The works discussed below address important parts of this problem, but differ in their execution environment, enforcement model, or treatment of generated workflows. These distinctions position the Brane Integrator as a bridge between governance-oriented project configuration and executable federated workflows, rather than as a general-purpose policy or compliance engine.

Compliance-aware federated data processing. (1) presents a lifecycle-oriented approach to compliance management for federated data processing. The proposed architecture connects BraneHub and Brane with policy-as-code techniques and LLM-assisted processing across stages before, during, and after data exchange. It identifies requirements including participant onboarding, policy translation, controlled data access, monitoring, audit, and post-exchange obligations. This provides the closest architectural context for the present thesis. However, the scope of that work is broader than the system evaluated here and does not centre on a controlled comparison of concrete BraneScript-generation strategies. In contrast, this thesis implements and evaluates a specific BraneHub-to-Brane bridge. It retrieves project and participant configuration, translates supported governance fields into workflow artefacts, flags unsupported requirements, validates generated BraneScript, and executes accepted workflows in a locally simulated Brane federation. Consequently, the contribution is a bounded and operational translation pipeline rather than a complete implementation of the wider compliance lifecycle.

Machine-actionable data-use policies. (22) extends the Data Use Ontology by representing its textual data-use conditions as explicit policies using the Open Digital Rights Language and the Data Privacy Vocabulary. The approach distinguishes permissions, prohibitions, duties, purposes, and legal concepts, and demonstrates how dataset offers

and data-use requests can be matched to produce an agreement. This work shows that attaching a code or label to data is insufficient when the intended policy consequences remain implicit. The present thesis addresses a related representation problem, but uses a smaller project-defined vocabulary for sensitivity, legal basis, jurisdiction, and identifiability. These values are represented as Brane metadata annotations and included in the workflow traceability report. Unlike the ODRL-based approach, the Integrator does not define formal permission, prohibition, obligation, conflict-resolution, or policy-matching semantics for those values. Moreover, the annotations are removed from the executable script because of the identified Brane/eFLINT incompatibility. They therefore support governance representation and review, but are not demonstrated runtime controls.

Static and dynamic policy enforcement. (23) introduces RuleKeeper, a GDPR-aware framework in which developers describe personal-data processing through a dedicated manifest. The system applies static analysis to check consistency between the manifest and application code and introduces dynamic mechanisms for enforcing supported requirements during execution. Its evaluation considers both the operation of the enforcement mechanisms and their effect on application performance and developer usability. RuleKeeper is relevant because it connects explicit policy representation to concrete enforcement points. The Brane Integrator differs in both purpose and demonstrated capability. It generates federated scientific workflows from project configuration and preserves participant-to-worker placement, dataset references, and governance traceability. Its validator checks selected structural and registry properties of generated BraneScript, but it does not perform semantic code-policy consistency analysis. Most importantly, the current prototype does not provide dynamic enforcement for the generated governance tags. Within the evaluated environment, runtime enforcement is demonstrated for Brane placement annotations, while broader privacy and legal requirements remain represented, flagged, or delegated to external procedures.

Traceable compliance verification. (14) proposes tree-based compliance verification for connecting compliance requirements to process execution. Its approach represents both executable process behaviour and compliance requirements through structured trees, allowing deterministic verification of properties concerning control flow, time, resources, and execution-level data flow. The resulting verification records explain how a requirement was satisfied or where a violation occurred. This is methodologically related to the Integrator's aim of making governance translation inspectable rather than treating workflow generation as an opaque step. The difference lies in the strength and scope of verification. The tree-based approach operates over explicit semantic representations of processes and requirements. The Integrator instead produces a field-level traceability report and

applies textual checks together with live package and function lookup. These checks detect the evaluated annotation, placement, dataset, function, and arity faults, but they do not constitute a formal semantics for BraneScript. They also do not prove complete information flow, numerical correctness, privacy, or legal compliance. The Integrator's traceability should therefore be understood as evidence of accounting and construct placement rather than as a general compliance proof.

Runtime guards for generated workflows. (24) introduces a two-phase method for enforcing company policies in agentic workflows. During the build phase, natural-language policy documents are converted into deterministic guards associated with tool use. During execution, those guards are invoked before the corresponding tool action, creating a concrete runtime decision point. The work also identifies an important limitation: guarding tool calls does not necessarily detect cases in which a required tool action is omitted entirely. This approach is relevant because it separates policy interpretation from runtime enforcement and requires an operational connection between the two. The present thesis makes the same conceptual distinction but reaches a different implementation boundary. The Integrator translates governance information into reviewable BraneScript annotations and validates their expected textual placement. However, those annotations currently cannot reach the eFLINT policy checker because Brane serialises the metadata with an arity that conflicts with its bundled ontology. The execution-time workaround removes the annotations while retaining the full reviewed script. Accordingly, this thesis does not claim runtime enforcement for the metadata vocabulary. Only the retained participant and coordinator placement constructs have demonstrated execution-time effects.

Validation and execution of LLM-generated artefacts. (18) introduces a deployability-centred approach to Infrastructure-as-Code generation. Rather than evaluating generated configuration files only for syntactic correctness, it uses staged feedback from format checks, syntax validation, and live deployment attempts. Its evaluation demonstrates that iterative execution feedback can substantially improve deployment success, while also showing that deployability does not imply complete requirement coverage or security compliance. This distinction closely parallels the evaluation problem in the present thesis: valid BraneScript is not necessarily executable, numerically correct, privacy-preserving, or policy-enforcing. The Brane Integrator differs by targeting federated scientific workflows and by comparing a deterministic template with an LLM generator over the same package and project configurations. It also records how governance fields were translated or flagged. The LLM path allows one validator-feedback retry, whereas the deterministic template directly instantiates the supported workflow structure. Execution and numerical comparisons are evaluated separately from static validity, and broader security and compliance claims remain outside

the demonstrated results.

Together, these works show that the thesis problem cannot be reduced to either policy representation or code generation alone. Machine-actionable governance requires explicit semantics, while runtime enforcement requires operational decision points. Generated executable artefacts require structural validation, environment-aware checks, and execution-based evaluation. The Brane Integrator occupies the intersection of these concerns by translating BraneHub configuration into traceable Brane workflow artefacts and by distinguishing deterministic generation, probabilistic generation, static validation, execution, and numerical evaluation. Its contribution is deliberately narrower than the formal policy and runtime-enforcement systems discussed above: it demonstrates governance accounting and federated placement for a bounded workflow pattern, while exposing unsupported obligations and metadata-enforcement limitations rather than treating them as satisfied.

9

Conclusion

This thesis investigated how governance information collected through BraneHub can be transformed into a traceable federated workflow for execution through Brane. The resulting Integrator is organised around a Package Manager, a Node Provisioner, and a Workflow Handler. Together, these components support package preparation, provision participant and coordinator Brane environments, retrieve project configuration, generate and validate BraneScript, record how governance fields were handled, and submit the resulting computation for execution. The implemented approach combines deterministic processing with LLM-assisted package generation, free-text extraction, and an alternative BraneScript generator.

For governance representation, all 324 populated in-scope field instances across the eight evaluated scenarios were accounted for as generated constructs, processing outcomes, or explicit flags. Construct-generating entries also contained source-line references. This shows that the evaluated reporting process did not silently discard those inputs, but it does not mean that every field was enforced. The nine-rule validator accepted all eight expected-valid workflows, rejected all ten deliberately invalid workflows, and detected both tested locality fault types. Runtime enforcement was partial. Brane enforced the generated node-placement annotations, while the participant and workflow metadata annotations were removed from the executable script because their serialised representation was incompatible with the corresponding eFLINT fact representation.

The generator comparison showed that the deterministic template was more reliable within its supported single-pass local-combine-optional-finalise pattern. All five template workflows executed successfully, compared with four of the five LLM-generated workflows. The LLM nevertheless expressed the three tested structural properties of a two-round iterative workflow that the fixed template could not represent, although those outputs were not executed or assessed for numerical correctness. The bridge-artefact experiments produced consistently schema-valid manifests, but also revealed unsupported function declarations,

imperfect governance extraction, and numerical deviations in generated Python. These findings show that LLM-generated artefacts can support the integration process, but require authoritative interface information, deterministic validation, execution-based testing, and human review. Privacy correctness remained unverified.

The principal contribution is therefore a bounded and auditable bridge between governance configuration and federated execution, rather than a general compliance compiler. Within the evaluated scenarios, governance information influenced workflow construction, placement, validation, traceability, and review, while unsupported requirements remained visible. The prototype was evaluated in a locally simulated federation on one physical host. Deployment across independent organisations was not evaluated. The results demonstrate the feasibility of policy-driven federated workflow generation through Brane, while also establishing clear boundaries around runtime enforcement, workflow coverage, generated-code reliability, privacy, and legal compliance.

References

[1] Natallia Kokash, Adam Belloum, and Paola Grosso. **Compliance Management for Federated Data Processing**, 2026. 1, 2, 5, 7, 12, 13, 29, 38, 42, 47

[2] Onno Valkering, Reginald Cushing, and Adam Belloum. **Brane: A Framework for Programmable Orchestration of Multi-Site Applications**. In *2021 IEEE 17th International Conference on eScience*, pages 277–282. IEEE, 2021. 1, 2, 6, 11, 15, 25, 29

[3] Ines Akaichi and Sabrina Kirrane. **A Comprehensive Review of Usage Control Frameworks**. *Computer Science Review*, **56**:100698, 2025. 5, 13, 39, 42

[4] Sathvik Joel, Jie Wu, and Fatemeh H. Fard. **A Survey on LLM-Based Code Generation for Low-Resource and Domain-Specific Programming Languages**. *ACM Transactions on Software Engineering and Methodology*, 2025. 13, 28, 40

[5] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. **Bugs in Large Language Models Generated Code: An Empirical Study**. *Empirical Software Engineering*, **30**(3):65, 2025. 13, 35

[6] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. **LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation**. *Proceedings of the ACM on Software Engineering*, **2**(ISSTA):481–503, 2025. 13, 23, 40

[7] Fang Liu, Yang Liu, Lin Shi, Zhen Yang, Li Zhang, Xiaoli Lian, Zhongqi Li, and Yuchi Ma. **Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code**. *IEEE Transactions on Software Engineering*, **52**(3):1037–1055, 2026. 13

[8] Ruben Opdebeeck, Mahmoud Alfadel, Akond Rahman, Yutaro Kashiwa, João F. Ferreira, Raula Gaikovina Kula, and Coen De Roover. **An Empirical Study of Policy as Code: Adoption, Purpose, and Maintenance**. In *Proceedings of the 23rd International Conference on Mining Software Repositories*. Association for Computing Machinery, 2026. 13

[9] Luke Newcomb, Alexandra Newcomb, and Omar Ochoa. **Preconditions and Postconditions as Design Constraints for LLM Code Generation**. *IEEE Access*, **13**:184259–184274, 2025. 13, 31, 40

[10] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. **LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation**. *IEEE Transactions on Software Engineering*, **50**(9):2254–2268, 2024. 13, 31

[11] Noble Saji Mathews and Meiyappan Nagappan. **Test-Driven Development and LLM-Based Code Generation**. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1583–1594. Association for Computing Machinery, 2024. 13

[12] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. **LLM-Based and Retrieval-Augmented Control Code Generation**. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 22–29, 2024. 16, 23, 30, 40, 42

[13] Lucas Teixeira Borges and Ricardo Rios. **One Step Forward, Two Steps Back: Regression Errors and Cost Inefficiencies in LLM Iterative Refinement for Code Generation**. In *I Can't Believe It's Not Better Workshop at ICLR 2026*, 2026. 24, 31

[14] Johannes Loebbecke, Juergen Mangler, and Stefanie Rinderle-Ma. **Tree-Based Compliance Verification: Bridging the Gap Between Compliance Requirements and Process Execution**. In *Conceptual Modeling – ER 2025*, **16189** of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2026. 24, 48

[15] Simone Leo, Michael R. Crusoe, Laura Rodríguez-Navas, Raül Sirvent, Alexander Kanitz, Paul De Geest, Rudolf Wittner, Luca Pireddu, Daniel Garijo, José M. Fernández, et al. **Recording Provenance of Workflow Runs with RO-Crate**. *PLoS ONE*, **19**(9):e0309210, 2024. 25, 30, 39

[16] Kamaluddeen Usman Danyaro, Maged Nasser, Abubakar Zakari, Shamsu Abdullahi, Atika Khanzada, Muhammad Muntasir Yakubu, Sara Shoaib,

et al. **LLM-Based Code Generation: A Systematic Literature Review With Technical and Demographic Insights**. *IEEE Access*, **13**:194915–194939, 2025. 28

[17] Li Zhong and Zilong Wang. **Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation**. In *Proceedings of the AAAI Conference on Artificial Intelligence*, **38**, pages 21841–21849, 2024. 28

[18] Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. **Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed Through LLM-Empowered DevOps Simulation**. *Proceedings of the ACM on Software Engineering*, **3**(FSE):321–343, 2026. 28, 49

[19] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. **On the Effectiveness of LLM-as-a-Judge for Code Generation and Summarization**. *IEEE Transactions on Software Engineering*, **51**(8):2329–2345, 2025. 29

[20] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. **CWEval: Outcome-Driven Evaluation on Functionality and Security of LLM Code Generation**. In *2025 IEEE/ACM International Workshop on Large Language Models for Code*, pages 33–40. IEEE, 2025. 31, 36, 41

[21] Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. **AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation Through Static Analysis and Fuzz Testing**. In *NeurIPS 2024 Workshop on Safe and Trustworthy Agents*, 2024. arXiv:2409.10737. 36

[22] Harshvardhan J. Pandit and Beatriz Esteves. **Enhancing Data Use Ontology (DUO) for Health-Data Sharing by Extending It with ODRL and DPV**. *Semantic Web*, **15**(4):1473–1498, 2024. 47

[23] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. **RuleKeeper: GDPR-Aware Personal Data Compliance for Web Frameworks**. In *2023 IEEE Symposium on Security and Privacy*, pages 2817–2834. IEEE, 2023. 48

[24] Naama Zwerdling, David Boaz, Ella Rabinovich, Guy Uziel, David Amid, and Ateret Anaby Tavor. **Towards Enforcing Company Policy Adherence in Agentic Workflows**. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 595–606. Association for Computational Linguistics, 2025. 49