Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

BraneHub: A Compliance-Aware Platform for Collaborative Federated Data Analysis

Author: Mingjiang Ma (UvA: 15322351, VU: 2804818)

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

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

August 1, 2026

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

Abstract

Federated analysis operates on sensitive data without moving it, which shifts the hard problem from computation to governance: deciding who is admitted to a project, with what data, and under what oversight. We present BraneHub, a compliance-aware platform that manages this governance lifecycle for cross-institution federated data projects, covering applicant onboarding, a human-approved analysis cycle executed through the Brane federated executor, an append-only audit trail under an independent auditor, policy-as-code evaluation, and GDPR right-to-erasure handling by in-place pseudonymisation. Its most novel component is a language-model relevance judge that advises a project owner whether an application fits the project it targets.

Because an admission gate cannot depend on a judge of unknown reliability, we evaluate the judge on two human-labelled LegalBench tasks against majority-class and lexical-overlap baselines, across four contemporary models spanning the cost range, and under an ablation that withholds the clause body, testing every comparison with McNemar's exact test under Holm correction. All four models clear both baselines by a wide margin, reaching 78.0–81.6% accuracy on the balanced privacy-policy task against a 50% baseline and 91.5–94.4% on contract natural-language inference. A free model matches the proprietary models, and the most expensive model is the least accurate on the privacy task. Withholding the clause body collapses privacy accuracy to the chance level, showing that the verdict depends on reading the text rather than on surface cues. A 196-pair résumé-matching pilot shows the judge ranks suitability well (area under the ROC curve of 0.921) but is miscalibrated, applying a far stricter absolute threshold than the human annotators. These results lead us to use the judge as advice to a human reviewer rather than as an automatic gate, and yield two methodological recommendations for evaluating language-model judges: test body-dependence by ablating the document, and report discrimination and calibration separately.

Acknowledgements

I am grateful to my first supervisor, Adam Belloum, and my daily supervisor, Natallia Kokash, for their guidance throughout this project. I am also grateful to everyone else who helped along the way, for the advice and discussion that shaped this work.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Motivation

Much of the data that research depends on cannot be shared. Hospitals, registries, and companies hold personal records that they cannot release for legal and ethical reasons, even though analysing these records together would be valuable. Federated analysis addresses this problem: instead of collecting the data in one place, the analysis is sent to each site that holds the data and runs locally, and only the aggregated results are returned, so that the raw data never leaves its owner (1). Frameworks such as Brane (2) and Vantage6 (3) now package and execute such analyses across a federation of sites.

These frameworks handle execution but leave the surrounding governance to each project. Before an analysis runs, someone must decide who is admitted and with what data; while it runs, the actions taken must be recorded and overseen; and throughout, the platform must meet the data-protection obligations that handling personal data carries. One decision recurs at the centre of this governance: whether an applicant, and the data they bring, are relevant and eligible for the project they wish to join. Today a project owner makes this decision by hand. A language model could assist with that decision, but an admission gate cannot use a judge that is confidently wrong, and the reliability of a language-model judge is not self-evident. We therefore pursue two goals in this thesis: to build a platform that governs the federated-analysis lifecycle with compliance designed in, and to measure whether the language-model relevance judge it provides can be trusted.

1.2 Problem Statement

The problem has two parts, one constructive and one empirical. The constructive part is that existing federated platforms focus on executing analyses and on technical access control, and leave the surrounding governance lifecycle to be improvised for each project: admission through onboarding, an auditable record with independent oversight, lawful erasure, and policy-based gating. We need a platform in which these concerns are designed in rather than added afterwards, and in which the human approvals that compliance requires are part of the system's protocol rather than a manual process around it.

The empirical part concerns the platform's most novel component, the language-model relevance judge. Such judges are usually validated only by their agreement with human labels, but agreement can arise for the wrong reason: a model can exploit surface cues instead of reading the content it is meant to judge. Whether the judge is reliable enough to inform a real admission decision is therefore unestablished. We address both parts: we build the platform, and we measure the judge.

1.3 Research Questions

The thesis is organised around two kinds of question. Two design questions concern requirements that shape the platform's architecture and have no off-the-shelf answer; the constructive chapters answer them. Four research questions concern the relevance judge and move from whether it works to why it works, what it costs, and how it fails; the evaluation answers them. The last of these is diagnostic: RQ4 characterises where the judge errs rather than predicting its errors.

DQ1 (Erasure versus accountability): How can a federated-analysis governance platform honour a data subject's right to erasure while preserving an append-only audit trail that, by its nature, references the very subjects whose data may have to be erased?

DQ2 (Consistency under change): How can a running federated analysis cycle be kept consistent with a participant set that changes beneath it—as applicants are admitted, withdraw, are removed, or are erased—without corrupting the run or its record?

RQ1 (Competence): Can a general-purpose language model judge whether a piece of compliance text satisfies a stated requirement more accurately than non-learning baselines, and by how much?

RQ2 (Evidence dependence): Does that judgement depend on reading the body of the text, or can it be reached from surface cues alone?

RQ3 (Cost and prompting): How does judging accuracy trade off against the cost of the model used, and does a chain-of-thought prompting strategy improve it?

RQ4 (Failure characterisation): What distinguishes the cases a model judges incorrectly from those it judges correctly, and do those errors point to a systematic weakness?

1.4 Contributions

This thesis makes three contributions, two constructive and one empirical.

1. **A governance platform for federated data projects.** BraneHub manages the full lifecycle of a cross-institution federated study: project creation, applicant onboarding and admission, a multi-state analysis cycle with human approval, and long-term audit retention. It integrates with a federated executor through a bidirectional contract built for safety, in which the project state is frozen into a snapshot at cycle start and machine-to-machine calls are authenticated, idempotent, and retried. Chapter 4 presents the platform.

2. **A compliance framework built into the lifecycle.** Compliance is realised as four mechanisms that run alongside the lifecycle: an append-only audit trail observed by a strictly read-only auditor; policy-as-code evaluation, in which per-project rules are versioned with integrity checks and evaluated by a dedicated engine; erasure that satisfies the right to be forgotten by pseudonymising personal data in place, so that the record of the platform's own actions survives; and a language-model relevance judge that advises project owners. Chapter 5 presents this framework.

3. **A rigorous evaluation of the language-model judge.** We assess the judge not on BraneHub's own data, which carries no ground truth, but against external, human-labelled legal benchmarks. We compare four models against non-learning baselines, ablate the evidence given to the judge, and test every comparison for significance. The study yields two methodological lessons of wider relevance: a judge benchmark must be shown by ablation to be body-decisive before its scores mean anything, and a judge can discriminate well yet be miscalibrated, ranking suitable cases above unsuitable ones while setting its absolute threshold in the wrong place. Chapters 6 to 8 present and bound this evaluation.

1.5 Thesis Structure

Chapter 2 introduces the background the thesis assumes. Chapter 3 gives an overview of BraneHub as a whole. Chapter 4 presents the platform and its lifecycle, and Chapter 5 the compliance framework layered over it. Chapter 6 evaluates the relevance judge against human-labelled benchmarks; Chapter 7 interprets the results and Chapter 8 states the threats to their validity. Chapter 9 positions the work against related research, and Chapter 10 concludes by answering the research questions and outlining future work.

2

Background

We review in turn the five concepts BraneHub builds on: federated analysis and the Brane framework, the FAIR principles and the FAIR Data Point, GDPR's right to erasure, policy-as-code, and the use of language models as automated judges, whose reliability we test empirically in Chapter 6.

2.1 Federated Analysis and Brane

Sensitive data—medical records, genomic data, and other personal records—often cannot be pooled into one location, whether for legal, ethical, or competitive reasons. Consent may not cover transfer, and moving identifiable records across an institutional or national boundary raises regulatory risk. Federated analysis moves the computation to the data rather than the data to the computation (1). An analysis is dispatched to each site that holds relevant records, executed behind the custodian's own controls, and reduced to aggregate results that are returned and combined. The best-known instance is federated learning, which trains a model across decentralised data by exchanging parameter updates rather than raw examples (4), but the principle is general and predates that application: the raw data never leave the custodian.

This control comes at the cost of coordination machinery. A federated computation must be packaged to run identically at every site, shipped to custodians running heterogeneous infrastructure, scheduled and monitored across administrative domains with no common operator, and reconciled into a single result. Coordination, not the analysis computation itself, dominates the engineering effort.

Brane is a programming framework that supplies this machinery (2). A workflow is expressed once, packaged with its dependencies, and executed across a federation of sites

without the author orchestrating the distributed execution by hand; the framework ships each unit of work to a site, runs it within that site's environment, and returns its outputs for combination. Brane handles execution: moving computation to data and running it reliably across independent domains. It does not address governance—which parties are admitted to a study, which datasets they may contribute, on whose authority, and under what audit record. BraneHub provides this governance layer above Brane's execution framework. BraneHub composes with Brane rather than replacing it: Brane runs the analysis, and BraneHub decides, through admission, oversight, and audit, whether it should run at all. The chapters that follow develop this layer.

2.2 FAIR Data and the FAIR Data Point

Data usable in a federated study must first be discoverable and machine-describable; the FAIR principles define what such description requires. Introduced by Wilkinson et al. (5) to improve the stewardship of scientific data, they require data to be Findable, Accessible, Interoperable, and Reusable, with an explicit emphasis on machine-actionability: described by rich, persistently identified metadata, retrievable over standard open protocols even where the underlying data are restricted, expressed in shared, formally specified vocabularies, and carrying clear provenance and explicit terms of use.

The principles govern metadata, which is separable from and typically far less sensitive than the data themselves. A FAIR Data Point operationalises this separation: it is a service through which a custodian publishes structured, machine-readable metadata about the datasets it holds, organised in a defined hierarchy and exposed over a standard interface so that external parties can locate a dataset and inspect its contents and access conditions without seeing any record (6). In a federated setting, separating description from content is a precondition. It lets a prospective study identify suitable data and partners by querying metadata, before any sensitive material is shared; access—the costly, legally constrained step—is granted only once a metadata-level match is found. Metadata publication is thus the entry point of the federated workflow, and the channel through which the candidate datasets that the later chapters reason over are first made visible.

2.3 Data Protection and the Right to Erasure

Because federated analysis typically processes personal data, it falls under data-protection law, principally the European Union's General Data Protection Regulation (GDPR) (7).

GDPR regulates the processing of personal data, binds the controller that determines the purposes and means of that processing, and grants data subjects enforceable rights over information relating to them. Article 17 establishes the right to erasure, which poses a central design problem in this thesis: under defined conditions—among them withdrawal of the consent on which processing rested, or the data no longer being necessary for the purpose for which they were collected—a data subject may obtain deletion of their personal data, and the controller must carry it out without undue delay (7). Deleting records on request is straightforward for an ordinary store.

The difficulty is that erasure conflicts with a second obligation a governed platform carries: maintaining an append-only audit log of what was done, by whom, and on what authority. Such a log necessarily records the identities of the data subjects whose personal data may later require erasure. Erasure and audit-immutability therefore pull in opposite directions: deleting freely breaks the integrity of the audit log, while preserving everything frustrates a right the law makes mandatory. Reconciling the two is a design problem rather than a matter of configuration, and we address it in Chapter 5.

2.4 Policy as Code

In most systems, authorisation checks are scattered inline through the application's own logic. Such checks are hard to audit: determining what the system actually permits requires reconstructing the policy from imperative code spread across many call sites. *Policy-as-code* separates the two. Authorisation rules are stated explicitly and declaratively, kept apart from the application logic, and evaluated by a dedicated engine that the application consults at each decision point. The Open Policy Agent (OPA), a graduated project of the Cloud Native Computing Foundation, is a widely used implementation: policies are written in a purpose-built declarative language, Rego, and the application queries the engine with the relevant facts and receives a decision in return (8).

The benefit for a compliance-sensitive platform is that externalised policy becomes an inspectable artefact: a body of declarative rules that can be read, reviewed, version-controlled, and tested against expected cases before deployment, independently of the system that enforces it. A reviewer (an auditor, a data-protection officer, or a collaborating institution) can examine what the platform permits without reading its source, and a change in policy becomes a visible, attributable revision rather than a buried edit to control flow. Because BraneHub's purpose is inspectable governance, we express its authorisation decisions as externalised OPA policy.

2.5 Language Models as Judges

We introduce the use of language models as automated judges, whose reliability the empirical part of this thesis examines. Large language models can evaluate text as well as generate it, and a growing body of work uses a model as a judge—scoring an answer against a rubric, ranking competing candidates, or deciding whether some content satisfies a stated criterion—in place of, or as a cheaper first pass ahead of, a human assessor. Work on automated evaluation of chat assistants reported that strong models agree with human preference judgements at rates approaching the agreement between human annotators themselves, while the broader information-retrieval and evaluation communities reach more guarded conclusions about the same substitution (9, 10). The appeal is cost and throughput: a model judge evaluates at far lower marginal cost and higher speed than a panel of human experts, making feasible kinds of routine assessment that manual labelling could not afford at scale.

A model judge can be confidently wrong, and this is what motivates the empirical chapters that follow. Its fluency does not guarantee a sound verdict. It may reach a correct label for the wrong reasons, agreeing with a human judgement on grounds that do not generalise. Its behaviour is also sensitive to nuisance factors that ought not to bear on a judgement: the particular model invoked, the precise wording of the prompt, the order in which candidates are presented, and the surface form of the text under assessment (9). A judge whose verdict shifts under such perturbations measures something other than the criterion it was asked to apply. We therefore treat judge reliability as an empirical question, measured against human ground truth under controlled variation. Chapter 6 takes up this evaluation for BraneHub's relevance judge.

3

Overview

BraneHub manages the human and procedural lifecycle of federated analysis: applicant onboarding, a data-readiness step, and a human-approved analysis cycle, under a compliance overlay of four mechanisms (an append-only audit trail, an Open Policy Agent policy engine, GDPR erasure by in-place pseudonymisation, and a language-model relevance judge). This chapter describes the setting, the actors, the lifecycle, and the compliance overlay, and then states the design problem the rest of the thesis addresses.

3.1 Setting and Actors

In federated analysis, data stays at the sites that hold it and the analysis is shipped to the data rather than the reverse (1). This keeps sensitive records under their custodians' control but moves the hard problem from computation to governance: who may take part in a study, with which data, and under whose oversight. We build BraneHub to govern that lifecycle, the human and procedural steps around federated execution rather than the execution itself.

The lifecycle has five actors:

1. **Applicant** — a user who requests to join a project.
2. **Owner** — a user who runs a project and decides who is admitted to it.
3. **Administrator** — a user who oversees the platform across projects.
4. **Auditor** — a user with read-only access to the audit record.
5. **System identity** — an internal account that performs automated actions (webhook callbacks, scheduled transitions) and never signs in.

The first four map onto three user-facing roles plus the ownership relation between a user and a project; the fifth keeps machine actions attributable and distinct from human ones.

3.2 Onboarding and Analysis Lifecycle

BraneHub manages two linked phases, onboarding and the federated analysis cycle. We model both as state machines with restricted transitions, enforced at the database level; Chapter 4 specifies them in full.

Onboarding. An applicant submits a request to a project; the owner accepts or rejects it; an accepted participant then marks their data ready for use. A request moves through one explicit state machine: a newly created request enters `submitted`, from which it reaches one of three decided states (`accepted`, `rejected`, or `withdrawn` by the applicant). We enforce at most one active request per user per project, so the admission record is unambiguous.

Analysis cycle. Once enough participants have marked their data ready, the owner opens a cycle. An integrator (the external, Brane-side party that supplies the analysis package) contributes a package, which is approved, rejected, or returned for regeneration; on approval the package runs across the participating sites. The cycle moves through ten states — one initial, five in-flight, and four terminal — with owner approval gating the transition into `running`.

3.3 Compliance Overlay

We layer a compliance overlay across both phases that a bare workflow tool lacks. It comprises four mechanisms, each detailed in Chapter 5:

1. **Audit trail.** We log every consequential action to an append-only audit trail that auditors can read but not alter.

2. **Policy engine.** A deterministic policy engine, OPA, run as a separate sidecar, evaluates project-scoped admission rules written in OPA's policy language, Rego, so that the conditions under which actions are permitted are stated explicitly rather than held in application code.

3. **Erasure.** We implement the right to erasure required by data-protection law by pseudonymising personal data in place rather than deleting records, so the audit trail survives after a person is removed.

4. **Relevance judge.** A language-model relevance judge produces an advisory opinion on each application, shown to the project owner.

3.4 Design Problem and Roadmap

This thesis has two aims:

1. **Constructive.** We build a platform that runs the lifecycle's state machines and the compliance overlay correctly, with compliance enforced by construction rather than added afterwards (Chapters 4 and 5).

2. **Empirical.** We measure the reliability of the relevance judge against human-labelled benchmarks (Chapter 6): an advisory opinion is useful only if owners can trust it, and that trust must rest on measured reliability rather than assertion.

Chapter 4 specifies the platform and its lifecycle; Chapter 5 the compliance mechanisms layered over it; Chapter 6 evaluates the relevance judge against human-labelled benchmarks. Chapter 7 interprets those results and Chapter 8 states their limitations; Chapter 9 sets the work against related research; and Chapter 10 concludes.

4

The Platform

We describe BraneHub first because Chapter 5's compliance mechanisms run on it and Chapter 6's relevance judge runs inside it; the architecture, data model, and workflows defined here are used throughout the thesis. We cover the architecture and its principal technology choices (Section 4.1), the onboarding and federated-analysis workflows (Sections 4.2 and 4.3), and the implementation and 2,069-case test suite that validates the platform as an engineering artefact (Section 4.4).

4.1 System Architecture

BraneHub supports a workflow in which a data owner publishes a federated data project, external researchers apply to participate, and an owner-supplied analysis is executed against each participant's data in place, so that the underlying records never leave their host institution. Three requirements distinguish this setting from a conventional data-management application.

1. **Runtime-configurable policy.** The rules that govern admission to a project are defined by the project owner and may differ between projects, so admission policy cannot be fixed in application code.

2. **Asynchronous coordination.** The execution of an analysis is asynchronous and long-running, and is coordinated with an external workflow framework, Brane (2), rather than completed within a single request–response cycle.

3. **Durable audit trail.** Because the platform mediates access to sensitive data, every decision of compliance significance must be recorded in a form that an independent auditor can later inspect.

We meet these requirements with a layered, server-rendered architecture, summarised in Figure 4.1. The presentation layer is generated on the server from Jinja2 templates, styled with Bootstrap and using Chart.js for the compliance dashboards; we confine client-side scripting to the small number of panels that require asynchronous updates. The application layer comprises seven Flask blueprints, each responsible for a single functional domain (Table 4.1), and enforces authorisation declaratively through role decorators applied at the entry point of each route. Persistence is provided by PostgreSQL through the SQLAlchemy object-relational mapper, with schema evolution managed by Alembic. We delegate policy evaluation to an OPA instance (8) running as a separate sidecar process, to which the application layer submits a structured description of each pending decision. A thin provider abstraction mediates calls to large language models, through which the relevance judge of Section 5.4 is invoked. We examine the two compliance mechanisms this architecture exposes, deterministic policy evaluation and language-model judgement, in Chapter 5.

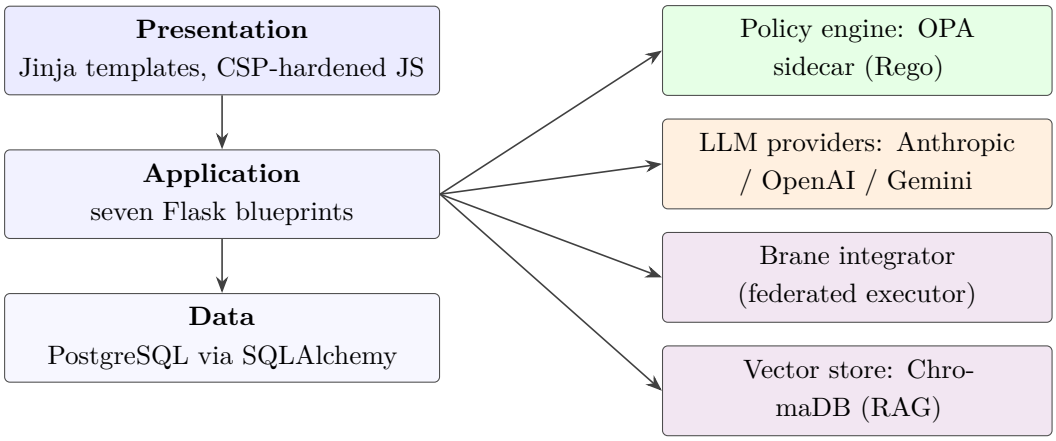

Figure 4.1: Layered architecture of BraneHub: the presentation, application, and data layers, with policy evaluation and language-model access delegated to out-of-process services reached from the application layer.

We partition the application into seven blueprints (Table 4.1) by functional domain rather than technical layer, so that each activity's code is localised to one blueprint and can be reasoned about in isolation.

We render the interface on the server rather than expose a JSON API consumed by a separate client-side application, because the workflows the platform supports are predominantly form- and page-oriented. Server-side rendering keeps the security-sensitive concerns behind BraneHub's compliance guarantees, authorisation, cross-site-request-forgery protection, and audit logging, within a single trust boundary, avoiding the duplication of that logic

Table 4.1: The seven Flask blueprints and their responsibilities.

Blueprint	Responsibility
`auth`	Registration, login, and logout, each recorded in the audit log
`projects`	Project creation and configuration, discovery, joining and leaving, policy upload, and the questionnaire editor
`onboarding`	The multi-stage onboarding forms, their policy evaluation, and the owner's accept or reject decision
`integration`	The federated analysis cycle: the outbound and inbound webhooks and state callbacks exchanged with the Brane framework
`api`	The research-assistant chat endpoint, backed by multiple language-model providers with optional retrieval-augmented context, consumed asynchronously by the browser
`admin`	User management and platform-wide operational oversight
`auditor`	The read-only compliance dashboards, audit-log views, and policy-integrity checks

across a client and a server. The one component that is inherently asynchronous, the federated analysis cycle of Section 4.3, we accommodate through dedicated status-polling endpoints rather than by adopting a client-side framework for the application as a whole.

Two further decisions shape the architecture.

Externalising policy. Because each project owner defines their own admission criteria, embedding those criteria in application code would require a code change for every project and would entangle compliance rules with request-handling logic. We instead externalise policy to OPA and evaluate it out of process: each project's criteria are expressed as a versioned Rego policy (8), uploaded at runtime and evaluated over a structured input through OPA's REST interface (Section 5.2). Treating policy as versioned, out-of-process code has two consequences for a compliance platform: (1) each policy becomes an artefact whose integrity can be checked and which is auditable in its own right (Section 5.1); and (2) the point at which a decision is made becomes explicit and therefore straightforward to record.

Representing project-specific data. The onboarding questionnaires, federated-data-provider configurations, and policy inputs and outputs differ in structure from one project to the next; modelling each as a fixed relational table would again require a schema migration whenever a project's requirements changed. We store these variable structures in PostgreSQL `JSONB` columns, which accommodate heterogeneous, project-defined shapes within a single column, while the stable relational core retains conventional typed columns

and the referential constraints that guarantee its integrity.

We model the platform's persistent state as ten entities in three groups (Figure 4.2), corresponding to the structure of this chapter and the next:

- **Project core** (`User`, `Project`, `OnboardingRequest`, `ProjectParticipant`): the actors and the membership relation that the onboarding workflow of Section 4.2 manipulates.
- **Integration** (`IntegrationCycle`, `IntegrationPackage`, `IntegrationScript`): the federated analysis cycle of Section 4.3.
- **Compliance substrate** (`PolicyVersion`, `OpaEvalLog`, `AuditLog`): the policies in force and the history of compliance-relevant actions, which underpin the mechanisms of Chapter 5.

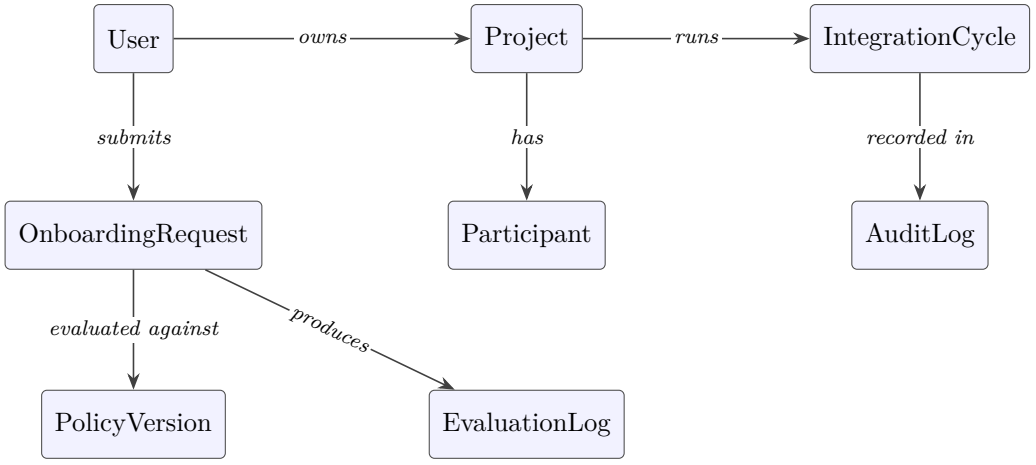

Figure 4.2: Entity-relationship overview of the ten persistent entities, grouped as project core, integration, and compliance substrate.

The `api` blueprint adds a research assistant: a single chat endpoint answered by one of several language-model providers, optionally enriched with retrieval-augmented context drawn from a local vector store of compliance material. We inject any context the user supplies from their own questionnaire or page, as distinct from the retrieved material, under a user role rather than a system role, so that user-controlled text cannot be read as a trusted instruction; and we dispatch each model call on a bounded thread pool under a timeout, so that a slow provider cannot exhaust the request workers. The assistant writes no compliance state and is not one of the compliance mechanisms of Chapter 5.

4.2 Onboarding and Participation

Onboarding admits an external researcher to a project and is where BraneHub applies its admission controls; it is the first of the two workflows that define the platform's behaviour. We describe the roles and authorisation model, the lifecycle of an onboarding request, and the multi-stage procedure through which a request is submitted, evaluated, and decided.

We define three user-facing roles and one internal service identity (Table 4.2): an ordinary *user*, an *auditor* with read-only access to the compliance dashboards and logs (Section 5.1), an *administrator*, and a *system* service account used for automated actions that never authenticates interactively.

Table 4.2: The user-facing roles and the internal service identity.

Role	Capabilities
`user`	Owns projects, applies to join others, and participates once accepted
`auditor`	Read-only access to compliance dashboards, audit logs, and policy-integrity checks; cannot alter state
`admin`	User-account management and platform-wide operational oversight
`system`	Internal service account for automated actions; cannot authenticate interactively

We enforce authorisation at two levels, reflecting a distinction between what a user *is* and what a user *owns*.

Role gate. Coarse-grained, role-based access control is applied declaratively: route handlers carry decorators (`role_required`, `admin_required`) that admit only the permitted roles and reject all others, before the handler body or any database query runs.

Ownership. A role check is insufficient because the privileges that matter most in BraneHub are relational: only the owner of a specific project may accept its applicants. A second, object-level layer therefore governs actions on individual resources: before acting on a request, we verify the actor's relationship to it, so that only a project's owner may decide its onboarding requests and only the applicant who submitted a request may withdraw it.

An onboarding request progresses through a small, explicit state machine, shown in Figure 4.3. A newly created request enters the `submitted` state, which the interface labels "Pending". From there it reaches one of three decided states: `accepted` or `rejected` on the project owner's decision, or `withdrawn` if the applicant withdraws before a decision is taken. Only `withdrawn` ends the request for good: an applicant who edits an accepted or rejected application returns it to `submitted` for a fresh decision. We enforce the legal-state

set with a database check constraint, not application code alone, so that no execution path, including data migrations and direct database access, can leave a request in an undefined state. A partial unique index enforces a further invariant: a user holds at most one *active* request (one that is submitted or accepted) for a given project, which prevents duplicate applications while still allowing re-application after an earlier request has been rejected or withdrawn.

BraneHub supports two admission paths, selected by the kind of project being joined. A *Simple* project carries a *join mode*, configurable by its owner and defaulting to open, which determines whether admission requires the owner's involvement at all. Under the open mode we admit a join request immediately: we record an onboarding request already in the `accepted` state, attributed to the owner, and create the participant record in the same transaction, so that no questionnaire is completed and no decision is awaited. Under the approval mode the same action instead creates a request in the `submitted` state with empty responses, which the owner must subsequently accept or reject. A *federated-data-provider* project has no such shortcut: we route every applicant through the full procedure, which ends in a manual decision. We describe the full federated-data-provider path below; the open-mode shortcut elides its intermediate stages.

We assemble a federated-data-provider request in stages before it is decided. The applicant first completes the project's onboarding questionnaire, whose responses we store as structured JSON, and then a second questionnaire that captures the format of the data the applicant intends to contribute. We evaluate the completed responses against the project's policy by OPA and record the structured result on the request; the evaluation mechanism itself is the subject of Section 5.2. When relevance judgement is enabled, we additionally annotate the request with a suggested relevance score and a natural-language rationale from the language-model judge of Section 5.4; the annotation is advisory and does not determine the outcome. The owner reviews the assembled request (the questionnaire responses, the policy-evaluation result, and any relevance suggestion) and records a decision together with the deciding user, a timestamp, and a reason on rejection. Acceptance creates a `ProjectParticipant` record, admitting the applicant to the project and making them eligible to take part in the federated analysis cycle of Section 4.3.

We protect the integrity of this decision against the concurrency that a multi-stage, separately edited request invites, with two properties.

Stale-review guard. When the owner opens a request for review the interface records the moment at which the request was last updated, and on submitting a decision we compare that recorded value against the request's current update time. If the applicant

has edited any stage in the interim the two disagree, we refuse the decision, and we return the owner to re-review the amended request rather than act on a stale view of it. This is optimistic concurrency control: we detect the conflict at write time rather than hold a lock across the owner's deliberation.

Edit demotes acceptance. Any edit to an already-accepted application automatically demotes it to the `submitted` state, clears the prior decision, and deactivates the corresponding participation, so that the revised application must be reviewed afresh before its author is readmitted. Acceptance therefore binds to a specific application version rather than the applicant in the abstract.

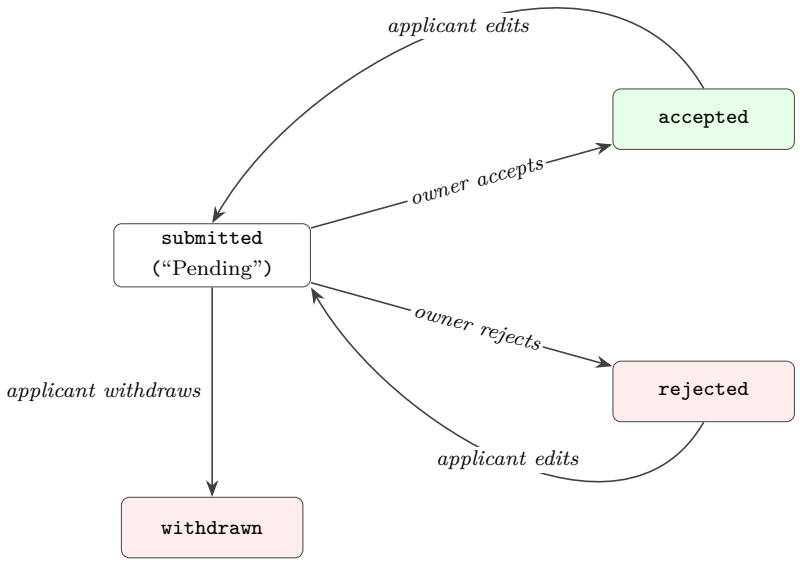

Figure 4.3: The onboarding-request state machine. The interface labels `submitted` "Pending"; an applicant's edit returns an accepted or rejected request to `submitted`, so acceptance binds to one version of the application and only `withdrawn` ends the request for good. The legal state set is fixed by a database check constraint.

4.3 The Federated Analysis Cycle

Once a project has admitted participants, its owner may run a federated analysis over their data. This is the platform's second core workflow: rather than gather the participants' data centrally, we coordinate the execution of an owner-supplied computation against each participant's data in place, through the external Brane framework (2), so that the records

never leave their host. Because the integrator runs outside our control and a run may take minutes (Section 4.3.2), the workflow is asynchronous and event-driven; the difficulty is reliable coordination with that external system. We describe the entities that represent a cycle, the lifecycle through which it progresses, and the webhook contract by which BraneHub and the Brane integrator exchange state.

Three entities represent the analysis cycle. The `IntegrationPackage`, of which a project has at most one, holds the computation to be executed, in one of two forms set by its `source_type`: `generate` (Python code and a container specification synthesised from the owner's description) or `upload` (an artefact the owner provides directly); its build lifecycle is the subject of Section 4.3.2. The `IntegrationScript` represents a concrete, runnable script produced by the integrator for a given cycle; scripts are versioned per cycle, so that a regenerated script supersedes its predecessor rather than overwriting it, and its `decision` moves from `pending` to `approved`, `rejected`, or `superseded`. The `IntegrationCycle` records each run of the workflow for a project: it is identified by a per-project cycle number; it stores a frozen snapshot of the project state at cycle start, so that the cycle stays reproducible and auditable as the live project changes; and, once the cycle ends, it records its terminal outcome.

A cycle has ten states (Figure 4.4): one initial state (`never_triggered`), five in flight (`waiting_for_integrator`, `awaiting_review`, `awaiting_regeneration`, `running`, and `awaiting_abort_confirmation`), and four terminal (`completed_success`, `completed_failed`, `aborted`, and `abandoned`). We do not store the state in a column of its own; we derive it in a single place in the code from the cycle's recorded fields, and the interface reads the same derivation, so that what a user sees and what the back end enforces cannot drift apart. The legal set of terminal values is in turn fixed by a database check constraint. No script executes until the owner approves it: owner approval gates the transition into `running`.

BraneHub and the integrator communicate over a webhook interface that is distinct from the user-facing application and authenticated by a shared API key rather than by a user session. Through this interface the integrator retrieves the current state of a project, submits a generated script for review, and reports the completion of a run, while a companion set of callbacks reports the per-participant status of the participating nodes. These messages arrive over the network from an independent system under at-least-once delivery, so we harden each handler against three failure modes.

- **Idempotency.** We detect a redelivered completion report and do not apply the same outcome twice.
- **Commit safety.** Each database write is wrapped so that a failed commit is rolled back and reported as an explicit error rather than left partially applied.
- **Serialisation.** We serialise concurrent deliveries with row-level locks, which prevents the lost updates that would otherwise arise when two callbacks modify the same record.

For the outbound calls BraneHub itself initiates, we choose a retry strategy by how each call's result is consumed. Acknowledgement-only notifications retry on a transient failure with jittered exponential backoff, spreading the retries of concurrent callers so that a recovering integrator is not met by a simultaneous burst. State-returning calls, whose responses carry state the platform must read back, are instead single-attempt: rather than retry inline, we return the integrator's reply to the caller and let the browser's existing status poll retry a transient failure. All of these calls share a single pooled HTTP client, so that the cost of establishing a connection is paid once and amortised across every webhook. The one synchronous call that drives a container build, which may run for several minutes (Section 4.3.2), is granted a correspondingly long timeout of 300 seconds, in contrast to the short timeout that bounds the remaining calls.

Schema boundary. The fields whose values BraneHub itself defines, the package source type, the script decision, and the cycle's terminal state, are constrained at the database level to their legal sets, whereas the fields that mirror the integrator's own vocabulary, such as the package's assessment and build status, deliberately carry no such constraint. Constraining these fields would force a coordinated migration whenever the integrator introduced a new status; we leave them unconstrained and validate them in application code instead, so that the integrator's vocabulary can evolve independently.

4.3.1 The Effective Participant Set and Reacting to Membership Change

A project's membership changes while a cycle is in flight, and each such change must be reconciled with the script that the integrator is preparing or running. We resolve this by defining a single derived notion, the *effective participant set*, and routing every membership change through one engine that decides what the running cycle should do about it. The effective set, computed by `effective_participant_user_ids` in `cycle_regen_service.py`, is the set of users who are simultaneously the subject of an accepted onboarding

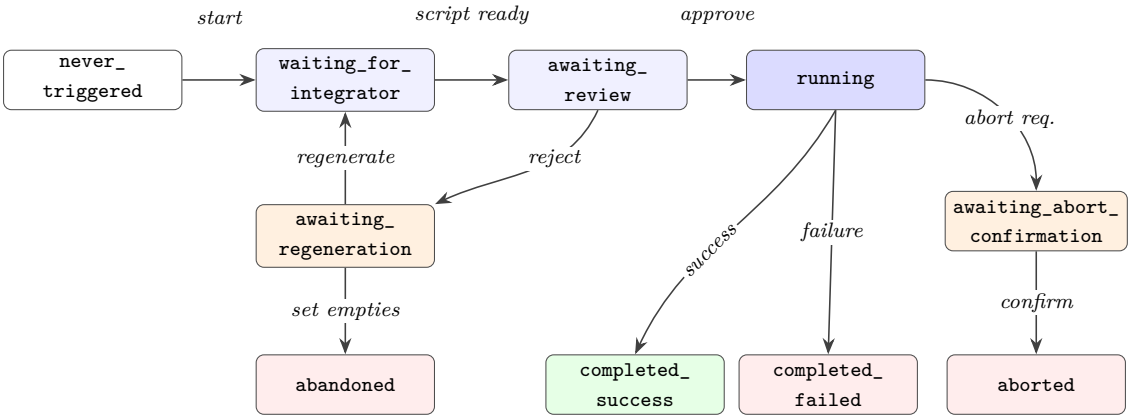

Figure 4.4: The federated analysis cycle state machine: one initial state, five in flight, and four terminal. Owner approval gates the transition into `running`.

request, an active participant (`is_active`), and marked data-ready (`data_ready`); it is the set of nodes against which a federated analysis would actually run, as distinct from the larger set of users who have merely been admitted. Because membership in this set is a conjunction of three conditions, a range of otherwise unrelated events can add or remove a member: an owner's accept or reject decision, a participant's withdrawal, a participant marking or unmarking their data as ready, an administrator removing, deactivating, or anonymising an account, a user deleting their own account, and the periodic inactivity scan that deactivates dormant participants. Each of these sites computes the effective set before its mutation, performs the mutation, and then calls the single entry point `auto_regen_if_effective_set_changed`, which recomputes the set and dispatches on the difference; we call this entry point from roughly ten such sites across the administration, onboarding, membership, authentication, and inactivity-scan code.

The engine must handle a cycle in any of its in-flight states, one that may be terminated concurrently by an inbound completion callback (Section 4.3.3). It therefore computes the new set first and only then branches, so that it never supersedes work before establishing that it should, and it acquires a row lock on the `IntegrationCycle` before any dependent mutation, re-reading the freshly locked `terminal_state` to detect a cycle that ended between the liveness check and the lock. Under that lock the engine performs exactly one of five actions:

1. *No-op*: it does nothing when the set is unchanged or no cycle is active.

2. *Warn-in-running*: it records an audit entry and warns the owner by email but leaves

the run untouched when the integrator may already be executing (the `running` and `awaiting_abort_confirmation` states).

3. *Supersede-and-regenerate*: it supersedes the pending script and re-fires script generation, after rebuilding the cycle's frozen snapshot so that the integrator's next read reflects the new membership, when the cycle is still in a pre-execution state.

4. *Abandon-emptied-cycle*: it force-abandons a cycle whose effective set has fallen to empty, because no node remains to run against, and tells the integrator to dismiss its work.

5. *Race-with-terminal*: it detects a race in which the cycle was made terminal by a concurrent write, abandons its own action cleanly, and records the race rather than corrupting the terminal record.

We lock the cycle before staging a supersession and commit the local mutations before the best-effort outbound webhook, so that the audit record stays correct under concurrency. The decision flow is set out in Figure 4.5.

4.3.2 The Computation Package and Its Build Lifecycle

A cycle requires a built computation, and preparing it is a lifecycle that gates the rest of the workflow. The `IntegrationPackage`, of which a project has at most one (a uniqueness constraint enforces the one-to-one relation), carries the computation in one of two forms set by its `source_type`: a `generate` package holds an owner-supplied natural-language description from which the integrator synthesises Python code and a container specification, whereas an `upload` package holds code the owner provides directly. In either case the owner-facing endpoints in `app/blueprints/projects/package.py` act as a server-side proxy to the integrator, so that the shared API key never reaches the browser, and each call upserts the locally held package state from the integrator's response. The owner drives the package through three steps:

1. **Register**: the owner registers the package, which asks the integrator to assess it and sets `assessment_status` and `build_status` to `pending`.

2. **Poll**: the browser polls the integrator until the assessment settles.

3. **Approve**: the owner approves the package, which dispatches a synchronous build.

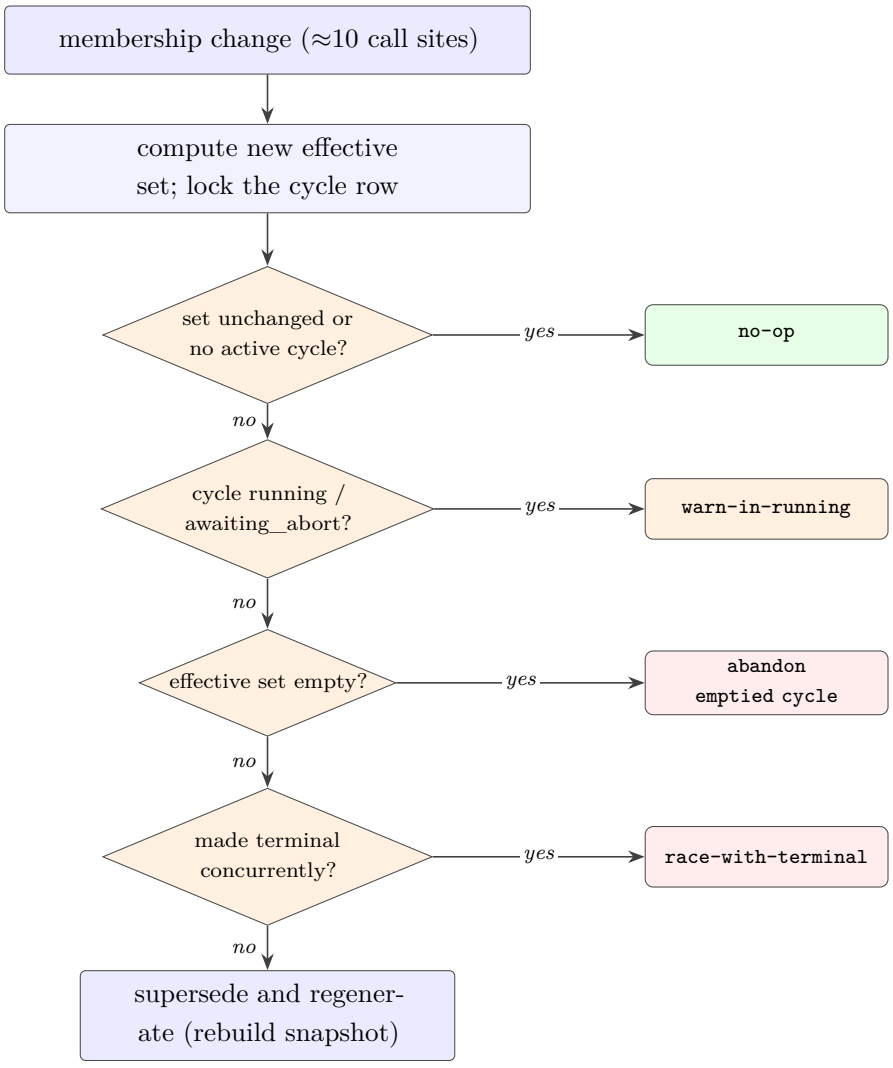

Figure 4.5: Decision flow of `auto_regen_if_effective_set_changed`. All five outcomes are decided under a row lock on the cycle.

The build dominates the platform's handling of failure, because compiling the computation into a container image takes between two and five minutes, longer than a typical request timeout. The approve handler therefore dispatches the build over an HTTP call deliberately made without holding a database row lock, so that a build does not serialise concurrent polls behind an open transaction, and reacquires the lock only briefly to record the result. Where the dispatch itself returns a transport-level timeout or connection failure, the handler does not treat the build as failed: it returns `202 Accepted` with a `Retry-After` hint, assuming the build is still running, and the browser resumes polling until `build_status` leaves `pending`. A clean response reporting failure, by contrast, demotes the package to `failed` so that the poll terminates rather than spinning. The value `build_status == 'built'` gates two later decisions: it is a precondition for accepting a participant into the project, and it is one of the conditions of the cycle start-gate of Section 4.3.3. Because the built image must match the code it was compiled from, any subsequent change to the computation, whether regenerating it, re-uploading code, or switching `source_type`, demotes `build_status` back to `pending` and so closes both gates until a fresh build succeeds; and we refuse each such mutation outright, with a `409` and an audit entry, while a cycle is in flight, so that the running analysis cannot be desynchronised from the image beneath it.

4.3.3 Node and Coordinator Provisioning, and the Start-Gate

Running a federated analysis requires not only a built computation but also provisioned infrastructure. BraneHub tracks two kinds of provisioning that, although both reported by the integrator, are deliberately kept orthogonal.

`node_status`. On each participant, `node_status` records whether the Brane runtime has been installed and made ready on that participant's machine, progressing through `provisioning`, `ready`, and `deprovisioned`.

`data_ready`. Separately, `data_ready` records whether the participant has confirmed that their research dataset is loaded in the declared format. Node-readiness and data-readiness are independent: a node may be technically ready without its data being ready, or the reverse. `data_ready` feeds the effective set of Section 4.3.1; `node_status` feeds the start-gate below.

`coordinator_status`. At the project level, `coordinator_status` records the provisioning of the central coordinator node, which the integrator establishes once the package is built and reports through a dedicated callback.

The integrator communicates all three through the inbound version-two callbacks of the integration blueprint, summarised in Table 4.3; we guard each by a monotonic `callback_at` check that drops a stale or replayed delivery, and each takes a row lock before its write so that two concurrent deliveries cannot lose an update.

Table 4.3: The inbound version-two callbacks for node and coordinator provisioning.

Callback	When the integrator sends it	Effect on BraneHub state
`participant-ready`	A participant's node has finished provisioning	Sets the participant's `node_status` to `ready` and records its `brane_node`
`participant-deprovisioned`	A participant's node has been torn down	Sets the participant's `node_status` to `deprovisioned`
`coordinator-ready`	The coordinator node has finished provisioning, or failed	Sets the project's `coordinator_status` to `ready` or `failed`

A single guard, `Project.brane_start_blocker`, decides whether a cycle may begin; both the server-side start handler and the interface that renders the start controls consult it. The guard returns the human-readable reason a start is blocked, or `None` when it may proceed, and it admits a start only when the coordinator is ready, the package is built, and at least one participant is both data-ready and node-ready, that is, when the effective set of Section 4.3.1 is non-empty and its members' nodes have been provisioned. We express the gate as one property, so that what the server enforces and what the interface displays cannot drift apart; this single-source-of-truth principle recurs across the cycle, governing the cycle-state derivation above and the transition table of Section 4.3.4.

4.3.4 The Transition Table

State derivation in one place answers *what state a cycle is in.* A second, orthogonal question, *which action is permitted from which state*, was originally answered by literals scattered across the route handlers and the cycle service, and is now consolidated into a single declarative table in `cycle_fsm.py` that every guard consults through `transition_allowed`. The table records, for each action, the set of states from which it is allowed: a cycle may be started only from the never-triggered or a terminal state, a script may be approved or rejected only from `awaiting_review`, and so on. It also encodes the one role-dependent asymmetry in the model. The states from which an owner may abandon a cycle are a strict subset of those available to an administrator. An owner may cancel a cycle that is still pre-execution, or one stuck in `awaiting_abort_confirmation` (an owner-side recovery path for a non-responsive integrator). An administrator, by contrast,

may additionally dismiss a `running` cycle as an emergency override. An equivalence-oracle test asserts that this single table yields the same decision as the original literals for every combination of action, state, and role; Section 4.4 returns to this oracle.

4.4 Implementation and Testing

We implement BraneHub in Python using the Flask framework; the principal components of its implementation stack and their versions are listed in Table 4.4. We define the relational schema through the SQLAlchemy object-relational mapper and evolve it through a sequence of versioned Alembic migrations. We render the browser-facing interface from Jinja2 templates styled with Bootstrap, and the language-model clients on which the language-model layer (Section 5.4) depends are provided by the respective vendor libraries; the details of that layer are deferred to Chapter 5.

Table 4.4: The principal components of the implementation stack.

Concern	Component (version)
Language and framework	Python, Flask 3.0.3
Templating and WSGI server	Jinja2 3.1.5, gunicorn 23.0.0
ORM and migrations	Flask-SQLAlchemy 3.1.1 (SQLAlchemy 2.x), Flask-Migrate 4.0.7 (Alembic)
Database	PostgreSQL via psycopg2-binary 2.9.9
Authentication and sessions	Flask-Login 0.6.3, Flask-Bcrypt 1.0.1
Request safety	Flask-WTF 1.2.1 (CSRF), Flask-Limiter 4.1.1
Policy engine	Open Policy Agent (Rego)

We deploy the platform as a live service, described in full in Section 4.5; its measured runtime performance under that deployment is reported in Section 6.8.

We establish BraneHub's standing as a contribution by its behaving correctly rather than by a study of its users; whether the language-model judge is reliable is a separate empirical question, addressed in Chapter 6. We establish the platform's correctness by construction and by an automated test suite, run against a dedicated PostgreSQL test database so that the tests neither depend on nor disturb development data. At the time of writing the suite comprises 2,069 automated test cases collected by the test runner, with a further six that require optional dependencies and are excluded from the default run.

We concentrate these tests where correctness matters most for a compliance platform, in five groups.

- **Schema invariants.** The legal state sets of the onboarding and cycle state machines, the foreign-key deletion policies that carry the platform's erasure behaviour, and the uniqueness constraints that forbid duplicate active requests, verified to hold rather than assumed. The cycle transition table of Section 4.3.4 is checked in the same group by an equivalence-oracle test, which asserts that the single declarative table admits exactly the transitions the previously scattered guards allowed.
- **Authorisation.** The authorisation model of Section 4.2, covering both the coarse-grained role gates and the object-level ownership rules.
- **Data rights.** The erasure and anonymisation paths and their interaction with the audit trail.
- **External coordination.** The points at which BraneHub coordinates with external components, the OPA evaluation and, in particular, the Brane webhook interface, whose idempotency, row-level locking, and commit safety (Section 4.3) are verified under the concurrent and repeated deliveries that a networked webhook must tolerate.
- **Audit and web-security.** The completeness of the audit trail together with a range of web-security hardening measures, from session-fixation resistance to content-security policy enforcement and rate limiting.

Table 4.5 gives the approximate distribution of the suite across these areas; the single largest share exercises the Brane integration and the concurrency that its webhook interface invites.

This suite establishes that the platform behaves as specified: that its invariants hold, that its concurrent paths are free of the lost-update and partial-commit hazards inherent in webhook coordination, and that its compliance machinery does what it claims. It does not establish that the relevance judgements produced by the language model are themselves accurate; that is a question not of software correctness but of model capability, and answering it is the purpose of the evaluation reported in Chapter 6.

The Brane integrator, which generates packages, runs the federated computation, and fires the lifecycle callbacks, was not available to drive locally during development. To close that gap we built a standalone, containerised simulator of the integrator, a single Flask service of roughly seven hundred lines, that impersonates the integrator side of both the version-one workflow path (snapshot retrieval, script and traceability upload, the simulated federated-learning run and its completion callback, and the abort, reject, and

Table 4.5: Approximate distribution of the 2,069-case test suite by area. The categorisation is a single coarse partition for orientation.

Area	Test cases	Share
Brane integration: webhooks, cycle FSM, locking, idempotency, package lifecycle, callbacks	868	42%
Evaluation harness: judge, baselines, statistics, loaders, prompts	214	10%
Authentication, session, and web-security	187	9%
GDPR erasure, anonymisation, inactivity, and audit logging	176	9%
Frontend, templates, and interface behaviour	147	7%
Onboarding, project, and administration routes	125	6%
Miscellaneous hardening	122	6%
Database schema, migrations, and seed data	68	3%
In-platform relevance judge: service, endpoint, configuration	60	3%
Notifications, unsubscribe, and profile	40	2%
OPA / Rego policy evaluation	36	2%
Assistant chat, health, and end-to-end checks	18	1%
User model and package concurrency	8	<1%
Total	**2,069**	**100%**

dismiss branches) and the version-two package path (registration, time-gated assessment, regeneration, build approval, and the callbacks that report coordinator provisioning, participant provisioning, and participant deprovisioning). Because the simulated work is time-driven rather than instantaneous, configurable delays on assessment, generation, the run, and node provisioning let a tester observe each intermediate state and mutate the participant set while a cycle is in flight. A set of failure-injection switches makes the otherwise hard-to-reach error branches reachable on demand: a failed run, a failed coordinator provisioning, a failed build, and a silent integrator that forces an abandon. Together these facilities enable the concurrency and idempotency tests catalogued above to run without a live integrator.

4.5 Deployment

We operate BraneHub as a live service rather than leave it as a local artefact, and its deployment underpins the security and performance claims of Sections 5.1 and 6.8. BraneHub runs on a single rented virtual private server, a Hetzner CX33 instance located in Helsinki, with four virtual CPUs, eight gigabytes of memory, and eighty gigabytes of disk, running Ubuntu 24.04, and is reached at the canonical address `branehub.bit.hospital`. We assemble the service from a small set of containers orchestrated by Docker Compose from a checked-in specification, which isolates each service in its own container and keeps the runtime reproducible.

Four containers make up the deployment (Figure 4.6). A Caddy reverse proxy terminates TLS at the network edge and forwards cleartext requests inward to the application; gunicorn serves the Flask application itself; PostgreSQL 16 and the Open Policy Agent provide persistence and policy evaluation as backing services. Only the proxy is exposed to the public internet, listening on ports 80 and 443; the application server, the database, and the policy agent communicate exclusively over a private Docker bridge network and publish no ports to the host, so that neither PostgreSQL nor OPA is reachable from outside the machine. Two firewalls, a network-level cloud firewall and a host-level firewall on the instance itself, admit inbound traffic only on the secure-shell, HTTP, and HTTPS ports and reject everything else. Caddy obtains and renews its certificate automatically from Let's Encrypt through the ACME HTTP-01 challenge (11), which is the reason port 80 is held open at all, and redirects bare-address and cleartext requests to the canonical HTTPS host, so that the edge enforces the transport security on which the content-security and strict-transport policies of Section 5.1 depend. The embedded vector store that backs the research assistant of Section 4.1 runs inside the application container rather than as a service of its own, and rate limiting is served from in-process storage, which is sufficient for the single-instance deployment operated here.

We size the application server to the shape of the workload it carries rather than to the bare core count of the host: gunicorn is configured with two worker processes, each running four threads, in preference to a larger number of single-threaded workers. The reasoning has two parts.

I/O-bound requests. The requests that dominate BraneHub's latency are the calls it makes to external language-model providers, which are bound by network and provider latency rather than by local computation. Each of a worker's four threads can hold one such request in flight while it awaits its response, so that threads absorb the I/O wait that would otherwise idle a process.

Memory. Each worker process that loads the assistant's embedding model pays the memory cost of a separate copy, so multiplying workers on an eight-gigabyte host would duplicate that model needlessly. Two workers of four threads each therefore bound the embedding-model memory cost while still serving concurrent requests. The latency this configuration delivers in practice is measured under the live deployment and reported in Section 6.8.

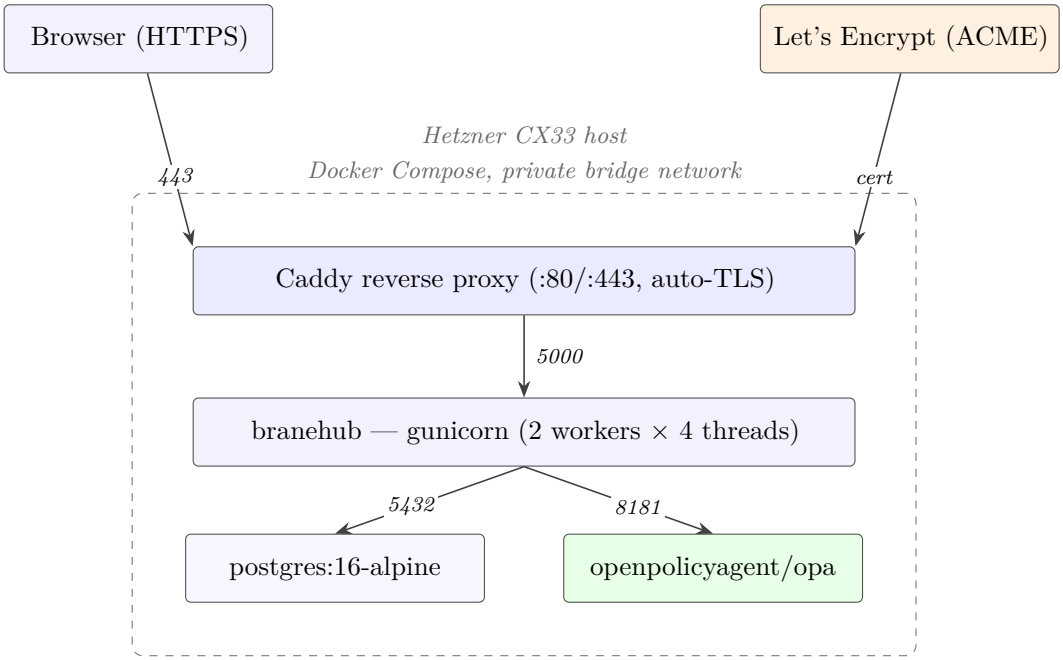

Figure 4.6: Deployment architecture on the Hetzner CX33 host. Only the Caddy proxy is reachable from the internet; the three backing containers publish no port to the host.

5

Compliance Mechanisms

We implement four compliance mechanisms in BraneHub: an append-only audit trail under an auditor role, deterministic policy evaluation through Open Policy Agent, data-subject erasure in the sense of the General Data Protection Regulation, and a large-language-model-based (LLM) relevance judge that scores whether an application matches a project's stated objective. The first is the shared accountability substrate the other three write into, which we define first in Section 5.1 because every mechanism that follows writes into it.

5.1 Auditor Oversight and Compliance Traceability

We make every compliance-relevant operation inspectable after the fact: we record it in three forms and expose those records to a read-only auditor role.

Audit log. The most general of the three records every significant operation with the acting user, the action performed, the type and identity of the affected object, the timestamp, and the originating network address, together with a structured detail field for action-specific context.

Policy-evaluation log. Each policy evaluation is recorded in greater detail: the exact input it was given, the full result it produced, the binary decision reached, and the policy version against which it was made, so that any past decision can be reconstructed rather than merely summarised.

Policy-version archive. We archive every version of every policy ever used rather than overwriting it, so the rule applied to a historical decision remains retrievable after the project's policy has since changed.

All three records are append-only: we expose no path by which an entry may be edited or deleted, so recorded history cannot be modified. Append-only retention conflicts with the

right to erasure that Section 5.3 implements. We resolve the conflict through the form our erasure takes: rather than deleting the rows a person's records refer to, we pseudonymise them in place (Section 5.3). The audit and evaluation records are retained and continue to attribute each action to the same row, but that row, and the personal data the records themselves hold, are stripped of identifying content.

Archived policies carry a further guarantee. When a policy is uploaded we compute a SHA-256 checksum of its Rego source and store it alongside the source. We verify integrity by recomputing the checksum from the stored source and comparing it against the stored value, detecting any post-upload modification of a policy's content in the database. The auditor's dashboard reports this check for every archived policy. The same control extends to what the browser may execute: we serve a content-security policy whose `script-src` directive is restricted to `'self'` with no `'unsafe-inline'`, so that no inline event handler or inline `<script>` runs and an injected script string cannot execute.

We separate the *auditor* role from the administrator who operates the platform. The auditor may read the entire audit trail, inspect the evaluation history, and verify policy integrity, but cannot alter platform state of any kind; the administrator manages the platform but is themselves subject to the audit trail. The auditing role therefore observes everything and changes nothing, which keeps it independent of the actions it audits.

The notification subsystem informs participants of the events that concern them, namely that an application was accepted or rejected, that a workflow has finished, or that the membership of a running cycle has changed, and is itself bound by the same accountability discipline. Each message is transactional and best-effort, dispatched only where the recipient has not opted out of that class of notification through a per-event preference, and each carries a signed, single-use unsubscribe link, revocable in bulk by advancing a per-user version counter. We write the act of sending into the audit trail as an action of the system rather than of the human who triggered the underlying event, keeping the automated correspondence distinguishable from a user's own actions. The dispatcher never raises: a failure to render or deliver a message is logged and any partial database work is rolled back, so an undelivered notification does not break the request that occasioned it. Notification is a convenient channel but never the authoritative record, which remains the audit trail itself.

Attribution defaults and their hazards. The application writes all audit entries through a single helper, so the trustworthiness of an entry depends on that helper's defaults. To spare its call sites from restating context that is almost always implicit, the helper supplies two defaults: where the caller names no acting user it attributes the action to the

authenticated user of the current request, and where the caller names no network address it records the address from which the request originated. These defaults are correct for entries that document a logged-in user acting on their own behalf, but each fails for a case that does not fit that pattern:

1. A side effect the system performs while a human is logged in, such as a re-evaluation triggered automatically as an owner clicks a button, would be silently mis-attributed to that human under the user default, blurring the line between automated and human action that the notification record above keeps distinct.

2. The address default would stamp the originating network address, itself personal data, onto rows deliberately retained beyond a subject's erasure.

Since the absence of an argument cannot be distinguished from a deliberate request for an empty value, the helper resolves the ambiguity with two explicit opt-out sentinels: a caller may pass a distinguished marker to force a null acting user, or another to force a null address, overriding the default rather than colliding with it. System-attributed actions thus declare their lack of a human actor explicitly, and the retained erasure records of Section 5.3 suppress the address that would otherwise outlive the subject they concern.

Deployment hardening. The accountability substrate above is an application-layer construction, and its guarantees hold only if the host on which it runs is itself uncompromised. We therefore back the application-layer controls introduced across Chapters 4 and 5, namely the content-security and transport-security headers, the cross-site-request-forgery protection, and the role-based separation between administrator and auditor, with host and container hardening:

- **Network.** A two-layer firewall, comprising the cloud provider's network filter and the host's own `ufw`, admits only SSH on port 22 and HTTP and HTTPS on ports 80 and 443, and rejects everything else.

- **Backing services.** PostgreSQL and the Open Policy Agent sidecar of Section 5.2 sit on a private Docker bridge network with no published host port, and the vector store used by the assistant is embedded within the application container rather than run as a separate service, so none of the data stores is reachable from outside the deployment.

- **SSH and patching.** `fail2ban` bans hosts that repeatedly fail to authenticate over SSH, and unattended security upgrades apply the operating system's patches without manual intervention.

- **Privilege.** The application container runs as an unprivileged, non-root user, so a compromise of the application process does not begin with root authority.

- **Secrets.** The application's signing key, the database password, and the third-party API credentials live in an environment file readable only by its owner and are injected into the containers as environment variables rather than baked into the images, keeping them out of the distributed build artefacts.

These measures support, but are not themselves, compliance mechanisms: an audit trail is only as authoritative as the integrity of the system that maintains it.

5.2 Policy-Based Evaluation with Open Policy Agent

For conditions a project can state as explicit rules — that an applicant be located in a permitted region, hold a particular accreditation, or contribute data in a required structure — we evaluate admission with a deterministic policy engine, Open Policy Agent (8), reserving the relevance judge of Section 5.4 for the requirements that cannot be reduced to such rules.

A project's admission rules are expressed in OPA's policy language, Rego. An owner may upload a policy of their own or rely on the default policy BraneHub ships for each kind of evaluation; in either case we store the policy as a versioned, checksummed artefact in the policy archive of Section 5.1, so the exact rule applied to any decision can be retrieved and its integrity verified after the fact. We deduplicate identical policy content and increment version numbers monotonically, so the archive grows only on a content change.

An evaluation proceeds in three steps.

1. We assemble a structured input describing the case to be judged: for an onboarding evaluation, the applicant's questionnaire answers together with the relevant project context; for a data-format evaluation, the structured description of the data the applicant proposes to contribute.

2. We submit the policy and the input to the OPA sidecar over its REST interface and read back the result.

3. The result is structured rather than a bare verdict: alongside the allow-or-deny decision, it carries the reasons for a denial, any outstanding requirements, and explanatory notes.

We record the result in full, both the input from which it was computed and the output it produced, in the immutable evaluation log of Section 5.1, and show it to the project owner as advice; it does not gate admission. The owner's request for an evaluation is itself written to the audit trail before the policy engine is consulted, so the fact of an evaluation having been requested survives even if the external call subsequently fails.

Namespace isolation. OPA compiles all loaded policy modules into one global namespace, so were every project's policy to declare the same package name their rules would be merged and concurrent evaluations of different projects could interfere. We rewrite each policy's package to a per-project name before loading it, isolating each project's rules. We do not unload modules on the evaluation path, where a concurrent evaluation of the same project could otherwise delete a module another evaluation was about to query; we defer cleanup to project deletion and to a separate sweep of orphaned modules.

Static compatibility check. Because a policy and the questionnaire it reads are authored separately, the two can drift out of alignment: a rule may consult a field the questionnaire never collects, or the questionnaire may gather an answer no rule inspects. In neither case does OPA error; it reads a missing input as absent. We therefore statically reconcile a policy against its questionnaire without executing either. The check parses the Rego source and extracts every leaf reference it makes into the evaluation input, recognising both the direct dotted form and the defensive accessor that guards against an absent key, and compares the resulting set of input paths against the set of question identifiers the questionnaire declares. We make the comparison tolerant of the structural gap between a flat questionnaire and a nested input: a question capturing a parent object covers any policy reference beneath it, and a reference to a parent is satisfied by the child questions that compose it. We report the result in three sets:

1. the identifiers that match,
2. the paths that a policy reads but the questionnaire never supplies,
3. the questions that the questionnaire collects but no policy consults,

so an owner editing either artefact can see, before committing a decision to it, exactly where the two have diverged. The check is advisory: it neither blocks an upload nor alters

an evaluation, and we compute it defensively so a questionnaire malformed by hand cannot turn the check itself into a failure.

Upload validation. We treat a custom policy an owner supplies as a validated artefact rather than as text accepted on trust. We reject an upload unless it is a Rego file within a fixed size bound and carries a package declaration, and we check the declared package against the slot it is being uploaded into, so a data-format policy offered as an onboarding rule, or the converse, is refused with a specific explanation. A further constraint binds the policy to the lifecycle of the project: the questionnaire a policy reads is frozen into the snapshot taken at the start of every analysis cycle, so we refuse to let an owner edit or reset that questionnaire while a cycle is in flight, recording the refused attempt in the audit trail rather than allowing a mid-cycle change that would desynchronise the frozen snapshot from the live configuration. Finally, the identifier under which a policy is stored in OPA becomes a segment of the REST path used to upload and query it, so we validate it against a strict character allow-list before constructing any request, preventing a crafted identifier from escaping its path segment; the owner-facing routes that upload and remove a policy are, in addition, rate-limited.

5.3 Data-Subject Erasure under the GDPR

The General Data Protection Regulation grants a data subject the right to obtain the erasure of their personal data (7). Erasure conflicts with the permanent audit trail of Section 5.1, which references the very users whose data may have to be erased. We reconcile the two by scrubbing the identifying content in place rather than deleting the row. Because the row and the references to it survive, the result is *pseudonymisation* rather than anonymisation in the strict sense of the Regulation: the retained record can no longer identify the subject on its own, but it is kept. That retention rests on Article 17(3), under which the erasure duty does not apply to the extent that processing remains necessary to comply with a legal obligation or to establish, exercise, or defend legal claims. Here that obligation is Article 5(2), which requires a controller to be able to demonstrate compliance — the accountability the audit trail exists to provide. We state the limits of this reconciliation at the end of the section.

We erase a user by overwriting every identifying field of the user's row rather than deleting the row: we replace the username and email address with neutral placeholders, set the password hash to a value that no input can satisfy, and mark the account inactive and erased. Retaining the row preserves the audit references while removing the identifying

content, so the audit and evaluation records that reference the user continue to point at a row stripped of identity. Because both the username and the email address are unique, we check each deterministic placeholder against the existing rows and append a random suffix on collision. A uniqueness conflict therefore cannot silently abort the erasure.

We added the email side of that check after a defect caught in review. An earlier version probed only for an existing account holding the candidate username, assuming the username and the email address would clash together or not at all. They need not: a legacy account predating the present naming rules can hold the canonical placeholder email while leaving the matching username free, and such an account passed a username-only probe. The erasure would then assign the already-taken placeholder email and, at the closing commit, violate the database's separate uniqueness guarantee on that column, rolling back the entire transaction and leaving the user un-erased with their personal data intact while the request itself failed. An erasure that silently does not erase is precisely what the right to erasure forbids, so we widened the probe to trigger the suffixing path on a conflict on either the username or the email held by any account other than the one being erased.

Erasing the user's own row is not sufficient, because the personal data a participant supplies propagates beyond it into several sinks:

- the free-text answers to the onboarding and data-format questionnaires;
- the metadata an applicant attaches to a request;
- the natural-language rationale produced by the relevance judge of Section 5.4, which may quote an applicant's organisation or dataset by name;
- the input and result snapshots held in the OPA evaluation log;
- the owner's contact email embedded in a project's configuration; and
- the snapshot of project state frozen at the start of every analysis cycle.

Erasure therefore cascades, within a single atomic transaction, to each of these: we clear the free-text fields, null the judge's score and rationale, redact the evaluation-log snapshots, and replace the identities frozen into the cycle snapshots of every project the user owned or joined with the same placeholders. Erasure must reach the derived and frozen copies in which personal data comes to rest, the judge's rationale and the cycle snapshot being copies that no foreign-key cascade would reach. Erasure additionally withdraws any onboarding

request the user still holds open, clearing its decision metadata, so no live application continues under an erased identity.

Bounded cascade. The cascade is at the same time deliberately bounded. Where a project's configuration holds both the owner's personal contact email and non-personal research metadata, such as the institution and the study's objective, we remove only the contact email. The research metadata describes the project rather than an individual, and is out of scope for erasure under the regulation.

The act of erasure must not itself become a means of recovering what was erased. The audit record that documents an erasure, permanent by the design of Section 5.1, records neither the prior identity nor the originating network address, since either would leave the subject recoverable from the trail intended to outlive them. We run the operation as a single transaction, so a partial erasure can never be left committed, and we gate it by two pre-conditions: a user who still owns active projects cannot be erased until those projects have been removed, and the sole remaining administrator cannot erase themselves. One mechanism serves three triggers, which differ only in what precedes the erasure and not in the erasure itself: a user erasing their own account, an administrator acting upon a request, and an automated scan that erases accounts dormant beyond a defined period.

Dormancy scan. The dormancy trigger runs in two phases, so erasure is never the consequence of a single scan. In the first phase, an account whose most recent activity, the later of its last login and its creation, lies beyond a configured warning threshold receives a single email announcing the date on which it will be erased. In the second phase, an account warned and left untouched for a subsequent grace period is pseudonymised. A user who logs in after being warned escapes erasure, since the second phase considers only those whose last login still precedes the warning, and we enforce a minimum interval between the two phases regardless of how the grace period is configured, so a single run can never both warn and erase the same account. The scan declines to act, rather than erase, in two cases: an account that is the last remaining active administrator, and one that owns a project with an analysis cycle still in flight. In either case it records the refusal and notifies the other administrators that a dormant account awaits manual attention, rather than removing data whose erasure would compromise the platform's governance or interrupt a running computation.

Limits of the reconciliation. Two limits are worth stating plainly. First, because the pseudonymised row keeps its primary key and the foreign keys that link it to the audit, evaluation, and participation records, the subject is not rendered anonymous in the sense of Recital 26: the record could in principle still be singled out and linked (12), so it

remains personal data held under the Article 17(3) basis above rather than placed beyond the reach of the Regulation. Second, the audit trail is retained without a fixed horizon; a production deployment should set a retention period or a periodic-review trigger to satisfy the storage-limitation principle of Article 5(1)(e), which this work does not address.

5.4 The Relevance Judge

Whether an application is genuinely relevant to a project's objective cannot be reduced to rules of the kind the previous two sections handle — a region, an accreditation, a data format. Whether the data an applicant offers and the study they describe bear on what the project sets out to do is a matter of judgement, and it is the judgement a project owner makes, application by application, by reading each one. We delegate a first pass at that judgement to a large language model. We describe this relevance judge here and evaluate its reliability in Chapter 6.

Why an LLM, not a trained classifier. We use a language model rather than a classifier trained for the task for two reasons. First, BraneHub has no corpus of labelled past decisions from which a supervised model could be trained, and could not readily assemble one, because each project defines relevance against its own objective, so labels from one project do not transfer to another. Second, the input is free text: an applicant's answers to an open questionnaire, weighed against a free-text statement of the project's aim. An LLM reads both and assesses their correspondence without task-specific training, which a supervised classifier cannot do here because the labelled data it would require is exactly what the setting cannot supply.

Inputs. We give the judge two things: the project's stated objective, which serves as the query, and a structured description of the application derived from the applicant's questionnaire answers, comprising a title drawn from the dataset name they provide, a description assembled from their answers, and the applicant as author.

Outputs. The judge returns a graded relevance score, one of three levels rather than a binary verdict, together with a natural-language rationale for that score and the identifier of the model that produced it. We store these on the onboarding request and present them to the owner alongside the policy-evaluation result of Section 5.2.

As in Section 5.2, the judge advises and the owner decides: the score and its rationale inform the owner, who retains the authority to accept or reject, and we record the suggestion but never act on it automatically. We treat an LLM's relevance judgement as advice from a fallible assistant, to be weighed by a person, not as an authority that gates admission.

We govern the judge by a feature flag, disabled by default. When the flag is off, the automatic trigger is a no-op and the on-demand endpoint reports that it does not exist, so the feature can be withdrawn from production should it misbehave. The flag also lets us evaluate the judge in isolation from the running platform, as a component in its own right, which Chapter 6 does. When enabled, the judge runs in two ways: automatically and in the background when an application is submitted, and on demand when an owner requests a fresh assessment. In production it runs on a single default provider, with an owner able to select from a fixed set of three providers when requesting a fresh assessment by hand, where the evaluation of Chapter 6 instead exercises several providers in order to compare them. We record every prediction the judge makes, successful or not, in the audit trail as an action of the system rather than of any user, while a person's request for a re-assessment is recorded separately as that person's own action, so the model's contributions and the human's remain distinguishable after the fact.

This design cannot establish whether the judge's suggestions are correct. The deployed mechanism produces a score and a rationale, but the platform holds no ground truth against which to check them: the real onboarding decision is the owner's, made for reasons we do not record, and the description the judge reads in production is itself derived from questionnaire answers rather than the curated metadata a careful evaluation would supply. We therefore evaluate the judge's reliability on external, human-labelled data. The evaluation reported in Chapter 6 exercises the same judging procedure the platform invokes, not a separate reimplementation, so what differs between evaluation and deployment is the data and the choice of model, not the mechanism under test. Chapter 6 reports that evaluation.

6

Evaluation

We evaluate the relevance judge on external, human-labelled benchmarks rather than on BraneHub's own onboarding data, which carries no ground truth; we treat the judge as a component to be measured under controlled conditions. We compare four language models against two non-learning baselines, ablate the evidence shown to the judge, and test every comparison with McNemar's exact test under Holm correction. Section 6.1 states the research questions. Section 6.2 describes how we selected the benchmark, including the benchmark we rejected (ACORDAR-2) and the body-decisiveness criterion that rejection produced. Section 6.3 sets out the experimental design, and Sections 6.4 to 6.7 report the findings for each research question in turn.

6.1 Research Questions

We pose four research questions, which move from whether the judge works, to why it works, to what it costs, and to how it fails.

RQ1 (Competence): Can a general-purpose language model judge whether a piece of compliance text satisfies a stated requirement more accurately than non-learning baselines, and how large is the difference?

RQ2 (Evidence dependence): Does that judgement actually depend on reading the body of the text, or could it be reached from surface cues alone? A benchmark on which a model scores well without reading the content would measure nothing about the capability of interest, so this question is a validity check on both the judge and the benchmark.

RQ3 (Cost and prompting): How does judging accuracy trade off against the cost of the model used, and does a chain-of-thought (CoT) prompting strategy, which asks the model to reason before answering, improve it?

RQ4 (Failure characterisation): What distinguishes the cases a model judges incorrectly from those it judges correctly, and do those error patterns point to a systematic weakness?

6.2 Choosing a Discriminative Benchmark

We required a benchmark whose task mirrors the one that the judge performs in BraneHub, deciding whether a piece of text satisfies a stated requirement, and on which a judge cannot score well without reading the content under test. We reached this requirement after rejecting one benchmark, whose rejection is itself load-bearing for the validity of what follows.

Rejected benchmark (ACORDAR-2). We first used ACORDAR-2, a collection for ad-hoc dataset retrieval (13). We ran it as a complete study under the same statistical protocol we later applied to LegalBench (Section 6.3): three judge models, each run under zero-shot, few-shot and chain-of-thought prompting, gave nine conditions and 34,426 individual relevance predictions. We scored predictions against human gold labels with agreement statistics (accuracy, macro-F1, Cohen's κ, quadratic-weighted κ, Krippendorff's α) and ranking metrics (nDCG at several cut-offs, MAP, MRR, precision, recall), against random and lexical BM25 baselines, with paired Wilcoxon signed-rank tests each under its own Holm-Bonferroni family. The judges cleared the baselines, reaching nDCG@10 of 0.62 to 0.65 against 0.32 for a random ranking and 0.45 for BM25, and agreed with the human annotators at the fair-to-moderate level.

What disqualified ACORDAR-2 as the *primary* benchmark was an empirical finding from that study. Holding model and prompt fixed, we reduced the judge's input from the full record (title, description, tags, author) to the title alone, and ranking quality barely moved: a descriptive drop of only Δ nDCG@10 ≈ -0.0186, within its bootstrap confidence interval and not significant after multiplicity correction. A judge could therefore score close to the right answer from a dataset's title and surrounding metadata without reading its fuller description, so ACORDAR-2 cannot cleanly separate a judge that reads the content from one that pattern-matches the surface. This metadata ablation is not the same manipulation as the clause-removal test we later apply to LegalBench (Section 6.4): ACORDAR-2 did not fall to chance but stayed above the random (0.32) and BM25 (0.45) floors throughout, and the title-only condition showed only insensitivity to the metadata fields, a small and non-significant change rather than a breakdown.

Selection criterion (body-decisiveness). We therefore adopted a single selection criterion, *body-decisiveness*: a usable judge benchmark must make it impossible to score well without reading the content under judgement. The underlying manipulation — withholding part of the input and checking that performance falls — is the familiar partial-input or hypothesis-only control of natural-language-inference evaluation (14, 15); our contribution is to promote it from a post-hoc diagnostic into an explicit precondition a judge benchmark must pass before its scores are trusted. We then searched for a benchmark that was body-decisive, external to this work, human-labelled, openly accessible, and discriminative. We ruled out two kinds of dataset. Labels produced by a language model would make the evaluation circular, measuring the judge against another model's output rather than against human judgement. Access-gated datasets would compromise reproducibility, because others could not obtain the benchmark to check the result.

Primary benchmark (LegalBench). The search settled on LegalBench (16), a suite of legal-reasoning tasks with expert annotations. We adopted two of its tasks: privacy-policy entailment, in which the judge decides whether a one-sentence description correctly characterises a privacy-policy clause (a balanced subset of 1,188 clauses), and contract natural-language inference (NLI), in which the judge decides whether a contract clause satisfies a fixed legal requirement (1,927 clauses across fourteen sub-tasks). These tasks resist surface matching through a structural property of their labels: 92% of negative privacy clauses contain a negation, but so do 35% of positive ones (Table 6.1), so a model or baseline that simply detects negation cannot succeed and the clause itself must be read. Table 6.1 reports the two tasks' clause lengths, the number of distinct requirements each poses, the label balance, and the negation asymmetry on which both turn.

We confirmed body-decisiveness on a small balanced pilot before running the full evaluation. On the pilot the language models scored in the mid-to-high eighties, whereas the keyword-overlap baseline scored exactly 50% even at its best-tuned threshold. The pilot set is balanced between the two labels, so 50% is exactly the accuracy of random guessing. The tuned baseline recovered no signal at all. Removing the clause body dropped the models themselves to that same level. LegalBench therefore had the property that ACORDAR-2 lacked, and only then did we run the full evaluation.

Structured-data pilot (TalentCLEF). We sought a third dataset with structured, multi-field content, in which the judge must weigh several fields rather than read a single block of prose. We rejected the popular résumé-screening datasets for carrying machine-generated labels or undocumented provenance, leaving TalentCLEF (17), a curated benchmark matching curricula vitae to job descriptions, as the only open, human-labelled

Table 6.1: Statistical properties of the two LegalBench tasks (16). The negation-word row is the asymmetry the body-decisiveness criterion is designed to neutralise.

Property	Privacy-policy entailment `privacy_policy_entailment`	Contract NLI (14 sub-tasks) `contract_nli`
Items evaluated	1,188 of 4,335 (all 594 Incorrect, 594 Correct sampled under a fixed seed)	1,927 (all test rows)
Fields shown to the judge	clause text + description statement	clause text + fixed requirement
Distinct requirements	54 description-statement types	14 fixed hypotheses (of ContractNLI's 17)
Label balance		
positive class	Correct: 594 (50%)	Yes: 845 (44%)
negative class	Incorrect: 594 (50%)	No: 1,082 (56%)
Clause length (characters)		
median	339	362
mean	477	465
minimum	30	65
90th percentile	988	921
maximum	3,671	2,493
Negation-word rate	92% of Incorrect vs 35% of Correct	52% of clauses under either label
Licence	CC-BY-NC-3.0	CC-BY-4.0
Source	Hugging Face dataset `nguha/legalbench`	

candidate. TalentCLEF is a closer structural analogue to BraneHub's onboarding gate than the legal-entailment tasks: matching a candidate to a post has the same shape as the decision BraneHub must make, in which the gate weighs an applicant's profile and proposed study against a project's stated requirement. Our first TalentCLEF protocol failed. Under a binary protocol, asking the judge whether a candidate was suitable for a post, the model scored the clear non-matches almost perfectly yet rejected a large share of the matches that human annotators had accepted. This was a calibration mismatch, not a discrimination failure: the model's absolute threshold for "suitable" was stricter than the annotators' more lenient, invitation-to-interview standard (mean rating 5.85 positive versus 1.06 negative, Table 6.2). We corrected this by abandoning the binary framing for a graded rating, scored as a ranking. Under that protocol the full-résumé judge reached AUC 0.921 versus the lexical baseline's 0.788, and, as the structured-data question demanded, the full résumé beat the title-only condition by 0.115 AUC (0.921 versus 0.806). Because this established a workable protocol rather than a complete result, we carry TalentCLEF in this thesis as a structured-data methodological finding (Section 9.3) and as future work (Section 10.3), not as a headline result; we answer the four research questions on LegalBench throughout. The rating-protocol figures are collected in Table 6.2.

Table 6.2: TalentCLEF graded-rating pilot (196 balanced pairs, seed 42) (17). Two of the four AUCs are computed over fewer than 196 pairs.

Condition	AUC	AUC vs hard negatives	Best-threshold acc	Mean rating (pos / neg)
Lexical-overlap baseline	0.788	0.652	73.0%	0.44 / 0.32
Sonnet 4.6, full résumé	0.921	0.883	87.3%	5.85 / 1.06
Sonnet 4.6, title only	0.806	0.777	77.5%	4.26 / 1.29
Haiku 4.5, full résumé	0.912	0.861	85.7%	6.16 / 1.92

Under the pilot's four-token output cap, 23 of the 196 Sonnet full-résumé ratings and 14 of the title-only ratings were unparseable and are excluded, so those two AUCs are computed over the parsed pairs (n=173 and n=182); the Haiku and lexical conditions had no parse failures.

Outcome. Benchmark selection produced the LegalBench primary benchmark proven to be body-decisive, the contract-NLI companion task, the TalentCLEF calibration finding (Sections 9.3 and 10.3), and the body-decisiveness criterion we apply to any judge benchmark.

6.3 Experimental Design

Tasks. We run the evaluation on the two LegalBench tasks of Section 6.2. For privacy-policy entailment we use a balanced subset of 1,188 clauses (all 594 negative clauses together with 594 positive clauses sampled under a fixed seed) in place of the naturally skewed full set. Balancing fixes the majority-class baseline at exactly 50%, so a model's accuracy reads directly as distance above chance rather than being inflated by a lopsided class distribution. We use contract NLI in full, 1,927 clauses across fourteen sub-tasks, which is naturally close to balanced.

Models. We compare four models spanning the cost–capability range:

1. Claude Sonnet 4.6, the larger proprietary model from one vendor;
2. Claude Haiku 4.5, the smaller and cheaper proprietary model from the same vendor;
3. GPT-5.5, a frontier model from another vendor;
4. Gemma-4-31B, an openly available small model offered on a free tier.

We access all four through one OpenAI-compatible interface (OpenRouter), so the model is the only factor that varies across conditions. This single interface corrects an earlier mistake. The ACORDAR-2 study (Section 6.2) routed the Claude models through a third-party proxy that prepended roughly 45,000 cache-read tokens to every Claude request while adding none to the GPT-5.5 requests, so the two model families were never judged under identical conditions; the unified interface used here removes that asymmetry. We vary two further factors. The prompting strategy is either zero-shot, in which the model is asked for its judgement directly, or chain-of-thought, in which it is asked to reason before answering (the implementation also provides a few-shot strategy, which this study does not exercise). For the ablation of RQ2, the clause shown to the judge is either left whole, truncated to its first half, or removed altogether and replaced by a placeholder while the description is left intact.

Baselines. We measure every model against two non-learning baselines: a majority-class baseline, which always predicts the more common label, and a lexical-overlap baseline, which decides from the word overlap between clause and description at its best-tuned threshold. The majority baseline measures what constant guessing achieves. Shallow word-matching is the shortcut that the body-decisiveness criterion is designed to rule out, and the lexical baseline tests directly whether that shortcut is by itself enough.

Metrics. We report four metrics per condition, because no single one is sufficient:

1. accuracy, the proportion of items judged correctly; the most intuitive of the four, but a figure that a skewed class distribution can inflate and that makes no allowance for chance agreement;

2. balanced accuracy, the mean of the accuracies computed separately within each class, which therefore cannot be inflated by such a skew;

3. macro-averaged F1, the unweighted mean of the two classes' F1 scores, each of which is the harmonic mean of that class's precision and recall, so that the smaller class counts for as much as the larger;

4. Cohen's κ, agreement with the human labels once chance agreement has been subtracted, so that 0 marks chance-level agreement and 1 perfect agreement.

Each estimate carries a 95% confidence interval from bootstrap resampling (5,000 resamples under a fixed seed), so that genuine differences can be told from sampling noise.

Significance testing. We test comparisons between systems with McNemar's exact test, because the systems judge the very same items: their outcomes are paired item by item, and an unpaired test would discard that pairing and lose power. Because we make many comparisons, we Holm-correct the p-values for multiple testing. We apply that correction within three separate families rather than across all comparisons at once: comparisons between models, comparisons across the ablation conditions, and comparisons between prompting strategies. Each family answers a distinct question and controls its own family-wise error rate; pooling them would enlarge the correction unnecessarily and cost power without making any individual claim safer.

Reproducibility. The experiment is deterministic by construction. We cache every judgement under a key that records the full set of factors that could change its outcome: the model, the prompting strategy, the item and task, the temperature, the output-token limit, a content hash of the prompt, and the ablation treatment. We fix the temperature at zero. A re-run therefore reads finished judgements from the cache and computes only what is missing, so the reported figures regenerate exactly and a paid experiment of several thousand calls can be interrupted and resumed without loss. We report costs at the providers' published list prices, as estimates rather than billed amounts, and token counts are approximations.

6.4 RQ1 — Competence against Baselines

On the balanced privacy benchmark all four models score between 78 and 82% against 50% for both baselines (Figure 6.1). Both the majority-class and the lexical-overlap baselines score exactly 50% (chance, with the lexical threshold tuned to its best), whereas the four models score 78.0–81.6% (Claude Sonnet 78.0, Claude Haiku 80.9, GPT-5.5 81.6, Gemma 81.2), a margin of roughly thirty points whose bootstrap intervals lie entirely above the baseline. On contract NLI the models reach 91.5–94.4%. Cohen's κ is 0.56 to 0.63 on privacy and 0.83 to 0.89 on contract (Table 6.3), once chance agreement is removed. We answer RQ1 affirmatively: a general-purpose language model judges requirement satisfaction on this compliance text far more accurately than either non-learning baseline. Table 6.3 gives the full figures, with balanced accuracy, macro-F1, Cohen's κ and bootstrap confidence intervals alongside raw accuracy.

Table 6.3: RQ1: model competence against non-learning baselines on both LegalBench tasks. An n/a cell marks a quantity we do not report for that baseline: macro-F1 for both baselines, and Cohen's κ and the bootstrap interval for the lexical baseline as well.

Model	Task	n	Accuracy	Balanced acc	Macro-F1	Cohen's κ	95% CI
Claude Sonnet 4.6	privacy	1188	0.7803	0.7803	0.7707	0.5606	[0.7576, 0.8039]
Claude Haiku 4.5	privacy	1188	0.8089	0.8089	0.8027	0.6178	[0.7862, 0.8308]
GPT-5.5	privacy	1188	0.8157	0.8157	0.8118	0.6313	[0.7929, 0.8367]
Gemma-4-31B (free)	privacy	1188	0.8123	0.8123	0.8091	0.6246	[0.7904, 0.8342]
Majority baseline	privacy	1188	0.5000	0.5000	n/a	0.0000	[0.5000, 0.5000]
Lexical baseline	privacy	1188	0.5000	0.5000	n/a	n/a	n/a
Claude Sonnet 4.6	contract	1927	0.9154	0.9080	0.9128	0.8261	[0.9030, 0.9284]
Claude Haiku 4.5	contract	1927	0.9216	0.9157	0.9196	0.8394	[0.9092, 0.9336]
GPT-5.5	contract	1927	0.9279	0.9237	0.9263	0.8526	[0.9165, 0.9393]
Gemma-4-31B (free)	contract	1927	0.9440	0.9414	0.9429	0.8858	[0.9331, 0.9543]

The competence is not an artefact of aggregating the contract task. Table 6.4 breaks contract NLI down by its fourteen sub-tasks and shows accuracy holding across requirement types, from 80.5% on the hardest (`permissible_copy`, GPT-5.5) to 99.4% on the easiest (`sharing_with_employees`).

A second pattern in the same results is counter-intuitive: the most expensive model is not the most accurate. On the privacy task Claude Sonnet, the largest and most expensive of the four, scores 78.0%, significantly below each of the other three under McNemar's Holm-corrected test (Table 6.5), while those three are statistically indistinguishable from one another. On contract NLI the free Gemma model scores highest at 94.4%; its bootstrap

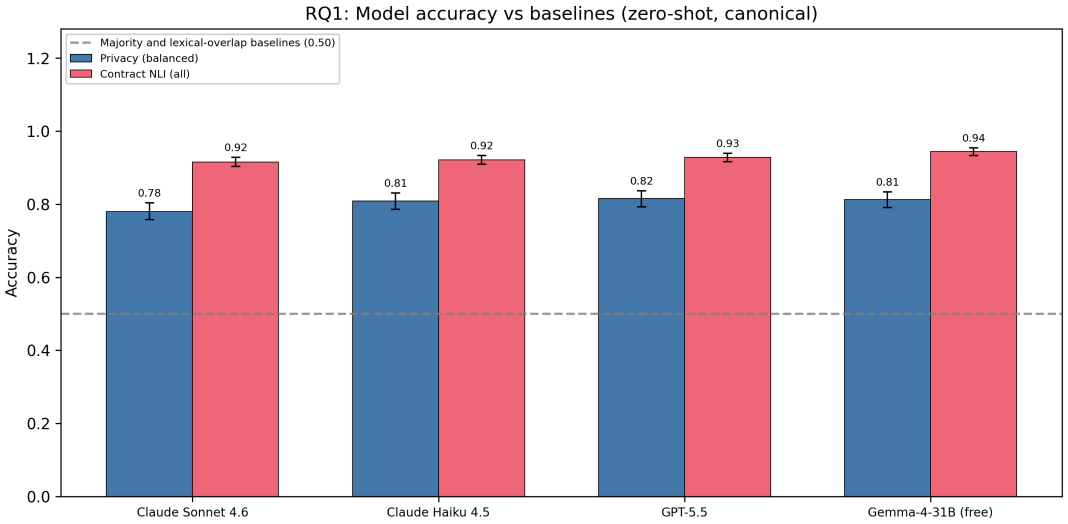

Figure 6.1: Per-model accuracy on the two LegalBench tasks against the majority and lexical-overlap baselines, with bootstrap confidence intervals. The two baselines coincide at the chance level, so a single line carries both.

Table 6.4: Contract-NLI accuracy by sub-task (zero-shot, canonical). The per-subtask n is from the dataset statistics (16); the four model columns are zero-shot canonical accuracy.

Sub-task	n	Sonnet 4.6	Haiku 4.5	GPT-5.5	Gemma-4-31B
confidentiality_of_agreement	82	0.8415	0.8049	0.9024	0.9146
explicit_identification	109	0.8899	0.8624	0.9174	0.9266
inclusion_of_verbally_conveyed_information	139	0.9137	0.9209	0.9496	0.9640
limited_use	208	0.9423	0.8990	0.9183	0.9471
no_licensing	162	0.8210	0.8580	0.8827	0.8951
notice_on_compelled_disclosure	142	0.9789	0.9789	0.9789	0.9789
permissible_acquirement_of_similar_information	178	0.8652	0.9438	0.9607	0.9719
permissible_copy	87	0.9080	0.8966	0.8046	0.8161
permissible_development_of_similar_information	136	0.8824	0.9853	0.9779	0.9706
permissible_post-agreement_possession	111	0.9910	0.9730	0.9730	0.9820
return_of_confidential_information	66	0.9091	0.9394	0.8182	0.9242
sharing_with_employees	170	0.9941	0.9765	0.9882	0.9941
sharing_with_third-parties	180	0.9333	0.9278	0.9167	0.9556
survival_of_obligations	157	0.9108	0.8917	0.8917	0.8981

interval lies entirely above Sonnet's 91.5%, an indication of advantage we did not confirm with a paired significance test on the contract task. RQ3 (Section 6.6) analyses this inverse relation between accuracy and price.

The paired significance tests behind these claims, together with those for the ablation (RQ2) and prompting (RQ3) analyses that follow, are collected in Table 6.5. They are Holm-corrected within three separate families, for the reasons given in Section 6.3.

Table 6.5: McNemar's exact test across the three Holm families. n01 and n10 count the items each system alone gets right; the model and ablation comparisons are on the zero-shot canonical privacy condition.

Family	Comparison	n01	n10	p_raw	p_Holm	sig
Models (m=6)	Sonnet 4.6 vs Haiku 4.5	56	22	0.0001	0.0007	yes
Models (m=6)	Sonnet 4.6 vs GPT-5.5	78	36	0.0001	0.0006	yes
Models (m=6)	Sonnet 4.6 vs Gemma-4-31B	75	37	0.0004	0.0017	yes
Models (m=6)	Haiku 4.5 vs GPT-5.5	53	45	0.4797	1.0000	no
Models (m=6)	Haiku 4.5 vs Gemma-4-31B	47	43	0.7520	1.0000	no
Models (m=6)	GPT-5.5 vs Gemma-4-31B	45	49	0.7572	1.0000	no
Ablation (m=2)	canonical vs truncated_50	27	153	0.0000	0.0000	yes
Ablation (m=2)	canonical vs removed	9	342	0.0000	0.0000	yes
CoT (m=3)	Sonnet 4.6: zero-shot vs CoT	43	17	0.0011	0.0021	yes
CoT (m=3)	Haiku 4.5: zero-shot vs CoT	6	133	0.0000	0.0000	yes
CoT (m=3)	GPT-5.5: zero-shot vs CoT	39	43	0.7407	0.7407	no

6.5 RQ2 — Evidence Dependence

Ablation. We withhold the clause body in three stages (whole, first half, removed) and measure accuracy. On the privacy task Claude Sonnet's accuracy falls monotonically from 78.0% to 67.4 to 50.0, which is exactly chance (Figure 6.2). Both steps of that decline are statistically significant under McNemar's test, Holm-corrected within the ablation family. With the clause removed and only the description left, the model does no better than chance, so the judgement depends on reading the content; LegalBench is body-decisive on the full data, confirming the pilot (Section 6.2) and the property ACORDAR-2 lacked. The collapse generalises: GPT-5.5 also falls to 50.0% on the body-removed privacy condition, and body-removed contract NLI drops to 0.5615 accuracy (Table 6.6).

We split the privacy clauses by negation to expose how the model reaches its answers; the split is made by a conservative surface pattern over common negation words, so the strata

are a diagnostic proxy rather than a definitive linguistic annotation. On the whole clause Sonnet scores 88.7% on negated clauses versus 59.5% on non-negated ones (Table 6.6), reflecting the negation asymmetry around which the benchmark is built. With the clause removed, negated-item accuracy stays at 72.8% while non-negated accuracy falls to 10.4%. With no clause to read, the model guesses from the negation cue of the description alone, which is why a benchmark whose labels could be recovered from that cue would have measured nothing.

Table 6.6 sets out the ablation ladder and the negation split together.

Table 6.6: RQ2: clause-treatment ablation with negation strata (privacy, n=1188 per row). GPT-5.5 reproduces Sonnet's removed-condition figures exactly. We stratify by negation on the privacy task only, so the contract row's strata are n/a.

Model	Treatment	Acc	Acc (negated)	Acc (non-neg)	n_neg	n_nonneg
Claude Sonnet 4.6	canonical	0.7803	0.8873	0.5945	754	434
Claude Sonnet 4.6	truncated_50	0.6742	0.8156	0.4286	754	434
Claude Sonnet 4.6	removed	0.5000	0.7281	0.1037	754	434
GPT-5.5	removed	0.5000	0.7281	0.1037	754	434
Claude Sonnet 4.6	removed (contract)	0.5615	n/a	n/a	n/a	n/a

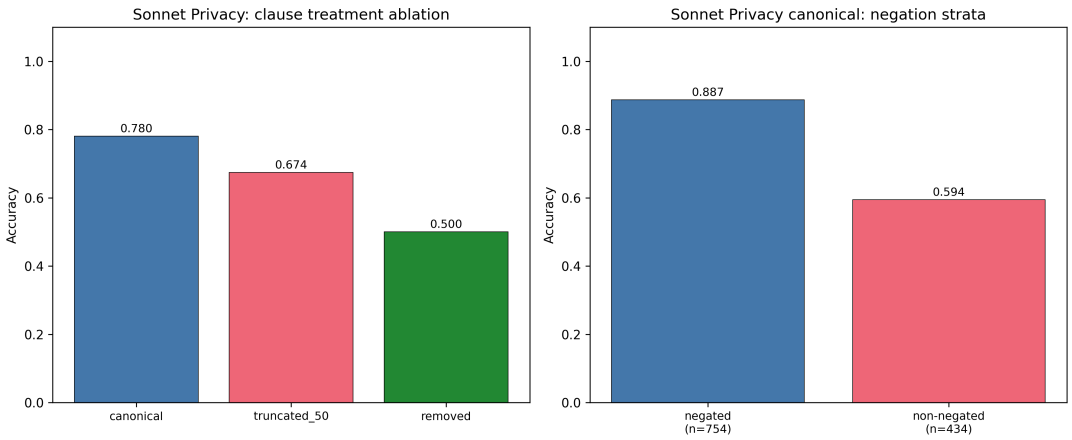

Figure 6.2: The ablation ladder: Claude Sonnet's privacy accuracy as the clause is reduced from whole to first-half to removed, shown overall and split by whether the clause contains a negation.

6.6 RQ3 — Cost and Prompting

The four models differ several-fold in price but span only four accuracy points (78–82%) on the matched privacy condition, and one of them is free. Expressed as accuracy per unit of cost (Figure 6.3), the cheaper Haiku returns 3.52 against Sonnet's 1.13, a 3.1× advantage (Table 6.7), and the free Gemma model scores 81.2% at no marginal cost at all. For this task, paying more buys very little: a free model is competitive with the frontier, a result with direct bearing on how an onboarding gate like BraneHub's might economically be run (Chapter 7). Table 6.7 reports the cost accounting behind this.

Table 6.7: Cost and accuracy-per-dollar. Token counts and total spend are blended over every condition run for each provider; cost-per-1k and accuracy are restricted to the matched zero-shot canonical privacy condition. Dollar figures are estimates at OpenRouter list prices (June 2026) and token counts are character-based estimates, not API-reported usage.

Provider	n calls	Tokens in	Tokens out	Total (USD)	Cost/1k (matched)	Privacy acc	Acc per $1k
Claude Sonnet 4.6	8606	1,495,422	138,148	$6.56	$0.6900	0.7803	1.1308
Claude Haiku 4.5	4303	988,285	149,548	$1.74	$0.2299	0.8089	3.5190
GPT-5.5	5491	1,117,756	74,905	$7.84	$1.1567	0.8157	0.7051
Gemma-4-31B (free)	3115	658,335	1,935	$0.00	$0.0000	0.8123	free

Chain-of-thought prompting is model-dependent rather than a uniform improvement (Table 6.5). Asking the model to reason before committing to an answer significantly helped Claude Sonnet, significantly hurt Claude Haiku (133 items flipped wrong against 6 flipped right, $p < 0.0001$), and made no significant difference to GPT-5.5 (McNemar's test within the prompting family, Holm-corrected). A prompting strategy must therefore be chosen per model by measurement, not transferred across models.

6.7 RQ4 — Characterising the Failures

We fit a logistic regression predicting, for each privacy clause, whether Claude Sonnet erred on it, from three features of the item: whether the clause contains a negation, the length of the clause, and the length of the description. Negation dominates (Figure 6.4): its standardised coefficient is −0.8275, the largest in magnitude and negative, so a negated clause is *less* likely to be misjudged, consistent with the finding in Section 6.5 that the model is strongest on negated clauses. Clause length has coefficient 0.3555 (longer clauses are judged slightly less reliably) and description length 0.1023 (Table 6.8). The model's

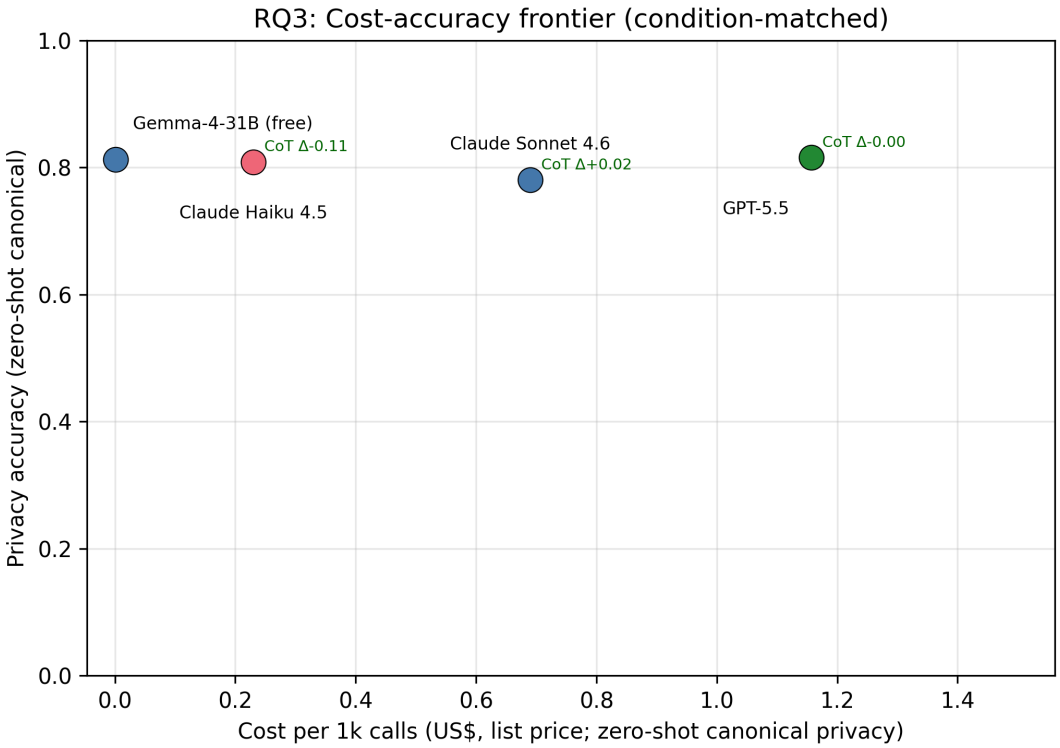

Figure 6.3: The cost–accuracy frontier: privacy-task accuracy against estimated cost per thousand calls, with the free model marked.

errors thus track negation, the linguistic feature the benchmark itself depends on. The fitted coefficients are given in Table 6.8.

Table 6.8: RQ4: standardised logistic-regression coefficients predicting whether Claude Sonnet errs on a privacy clause (fit on n=1188). Holdout accuracy is 0.7647 on a 30% stratified split (n=357, seed 42), against a majority-class share of 0.7815.

Feature	Standardised coefficient
negated	−0.8275
len_text	0.3555
len_description	0.1023
intercept	−1.5038

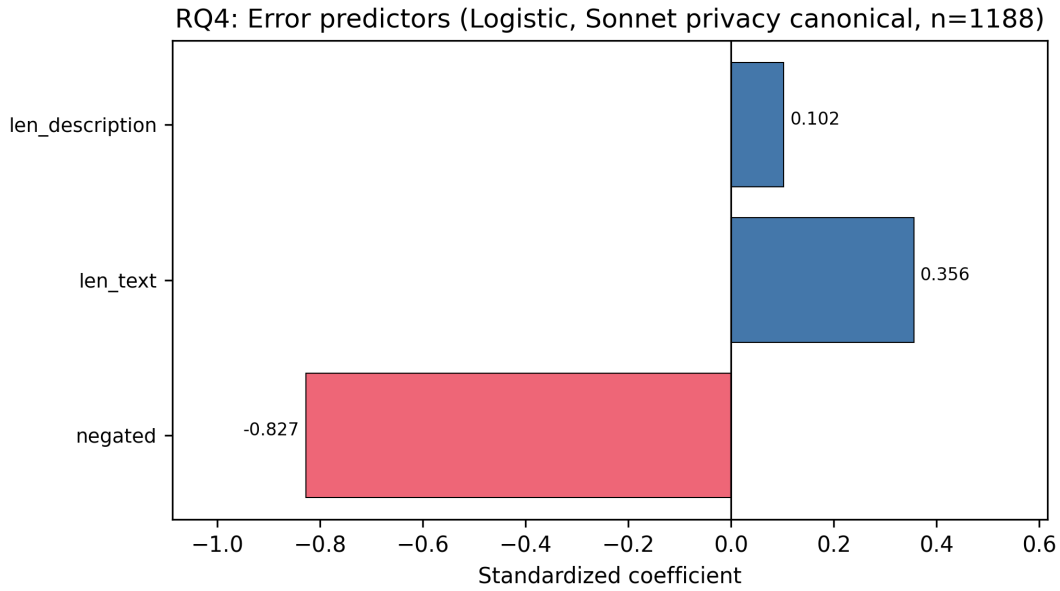

Figure 6.4: Standardised logistic-regression coefficients for the three error features (negation, clause length, description length), with negation the dominant, protective term.

The regression ranks which features are associated with error; it does not predict error out of sample. On a 30% stratified holdout (n=357) its accuracy is 76.5%, below the majority-class rate of 78.2% (Table 6.8), so it adds essentially no predictive power over the guess that the model is usually right. What it establishes robustly, through the size of that one coefficient, is that negation, rather than clause length or description specificity, is the feature most associated with whether the model succeeds or fails. This is a coherence check rather than an independent discovery: negation is also the axis along which the

benchmark's own difficulty is constructed (Table 6.1), so the finding confirms that the judge tracks the testbed's discriminative feature rather than exposing a weakness that would transfer beyond it.

Summary. Across the four questions: a general-purpose language model judges this compliance text 78.0–81.6% on privacy and 91.5–94.4% on contract, far above the 50% non-learning baselines (RQ1); it reads the content, falling to 50.0% when the clause is removed (RQ2); a free model is competitive with the frontier, while chain-of-thought prompting helps Sonnet, hurts Haiku, and does nothing for GPT-5.5 (RQ3); and what most distinguishes its errors is the handling of negation (RQ4). Chapter 7 discusses the implications for BraneHub's onboarding gate and for the use of language-model judges more generally. All figures in Sections 6.4–6.7 trace to the prediction cache `runs/legalbench/cache.sqlite`; the run is deterministic and regenerable (Section 6.3).

6.8 System Performance

Sections 6.1 to 6.7 evaluated the relevance judge. We now measure the platform that hosts it, reporting its runtime behaviour (request-processing latency, the cost of a policy evaluation, and behaviour under concurrent load) so that the constructive contributions of Chapters 4 and 5 are assessed empirically and not only by the correctness tests of Section 4.4.

Method. These are indicative in-process measurements, not deployment figures. We measure request latency in process, through the application's test client, against the PostgreSQL test database seeded with a fixed synthetic dataset (ten projects, sixty-one onboarding requests, fifteen hundred audit entries, two hundred evaluation-log entries); it therefore captures the application's own processing time, excludes network transit and the production server, and is a lower bound on end-to-end latency rather than a deployment measurement. Each figure is the median, ninetieth, and ninety-ninth percentile over two hundred repetitions after a warm-up. Policy-evaluation latency is the real round-trip to a local Open Policy Agent server. We exercise concurrency with in-process worker threads, which indicates scaling on a single machine rather than the throughput a multi-worker deployment would reach. Finally, because the platform is also live in production, we measure its real end-to-end latency directly against the running deployment (Table 6.12).

The governance layer is not a performance bottleneck. Server-side processing is sub-millisecond for the common owner and authentication paths and stays in the low single-digit milliseconds for the heavier compliance views (Table 6.9). The most expensive read is

Table 6.9: Server-side request-processing latency in milliseconds (n=200, in-process test client, seeded test database).

Operation	Role	p50	p90	p99
`GET /health`	anonymous	0.16	0.17	0.22
`POST /login`	anonymous	0.34	0.46	0.68
`GET /dashboard`	owner	0.69	0.92	0.97
`GET /explore`	owner	1.31	1.53	1.59
`GET /onboarding-requests`	owner	1.90	2.11	2.16
`GET /auditor/dashboard` (7 charts)	auditor	7.12	8.22	9.91
`GET /auditor/audit-logs`	auditor	2.82	3.17	3.41
`GET /auditor/opa-eval-logs`	auditor	4.07	4.63	5.81
`GET /admin/dashboard`	admin	3.24	3.80	4.72

Table 6.10: OPA policy-evaluation latency in milliseconds (n=200, real round-trip to a local OPA server).

Step	p50	p90	p99
Policy upload (PUT module)	0.48	0.76	1.12
Decision query	0.31	0.42	0.59
Full evaluation (upload + query)	0.83	1.23	1.55

Table 6.11: Throughput and latency of `GET /dashboard` under in-process concurrent load (800 requests per level). Indicative single-machine scaling, not a production throughput figure.

Concurrency	Throughput (req/s)	p50 (ms)	p90 (ms)
1	653	1.49	2.36
2	1050	1.78	2.81
4	1447	2.68	3.86
8	1766	4.26	6.24

the auditor dashboard at p50 7.12 ms (Table 6.9); it computes seven chart aggregations over the audit and evaluation logs within a single request, so its cost is dominated by those aggregation queries and not by the framework. A policy evaluation, uploading the per-project Rego module and querying the decision, completes in under a millisecond against a local agent (Table 6.10), so the compliance gate adds under 1 ms to an onboarding submission. Under concurrent load (Table 6.11) throughput rises from 653 to 1,766 requests per second as in-process concurrency increases from one to eight, while median processing latency rises from 1.5 to 4.3 ms; the sub-linear scaling reflects the single Python process (the interpreter lock and a shared database connection), not the application, and a multi-worker deployment would scale further. Correctness under concurrency, namely that two deliveries contending on the same cycle cannot lose an update, is not a question of throughput and is established separately by the row-level-locking tests of Section 4.3 and Appendix A.

The figures so far isolate the application's own processing time. BraneHub also runs as a live service (served by gunicorn behind TLS, with HTTP/2 and HTTP/3, HSTS, and the strict content-security policy of Section 5.1 in force), so we measure its real end-to-end latency directly against the running deployment, and Table 6.12 reports it for the two public endpoints over a warm, connection-reused client.

Table 6.12: Production end-to-end latency in milliseconds against the live deployment (gunicorn + TLS) over the network (n=40 warm, n=12 cold). "Warm" reuses a kept-alive connection; "cold" pays a fresh TLS handshake per request.

Endpoint / condition	p50	p90	p99
`/health` (warm connection)	34.4	36.1	86.5
`/login` (warm connection)	38.4	61.4	80.2
`/login` (cold connection, incl. TLS handshake)	266.2	293.8	—

A request therefore completes end to end at p50 38.4 ms for the login page and 34.4 ms for the health check (Table 6.12), of which the application's own processing is the one-to-seven milliseconds of Table 6.9 and the remainder is network transit and TLS; only the first request on a fresh connection pays the one-time connection-and-handshake cost, p50 266.2 ms (Table 6.12). The warm median stays at about 38 ms as concurrency rises from two to eight clients in the same measurement, so the deployed server absorbs concurrent load without latency degradation. A usability evaluation of the auditor and owner interfaces remains future work (Section 10.3).

7

Discussion

7.1 Summary of Findings

The evaluation asks whether the relevance judge of Section 5.4 can be trusted, and the four research questions together return a qualified positive answer: the judge clears both baselines (RQ1), depends on reading the clause body (RQ2), is matched by a free model (RQ3), and is both strongest and most error-prone on negation (RQ4). We take each in turn.

RQ1: competence. On the balanced privacy task the four models reach 78.0–81.6% accuracy (Sonnet 78.0, Haiku 80.9, GPT-5.5 81.6, Gemma 81.2), against 50.0% for both the majority-class and the lexical-overlap baselines, a margin of roughly thirty points whose bootstrap intervals lie far clear of the baseline (Table 6.3). On contract NLI the models reach 91.5–94.4%, with Cohen's κ rising from 0.56–0.63 on privacy to 0.83–0.89 on contract. We tuned the lexical baseline to its best overlap threshold on the same clauses, so the thirty-point gap cannot be attributed to shared vocabulary; the majority baseline establishes what guessing the common label achieves. Both baselines sit at chance while the models sit far above it, which means the models read requirement-satisfaction from the clause rather than from surface cues the heuristics could exploit.

RQ2: evidence dependence. RQ2 tests whether the RQ1 margin reflects clause content rather than the prompt template. We ablate the clause body in stages: Sonnet's privacy accuracy falls from 78.0% on the whole clause to 67.4% with the first half and to 50.0% — exactly chance — with the clause removed (Table 6.6, Figure 6.2), and both steps are significant under McNemar's test within the ablation family. The graded decline shows the dependence is genuine rather than incidental: removing half the clause already costs ten points, and removing all of it reaches chance. Without this ablation the RQ1

margin could equally reflect a benchmark artefact; the body-decisiveness test rules that out. The negation strata sharpen the point. With the clause removed, accuracy on negated items stays at 72.81% while non-negated accuracy collapses to 10.37%, so the only signal that survives the ablation is the surface negation cue — which is exactly why a benchmark whose labels could be recovered from that cue alone would have measured nothing.

RQ3: cost and prompting. The free Gemma-4-31B scores 81.2% on privacy and 94.4% on contract NLI (Table 6.3), matching or exceeding every proprietary model; the largest and most expensive model, Sonnet 4.6, is the weakest on privacy at 78.0%. Capability and price diverge here: expressed as accuracy per dollar, Haiku returns roughly three times the value of Sonnet, and Gemma sits on the frontier at no marginal model cost (Table 6.7). Judging this compliance text therefore does not require the largest or most expensive model. Chain-of-thought prompting admits no such single recommendation: it significantly helps Sonnet, significantly hurts Haiku, and does nothing for GPT-5.5 (Table 6.5). Its effect is an interaction between the technique and the model, not a property of the technique, so we tune the prompting strategy per model rather than adopt one default.

RQ4: failure characterisation. Negation is both the feature the judge handles best and the feature most associated with its errors. On the whole privacy clause Sonnet reaches 88.73% accuracy on negated clauses against 59.45% on non-negated ones (Table 6.6), and in the error-prediction regression negation carries the largest standardised coefficient at -0.83, protective rather than harmful (Table 6.8). Because negation appears in 92% of the Incorrect privacy clauses and 52% of all contract clauses (Table 6.1, Section 6.2), this is a property of the compliance and legal domain in which the judge would operate, not a marginal quirk.

In the structured-data pilot (Section 6.2), asked for a binary suitability verdict, the model identified the clear non-matches well but rejected a large share of the matches the annotators had accepted. This is a calibration error, not a discrimination error: under a graded ranking the model separates the classes at AUC 0.921, well above the lexical baseline's 0.788 (Table 6.2). The model's implicit threshold for "suitable" ran stricter than the annotators' more lenient, invitation-to-interview standard, and re-casting the task as a graded ranking rather than a binary cut recovered, from that separation, a best-threshold accuracy of 87.3%. A well-discriminating judge can thus still return wrong answers when the point at which it commits is miscalibrated against the human standard it approximates; Section 7.2 turns on this distinction directly.

7.2 Implications

These results support the advisory design of Section 5.4 on measured grounds rather than out of caution. We draw out the cost implication first, then the more consequential one for the judge's authority.

Cost. Because the free Gemma-4-31B matches the frontier (Section 7.1, Table 6.7), we can run the relevance suggestion at no marginal model cost, and need not ration judge invocations across a project's applications. This economy holds only for the task we evaluated: zero-shot, discriminative judgements over short legal-style clauses, in which a negation cue carries much of the signal and accuracy depends on reading the clause body. It does not show that a free model remains competitive on the longer-context or more subjective reasoning a production application might require; Chapter 8 examines that limitation.

Authority. The accuracy and calibration results together rule out automatic gating. An accuracy of 78.0–81.6% on privacy (Table 6.3) is high enough to direct a project owner's attention usefully but too low to gate admission unaided, since roughly one application in five would be misjudged by a mechanism acting alone — a bound the LegalBench accuracy results establish on their own. The calibration gap from the structured pilot, though preliminary, supplies the further reason this residual error is the wrong shape for an automatic threshold. A symmetric, unbiased error might be tolerable behind a conservative cut-off; the observed skew is not. A model whose bar for "suitable" runs stricter than the human standard does not fail at random but excludes acceptable applicants quietly and consistently. The platform cannot catch those exclusions, because it holds no ground truth against which to check them. The advisory posture of Section 5.4, in which the judge suggests and a person decides, is therefore what the measured reliability supports, and the calibration finding supplies the specific reason an automatic gate would be unsafe rather than merely imperfect. The design choice and the evidence arrive at the same place.

Advisory is not self-enforcing. Surfacing the verdict as advice avoids an automated decision in the sense of Article 22 (18), but only to the extent that the human review it defers to is genuine. The calibration gap that makes an automatic gate unsafe also leaves the advisory posture fragile: an owner who comes to rely on a usually-helpful suggestion may defer to it from habit, so a quietly miscalibrated judge can still shape admissions through automation bias with a person only nominally in the loop. The safeguard therefore rests on the review staying meaningful — the owner retaining and using the authority to overrule the suggestion. The platform supports that review by recording the suggestion

alongside the decision it informs, but it cannot guarantee that the review is genuine. How owners actually weigh the suggestion we do not measure; whether the advisory framing withstands automation bias in practice is left open, and Chapter 8 records it among the study's limits.

Lessons for evaluating LLM judges. Two cautions transfer beyond BraneHub, and both replace reputation with measurement. First, a judge benchmark is informative only if it is body-decisive, and this property must be verified by ablation rather than assumed: the abandoned ACORDAR-2 benchmark of Section 6.2 looked reasonable until its ablation came back flat, and the difference between a benchmark that measures the capability of interest and one that measures a surface artefact was visible only once the body was withheld. Second, neither the choice of model nor the choice of prompting strategy transfers reliably between settings. The cost result shows the strongest model need not be the most expensive (Section 7.1), and the prompting result shows a strategy that helps one model can degrade another, so both decisions are better resolved by measurement on the task at hand than by convention. A domain-specific corollary follows: a judge deployed over compliance and legal text, dense with negation, should expect negation to be at once where it is strongest and where it is most exposed.

Bound. One bound frames all of the above and Chapter 8 takes it up in full. The figures of Chapter 6 measure the judging procedure on curated benchmark text, whereas in production it reads descriptions assembled from applicants' own answers (Section 5.4). They are therefore best read as the capability of the procedure on well-formed input, not as a direct measurement of what the deployed gate would achieve.

8

Threats to Validity

We bound the conclusions of Chapter 6 along the four validity dimensions of Wohlin et al. (19):

1. **Internal validity**: whether the measurements reflect what the model did.
2. **External validity**: how far the results generalise.
3. **Construct validity**: whether the measured quantity (agreement with a benchmark label) stands for the onboarding judgement of interest.
4. **Conclusion validity**: the statistical inferences drawn from the measurements.

8.1 Internal Validity

Label noise. The most material internal threat is noise in the gold labels. The privacy labels derive from APP-350 (20), where annotations are made at the level of a document's data practices and then associated with the clauses that express them; a clause labelled *Correct* for a given practice does not always contain, in isolation, the explicit text a reader would need to confirm that practice. We judge that some of the cases we score as failures are disagreements with an imperfect label rather than genuine errors, so the accuracy ceiling reported in Chapter 6 reflects label imperfection as well as model limitation.

Verdict extraction. A second, smaller threat is extracting a discrete label from the model's free-text reply. We resolve ambiguous replies conservatively, preferring the negative class when both terms appear, but mis-extraction remains possible. We encountered one concrete instance: a reasoning model returned no parseable verdict until we handled its reasoning output explicitly. We caught and corrected it, but the episode confirms that the measurement depends on parsing the model's surface form.

8.2 External Validity

Evaluation–deployment input gap. The principal external threat is the gap between the evaluation input and the production input (Section 5.4; Chapter 7). We measured the judge on curated benchmark clauses, whereas in deployment it reads a description assembled from an applicant's free-form questionnaire answers. The judging code is the same, but the input distributions are not, and we do not establish how much of the measured accuracy survives that shift.

Narrow coverage. Coverage is narrow in two further respects.

1. Our human-labelled results rest on two English legal-text tasks, privacy-policy entailment and contract NLI, and the structured result on a single English curriculum-vitae matching pilot, so generalisation to other compliance domains, document genres, or languages is untested.

2. The structured pilot uses synthetic curricula and postings, so behaviour on real applicant material may differ.

Benchmark, not field study. By design we measure the judging procedure on benchmarks, not the deployed gate in the field; the quality of the human decision the suggestion informs is outside our scope. For the same reason we report the platform's runtime behaviour in Section 6.8 in two forms: as in-process processing latency (request handling, policy evaluation, and concurrent scaling), and as end-to-end latency measured against the live production deployment. We establish the platform's correctness by construction and by the test suite of Chapter 4 and Appendix A; we do not measure the usability of its auditor and owner interfaces. Three items remain future work (Section 10.3):

1. a sustained high-load stress evaluation of the deployment;

2. a usability evaluation of the auditor and owner interfaces;

3. an evaluation of the judge on BraneHub's own onboarding data rather than external benchmarks.

RQ3 task scope. A final external bound concerns RQ3. The conclusion that a cheap or free model is competitive holds for the task measured here: zero-shot, discriminative judgements over short legal-style text in which a negation cue carries much of the signal. It does not establish that such a model would remain competitive on the longer-context, multi-step, or more subjective reasoning a production judgement might demand.

Simulator vs. production integrator. A further external bound concerns the integration rather than the judge. Because the production Brane integrator was not available to drive the integration locally during development, we obtained the integration results against the containerised simulator of Section 4.4 rather than against the production integrator. The simulator reproduces the integrator's documented request and callback contract for the full version-one and version-two package and federated-analysis lifecycle, including its failure modes, so it exercises the same BraneHub-side behaviour the real interface would: idempotent webhook handling, the cycle state machine, row-level locking, and the package and provisioning callbacks. It does not establish that the production integrator honours that contract faithfully; confirming the integration against the live integrator remains future work (Section 10.3).

8.3 Construct Validity

Label as proxy. Construct validity depends on whether agreement with a benchmark label measures the judgement BraneHub actually needs. The benchmark's discrete label (*Correct*/*Incorrect*, or the graded relevance score) is a proxy for the context-dependent assessment a project owner makes when deciding whether an applicant belongs in a study, and agreement with that label is an imperfect stand-in for the quality of that assessment. The proxy is further weakened by treating benchmark gold as ground truth when the gold is itself human annotation carrying the noise discussed in Section 8.1.

Analogy, not instance. The tasks are analogues, not instances, of the onboarding decision. Entailment over legal clauses is structurally similar to judging whether a stated purpose fits a project, but high accuracy on the former does not by itself establish usefulness on the latter; the inference is one of analogy.

8.4 Conclusion Validity

Single-run determinism. We ran each condition once at temperature zero. With caching this makes the results reproducible but provides no estimate of the model's own sampling variability: the bootstrap confidence intervals quantify uncertainty over the sampled items, not over model stochasticity, so behaviour under sampling is unmeasured.

Multiplicity correction. We Holm-correct the McNemar comparisons within three families (across models, across ablations, and across prompting strategies), for the reasons given in Section 6.3. Pooling them into a single, stricter family would raise the threshold,

but it would not overturn any comparison we report as significant: each such comparison has a raw p no greater than 0.0011 and so survives even the single-family correction. The comparisons that are non-significant here — for instance Haiku versus GPT-5.5, or chain-of-thought on GPT-5.5 — likewise remain non-significant. The family choice changes the reported thresholds, not which conclusions hold.

Baseline strength. Our two baselines are non-learning — a lexical-overlap rule tuned to its best threshold on the same clauses, and the majority class — and both sit at chance on the balanced privacy task. The thirty-point margin of Chapter 6 is accordingly a margin over non-learning heuristics, not over a trained discriminative model. A supervised skyline, such as TF-IDF features with logistic regression, would set a higher and more demanding reference; we did not train one, and so read the RQ1 result as evidence that the judge defeats naive surface heuristics rather than as a measure of how it compares with task-specific supervised learning. A trained skyline is the natural stronger baseline and we leave it to future work (Section 10.3).

Gemma access channel. We obtained the Gemma contract figures through the paid access route after the free route's retry behaviour made a complete run impractical. The model is identical and only the access channel differs, recorded in the run's methodology notes, but we flag it as a source of heterogeneity.

ACORDAR-2 proxy confound. The abandoned first study carried a more serious channel heterogeneity, which we record here. In the ACORDAR-2 evaluation (Section 6.2) we reached the Claude judges through a third-party proxy that injected roughly 45,000 cache-read tokens of hidden context into every Claude request but none into the GPT-5.5 request, so that study's cross-model cost and quality comparison was confounded: we did not run it under identical conditions. The confound does not propagate to the present results. It motivated our move to a single OpenAI-compatible access channel (Section 6.3) for the LegalBench study on which the RQ3 conclusions rest, where every model receives the same prompt with no injected context. We disclose the earlier asymmetry for completeness.

Pilot size. The structured result is a pilot of fewer than two hundred pairs, so its effect sizes are indicative rather than settled. The pilot's four-token output cap additionally cost the two Sonnet conditions 23 and 14 unparseable ratings respectively (Table 6.2), so their AUCs rest on 173 and 182 parsed pairs rather than the full 196, and the full-versus-title comparison is between two slightly different item subsets.

Cost estimation. We compute the cost and token figures as list-price estimates from character counts rather than billed amounts; they support the order-of-magnitude comparison they are used for but are not exact.

8.5 Ethical and Regulatory Scope

The validity threats above concern whether the measurements support their conclusions. A final group of limits concerns the platform's conduct and obligations beyond what we evaluated. These are deployment prerequisites rather than threats to the validity of the evaluation; we record them so that the compliance framing of this thesis is not read as more complete than it is.

Fairness and disparate impact. The relevance judge ranks people's applications, and the structured pilot in particular casts it as résumé screening — a setting with a well-documented potential for disparate impact across protected groups. We evaluated the judge for aggregate accuracy and calibration only, not for differential error rates across applicant subgroups, and the benchmarks we used carry no group annotations from which such rates could be computed. A judge that is well calibrated on average can still err systematically against a subgroup, so a fairness audit on group-annotated data is a precondition for any non-advisory use. We name it as a limitation here and return to it as future work (Section 10.3).

Adversarial robustness. The judge reads text the applicant supplies. Unlike the in-platform assistant, which isolates user-controlled text under a non-privileged role precisely to limit prompt injection, the relevance judge concatenates the applicant's questionnaire answers directly into the description it scores. A crafted application could therefore attempt to steer the verdict rather than describe a study faithfully. We did not evaluate adversarial robustness, and while the advisory posture bounds the damage — a manipulated suggestion still only advises a human — it does not remove it. Hardening the judge against applicant-side injection is a deployment concern and future work.

Regulatory coverage. Our treatment of compliance is bounded to the mechanisms BraneHub implements and does not exhaust the obligations a production deployment would carry. We do not work through the special-category regime of Article 9 that federated health or registry data would often trigger, the allocation of controller and processor responsibilities among the participating institutions, or the applicant's own rights of information and recourse against an adverse relevance judgement. These are necessary parts of a complete regulatory case and lie outside the scope of this thesis; we flag them so that the framework of Chapter 5 is read as a set of implemented mechanisms rather than as a finished legal solution.

9

Related Work

We relate BraneHub to four research threads:

1. language models as judges (Section 9.1);
2. legal and compliance natural language processing (NLP) benchmarks (Section 9.2);
3. structured candidate matching (Section 9.3);
4. federated data platforms and policy as code (Section 9.4).

We contribute not a new judging method, benchmark, or platform, but an ablation-grounded assessment of a language-model judge embedded in a working compliance platform. For each thread we name the concrete systems that define it and identify the gap we occupy.

9.1 Language Models as Judges

Three lines of work use language models to evaluate rather than generate. Zheng et al., in the MT-Bench and Chatbot Arena study, used a strong model as a stand-in for human preference and reported that its pairwise verdicts agree with human raters at a rate comparable to the agreement between humans themselves, while documenting characteristic failure modes such as position and verbosity bias (9). Faggioli et al. examined whether a model can supply relevance judgements for information-retrieval evaluation, and identified a circularity risk when the same model family both retrieves and judges (10). RankGPT and the listwise rerankers that followed it replace per-item scoring with one-pass ordering of the full candidate list, treating relevance as a permutation rather than a classification problem (21). All three target open-ended generation quality or retrieval relevance over

general and web content, and all validate the judge by agreement with human labels or preference.

We differ in domain and method. Our judge does not rank chatbot answers or web documents; it gates admission to a project over legal-style text, emitting accept or reject rather than an ordering. We also do not validate by agreement alone. We ablate the document body and find that accuracy collapses towards the chance level, confirming that the verdicts depend on body content rather than on length, formatting, or other surface cues (Chapter 6). Agreement can be high for the wrong reasons: our first benchmark (Section 6.2) reached respectable agreement, yet a body-withheld control recovered the label without reading the body, so we discarded it. Prior work in this thread certifies a judge by correlation with human labels; we add a body-dependence test as a precondition, so that agreement counts only once we have ruled out shortcut behaviour. The test adapts a control already established in natural-language-inference evaluation, where hypothesis-only baselines expose datasets whose labels can be recovered without reading the premise (14, 15); we carry it over from diagnosing a dataset to certifying a judge.

9.2 Legal and Compliance NLP Benchmarks

We draw on human-labelled resources from legal and compliance NLP. Three span the field's range. LegalBench, assembled by Guha et al. through a collaboration across many institutions, collects a large and heterogeneous suite of legal-reasoning tasks hand-built by legal professionals (16). ContractNLI, introduced by Koreeda and Manning, restricts to one document type, framing contract understanding as natural-language inference: it pairs a non-disclosure agreement with hypotheses a model must judge as entailed, contradicted, or not mentioned (22). APP-350, the corpus of Zimmeck et al., annotates privacy policies with expert labels for the data practices each policy describes (20). Each defines its own success criteria and leaderboard for legal-text understanding.

We propose no new benchmark. We repurpose these resources as a discriminative testbed for our judge; our methodological contribution is dataset selection, not construction. We did not adopt a dataset merely because it was available and legal in subject. We required that the gold label depend on the document body, verified by the same body-withholding ablation we apply to the judge, so that strong performance could not be explained by surface cues correlated with the label. We set aside several otherwise reasonable candidates because a body-withheld control recovered the label too readily, showing the task was decidable from metadata or template structure. Our contribution is the selection criterion

(Section 6.2): a testbed must survive a body-withholding ablation before it can certify a judge.

9.3 Structured Candidate Matching

Our structured pilot builds on curriculum-vitae and job-description matching. TalentCLEF is a recent shared task posing the alignment of candidate profiles to job requirements as a multilingual ranking and skill-matching problem with a shared evaluation harness (17). Such evaluations score a system by how well it ranks or retrieves suitable candidates, not by the absolute accept or reject verdicts a deployed gate must emit.

Our pilot separates discrimination from calibration, a gap that ranking-centred evaluation hides. On the matching data the model discriminated well, ordering suitable candidates above unsuitable ones, which a ranking metric alone records as success. Yet its absolute threshold for declaring a candidate "suitable" was markedly stricter than the human annotators', so a deployed system that acted on the model's binary verdict would have turned away applicants the annotators deemed acceptable (Sections 6.2 and 7.2). A ranking metric cannot expose this, because reordering the same items can be excellent while the cut point is miscalibrated. We contribute the measured discrimination–calibration gap and its consequence for a gate that must emit verdicts rather than rankings.

9.4 Federated Data Platforms and Policy as Code

BraneHub relates to two infrastructure lines: systems that execute analysis across data they do not centralise, and mechanisms that govern such execution as code.

Execution. DataSHIELD runs statistical computations at the holding site and returns only non-disclosive aggregates (1); the Personal Health Train moves analyses ("trains") to data stations rather than the reverse (23); and Vantage6 provides a client-server implementation for orchestrating federated tasks across organisations, with its own authorisation layer governing which parties may run which tasks (3). OpenSAFELY goes furthest towards governance: it runs analyses in situ against electronic health records held by their custodian, and enforces reproducibility by requiring that all analysis code be public and by keeping an audit trail of every query run against the real data (24). The Brane framework, on which BraneHub is built, supplies the federated execution and workflow orchestration the system relies upon (2), while the FAIR Data Point model BraneHub adopts for onboarding

(Section 2.2) lets each custodian expose dataset metadata for discovery without revealing the records themselves (6).

Governance. Policy-as-code engines such as the Open Policy Agent externalise authorisation decisions into a declarative, reviewable rule language evaluated at request time (8). At the complementary level of data-use terms rather than request-time authorisation, the GA4GH Data Use Ontology standardises machine-readable consent and permitted-use codes, so that a dataset's allowable uses are represented consistently, and access requests matched against them automatically, wherever the standard is adopted (25).

With the partial exception of OpenSAFELY's audit and reproducibility controls, these systems concentrate on the execution of federated analysis and on technical access control at the moment of action. We govern the full lifecycle around execution:

1. admission through onboarding;
2. an append-only audit trail under independent auditor oversight;
3. lawful erasure reconciled with that trail;
4. policy-based evaluation.

Within that lifecycle we embed a language-model relevance judge whose reliability we measure (Chapter 6) rather than assume. A policy-as-code engine answers "is this action permitted", a binary that follows deterministically from rules and request attributes. The judge answers a question policy engines cannot express: "is this application relevant to this project", which has no rule-form answer and must be decided by reading natural-language content. We answer this second question empirically and hold its answer to an ablation-grounded standard of reliability, which distinguishes BraneHub from both the execution platforms and the policy engines it composes.

9.5 Positioning

Table 9.1 summarises the comparison along the six dimensions of Sections 9.1 to 9.4: federated execution, governance lifecycle, compliance overlay, the relevance-versus-permission question, the presence of a language-model relevance judge, and the rigour of its evaluation. In the judge-evaluation column, prior LLM-as-judge work validates by agreement, whereas we add a body-withholding ablation as a precondition for trusting that agreement. No prior system combines a federated-analysis governance lifecycle, a compliance overlay,

and a language-model relevance judge subjected to an ablation-grounded evaluation. We contribute that combination, not any single column.

Table 9.1: Positioning of BraneHub against the federated-execution, policy-as-code, and LLM-as-judge lineages, along the six dimensions of Sections 9.1 to 9.4.

System	Federated execution	Governance lifecycle	Compliance overlay	Relevance question	Language-model relevance judge	Rigorous judge evaluation
DataSHIELD (1)	✓	partial	partial	—	—	—
Personal Health Train (23)	✓	partial	partial	—	—	—
Vantage6 (3)	✓	partial	partial	—	—	—
OpenSAFELY (24)	✓	partial	partial	—	—	—
Brane (2)	✓	—	—	—	—	—
OPA (policy-as-code) (8)	—	—	✓	—	—	—
LLM-as-judge work (9, 10, 21)	—	—	—	partial	✓	partial
BraneHub (this work)	✓	✓	✓	✓	✓	✓

"Partial" marks a capability present only in limited or implicit form; "—" marks its absence. OPA decides whether an action is permitted rather than whether content is relevant, and prior judge work assesses relevance or quality outside a governance setting, validated by label agreement alone without an ablation establishing body-dependence.

10

Conclusion

We pursued two goals: (1) build a platform that governs the lifecycle of a federated data project with compliance built in, and (2) determine whether the language-model relevance judge in the platform's compliance layer can be trusted. We built the platform (Chapters 4–5) and measured the judge (Chapters 6–8).

10.1 Answering the Research Questions

We answer the two design questions architecturally.

DQ1. We reconcile erasure with an append-only record not by deleting rows but by pseudonymising the personal data in retained rows in place (Section 5.3). The audit trail keeps its referential integrity, the subject can no longer be identified from the record on its own — a pseudonymised accountability record retained under Article 17(3) — and the erasure cascades into the derived frozen artefacts (judge rationales, evaluation-log snapshots, and cycle snapshots) that no foreign-key rule would reach.

DQ2. A single engine keeps a running cycle consistent with changing membership (Section 4.3). On every membership change it recomputes the effective participant set and, under a row lock, takes one of five actions: (1) leave the run untouched, (2) warn the owner, (3) regenerate the pending analysis, (4) abandon a cycle whose participants have all departed, or (5) stand down and record the race if the cycle reached a terminal state concurrently.

We answer the four empirical research questions against the evaluation tables.

RQ1. A general-purpose language model judges compliance text far more accurately than non-learning baselines. On the balanced privacy-policy task all four models score between 78.0% and 81.6% accuracy against a 50% chance baseline; on contract NLI they

reach 91.5% to 94.4% (Cohen's κ of 0.56–0.63 on privacy and 0.83–0.89 on contract), every bootstrap interval clear of the baseline (Table 6.3).

RQ2. This competence is body-decisive. As the privacy clause is withheld in stages, Claude Sonnet's accuracy falls monotonically from 78.0% to 67.4% to 50.0% (exactly chance), and GPT-5.5 likewise collapses to chance on the body-removed condition (Table 6.6). The judge reads the content rather than exploiting surface cues.

RQ3. Accuracy and cost are not monotonically related. On the matched privacy condition the four models span just four points (78–82%) while their prices differ several-fold: the free Gemma-4-31B sits on the accuracy frontier at no marginal cost, and Haiku returns the most accuracy per dollar, roughly three times Sonnet's (Table 6.7). Chain-of-thought prompting is model-dependent: it significantly helps Sonnet, significantly hurts Haiku, and does not move GPT-5.5 (Table 6.5). Neither the most expensive model nor a single prompting strategy can be assumed best.

RQ4. The model's errors are driven above all by negation, the feature this domain depends on, rather than by clause or description length: negation carries the dominant standardised logistic-regression coefficient at -0.83 and is protective (Table 6.8).

A structured-data pilot adds a fifth result. The judge discriminates well, reaching an AUC of 0.921 on full résumés against 0.788 for lexical overlap, but is miscalibrated, setting its absolute bar for "suitable" markedly stricter than human annotators do (Section 6.2, Table 6.2).

10.2 Contributions

We make three contributions.

1. **A governance platform for cross-institution federated data projects.** It manages onboarding and admission, a human-approved analysis cycle, and long-term audit retention, and integrates with a federated executor through a snapshot-based, idempotent, retrying contract, exercised in this work against a containerised simulator of that executor rather than the production integrator (Section 8.2).

2. **A compliance framework integrated into that lifecycle**, with four components: append-only audit under read-only auditor oversight, versioned policy-as-code evaluation, lawful erasure by in-place pseudonymisation, and the language-model relevance judge.

3. **A rigorous evaluation of the judge against external human-labelled benchmarks**, yielding two transferable methodological findings: (1) a judge benchmark must be shown body-decisive by ablation before its scores carry meaning; (2) a judge's discrimination and its calibration are independent properties, either of which can fail alone.

10.3 Future Work

Production-text validity. We measured the judge on curated benchmark text, but in production it reads descriptions assembled from applicants' free-form answers (Chapter 8). The accuracy that survives this distribution shift should be measured on real onboarding data once enough has accrued to label.

Calibration before automation. Turning the judge's suggestion into an automatic threshold requires first calibrating that threshold against human decisions rather than taking the model's raw verdict.

Broader coverage. We could extend the evaluation beyond the two English legal-text domains to other domains and languages, and repeat it under sampling rather than a single deterministic run to quantify the model's own variability.

Platform-corpus evaluation. The production-text validity above can be tested directly by building a BraneHub onboarding corpus that pairs project objectives with applicants' questionnaire answers and human relevance labels, then measuring whether the capability seen on external benchmarks transfers to the platform's own input.

System and usability evaluation. Beyond the indicative measurements of Section 6.8, the platform itself warrants a sustained high-load stress test of the deployment and a usability study of the auditor and owner interfaces, neither of which we address here.

Demonstrated validation of the design guarantees. We answer DQ1 and DQ2 by construction and by the test suite of Chapter 4, not by a reported experiment. The natural next step is to demonstrate them under load: driving the membership-change engine with a scripted storm of concurrent admissions, withdrawals, removals, and erasures against the simulator of Section 4.4, and confirming that the five reconciliation actions and the in-place erasure preserve their invariants. This would turn the architectural argument of Section 10.1 into measured evidence.

Fairness and adversarial robustness. Two evaluations named as limitations in Chapter 8 are prerequisites for any move beyond an advisory judge. The first is a fairness audit: measuring the judge's error rates across applicant subgroups on group-annotated

data, since calibration that holds on average can still mask the systematic exclusion of a protected group. The second is adversarial robustness: testing whether applicant-supplied text can steer the verdict through prompt injection, and hardening the judge's input handling if it can. Neither is addressed here.

None of these directions alters the present recommendation that the judge remain advisory.

10.4 Closing

The judge at the centre of BraneHub reads what it is given and judges it accurately and cheaply: it collapses to chance when the body is withheld, runs at zero marginal cost on the free model and returns the most accuracy per dollar on Haiku (Table 6.7), yet it is bounded, scoring only 78.0–81.6% on the harder privacy task, sensitive to negation, and miscalibrated in its absolute standard. For this reason we surface its verdict as advice to a person rather than as an automatic gate, placing the judge where a human can weigh it alongside the evidence that says how far it should be trusted.

References

[1] Amadou Gaye, Yannick Marcon, Julia Isaeva, Philippe LaFlamme, Andrew Turner, Elinor M Jones, Joel Minion, Andrew W Boyd, Christopher J Newby, Marja-Liisa Nuotio, et al. **DataSHIELD: taking the analysis to the data, not the data to the analysis**. *International Journal of Epidemiology*, **43**(6):1929–1944, 2014. doi:10.1093/ije/dyu188. 1, 5, 9, 73, 75

[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 e-Science (eScience)*, pages 277–282. IEEE, 2021. doi:10.1109/eScience51609.2021.00056. 1, 5, 13, 19, 73, 75

[3] Arturo Moncada-Torres, Frank Martin, Melle Sieswerda, Johan van Soest, and Gijs Geleijnse. **VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange**. In *AMIA Annual Symposium Proceedings*, **2020**, pages 870–877, 2020. 1, 73, 75

[4] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. **Communication-Efficient Learning of Deep Networks from Decentralized Data**. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1273–1282, 2017. 5

[5] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, et al. **The FAIR Guiding Principles for scientific data management and stewardship**. *Scientific Data*, **3**(1):160018, 2016. doi:10.1038/sdata.2016.18. 6

[6] Luiz Olavo Bonino da Silva Santos, Kees Burger, Rajaram Kaliyaperumal, and Mark D. Wilkinson. **FAIR Data Point: A FAIR-Oriented Approach**

for Metadata Publication. *Data Intelligence*, **5**(1):163–183, 2023. doi:10.1162/dint_a_00160. 6, 74

[7] European Parliament and Council of the European Union. **Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)**. Official Journal of the European Union, L 119, pp. 1–88, 2016. 6, 7, 38

[8] Open Policy Agent Authors (Cloud Native Computing Foundation). **Open Policy Agent: Policy-based control for cloud native environments**. https://www.openpolicyagent.org/, 2024. Graduated CNCF project since 29 January 2021, https://www.cncf.io/projects/open-policy-agent-opa/. Accessed 2026. 7, 14, 15, 36, 74, 75

[9] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. **Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena**. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. 8, 71, 75

[10] Guglielmo Faggioli, Laura Dietz, Charles L. A. Clarke, Gianluca Demartini, Matthias Hagen, Claudia Hauff, Noriko Kando, Evangelos Kanoulas, Martin Potthast, Benno Stein, and Henning Wachsmuth. **Perspectives on Large Language Models for Relevance Judgment**. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval (ICTIR)*, 2023. doi:10.1145/3578337.3605136. 8, 71, 75

[11] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. **Automatic Certificate Management Environment (ACME)**. RFC 8555, Internet Engineering Task Force (IETF), 2019. 30

[12] Article 29 Data Protection Working Party. **Opinion 05/2014 on Anonymisation Techniques (WP216)**. European Commission, WP216, 2014. Adopted 10 April 2014. 40

[13] Qiaosheng Chen, Weiqing Luo, Zixian Huang, Tengteng Lin, Xiaxia Wang, Ahmet Soylu, Basil Ell, Baifan Zhou, Evgeny Kharlamov, and Gong Cheng. **ACORDAR 2.0: A Test Collection for Ad Hoc Dataset Retrieval with Densely Pooled Datasets and Question-Style Queries**. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 303–312, 2024. doi:10.1145/3626772.3657866. 44

[14] Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel R. Bowman, and Noah A. Smith. **Annotation Artifacts in Natural Language Inference Data**. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 107–112, 2018. 45, 72

[15] Adam Poliak, Jason Naradowsky, Aparajita Haldar, Rachel Rudinger, and Benjamin Van Durme. **Hypothesis Only Baselines in Natural Language Inference**. In *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics (*SEM)*, pages 180–191, 2018. 45, 72

[16] Neel Guha, Julian Nyarko, Daniel E. Ho, Christopher Ré, Adam Chilton, Aditya Narayana, Alex Chohlas-Wood, Austin Peters, Brandon Waldon, Daniel N. Rockmore, et al. **LegalBench: A Collaboratively Built Benchmark for Measuring Legal Reasoning in Large Language Models**. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023. 45, 46, 51, 72

[17] Luis Gascó, Hermenegildo Fabregat, Laura García-Sardíña, Paula Estrella, Warre Veys, Casimiro Pío Carrino, Matthias De Lange, Daniel Deniz Cerpa, Álvaro Rodrigo, Jens-Joris Decorte, and Rabih Zbib. **Overview of the TalentCLEF 2026: Skill and Job Title Intelligence for Human Capital Management**, 2026. Overview of the TalentCLEF lab at CLEF 2026; Task A is Contextualized Job-Person Matching. arXiv:2606.31692. 45, 47, 73

[18] Article 29 Data Protection Working Party. **Guidelines on Automated individual decision-making and Profiling for the purposes of Regulation 2016/679 (WP251rev.01)**. European Commission, WP251rev.01; endorsed by the European Data Protection Board, 2018. Last revised and adopted 6 February 2018. 63

[19] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-29044-2. 65

[20] Sebastian Zimmeck, Peter Story, Daniel Smullen, Abhilasha Ravichander, Ziqi Wang, Joel R. Reidenberg, N. Cameron Russell, and Norman Sadeh. **MAPS: Scaling Privacy Compliance Analysis to a Million Apps**. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, **2019**(3):66–86, 2019. doi:10.2478/popets-2019-0037. 65, 72

[21] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. **Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents**. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. 71, 75

[22] Yuta Koreeda and Christopher D. Manning. **ContractNLI: A Dataset for Document-level Natural Language Inference for Contracts**. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 1907–1919, 2021. doi:10.18653/v1/2021.findings-emnlp.164. 72

[23] Oya Beyan, Ananya Choudhury, Johan van Soest, Oliver Kohlbacher, Lukas Zimmermann, Holger Stenzhorn, Md. Rezaul Karim, Michel Dumontier, Stefan Decker, Luiz Olavo Bonino da Silva Santos, and Andre Dekker. **Distributed Analytics on Sensitive Medical Data: The Personal Health Train**. *Data Intelligence*, **2**(1–2):96–107, 2020. doi:10.1162/dint_a_00032. 73, 75

[24] Elizabeth J. Williamson, Alex J. Walker, Krishnan Bhaskaran, Seb Bacon, et al. **Factors associated with COVID-19-related death using OpenSAFELY**. *Nature*, **584**(7821):430–436, 2020. doi:10.1038/s41586-020-2521-4. 73, 75

[25] Jonathan Lawson et al. **The Data Use Ontology to streamline responsible access to human biomedical datasets**. *Cell Genomics*, **1**(2):100028, 2021. doi:10.1016/j.xgen.2021.100028. 74

Appendix A

Engineering Rigour and Quality Assurance

A.1 A Repeatable Quality-Assurance Process

We did not implement BraneHub in a single pass followed by incidental bug-fixing. We applied a five-step quality-assurance loop repeatedly, in named waves, until a round found no defect above the Critical/High threshold. We labelled each wave in its commit subjects—EX1–EX10, W1–W9, FX1–FX14, B1–B10, the FRESH-1–11 series, and the P4 audit rounds—so the procedure is reconstructable from the version-control history rather than merely asserted. Across roughly 800 commits the same loop recurs under these identifiers.

Our loop has five steps.

1. **Audit.** We conduct a multi-domain audit and write it to a tracked audit document rather than to scattered notes, for example the commit "docs(audit): industrial reverification of Phase-2 — evidence-driven".

2. **Triage.** We sort the findings into five severity tiers (Critical, High, Medium, Minor, Nit) and record each tier's disposition, so the close-out commits read "docs(audit): close out Medium tier — all 15 Medium items fixed in W8" and "docs(audit): close out Minor/Nit tier — 18/24 fixed in W9, 6 documented-as-deferred".

3. **Spec then plan.** Before changing any code we write a specification and then a plan; the corpus contains on the order of 100 commits under the `docs(spec)` and `docs(plan)` prefixes, so a fix's intended behaviour is fixed in prose before we attempt the implementation.

4. **Repair.** We repair each defect in a single commit paired with a regression test. The one-defect-per-commit discipline is visible in the monotonic wave numbering, such as the FX13 series (FX13a through FX13h) and the B series (B1 through B10).

5. **Close.** We close the wave with a stage-two quality review and a full-suite run.

We gate each fix on a regression test that demonstrably exercises the bug, not on the author's belief that the bug is repaired. We report the honest failure mode of that commitment here rather than concealing it. Our automated test-generation step occasionally produced a "false-oracle" test: one that passed whether or not the defect was present, and so certified nothing. We caught these in the stage-two review and rewrote them so that the test fails before the fix and passes after—the distinction the history marks as a "true-oracle" test, as in "test(onboarding): B7 follow-up — rewrite tests to actually exercise the bug path" and "fix(brane): close the BUG-2 C1a race for real — populate_existing on the locked read (+ true-oracle test)". We treated an inadequate verification step as a defect to be fixed in its own right.

The suite now collects 2,069 test cases, a figure verified by a collection run on the audited tree and reported in Section 4.4, up from the "1357 passed 0 failed" that an early audit-document entry records (commit "docs(audit): append Phase-5 outcome — 12/14 Critical+High fixed, 1357 passed 0 failed"). The growth is almost entirely additional regression tests, since the discipline above adds at least one test per defect repaired.

We generate the demonstration database deterministically rather than hand-curating a fixture that drifts as the schema evolves. We seed the data with a deterministic program of roughly 6,300 lines that is byte-stable across runs: every timestamp is offset from a single pinned reference instant rather than the wall clock, and every identifier is drawn from a seeded pseudo-random source, so two runs on different days produce an identical database. The program does not only insert rows; it enforces 55 named correctness invariants, which a companion checker re-asserts against the seeded database, exiting non-zero should any fail. The 55 invariants encode the same properties the production code must uphold:

1. auto-increment identifiers are monotonic with creation time;

2. no seeded timestamp post-dates the pinned anchor;

3. every onboarding and cycle record occupies a legal state of its respective machine; and

4. the generated projects and cycles are consistent with those state machines.

The resulting corpus is realistic in shape—on the order of 100 projects with their associated onboarding requests, participants, analysis cycles, and policy versions—and we derive its audit trail from those real business rows rather than fabricating it independently, so the auditor-facing history that a demonstration shows corresponds to the events that produced it. The exact database against which we demonstrated the platform regenerates from the repository, and every one of the 55 invariants is checked against it.

A.2 A Taxonomy of Defect Classes

The waves did not find an undifferentiated set of bugs. They found a small number of recurring defect classes, each with a characteristic failure mode and, in the better cases, a systemic fix that generalised across the codebase rather than a point patch. We evidence each class below by commit subjects verified against the history.

Concurrency and lost-update races. The recurring failure was that two requests acting on the same cycle or participant could interleave so that one overwrote the other's committed state. We converged on a single pattern: a `SELECT ... FOR UPDATE` on the contended row followed by a `populate_existing` read, so that the locked row rather than a stale identity-map copy drives the decision. The pattern is visible in "fix(race): FX13a — FOR UPDATE in approve/reject/abort brane routes" and "fix(brane): close the BUG-2 C1a race for real — populate_existing on the locked read". We then applied the same pattern across the onboarding, project, and integration routes in subsequent waves, so the fix generalised rather than chasing one route at a time.

ORM identity-map pitfalls. On SQLAlchemy 2.0, the object-relational identity map and the automatic `onupdate` timestamping interacted to write an unintended `updated_at`. We made the intended persistence explicit, as in "fix(concurrency): FRESH-6 F4-A2 — flag_modified updated_at to suppress onupdate", which forces the framework's hand rather than relying on its inferred behaviour.

GDPR Article 17 residual personal data. BraneHub's compliance value rests on a permanent audit trail, which conflicts with the right to erasure (Section 5.3). The recurring failure was that anonymisation cleared the primary field but left personal data residual in a snapshot, an evaluation log, or a configuration record. We traced and cut every residual path, in "fix(gdpr): close Art.17 residual PII ... (BUG-10/11/12, CRITICAL)" and "fix(gdpr): FX13b — cascade anonymize_user into IntegrationCycle.snapshot_json"; the latter makes the cascade follow the data into the immutable snapshot rather than stopping at the live record.

A. ENGINEERING RIGOUR AND QUALITY ASSURANCE

Audit-trail completeness. Because the audit trail is the platform's evidentiary product, we treated any state change that mutated data without writing an audit row as a defect in itself. Two commits exemplify the class: "fix(audit-gaps): FX13c — close 4 silent state-mutation paths" and "fix(audit): B10 — completeness for build_failed / callback drops / webhook fails", the latter extending coverage to the failure paths that are easiest to leave unaudited.

Database invariants and finite-state validation. Cycle and onboarding state had originally been guarded by checks scattered across the routes, which is where divergence accumulates. We replaced the scattered guards with one declarative transition table validated by an equivalence oracle—"refactor(brane): single declarative FSM transition table ... + equivalence oracle"—and added a database-level `CHECK` constraint and terminal-transition validation as a backstop, "fix(invariants): onboarding.status CHECK + mark_cycle_terminal validation". The equivalence oracle confirms that the consolidated table accepts exactly the transitions the old scattered guards accepted.

Webhook idempotency and commit/rollback discipline. Inbound webhooks could commit partial state on an error path, and we mis-classified certain malformed responses. We wrapped inbound processing so that a failure rolls back cleanly—"fix(integration): B6 — wrap inbound webhook commits in try/except rollback"—and reclassified an unparseable success as a transport error rather than a logical one, "fix(webhooks): B9 — demote 2xx + unparseable JSON to transport_error".

Cross-cutting hardening. A residual group of defects did not form a single behavioural family but shared the property of being fail-safe omissions. Each closes a path reachable only under an unusual but reachable condition—precisely the kind of defect an incidental bug list tends to miss.

1. We degrade client-side rendering to `textContent` when its sanitiser is absent ("fix(js): _setHtmlSafe degrades to textContent on missing DOMPurify").

2. We enforce snapshot immutability by copying rather than trusting callers ("fix(integration): snapshot freeze deepcopy + isinstance defense").

3. We make migrations portable by adopting an explicit naming convention and renaming the legacy auto-generated constraint names ("fix(schema): SQLAlchemy naming_convention + rename legacy auto names").

4. We enforce least privilege where an auditor role could otherwise mutate state (“fix(security): restrict OPA-eval POST routes to OWNER_OR_ADMIN — auditor (read-only) must not mutate”).

Content-security-policy hardening. This class is a permissive default rather than a wrong behaviour: the platform’s `script-src` directive still admitted `'unsafe-inline'`, which permits execution of injected script. Closing it required multiple steps, because the directive cannot be tightened while any inline script remains.

1. We first vendored locally the third-party JavaScript that had been pulled from a content-delivery network, so that an off-origin `script-src` could be excluded.

2. We then migrated every inline `on*=` handler and every inline executable `<script>` to a delegated external handler.

3. We externalised the configuration the markup had previously interpolated into inline script (chart definitions and editor seed values among them) into inert `type="application/json"` data islands that the modules read at parse time.

Only once the inline surface was empty did we set the directive to `script-src 'self'`. A structural regression test, `tests/test_p2b_csp_hardening.py`, then holds the end state in place: its gate fails if any inline handler or inline executable `<script>` reappears anywhere in the templates, so a later contributor cannot silently reintroduce the pattern we removed.

A.3 Verification and Severity Accounting

Table A.1 reports per-wave severity counts drawn directly from close-out commit subjects in the tracked audit document rather than reconstructed. Where a precise count is not recorded, the row is described qualitatively rather than supplied with an invented number.

Two features of this accounting matter more than any single figure.

Deferral was explicit. We documented the two Phase-5 items not fixed and the six Minor/Nit items closed out in W9 as deferred, so the residual risk was acknowledged and bounded rather than forgotten.

Suites were green before close. We ran the post-wave suite, including the new regression tests, to a clean state before declaring any wave closed, so “fixed” is backed by a green suite rather than by an assertion that the fix was applied.

Table A.1: Per-wave severity accounting drawn from tracked audit-document close-out commit subjects.

Wave	Critical/High found	Fixed	Deferred	Post-wave suite
Phase-5 audit	14 Critical+High	12	2 (documented)	1,357 passed, 0 failed
W8 (Medium tier)	15 Medium	15	0	full suite green
W9 (Minor/Nit tier)	24 Minor/Nit	18	6 (documented)	full suite green
FRESH-6 (P4)	4 Critical + 3 Important	7	0	full suite green
FRESH-7 (P4)	5 Critical + 6 Important	11	0	full suite green
Current state	—	—	—	2,069 collected (Section 4.4)

A.4 How This Differs from Prior Practice

Comparable systems dissertations typically treat quality assurance as a short subsection listing a few "implementation issues" or "experiment issues" encountered along the way. Such a list records incidents: it tells the reader that some problems arose and were dealt with, but it gives no method by which the problems were found and no assurance that problems of the same kind were sought systematically elsewhere. A reader cannot tell whether the listed issues are the defects that happened to be noticed or the defects that exist.

Our method differs in four respects.

1. We surface defects by a repeatable, severity-tiered audit run in multiple rounds rather than encountered by chance.
2. We root-cause and repair each defect under a written specification and plan rather than patching it in place.
3. We lock each fix with a regression test constructed to fail before the fix and pass after, and we catch and rewrite the inadequate tests in review.
4. We re-verify each wave against the full suite before closing it, and record deferrals explicitly.

The product is therefore two things an incident list is not. The first is a reproducible methodology, which another engineer could re-run against the same codebase. The second is a defect taxonomy in which the recurring failure modes of Section A.2 are named and each closed by a fix that generalised across the codebase. That this convergence happened, and that it converged on patterns rather than on a longer list of patches, is the substantive claim this appendix makes for the engineering rigour of the work.