A Framework for Defining and Enforcing Governance Policies in Human-Agent Software Development

ADEM AIT, University of Luxembourg, Luxembourg
GWENDAL JOUNEAUX, Luxembourg Institute of Science and Technology, Luxembourg
JAVIER LUIS CÁNOVAS IZQUIERDO, Universitat Oberta de Catalunya, Spain
JORDI CABOT, Luxembourg Institute of Science and Technology, University of Luxembourg, Luxembourg

Software development increasingly involves heterogeneous stakeholders, including diverse human contributors and AI-powered agents. This amplifies governance challenges around transparency, participation, and accountability, particularly in Open-Source Software (OSS) projects where policies are usually implicit. Building on our prior vision paper, we present a framework for defining and enforcing governance policies via a Domain-Specific Language (DSL) and an executable decision engine. The DSL captures scopes, roles and individuals (with weighted participation and profiles), policy composition and deadlines, and platform-aware conditions. The engine operationalizes policies over collaboration events and records votes, rationales, and timestamps for traceability. We extend the core language with bindings for GitHub and GitLab and provide a form-based editor to ease authoring. We evaluate expressivity by formalizing existing `GOVERNANCE.md` files from a curated dataset and demonstrate enactment by reproducing three governance policies from different GitHub repositories.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Open source model*; **Documentation**; *Maintaining software.*

Additional Key Words and Phrases: Governance, Domain-Specific Language, Diversity, Human-Agent Collaboration

ACM Reference Format:
Adem Ait, Gwendal Jouneaux, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2026. A Framework for Defining and Enforcing Governance Policies in Human-Agent Software Development. 1, 1 (October 2026), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large-scale software development is a collaborative activity, where multiple participants contribute to the evolution of a codebase. The collaborative nature of software development, especially in Open-Source Software (OSS), requires proper governance to ensure effective decision-making and accountability. Governance in software projects defines the roles, responsibilities, and processes for managing contributions, resolving conflicts, and maintaining the integrity of software projects [6]. Governance policies formalize decision rights and procedures so that responsibilities are transparent to all participants. Yet many OSS projects either lack such policies or document them only superficially, obfuscating how

Authors' addresses: Adem Ait, adem.ait@uni.lu, University of Luxembourg, Esch-sur-Alzette, Luxembourg; Gwendal Jouneaux, gwendal.jouneaux@list.lu, Luxembourg Institute of Science and Technology, Esch-sur-Alzette, Luxembourg; Javier Luis Cánovas Izquierdo, jcanovasi@uoc.edu, Universitat Oberta de Catalunya, Barcelona, Spain; Jordi Cabot, jordi.cabot@list.lu, Luxembourg Institute of Science and Technology, University of Luxembourg, Esch-sur-Alzette, Luxembourg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

decisions are taken [6]. Furthermore, even when documented, policies are often scattered across project resources, making it hard for contributors to discover and follow the project's rules and procedures [5].

At the same time, teams increasingly rely on automation (a.k.a. bots) [24, 28] and AI-powered agents that participate in development workflows, introducing new governance challenges around transparency, accountability, and human-agent collaboration [22]. Thanks to advancements in AI, agents, powered by Large Language Models (LLMs), are becoming more capable of performing complex tasks, collaborating, and even participating in decision-making [29]. Moreover, human diversity is increasingly recognized as a key factor for effective governance, as diverse teams bring different perspectives and experiences that can lead to better decision-making [31], while also benefiting end-users [15]. Although human-agent collaboration and greater human diversity can drive innovation and productivity, they simultaneously expose a key challenge: the absence of governance mechanisms that explicitly and comprehensively address such heterogeneous participation.

This paper addresses this gap by extending our prior vision paper [1] with a comprehensive framework for defining and enforcing governance policies in software projects, with a particular focus on supporting diversity among participants, including both human contributors and AI-powered agents. The contributions of this paper are:

- A survey to understand the needs and challenges of governance in software projects, and a discussion of the corresponding results.
- A Domain-Specific Language (DSL) for defining governance policies that capture roles and individuals, weighted participation, profiling, and policy constraints.
- Platform-specific extensions to the DSL syntax for GitHub and GitLab that bind core concepts to platform entities (e.g., repositories, pull requests), enabling in-repository enforcement.
- An alternative form to easily define governance policies using our DSL via a form-based editor.
- A decision engine that evaluates policies over development events in GitHub and GitLab, applies decisions across phases, and records votes, rationales, and timestamps for traceability.
- A validation of the DSL expressivity through analysis of real governance files from OSS projects, and an evaluation of the decision engine's applicability in real-world scenarios.

In our previous above-mentioned work [1], we presented the core concepts of a governance DSL and decision engine to support human-agent collaboration. Of these contributions, the survey, the platform-specific extension, the form-based editor and the validation are novel, while the DSL syntax has been enhanced and the decision engine has been extended to also support GitLab in addition to GitHub.

The rest of the paper is structured as follows. Section 2 motivates our work and discusses the survey results. Section 3 reviews the state of the art, while Section 4 describes our approach and presents a running example. Section 5 and Section 6 present the design of the DSL and extend it to provide support for code-hosting platforms, respectively. Section 7 details the implementation of the decision engine. Section 8 evaluates the expressivity of our DSL and the applicability of the decision engine. Finally, Section 9 concludes the paper and discusses future work.

2 TOWARDS EXPLICIT GOVERNANCE IN SOFTWARE PROJECTS

2.1 Challenges of OSS Governance

Due to the open nature of OSS development, OSS projects often face challenges in decision-making and management, such as the participation inequality problem: everybody uses OSS, but few contribute back to its development [12, 19]. The lack of key governance information deters potential contributors from joining the project, as they are unsure

about the project's decision-making processes and how to effectively contribute [6]. Documentation is therefore key for OSS projects to communicate about the goals, organization, and processes of the project. Gaughan et al. [13] findings suggest that more substantive project documentation may be associated with subsequent project growth. Recently, formal governance systems have been increasingly adopted by communities [8]. The `GOVERNANCE.md` file[1] aims to provide a standardized format for documenting governance in OSS projects. However, despite the growing interest in governance, it is still not a common practice to report it. Yan et al. [30] analyzed 710 `GOVERNANCE.md` files from GitHub repositories, which also released as a dataset, and they conclude that while adoption is growing, many projects still lack comprehensive governance documentation. Moreover, those that report it still lack key governance information [1].

In parallel, governance frameworks must account for the fact that teams are more diverse (e.g., gender or ethnicity) [3, 23]. Proper representation of the team diversity could correlate with innovation and productivity improvement [7, 25, 31]. Beyond human diversity, bots, and agents contribute to development workflows [28], and, advances in LLMs, enable their participation in complex decision-making [16, 29]. This profound shift introduces an unprecedented dimension of diversity, presenting novel and urgent conceptual challenges for governance that existing frameworks are missing.

2.2 Practitioner Expectations for Governance

To obtain initial insights into governance-related challenges and expectations in software projects, we carried out a preliminary survey. The survey comprised multiple-choice questions plus brief free-text fields, and was structured into three sections: (i) project dynamics and challenges, (ii) AI in project processes, and (iii) language and automation for governance. The survey was distributed among software practitioners, including industry professionals, academics, and open-source contributors. To ensure broad representation, we contacted participants through several channels: we reached out to MOSAICO[2] project partners, shared the survey openly on professional networks like LinkedIn, and targeted OSS maintainers of projects containing a `GOVERNANCE.md` file. For the latter, we leveraged an extended version of the dataset provided by Yan et al. [30], which collects `GOVERNANCE.md` files from GitHub repositories (the extension process is described in Section 8.1.1). We contacted those repositories with publicly available contact information, resulting in 425 emails sent.

We received 32 responses: 10 from industry, 4 from academia, 8 from the open-source community, 5 from freelancers, and 5 did not disclose their affiliation. Both small (1-50 members or employees) and large (250+) communities and organizations were represented. Reported sectors of activity span R&D of new technologies, OSS development, client-specific consulting, and development of commercial software products and services. In the following, we report the main findings of the survey for each section, and then we discuss the results. When reporting the results, we provide the percentage of respondents that selected each answer.

2.2.1 Project dynamics and challenges. This section explores the day-to-day realities of software project coordination. We aim to understand the most common challenges as well as their current practices for making decisions and documenting governance practices.

We first asked respondents about the most common challenging situations or recurring complex aspects in their software development life-cycle. The most frequently cited challenges are keeping documentation up-to-date (68.75%), followed by managing conflicting priorities and aligning on technical direction (56.25% for both). Because maintaining clear guidelines and documentation is a foundational aspect of project governance, written policies are most frequently

[1] https://governance.md/
[2] A European project on enabling collaboration among communities of AI agents, see https://mosaico-project.eu/

used for coordination-heavy or maintenance-heavy activities. These include release management (59.38%), feature prioritization (56.25%), documentation standards (56.25%), and issue tracking (53.13%).

Then, we investigated how decisions are made within these projects. We found that decisions are typically driven by leaders (75%) and senior developers (53.13%), while AI agents (21.88%) and clients (12.5%) rarely participate. One respondent added that what is most important to decision-making is "respecting the input of expert developers and end users". Formal voting is uncommon (18.75%); instead, informal consensus (65.63%), leader-driven (50%), or expert-opinion (37.5%) processes prevail. Nevertheless, participants value transparency (65.63%), accountability (53.13%), and managing conflicts (50%), highlighting a gap between desired governance qualities and current informal practices.

Finally, we asked about the actual formalization of these governance practices. Among those maintaining OSS projects (78.13%), 20% of respondents reported that they do not document governance practices in their projects, while 8% did not answer. Of those that report documentation (72%), 66.67% provide a structured governance document following `GOVERNANCE.md` common practices , while 27.78% document them informally via notes or internal documents and 5.56% use a formal language.

2.2.2 AI in project processes. In the second section, we aimed to investigate the current adoption and role of AI agents within development workflows. Specifically, we wanted to determine whether practitioners rely on AI primarily for code-level tasks or if these agents are beginning to participate in broader decision-making and project governance.

A total of 26 out of 32 respondents (81.25%) report using AI agents in some part of their process. However, this usage is primarily restricted to code contribution and code analysis (88.46%). Use of agents as participants or leaders in decision-making is a minority (34.62%). Nevertheless, two respondents report that, while not using them for decision-making, they use AI agents for "obtaining the necessary reference opinions" and "summarizing, under rules, what needs decision", respectively.

For the respondents not using AI agents (18.75%), we explored the underlying reasons preventing adoption. The main barrier to adoption is the lack of trust in AI capabilities and potential AI bias (83.33%), while other concerns include high cost and unclear governance when evaluating human-agent and agent-to-agent interactions (66.67%).

2.2.3 Language and automation for governance. The final section examines how practitioners view the formalization of governance through defined languages and automated enforcement. Our goal was to identify both the anticipated benefits and the reservations practitioners expressed.

When asked about adopting a specialized language for governance policies, practitioners perceive several strong benefits. It is expected to improve clarity (65.63%) and consistency (59.38%) in applying rules, while also facilitating onboarding (46.88%) and documentation of governance practices (32.75%). On the other hand, the main drawback perceived is the risk of policies becoming too rigid or bureaucratic (62.5%). Respondents also worry about the overhead of maintaining them (50%), the difficulty in expressing complex governance scenarios (46.88%), and the steep learning curve for maintainers (32.75%).

Regarding the automation required to enact these policies, the perceived benefits align with efficiency: improved compliance and reduction in manual effort (56.25%). However, concerns center heavily on fault tolerance (68.75%), the risk of over-reliance on automation reducing human judgement (59.38%) and security concerns with automated systems making decisions (59.38%).

2.2.4 Discussion of the results. Overall, the survey reveals a clear gap between practitioner expectations and current realities. Despite highly valuing transparency and accountability, 31.25% of respondents still operate under informal

or undocumented governance practices. Additionally, while written policies do exist, they are largely confined to coordination-heavy activities, leaving broader decision-making to informal consensus or leader-driven processes rather than structured frameworks.

Furthermore, as projects evolve, this lack of formal structure clashes with the increasing integration of AI. Although a vast majority of respondents (81.25%) use AI agents in their workflows for code generation and analysis, their integration into project governance and decision-making remains severely limited. This limitation arises primarily from a lack of trust, bias concerns, and most notably, the absence of clear governance mechanisms to regulate human-agent and agent-agent interactions.

Accordingly, practitioners recognize the need for formal solutions. They consider that a specialized policy language and corresponding automation could significantly improve clarity, consistency, and compliance while reducing manual effort. However, this perspective is accompanied by a significant hesitation: the fear of introducing bureaucratic rigidity, steep learning curves, and ongoing maintenance overhead.

We believe these findings motivate our work, as they highlight the need for a governance framework that balances structure with flexibility. Specifically, a governance policy language expressive enough to capture diverse governance scenarios (including the participation and influence of both human and AI agents), a user-friendly interface to minimize overhead and facilitate adoption, and an automation engine to reduce the workload on humans while preserving oversight.

3 STATE OF THE ART

In this section we review how governance models are adopted and what tool support exists aimed to specify and enforce governance policies, with an emphasis on how far existing solutions address the needs outlined in Section 2.

Governance models in software projects draw from organizational and political theory, where various typologies have been established. Traditional models include *representative democracy*, where elected representatives make decisions on behalf of constituents; *direct democracy*, enabling all members to vote on decisions; *meritocracy*, where authority derives from demonstrated competence; *oligarchy*, where power concentrates in a small elite group; and *autocracy* or *benevolent dictatorship*, concentrating decision power in a single leader [10, 17, 20]. OSS projects often adapt and blend these models, such as using meritocratic principles for technical decisions while applying consensus-based approaches for community matters [11, 18]. However, existing frameworks assume relatively homogeneous human participation and do not account for the increasing diversity of contributors, including varying expertise levels, cultural backgrounds, and more recently, AI agents capable of autonomous decision-making. This gap motivates governance systems that can explicitly model heterogeneous participation.

In order to support the automation and supervision of governance in OSS projects, Cánovas Izquierdo and Cabot [5] proposed a DSL to define and enforce governance rules in OSS projects. The language captures project entities and constraints to check or trigger actions, making governance rules machine-checkable and executable. However, the proposal does not consider diversity among participants or the inclusion of AI agents in governance processes.

Complementing this line, further research has explored governance-as-code systems for online communities and Git-based ecosystems. For instance, PolicyKit [32] encodes roles, proposals, and procedural rules to make community governance executable, while also providing Pika [26], a visual programming interface to facilitate the creation of governance policies for non-programmers. Gov4Git[3] stores governance state and protocols in-repo to enable auditable,

[3]https://gov4git.org/

verifiable processes. These works share the goal of making governance explicit, machine-checkable, and enforceable. However, none of these proposals provide support to assess the diversity of the participants involved in the decision-making process, which are central to the challenges identified in Section 2.

In practice, projects operationalize governance through policy-as-code and platform controls: (1) repository rules such as CODEOWNERS[4] and branch protection or required reviews; (2) rules engines like Mergify[5] and Policy-Bot[6] that declaratively specify merge gates and approval constraints; and (3) general policy frameworks (e.g., Open Policy Agent[7]) integrated into CI to enforce organization-wide policies on configuration and process. While effective at enforcing pre-defined checks, these approaches typically do not model diversity of participants or human-agent collaboration, which motivates our proposal.

In our prior vision paper [1], we introduced the foundational concepts of a governance DSL and decision engine focused on human-agent collaboration. This work advances that preliminary proposal by enhancing the DSL syntax to fulfill the needs identified in a preliminary survey and extending the decision engine's capabilities to support multiple platforms (adding GitLab support alongside GitHub). Furthermore, this article presents platform-specific extensions for binding policies to repository entities, a form-based editor for accessibility, and a validation of the framework's expressivity and applicability.

4 OUR APPROACH

In this section, we present our approach aimed at providing a comprehensive framework for defining and enforcing governance policies in software projects, with a particular focus on supporting diversity among participants, including both human contributors and AI-powered agents.

We begin by providing an architectural overview of our approach, followed by a running example that illustrates its application.

4.1 Framework Overview

In the context of governance, a DSL can serve as a formal mechanism for specifying policies, roles, and decision-making procedures, thereby enhancing the transparency and reliability of the interactions. A DSL comprises three main elements [27]: (1) abstract syntax, defining the concepts and relationships of the domain; (2) concrete syntax, defining the notation; and (3) semantics, defining the meaning and execution of constructs. In our approach, we define both abstract and concrete syntaxes for our DSL (Section 5), while the operational semantics (i.e., how policies are enforced at runtime) are addressed separately through a dedicated governance engine (Section 7). Our design is guided by AI governance frameworks and principles [9], which we operationalize in the language through explicit indicators attached to agent participants. In particular, we incorporate constructs to express autonomy level [21], explainability [2], and confidence, enabling policies to weight participation, gate rights, or trigger safeguards based on these attributes. The DSL also supports modeling human diversity (e.g., profiles and roles) so that policies can encode fairness-aware participation and tailored decision procedures. Two distinct dimensions of diversity are therefore supported. First, agent diversity refers to the heterogeneity of AI agents (e.g., confidence, autonomy, explainability) to manage technical risks, such as adapting the voting weight of agents based on their confidence in their contributions. Second, social diversity refers to human attributes (e.g., demographics or background) that can be used to promote inclusion and representation

[4]https://docs.gitlab.com/user/project/codeowners/
[5]https://mergify.com/
[6]https://github.com/palantir/policy-bot
[7]https://www.openpolicyagent.org/

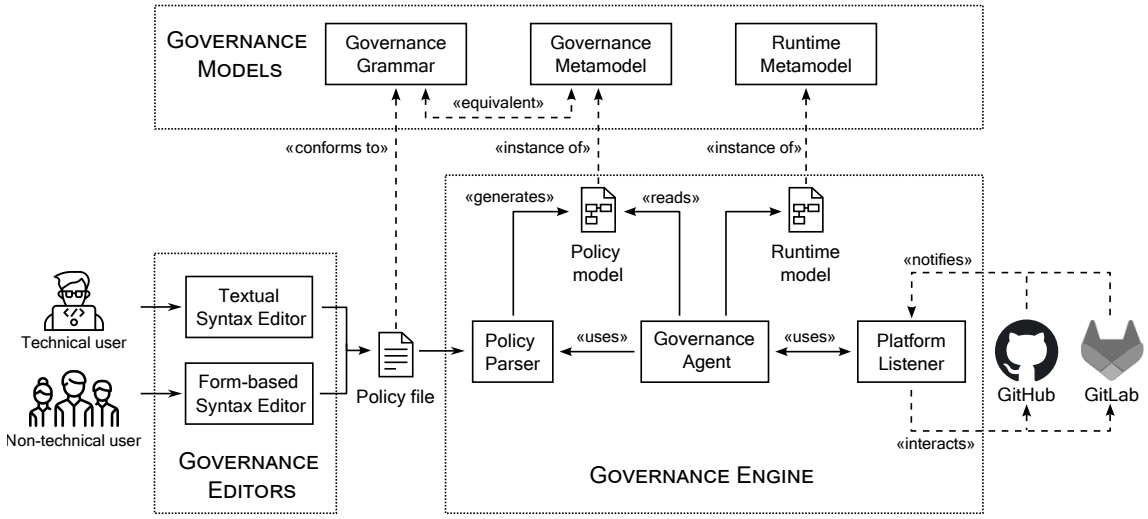


Fig. 1. Overview of our approach for defining and enforcing governance policies in software projects.

in decision-making. For instance, a policy could give higher voting weight to underrepresented groups to promote diversity. While these dimensions are conceptually distinct, one rooted in social fairness and the other in technical reliability, our DSL allows them to be combined in policies that account for both human and agent characteristics.

Our framework leverages metamodeling techniques to define both the structure of governance policies and their runtime execution. Metamodels restrict the structure of valid models (i.e., DSL instances, that are, in our case, concrete policies) and define the relationships between the different elements of the language (i.e., the different concepts that can be used to define a policy and how these concepts can be linked to each other) [4]. Thus, models conforming to the Governance metamodel represent specific sets of governance policies. Figure 1 illustrates the overall architecture, which consists of three interconnected component-based groups:

- **Governance Models**: We define two metamodels: (1) the *Governance metamodel* capturing the abstract syntax of our DSL, and (2) the *Runtime metamodel* representing the runtime state of ongoing decision-making processes. We then define a *Governance grammar* to formally define the concrete syntax of our DSL, representing the metaclasses of the governance metamodel.
- **Governance Editors**: Users can define policies using either a *textual syntax*, or a *form-based editor* for non-technical stakeholders. Both syntaxes produce instances (i.e., *Policy file*) that conform to our governance grammar.
- **Governance Engine**: Driven by a *Governance agent*, the engine receives policy files and uses a *Policy parser* to extract the corresponding *Policy model* (i.e., an instance of the governance metamodel). It then creates and maintains a dynamic *Runtime model* (i.e., an instance of the runtime metamodel) that tracks ongoing decision-making processes. By continuously updating this model based on incoming platform events, the engine evaluates governance rules and automatically enforces decisions when policy conditions are satisfied. The *Governance agent* is part of the decision engine's implementation; it differs from agent participants who are governed by policies. The governance agent is responsible for interpreting policies and enforcing them, while agent participants are entities whose actions and votes are subject to the policies.

4.2 Running Example

To illustrate our approach, we will use a running example that shows the evolution of an OSS project's governance from a flat structure to one that incorporates diversity and human-agent collaboration. This mirrors how OSS documentation often starts simple and matures over time [13].

4.2.1 Early Days: Flat Governance Structure. In the early days of the project, the governance structure is simple and flat. The project consists of few contributors, code reviews must be approved by at least one maintainer. No time limits are enforced.

4.2.2 Growth: Weighted Voting. As the project grows, decision-making is formalized through weighted voting. Contributors have a voice, but maintainers retain stronger influence in code reviews. In this case, maintainers have a voting weight of 2, while contributors have a voting weight of 1. Decisions require a majority of the total voting weight to be approved (e.g., 3 out of 5, for one maintainer and three contributors), and must be made within four days.

4.2.3 Maturity: Diverse Roles. As the project matures, it adopts a more complex governance structure that incorporates diverse roles. It acknowledges diverse roles, skills, and agents. Governance rules assess human diversity and agent contributions, introducing new rules. First, changes to Chinese documentation follow a supermajority majority voting process where contributors with a language expertise in Chinese have a vote weight of 2, while others have a vote weight of 1. It is required at least 60% of the total voting weight to approve changes, and the decision must be made within 7 days. Second, agents can participate in the decision-making, but their votes are calibrated by their confidence in the answers they provide. The confidence is self-assessed by the agent and ranges from 0 to 1. An agent's vote weight is: $w_{\text{agent}} = 0.5 \times \text{confidence}$, where confidence $\in [0, 1]$ (e.g., an agent with a confidence score of 0.8 has a vote weight of 0.4).

5 DEFINING GOVERNANCE POLICIES

In this section, we present (1) the formalization of the abstract syntax that captures scope, participants, policies, and conditions; (2) the definition of a concrete textual and a form-based syntax, and their illustration with the running example; and (3) the syntax implementation. In the next section, we present the extension of this approach to provide support for different platforms.

5.1 Abstract Syntax

The abstract syntax of our DSL is defined via the metamodel shown in Figure 2. Mainly, the metamodel is aimed at representing four main aspects: (1) scope (i.e., where the policy is applied); (2) participants (i.e., who is involved in the policy), (3) policies (i.e., what the policy is about), and (4) conditions (i.e., when it should be applied). Next we describe the main metaclasses of the metamodel according to these four aspects.

Scope. To represent the first aspect, we propose the `Scope` generalization. We envisioned a three-layered structure consisting of the: (1) `Project` metaclass representing the whole system; (2) `Activity` metaclass representing a subset of a project (e.g., development activities); and (3) `Task` metaclass representing the atomic part of a project (e.g., a pull request in the context of OSS projects). This structure allows the extension of the metamodel to include other scopes in the future. Each metaclass represents one level of granularity, forming a hierarchical structure where tasks can belong to activities, and activities can belong to projects. We also defined a `StatusEnum` enumeration to represent the state of a

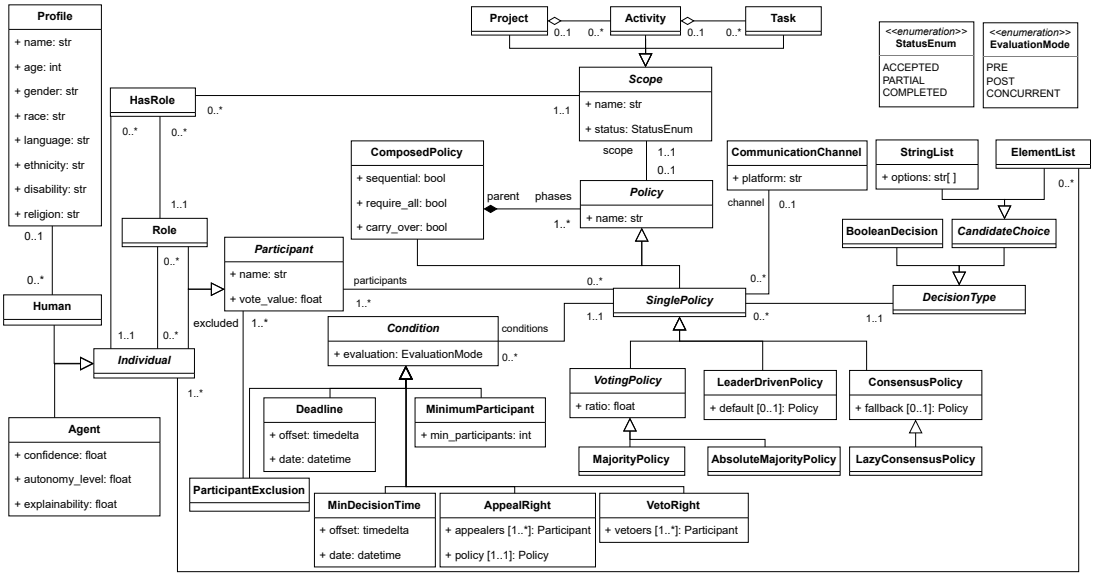


Fig. 2. Governance metamodel. For the sake of clarity, some associations are represented as class attributes.

scope during the decision-making process: *accepted* indicates the scope is ready for policy application; *partial* indicates the decision process is ongoing; and *completed* indicates the decision process has concluded. This attribute is optional but enables the decision engine to track the lifecycle of scopes. Additionally, to explicitly represent the communication channel where the policy is applied (e.g., Discord, GitHub issues, etc.), we can use the `CommunicationChannel` metaclass.

Participants. To represent who is involved in a policy, we define the `Participant` abstract metaclass, which is a generalization of the `Role` and `Individual` metaclasses. The former is intended to represent a group of individuals that share a common role in the project (e.g., contributors, maintainers, etc.), while the latter is intended to represent a specific individual (e.g., an owner). To ensure precise representation, we define an attribute to favor contributions from certain participants in a decision (see `vote_value`). The value of this attribute is a positive decimal, with a default value of 1.0. For instance, a project could set the `vote_value` of maintainers to 2.0 to give maintainers greater influence over contributors, or set the `vote_value` of low-confidence agents to 0.5 to reduce their impact on decisions. We can further identify human individuals by their profile (see `Human` and `Profile` metaclasses), based on their gender, language, or other characteristics. These attributes represent a pragmatic abstraction of profiling dimensions derived from diversity and inclusion research, allowing policies to account for underrepresentation and fairness concerns [14]. These attributes are optional, but they can be used to further characterize human individuals and apply positive discrimination policies if needed (e.g., favoring underrepresented groups). For instance, a policy could give higher voting weight to individuals with a specific profile or exclude certain profiles from participating in a decision. This way we can encode fairness-aware participation in the decision-making process. To represent agents (see `Agent` metaclass) in the project, we extend the `Individual` metaclass. We defined a set of attributes, derived from AI governance frameworks and principles, and the concerns of our survey, to characterize agents according to their capabilities and reliability. These quantitative

indicators (i.e., `autonomy_level`, `explainability`, and `confidence`) can be used in the engine in order to properly assess the participation of agents in the decision-making. Their values are positive decimals, with a range between 0.0 and 1.0. These attributes are used by the decision engine to calculate the voting weight of agents (see Section 7.1) Additionally, we defined a ternary association between the `Individual`, the `Scope` and the `Role` metaclasses (see `HasRole` metaclass) to represent the particular role an individual might have on a specific scope (e.g., a user might be a contributor for development but a reviewer for documentation).

Policies. To define policies, we define the policy hierarchy, which allows defining two kinds of policies: single policies and composed policies (i.e., policies consisting of a set of policies, either single or composed). Depending on the resolution method, the policy can be based on: (1) voting, where each participant selects one (or more) candidates among different possibilities; (2) consensus, which is focused on a cooperative dynamic where all participants work together to make the best possible decision for the group; or (3) leader-driven, where one participant makes the decision.

In our proposal, we generalized the evaluation of voting policies in two kinds: (1) by the number of participants involved in the voting process (see `AbsoluteMajorityPolicy` metaclass) or (2) by the number of participants that voted (see `MajorityPolicy` metaclass). In a nutshell, the decision is made by assessing whether the most voted option surpasses a certain threshold based on the number of potential voters or the number of actual voters. For instance, in a voting policy with 10 participants, with a threshold of 50%, if 4 of them vote for approval and 2 for rejection, the voting policy will be approved if we define a majority policy, but it will not be approved if we follow an absolute majority policy. The threshold can be modified with the `ratio` attribute, whose value is a positive decimal ranging from 0.0 to 1.0, with a default value of 0.5 (i.e., threshold of 50%).

Consensus policies are based on the idea that all participants should agree on a decision before it is made. This is particularly useful in situations where the decision has a significant impact on the project or the community. We also provided support for lazy consensus [8] (see `LazyConsensusPolicy` metaclass), which is a form of consensus where a decision is made if no one objects. It is, as mentioned in the Apache Foundation, one of the most preferred consensus methods. Furthermore, if consensus cannot be reached, we can define an optional fallback policy (see `fallback` attribute) to be applied in that case.

Leader-driven policies represent the case where a participant, playing the role of a leader, makes the decision for the group. Because of this decision-making process relies on a single individual, the leader can define a default policy (see `default` attribute in `LeaderDrivenPolicy`) in case the decision is not made by the deadline.

In order to express what the policy is selecting, we defined a set of decision types (see `DecisionType` hierarchy). A policy must have one decision type. Policies looking for an acceptance or rejection of a proposal can be represented as a `BooleanDecision` (e.g., a pull request review). Policies looking for a selection among a set of options can be represented as a `CandidateChoice` (e.g., who will become the new leader of a community). Thus, the set of options can be represented as a list of individual elements of our metamodel (see `ElementList` metaclass) or as a set of strings (see `StringList` metaclass).

These policies can be combined to create more complex policies. Composed policies (see `ComposedPolicy` metaclass) are defined by a set of policies (either single or composed). The evaluation of the composed policy is based on the evaluation of its sub-policies. For this, we propose a set of boolean attributes to define the relationship between the sub-policies: (1) `sequential`, sub-policies are evaluated by order of declaration; (2) `require_all`, all policies must be

[8] https://www.apache.org/foundation/glossary.html#LazyConsensus

satisfied; and (3) `carry_over`, which requires `sequential` attribute to be true, states that the outcome of the first policy is used as input for the next one (e.g., the votes of the first policy are considered in the second one).

Conditions. we defined the condition hierarchy (see `Condition` metaclass) to express the determinants of the policy (e.g., a policy to be resolved before two weeks). The condition hierarchy is composed of several conditionals, and structural properties that define the policy (see, for instance, `ParticipantExclusion` metaclass). Conditionals can be evaluated before or after the policy is executed, or during its execution (see `evaluation` attribute). If they are fulfilled, the engine calculates the outcome. For instance, in a policy that requires three participants (see `MinimumParticipant` metaclass), it will not be evaluated until this condition is fulfilled (which means that by the end of the deadline, if exists, the policy will be rejected in case the condition is not fulfilled). Structural properties are considerations that will be taken in the evaluation of the policy. For instance, in a policy, we can define an exclusion of certain participants (see `ParticipantExclusion`) to avoid biases in their approval (e.g., excluding the one proposing the change). Similarly, a policy can have a deadline or a minimum time to take the decision (see `Deadline` and `MinDecisionTime` metaclass, respectively), defined by a fixed date or an offset (e.g., two weeks). Furthermore, we can specify which participants hold veto rights (see `VetoRight` metaclass) or may appeal a decision (see `AppealRight` metaclass), thereby triggering the associated appeal policy (see `policy` association).

5.2 Concrete Syntax

As concrete syntax, we use a textual notation following a typical block-based structure. The syntax is therefore formally defined by a grammar. Each instance of the metaclass is textually represented by its keyword and a block that contains the properties of the instance. Containment references are represented as nested blocks, while non-containment references use an identifier to refer to the target element.

To define an instance of a governance policy, we need to define, in order: the scope of the policy, the participants involved, and the method in which to resolve the policy, along with its conditions that must be fulfilled and parameters, if any.

We illustrate the concrete syntax of our DSL with a running example presented in Section 4.2. Listing 1 shows the initial version of the policy, where only maintainers can vote to approve or reject a code review. As we see, we can define the code review as a task scope, the maintainer role as a participant, and a majority voting policy with a boolean decision type. Since there are no conditions, as soon as a maintainer votes (i.e., approves the changes), the decision is made.

Listing 2 shows an updated version of the policy, where we added contributors as participants, and maintainers now have a vote value of 2.0 (i.e., their votes count double). We also added a deadline condition of 4 days, meaning that by the end of this time frame, the decision will be made by the cast votes (i.e., approvals).

Listing 3 shows the final version of the policy, where we define a documentation activity scope, a profile for Chinese-speaking individuals, and the different participants (i.e., contributors consisting of humans and agents). We also added a minimum ratio of 0.6, meaning that at least 60% of the votes must be for approval for the policy to be accepted. Finally, we set a deadline of 7 days to make the decision.

Besides the textual notation, we also provide a graphical, form-based editor to facilitate the creation of policies. The editor is web-based[9] and checks the validity of the policy while it is being created, ensuring that only well-formed policies can be defined. Furthermore, attributes are defined using appropriate widgets (e.g., sliders for decimal attributes)

[9]Available at: https://besser-pearl.github.io/GovernanceDSL/

```
Scopes:
    Tasks: codeReview
Participants:
    Roles : maintainer
MajorityPolicy CRApproval {
    Scope: codeReview
    DecisionType as BooleanDecision
    Participant list : maintainer
}
```

Listing 1. Early days

```
Scopes:
    Tasks: codeReview
Participants:
    Roles : contributor, maintainer {
        vote value : 2.0
    }
MajorityPolicy CRApproval {
    Scope: codeReview
    DecisionType as BooleanDecision
    Participant list : contributor,
        maintainer
    Conditions:
        Deadline : 4 days
}
```

Listing 2. Weighted voting

```
Scopes:
    Activities: documentation
Participants:
    Profiles: langCN {
        language : Chinese
    }
    Roles : contributor
    Individuals :
        Mike {
            vote value : 2.0
            profile : langCN
            role : contributor
        },
        (Agent) DocAgent {
            confidence : 0.8
            role : contributor
        }

MajorityPolicy CRApproval {
    Scope: documentation
    DecisionType as BooleanDecision
    Participant list : contributor
    Conditions:
        Deadline : 7 days
    Parameters:
        ratio : 0.6
}
```

Listing 3. Running example: Diverse roles

to enhance usability, and they are only visible when applicable (e.g., ratio for voting policies). Figure 3 shows a screenshot of the form-based editor.

5.3 Policy Parser and Syntax Implementation

The Policy parser follows a two-phase process: a grammar-driven parser that produces an abstract syntax tree (AST), followed by a tree-walker that creates the policy model. First, a grammar-based parser translates each policy file into an AST that captures the structure of the language declarations of scopes, participants, and policy definitions (single or composed) along with their decision type, communication channel, conditions, and parameters, if any. Second, a tree-walking phase interprets the AST and initializes a policy model by mapping tree nodes to typed elements of the governance metamodel (see Section 5.1). This phase resolves cross-references, producing a complete policy model with associated scope, participants, decision type, communication channel, and conditions, ready for the decision engine.

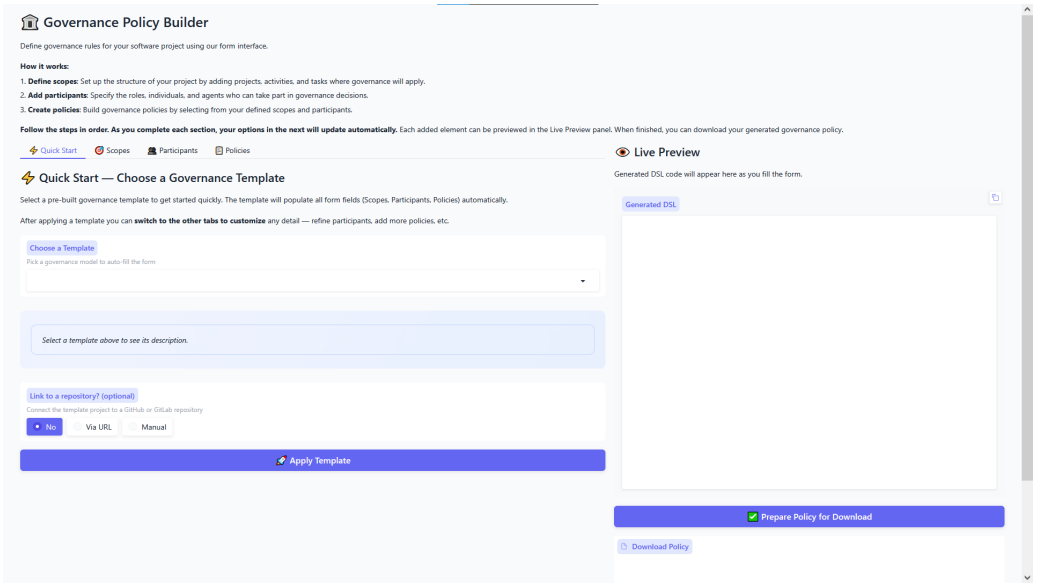


Fig. 3. Screenshot of the form-based editor for writing governance policies.

The governance metamodel is implemented as a set of Python classes, while the concrete syntax was implemented using ANTLR[10], a tool for creating language recognizers, interpreters, and translators from grammars. The form-based editor is implemented as a webpage using Gradio[11], a Python library for building interactive web applications. The complete implementation is available in the tool repository, including the grammar and editor[12].

6 EXTENDING THE GOVERNANCE METAMODEL

Defining the abstract syntax of our governance DSL with a metamodel allows to easily extend it to cover platform-specific elements. In this section, we illustrate an extension of our metamodel to provide support for code-hosting platform constructs, namely GitHub and GitLab. The extension introduces new metaclasses to represent platform-specific elements (e.g., repositories, pull requests) and conditions. These metaclasses are linked to the core metamodel through extensions, allowing to capture the relationships between the core governance concepts and the platform-specific elements.

Figure 4 shows the extended metamodel. We introduce the `Repository` metaclass to represent a GitHub repository, which can contain multiple `Patch` or `MemberLifecycle` tasks. We leverage the existing `Project–Task` hierarchy to represent the encapsulation of patch tasks within repositories. We define the `Patch` metaclass to represent the changes introduced in a pull request, thus linking it to the `PullRequest` metaclass. This way, we can define governance policies that apply only to pull requests within a repository. We also define the `MemberLifecycle` metaclass to represent tasks related to the lifecycle of members in a repository, such as becoming or being removed from a certain role.

In order to define GitHub-specific conditions for governance policies, we introduce three new conditions: (1) `CheckCiCd`, which checks if a pull request has passed all required tests; (2) `LabelCondition`, which checks if a pull

[10] https://www.antlr.org/
[11] https://gradio.app/
[12] https://github.com/BESSER-PEARL/governanceDSL

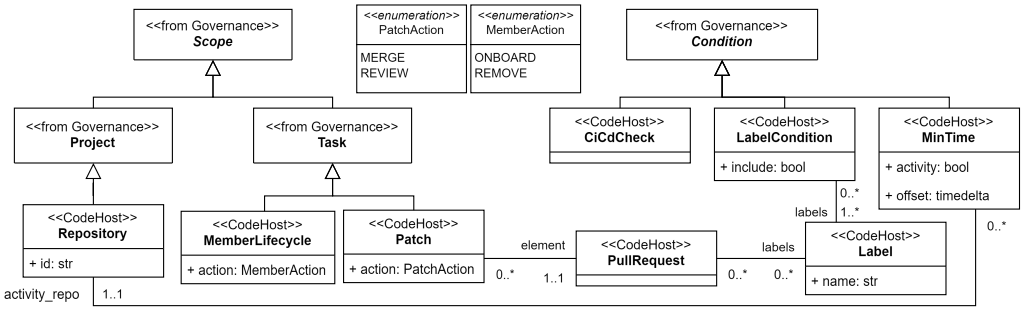


Fig. 4. Extended governance metamodel for code-hosting platforms.

request has (or has not) a specific set of labels; and (3) `MinTime`, which checks if there has been a minimum amount of activity or not on a repository for a specific time frame. `CheckCiCd` implicitly waits for all the tests to be passed, while `LabelCondition` has an associated set of `Label` instances and a boolean attribute `include` to indicate whether the presence or absence of the labels is required. `MinTime` has an `offset` attribute to indicate the time frame to consider (e.g., 3 months) and an `activity` boolean attribute to indicate whether to check for activity or inactivity. It is meant to be used for policies that require a minimum level of activity or inactivity in the repository to be applied (e.g., becoming or removing a maintainer). These conditions can be used in conjunction with the existing conditions from the core metamodel to create complex governance policies that consider both general and platform-specific aspects.

```
Scopes:
    Projects:
        GhRepo from GitHub : BESSER-PEARL/governanceDSL {
            Activities: development {
                Tasks: codeReview : PullRequest {
                        Action : review
                        Labels : unresolved, needs-review
                    }
            }
        }
Participants:
    Roles : maintainer
MajorityPolicy CRApproval {
    Scope: codeReview
    DecisionType as BooleanDecision
    Participant list : maintainer
    Conditions:
        CheckCiCd : true
        LabelCondition pre : unresolved, needs-review
}
```

Listing 4. Running example: Early days in GitHub.

Listing 4 instantiates the early days running example (see Section 4.2) with GitHub-specific conditions. In this example, we define a GitHub repository named `BESSER-PEARL/governanceDSL` as a project scope. Within this repository, we define a development activity that contains a task representing a pull request for code review. We specify that the task has the action `review` and can have labels `unresolved` and `needs-review`. We also define conditions to check if the pull request has passed all required tests and if it has the specified labels before a maintainer can approve it.

7 ENFORCEMENT OF THE POLICIES

To enforce the described policies, the runtime of the language needs to track interactions related to the decision-making process. This dynamic information is used to update a representation of the state of the decision-making process. To store this information, we designed a runtime metamodel representing the possible states of all the collaborations at stake. Based on the information stored in the runtime model (instance of the runtime metamodel), policies can be enforced by applying the rules defined in the governance language.

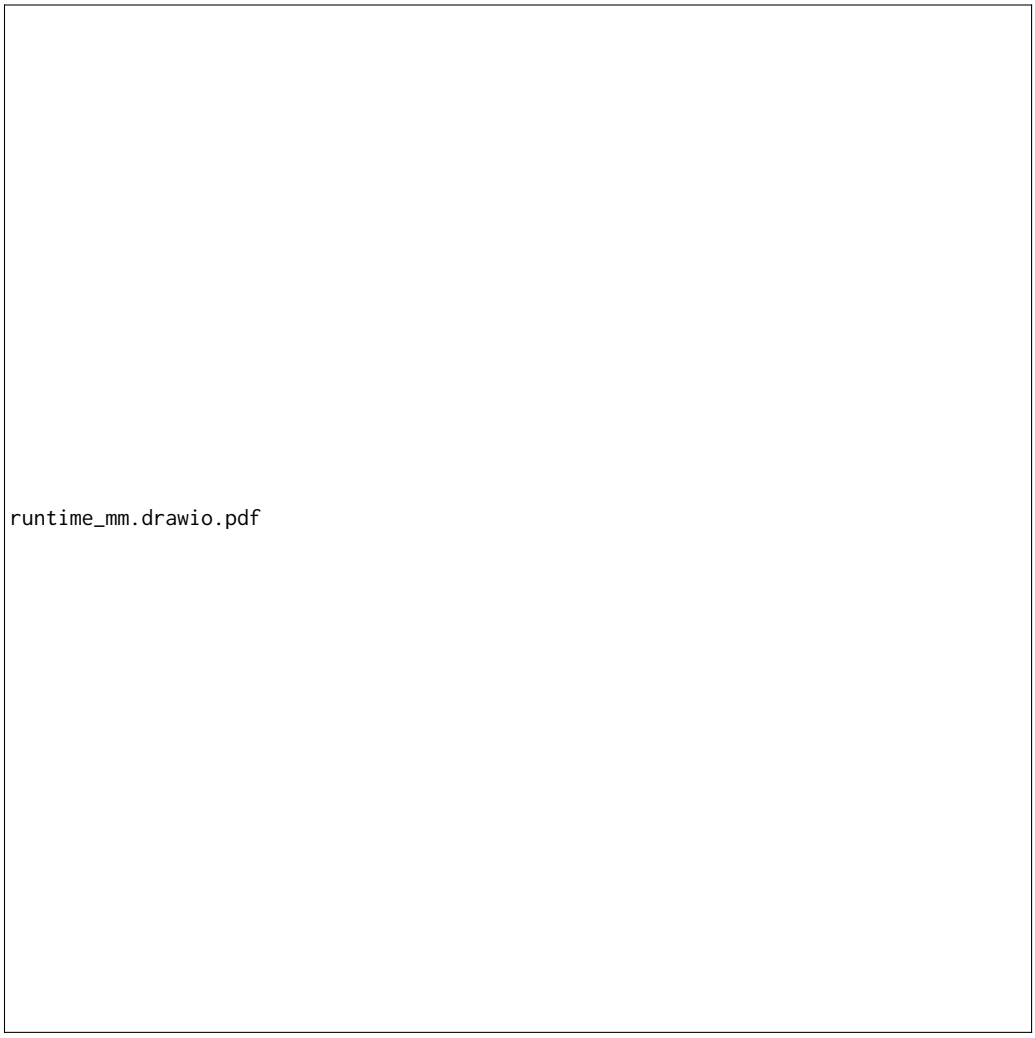


Fig. 5. Runtime metamodel. For the sake of clarity, method parameters are omitted from the diagram, and some attributes are shown with union types.

7.1 The Runtime Metamodel

The runtime metamodel, depicted in Figure 5, defines the set of possible state of decision processes. The runtime state includes all ongoing collaborations, individuals involved, votes cast, and decisions made. Mainly this metamodel is aimed at the four main dynamic aspects of the decision-making process: (1) collaborations, (2) individuals, (3) votes, and (4) decisions. On top of these four aspects, the `Interaction` metaclass is used to aggregate collaborations, individuals, and decisions and provide the methods to create and manage those aspects.

Collaboration. The `Collaboration` metaclass represents collaborations among multiple individuals requiring collective decision-making. Collaborations are the elements that call to action the governance rules defined in the policies. They are represented by a unique identifier, a rationale expressed as text, and a `Scope` representing the concrete scope in which this collaboration happens. When a collaboration is created (e.g., when a pull request is opened), the engine searches for the appropriate governance rule by matching the collaboration's scope against available policies. The scope matching follows a specificity hierarchy: the engine checks whether each rule's scope matches, includes, or differs from the collaboration scope. For instance, a pull request labeled *code* would match a policy scoped to *code* pull requests, be included in a repository-wide policy, but differ from a *docs*-scoped policy. The selected policy is the one with the most specific scope; if multiple policies have equivalent specificity, the first defined is chosen. This situation typically indicates incomplete policy coverage, for instance, when policies exist for individual labels but not their combinations.

In addition, collaborations contain a set of ballot boxes (see `ballot_boxes` attribute) to store the votes of each subsequent rule to enforce. Each ballot box maps a specific governance rule to the votes collected for that rule in the context of this collaboration. This mechanism allows managing composed policies that require multiple voting resolution through sequential or parallel application of rules. For sequential policies with the `carry_over` option enabled (see Section 5.1), votes from one phase are automatically transferred to the next phase's ballot box, allowing participants' previous votes to count toward subsequent decision stages without requiring re-voting. Collaborations are also associated with the individual who initiated the collaboration, the leader of the collaboration (the initiator by default), and the final decision made in the context of this collaboration.

Individuals. The `DynamicIndividual` metaclass represents the individuals participating in the decision-making process. This metaclass extends the `Individual` metaclass defined in the governance metamodel (see Section 5.1). It adds associations to the votes cast by the individual, the collaborations they initiated or are leading, and the role they enact in specific governance rules. Using this abstraction, we can represent the dynamic aspects of individuals in the decision-making process and keep track of its actions to provide explainability of the decision processes and accountability.

Votes. The votes contained in the ballot boxes are represented using the `Vote` metaclass. This metaclass represents the vote of an individual in the context of a collaboration. It is represented by the concrete agreement of the voter, a text expressing the rationale, the timestamp and the weight of the vote. For agent participants, the engine computes a voting weight derived from their runtime attributes. In the default strategy implemented for this validation, we use the formula $W_{vote} = 0.5 \cdot C + 0.3 \cdot A + 0.2 \cdot E$ (where C, A, E represent the values of confidence, autonomy, and explainability attributes, respectively), ensuring that agents with lower transparency or certainty have reduced influence on the decision. The coefficients prioritize confidence, the primary trust concern identified in our survey (83.33% of non-adopters); followed by autonomy as a governance risk calibrator for agents operating with reduced human

oversight, and explainability as a transparency property that aids accountability after decisions are made. Each vote is also associated to the individual that cast it and the decisions it was involved in. When a vote is cast (e.g., a code review is submitted), the engine retrieves the relevant collaboration from the runtime model and verifies that the voter is allowed to participate according to the governance rule. If authorized, a `Vote` instance is created and added to the appropriate ballot box.

Decisions. The `Decision` metaclass represents a decision made in the context of a given governance rule. It is characterized by the result of the decision process, the governance rule applied to make the decision, the set of votes used to make the decision, and the timestamp of the decision. In addition, each decision is associated to the collaboration it resolves, or partially resolve if the rule was a phase of a composed policy.

To resolve a collaboration, the engine analyzes the votes in the ballot box according to the governance rule's resolution type and conditions. For example, if a deadline condition is defined, the decision process triggers when the deadline is reached. When sufficient information is available, a `Decision` instance is created, linking the collaboration and counted votes. Based on the rule's scope and action specification, the engine may enact the decision (e.g., merging a pull request).

For composed policies, resolution depends on their phases. Sequential policies trigger subsequent phases after each resolution, while parallel policies start all phases simultaneously. The `require_all` attribute determines whether early termination is possible (e.g., a single rejection in a conjunction). If `carry_over` is enabled, votes from one phase are duplicated into the next phase's ballot box.

7.2 Decision Engine Implementation

To implement the decision engine of our framework, we need to receive the interaction data to update the runtime model. For this first implementation, we rely on GitHub and GitLab as collaboration platforms, using their webhook mechanism and REST API to monitor interaction and enact decisions. The complete implementation is available at the tool repository[13].

7.2.1 The Governance Agent. The governance agent was built using the BESSER Agentic Framework[14]. This framework allows defining agents using a state-machine formalism in Python. Figure 6 present the overview of the implemented state-machine in the context of the GitHub/GitLab extension.

The entry point of the agent is the `Init` state, initializing the runtime model and waiting for the first policy file to be sent. The `Policy` state updates the current policy file being used and the associated set of governance rules to apply. It calls the Policy Parser, and the resulting Policy model is stored in the agent for further use (e.g., policy lookup). The `Idle` state is the state managing the dispatch of the possible events to its associated managing state. The `Platform` state represents a family of platform-specific states (one per supported platform, e.g., `GitHub`, `GitLab`) responsible for managing platform events representing interactions during a collaboration. For instance, events such as pull request opening and pull request review submission representing the emergence of a new collaboration and participation in the decision-making process (i.e., a voting action), respectively. Each platform state's role is to convert the raw data received from its platform to a structured workflow event for the decision engine. The remaining states are responsible for enforcing governance rules by maintaining the runtime model and triggering decisions when policy conditions are satisfied. The `NewCollab` , `Vote`, and `Decide` states handle the different workflow events occurring, such as the start of a new collaboration, a vote for a given collaboration, or the resolution of a collaboration leading to the final

[13] https://github.com/BESSER-PEARL/GovernanceDecisionEngine
[14] https://github.com/BESSER-PEARL/BESSER-Agentic-Framework

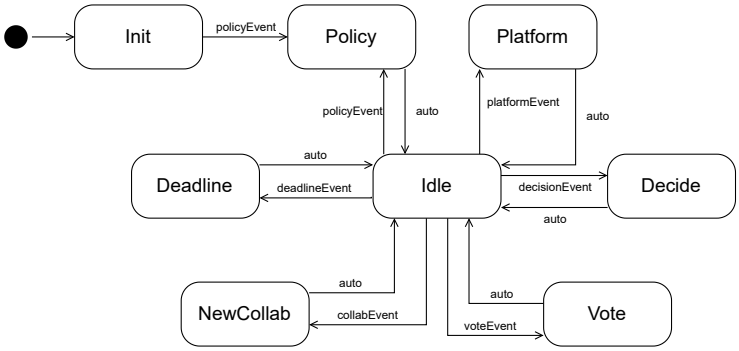


Fig. 6. Governance agent state-machine.

decision, respectively. Finally, the `Deadline` state is used to manage timed rules. This state is triggered when a deadline condition defined in a governance rule is reached. Its role is to forcefully trigger the decision process for the concerned collaboration.

To navigate through the state machine, we defined three types of events: (1) policy events, (2) platform events, and (3) workflow events. In addition to these events, the agent uses auto-transitions to return to the `idle` state after processing events, enabling continuous monitoring of new interactions.

Policy events. Policy events are used to update the current policy file being used by the agent. These events are sent to the agent when a new policy file is uploaded on the configuration panel of the agent. Also, when a policy event is received, the agent transitions from the `idle` state to the `policy` state.

Platform events. Platform events represent events sent by external platforms, such as GitHub or GitLab, to notify the agent of interactions happening on these platforms. In our implementation, these events are received using webhooks. When a platform event is received, the agent transitions from the `idle` state to the corresponding `platform` state based on the source platform of the event. Each platform state is responsible for converting the platform-specific event to a structured workflow event.

Workflow events. Workflow events represent high-level events used by the decision engine to update the runtime model and enforce the governance rules. There are four types of workflow events: (1) collaboration proposal events, (2) voting events, (3) decision events, and (4) deadline events.

Collaboration proposal events are used to signal the start of a new collaboration and contain the required information to create a new collaboration. Namely, a unique identifier, the rationale of the collaboration, the scope in which it happens, and the individual that initiated the collaboration. These events are used to transition from the `idle` state to the `newCollab` state.

Voting events are used to signal a voting action in the context of a collaboration. These events contain the identifier of the collaboration, the individual that cast the vote, the agreement of the vote, and a rationale for the vote. Voting events are used to transition from the `idle` state to the `vote` state.

Decision events are used to trigger the decision process in the context of a collaboration. These events are created when enough data was collected to make the decision or if a deadline was reached and the decision needs to be taken.

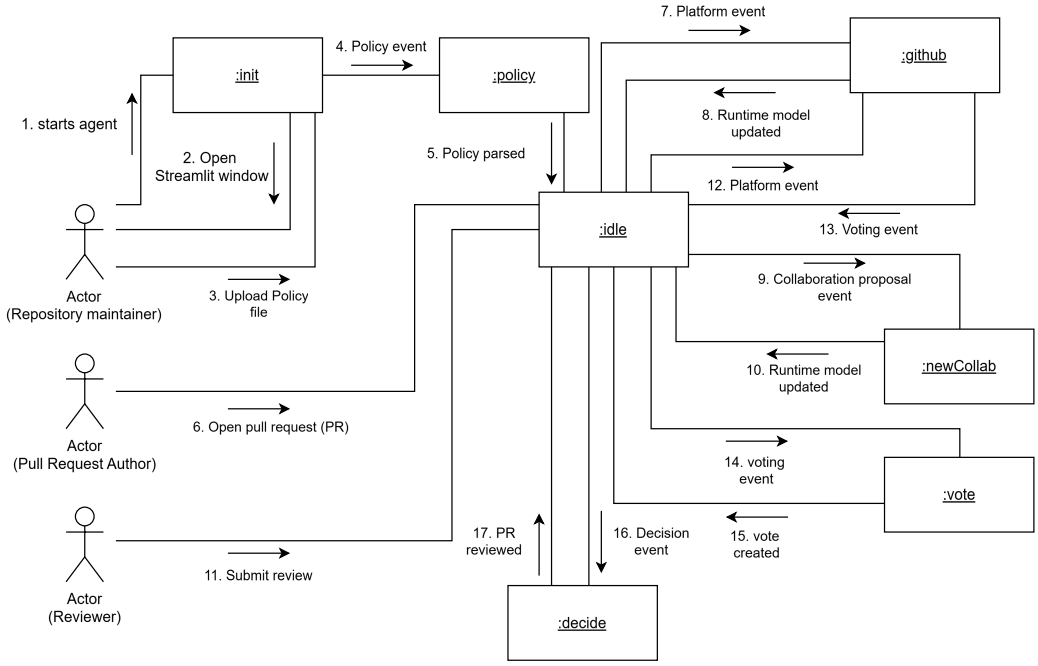


Fig. 7. Collaboration diagram of the example of policy enforcement.

They contain a reference to the collaboration for which the decision process needs to be triggered and the associated governance rule to process. Decision events are used to transition from the `idle` state to the `decide` state.

Deadline events are created when a deadline condition is defined in a governance rule. When this event condition is valid (i.e., the deadline is reached) a decision event is created to forcefully trigger the decision process for the concerned collaboration. These events are used to transition from the `idle` state to the `deadline` state.

7.2.2 Example of Policy Enforcement. To exemplify the enforcement of policies using the governance agent, we present in the following an example of use of the decision engine. Figure 7 presents the collaboration diagram of the example. Each action of the example is represented with a link and numbered in the diagram.

First, when the governance agent is started (cf. 1), it initializes the runtime model and waits in the `init` state for a policy file to be uploaded. On the side, the agent opened a Streamlit[15] web application to allow the user to upload a new policy file (cf. 2). When a new policy file is uploaded (cf. 3), a policy event is sent to the agent, allowing the transition from the `init` state to the `policy` state (cf. 4). In this state, the uploaded policy file is parsed and the governance rules extracted and stored in the agent for further use. Then, the agent transitions to the `idle` state, waiting for new events (cf. 5).

At this point, the agent is ready to receive platform events from the monitored collaboration platforms. For instance, when a new pull request is opened on a monitored repository (cf. 6), a platform event is sent to the agent using a webhook (cf. 7). This event allows the transition from the `idle` state to the corresponding `platform` state (`GitHub` in this case) based on the source platform of the event. In the `GitHub` state, the pull request information is extracted from

[15]https://streamlit.io/

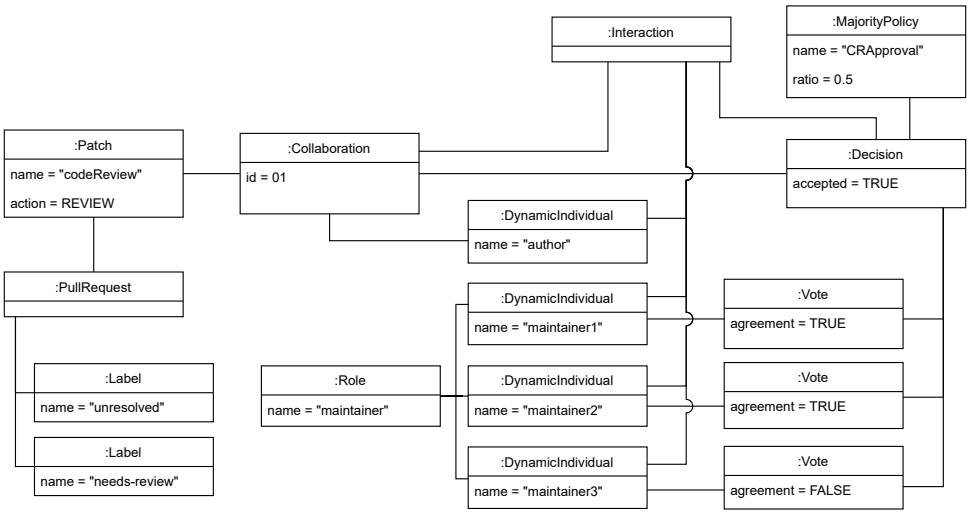


Fig. 8. Runtime model of the example.

the event, a `Scope` object representing the pull request scope is created, and a collaboration proposal workflow event is created (cf. 8). Then, the agent transitions back to the `idle` state, and the collaboration proposal event allows the transition to the `newCollab` state (cf. 9). In this state, the runtime model is updated to create a collaboration and the appropriate policy is searched in the available governance rules, and then transitions back to the `idle` state (cf. 10).

For this example, we will consider the governance rules defined in Listing 4. In this case, if the pull request is not labeled with *unresolved* and *needs-review*, no rule matches the scope of the collaboration and the collaboration is ignored. If the pull request is labeled with both *needs-review* and *unresolved*, the rule `CRApproval` matches the scope of the collaboration and its corresponding ballot box in the collaboration is opened. In the rest of the example, we will consider that the pull request is correctly labeled and the rule `CRApproval` is selected. After this processing step, the agent transitions back to the `idle` state, waiting for new events.

When a new review is submitted on the pull request (cf. 11), a platform event is sent to the agent using a webhook (cf. 12). This event allows the transition from the `idle` state to the corresponding `GitHub` state. In the `GitHub` state, the review acceptance is used as agreement of the vote. A voting workflow event is created based on the result of the review. Then, the agent transitions back to the `idle` state (cf. 13), and the voting event allows the transition to the `vote` state (cf. 14). In the `vote` state, we first check that the maintainer is allowed to vote in the context of the `CRApproval` rule. If the maintainer is allowed to vote, the runtime model is updated to create the vote instance and add it to the ballot box of the `CRApproval` rule. Then, the agent transitions back to the `idle` state (cf. 15). This voting process is repeated for each review submitted on the pull request. For this example, we consider that three reviews were submitted: two approvals and one change request.

Finally, when a vote ensures the acceptance or rejection of the collaboration (impossibility to be below, or above, the threshold defined in the rule respectively), a decision workflow event is created and sent to the agent (cf. 16). This event allows the transition from the `idle` state to the `decide` state. In the `decide` state, the votes contained in the ballot box of the `CRApproval` rule are analyzed. In this case, the two approvals represent a ratio of 0.66 of the voters, exceeding

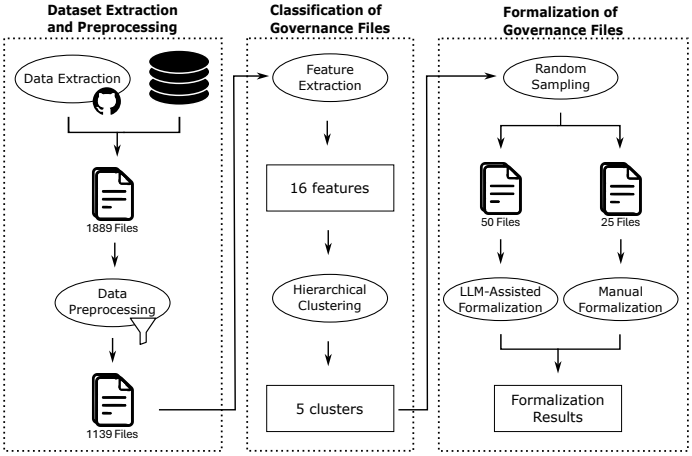


Fig. 9. Overview of the validation process for evaluating the coverage and expressivity of our DSL.

the 0.5 default ratio. The runtime model is updated to create the decision instance associated to the collaboration, the `CRApproval` rule, and the three votes. Finally, based on the scope of the rule, the pull request is left untouched as the action specified was `Review` leaving the management of the accepted pull request to humans or other tools. Then, the agent transitions back to the `idle` state (cf. 17), allowing new collaborations to be handled. Figure 8 presents the resulting runtime model after the enforcement of the example. For the sake of clarity, we omitted the `rationale`, `timestamp` and `ballot_boxes` attributes, as well as `vote_value` attribute since all participants have its default value (1.0).

8 VALIDATION

8.1 Coverage and Expressivity of the DSL

Our proposed DSL addresses the governance needs and expectations identified in the practitioner survey presented in Section 2.2. To further evaluate the expressivity and real-world applicability of our DSL, we formalized existing governance policies from GitHub projects using our DSL. To this aim, we use our DSL to represent a subset of `GOVERNANCE.md` files from a curated dataset.

We used the dataset provided by Yan et al. [30], extended it, and classified the files based on their content. This classification process resulted in five distinct clusters. From these clusters, we randomly selected 75 files, with equal representation, and formalized them using our DSL, both manually and with the help of an LLM, to check if our language was able to capture the information contained in these governance policies. Figure 9 illustrates this process. In the following we explain each step in more detail.

8.1.1 Dataset Extraction and Preprocessing. The dataset provided by Yan et al. [30] is a longitudinal collection of 710 OSS projects hosted on GitHub, each containing a `GOVERNANCE.md` file. The dataset includes information about project activity (i.e., commit history and issue tracking) as well as the history of modifications in governance files. However, since our analysis focused strictly on the content of the governance files rather than their evolution or project activity, we re-executed only the importer responsible for identifying and extracting repositories with a `GOVERNANCE.md` file (see

Data Extraction). This targeted extraction allowed us to extend the dataset with 1,179 additional repositories, providing a total of 1,889 files.

Next, we preprocessed the dataset (see *Data Preprocessing*). We first filtered out non-English files, obtaining a total of 1,712 `GOVERNANCE.md` files. Then, we analyzed the filename of the retrieved files, since the match is non-exact. We classified the filenames into four groups: (1) those with a prefix not related to project governance (e.g., `data-governance.md`); (2) those with a date prefixed, which typically indicate blog posts or articles rather than project-level governance documents, as we checked; and (3) those with exact matches of `GOVERNANCE.md`; or (4) those that could not be classified. We then removed those files with a prefix not related to project governance and those with a date prefixed, resulting in 1,427 files. After filtering out those files that were not related to project-level governance, we checked for repositories having more than one governance document. For these, we kept only the files named `GOVERNANCE.md` (case insensitive), resulting in 1,362 files. Finally, we filtered out the files with less than 400 characters, resulting in 1,139 files. The discarded files contained only references to external websites where the governance documentation was hosted.

8.1.2 Classification of Governance Files. In order to obtain a representative sample of governance files, we first classified the 1,139 files into clusters based on their content. Before applying clustering, we needed to define a set of features that would allow us to capture the main governance topics present in the documents (see *Feature Extraction*).

To identify the governance topics present in each document, we defined eight keyword dictionaries covering core governance concepts: roles and hierarchy, decision-making, contribution process, conflict resolution, elections and lifecycle, meetings and communication, code of conduct, and change and amendment. The dictionaries were created based on the indications of initiatives such as the `GOVERNANCE.md` one[16], prior knowledge of the domain, and an iterative process of refinement based on the contents of governance files. This creation process was discussed and validated by the first and third authors until reaching full agreement. We then calculated the proportion of keywords matching each dictionary, yielding eight concept density features for each file (see Table 1a). We complemented these features with the log-transformed token count (i.e., length of the document) and seven binary features indicating whether governance-related headings appeared in the document (see Table 1b), for a total of 16 features. Regarding the seven binary features, we extracted the headings from the documents and matched them against seven structural categories (i.e., roles, decisions, scope, meetings, code of conduct, elections, and changes). The categories were defined based on the most common headings found in the documents.

After z-standardizing the 16 features, we used hierarchical agglomerative clustering with Ward's linkage and Euclidean distance to create the clusters (see *Hierarchical Clustering*). This method minimizes within-cluster variance, naturally producing compact, well-separated groups that represent distinct governance archetypes. To determine the absolute number of clusters, we evaluated the resulting hierarchy across a range of possible partitions ($k = 2..10$) using the mean silhouette width, ultimately identifying five distinct governance clusters as the optimal fit (= 5).

8.1.3 Formalization of Governance Files. After classifying the governance files into clusters, we randomly selected 15 files from each cluster, resulting in 75 files (see *Random Sampling*). We first manually formalized 25 files, with equal representation from each cluster. The formalization was performed by the first author and validated by the third author until reaching full agreement. Then, to scale the formalization to the remaining 50 governance files, we employed an LLM-assisted approach using Copilot with GPT-4.1 (see *LLM-Assisted Formalization*). We provided the LLM with a

[16] https://governance.md/

Table 1. Features used for governance-file classification: (a) keyword-density features and (b) heading-based structural features.

(a) Keyword-based features.

Feature	Example Terms
Roles & Hierarchy	role, member, tsc…
Decision-making	decision, consensus, vote…
Contribution Process	contribute, pull, merge…
Conflict Resolution	conflict, dispute, mediate…
Elections & Lifecycle	elect, nominate, term…
Meetings & Communication	meeting, call, agenda…
Code of Conduct	conduct, behavior, harassment…
Change & Amendment	amend, revision, bylaw…

(b) Heading-based features.

Feature	Matched Headings
Roles	"Role", "Contributor", "Collaborator"…
Decisions	"Voting", "Consensus", "Decision"…
Scope	"Scope", "Purpose", "Mission"…
Meetings	"Meeting", "Call", "Sync"…
Code of Conduct	"Code of Conduct" or "Conduct"
Elections	"Election", "Nomination", "Removal"…
Changes	"Amendment", "Updating Governance", "Change"…

prompt containing a brief description of our DSL, its main concepts, and examples of governance policies written in our DSL. We anticipated that the LLM might generate invalid DSL constructs to capture information present in the files but not supported by our DSL. However, we believe these hallucinations would help us identify potential gaps in our DSL. The LLM generated initial formalizations, which were then manually reviewed by the first and third authors to ensure accuracy. While the LLM-generated formalizations required refinement, they successfully captured the core governance aspects and helped identify concepts commonly found in governance files that are descriptive rather than actionable (e.g., mission statements, codes of conduct, licensing information, meeting schedules). These descriptive elements, while often included in `GOVERNANCE.md` files, fall outside the scope of our DSL, which focuses specifically on actionable governance rules that can be enforced by the decision engine.

This formalization process revealed that our DSL was able to capture the actionable governance information present in these policies, including role definitions, decision-making processes, and conflict resolution mechanisms. However, we identified four specific concepts that were not explicitly supported by our core DSL. Their prevalence in the dataset ($N = 75$) was as follows: (1) *relationships between roles* ($n = 2$, 2.7%), as in the `rust-lang/rfcs` governance where it requires each "subteam" to be led by a "core team" member; (2) *organizational representation limits* ($n = 11$, 14.7%), for instance `AcademySoftwareFoundation/OpenImageIO` governance states that if an organization loses its sole Technical Steering Committee (TSC) representative, it must nominate a replacement; (3) *cardinality constraints on roles* ($n = 6$, 8.0%), as in `Subbuleo23/helix` where the TSC is capped at 9 members or that it requires a minimum of 3 maintainers in the project; and (4) *recurring policies* ($n = 8$, 10.7%), e.g., `keylime/keylime` dictates that a new project lead must be elected annually. Since the prevalence of each of these concepts was low (less than 15% of the analyzed files), we argue that they are best addressed by extending the metamodel (see Section 6) rather than increasing the complexity of the core DSL.

In summary, across all 75 analyzed governance files, our DSL demonstrated sufficient expressivity to capture the actionable governance policies, with some exceptions that could be captured with an extension. Additional non-actionable documentation (often stored in separate files like `CODE_OF_CONDUCT.md` or `LICENSE.md`) was appropriately excluded from the DSL scope.

8.2 Engine Evaluation

In this subsection, we illustrate how our engine is capable of executing their governance policy by artificially generating a sequence of events that simulate the review and approval process of existing pull requests, and compare the result of our

engine to the ground truth observable on GitHub. We assess its applicability in real world scenarios. We selected three GitHub repositories with different governance policies. The first one is the `kubernetes/kubernetes` repository. In our previous paper [1], we analyzed the governance of the top 25 starred projects in GitHub, and one of the most complete governance policies we found was the one used in the `kubernetes/kubernetes`. Furthermore, the governance aspect is supported through their own CI/CD system,[17] implemented in GitHub with bot `k8s-ci-robot`, that automatically approves pull requests when certain conditions are met. For those reasons, we selected the governance policy of the `kubernetes/kubernetes` repository as a case study to evaluate our governance engine. Since the `kubernetes` repository governance follows a voting system (i.e., review and approve the pull request), we also wanted to evaluate the applicability of our engine in a leader-driven and consensus-based governance policies. For this, we leveraged the governance file dataset from Yan et al. [30] and selected the `interpretml/interpret` which has a consensus-based policy, and the `EthicalSource/ethicalsource.dev` which has a benevolent dictator model (i.e, leader-driven)[18]. Besides, these repositories also differ in size, `kubernetes` being the largest with 4012 contributors, `interpret` with 49 contributors, and `ethicalsource` with 38 contributors, which allows us to evaluate the applicability of our engine across different project sizes.

8.2.1 Experimental setup. To reproduce the review and approval process, we first formalized their governance policy into our DSL. The formalizations can be found in Appendix A. The `kubernetes` governance policy consists of three phases that must be executed sequentially. In the first phase, at least two reviewers (who cannot be the author of the pull request) must review the pull request, and posteriorly the label `lgtm` (looks good to me) must be added. Note how the `Author` individual is automatically captured as the author of the scope element, in this case the pull request. As soon as a reviewer vote in favor of the decision, we can advance to the next phase (i.e., `ratio` set to 0.0). GitHub provides support for reviewing pull requests, thus the votes are understood as reviews in favor of the pull request. In the second phase, at least one approver must approve the pull request, and once approved, the `approved` label is added. All assigned approvers must vote in favor of the decision, thus the policy is a `Majority` policy with `ratio` 1.0, since it needs the votes from all assigned approvers. At last, in the third phase, the bot `k8s-ci-robot` can automatically merge the pull request if it has both `lgtm` and `approved` labels, and it does not have the labels `do-not-merge/hold` or `needs-rebase`.

The `interpret` governance follows a consensus-based model, where explicit agreement of all maintainers is not required (i.e., lazy consensus). Because opinions may evolve during deliberation, so our engine does not enact decisions based on intermediate positions alone. We therefore introduced a 7-day deadline to trigger decision process at the end of the deliberation period. Note, lazy consensus is a governance model where a proposal is considered accepted if no objections are raised, so it is important to specify a time frame for the decision to be made. Any community member may appeal a decision, and appeals are first resolved by maintainer consensus. If maintainers deny an appeal, it can be escalated to the TSC.

Finally, the `ethicalsource` governance follows a benevolent dictator model, where the decisions are made by a single individual or a small group of individuals (i.e., leaders) who have the final say in decision-making. In this case, the benevolent dictator (i.e., CoralineAda) has appointed authoritative representatives (i.e., tobie and DonGoodman-Wilson) and they are empowered to make decisions on behalf of the project.

[17] https://docs.prow.k8s.io/

[18] Policies can be found in: https://github.com/kubernetes/community/blob/master/contributors/guide/owners.md#code-review-using-owners-files, https://github.com/interpretml/interpret/blob/main/GOVERNANCE.md and https://github.com/EthicalSource/ethicalsource.dev/blob/release/GOVERNANCE.md.

To enact this governance policy, we developed a testing framework providing: (1) a fluent API to connect and simulate actions and mock queries to GitHub REST API, (2) an extension of the engine to communicate collaboration state, and (3) abstractions to formulate assertions on the governance decisions made by the engine. We used this framework to simulate the review and approval process of a set of 20 pull requests each repository, selected randomly among those created in the last month.

8.2.2 Results and discussion. The results of the experiment showed that our governance engine was able to successfully execute the governance policy of the three repository, correctly processing the simulated events and making decisions in accordance with the defined policy. However, we observed some cases where the decision process did not finish or the test did not pass.

In the case of the `kubernetes` repository, among the 20 simulated pull requests, 15 were decided successfully, while 5 did not finish the decision process. After analysis of the pull requests, the 5 that did not finish the decision process were due to the lack of required reviews or approvals. The current implementation of the `k8s-ci-robot` reject automatically the pull request and revise its opinion when reviewers use the `lgtm` command and approvers use the `approved` labels. This means that when refused, the pull request is stalled until the author close the pull request. In this sense, we could transcribe that in the form of an additional deadline condition in the policies of the governance rule stating a maximum stalling time before the pull request is automatically closed. However, as this was not part of the original governance defined by `kubernetes` maintainers, we decided to keep the evaluation as is.

In the case of the `interpret` repository, all the simulated pull requests were decided successfully, with the maintainers making the final decisions as expected. The decision process was triggered by the deadline condition, and the maintainers were able to make their decisions based on the simulated events. There was no case where the appeal process was triggered. Additionally, we observed that agents were code contributors but they did not participate in the decision-making process.

Finally, in the case of the `ethicalsource` repository, all the simulated pull requests were decided successfully, with the benevolent dictators making the final decisions as expected but one. This pull request was approved and merged by a contributor, rather than one of the leaders, which is not compliant with the governance policy. This shows how our framework can be used to identify governance violations, which can be useful for project maintainers to ensure that the governance policies are being followed and to take corrective actions when necessary. However, since we are evaluating the applicability of our engine, we decided to include these violations in our analysis.

8.2.3 Live Deployment and Human-Agent Playground. While the simulation described above demonstrates the engine's ability to enforce policies correctly, these scenarios lacked active agent participation in decision-making. To evaluate agent participation and the engine applicability beyond pull request reviews (e.g., role nomination), we established a public GitHub playground repository[19] governed by our engine.

We deployed four community agents using the BESSER Agentic Framework (powered by `gpt-4o-mini` LLM) that participate as community members with diverse capabilities: voting on contributor promotions or demotions, resolving triage issues, and changes in documentation.

Each agent is defined in the governance policy with specific roles and attributes. Based on the LLM temperature, we assigned a confidence score, and based on the provided reasoning, we assigned an explainability value. This design allows us to observe how the engine leverages agent attributes to calculate the voting weights and make

[19] https://github.com/BESSER-PEARL/GovernancePlayground

decisions. The environment is specifically designed to exercise the human-agent collaboration capabilities of our DSL, enabling evaluation of governance scenarios that simulations of real-world scenarios could not cover. Additionally, for demonstrability purposes, we configured the deadlines to be evaluated in minutes instead of days, which allows to observe the decision process in a shorter time frame.

The playground is currently active, and we are continuously monitoring the interactions and decisions made by the agents and the engine. We observed the promotion of `besser-bot` agent to the role of maintainer, and its demotion, the triage of an issue trying to improve the documentation, and two pull requests proposing first an addition to the documentation, and then a fix of the documentation.

8.3 Threats to Validity

We reflect on the validity of our evaluation across coverage, expressivity, and engine enactment.

Coverage and expressivity evaluation. First, we translated 75 `GOVERNANCE.md` files from the filtered dataset of 1,139 files, which may not represent the full diversity of governance practices across all repositories and organizations. For this, we used stratified random sampling, reducing the risk of bias by ensuring representation across different kinds of governance files. Second, we relied on an LLM (Copilot with GPT-4.1) to scale the translation from 25 to the 75 `GOVERNANCE.md` files. To mitigate this threat, both the manual and LLM-assisted translations were produced by the first author and reviewed by the third author until reaching full agreement.

Engine evaluation. First, we limited our experiment to three governance models from three different repositories, representing each a decision model (i.e., voting, leader-driven, and consensus). Although the selected governance policies represent different decision models and the `kubernetes` policy is one of the most complete we found in our previous study, it may not represent the diversity of governance policies used in other repositories or organizations. Second, we only focused on pull requests merge, which is just one aspect of software project governance. To mitigate this threat we also provided a playground repository with a governance policy addressing other governance activities such as role nomination. Finally, the set of policies and conditions used in this experiment do not cover all the possible policies and conditions supported by our governance engine. While it should not be, some combination of policies and conditions may lead to different applicability results due to complex interactions.

9 CONCLUSION

In this paper, we have presented a framework for defining and enforcing governance policies in software projects, with a particular focus on supporting diversity among participants, including both human contributors and AI-powered agents. The framework comprises a DSL for specifying governance policies, a decision engine for enforcing these policies, platform-specific extensions for GitHub and GitLab, and a form-based editor for easily defining governance policies without requiring knowledge of the DSL syntax. Our approach extends previous work by enabling the specification of participant profiles and agent attributes, thus facilitating more equitable and transparent decision-making processes. We have evaluated the expressivity of our DSL by formalizing existing `GOVERNANCE.md` files from a curated dataset, demonstrating that our language can capture the actionable portions of real-world governance policies. Furthermore, we have validated the applicability of our decision engine by reproducing three real-world governance policies, and provided a playground repository where agents are active participants. Our approach implementation, the survey results, and the validation artifacts have been made available on a Zenodo package[20].

[20] https://doi.org/10.5281/zenodo.19364904

As future work, we aim to explore the integration of our tool with existing software development tools and platforms to facilitate its adoption and use in practice. We also intend to assess its impact on the inclusivity and diversity of software project communities, particularly in relation to the participation of AI agents. For instance, we aim to investigate different mechanisms to dynamically adjust agent's attributes (i.e., voting weight , explainability, confidence and autonomy level) based on performance (e.g., agents that consistently make poor decisions could have their voting weight reduced). Similarly, we want to explore the possibility of allowing users to customize the agents' voting weights based on their attributes (e.g., giving more weight to agents with higher explainability). Additionally, we want to extend our language to support more sophisticated consensus mechanisms and different consensus types (e.g., unanimous consent, rough consensus). On top of that, we also plan to enhance its capabilities by adding features such as providing reusable templates of common governance policies, and populate the form from a given existing policy. Furthermore, we envision on assessing the adoption of our framework in other domains, such as making explicit the governance systems of organizations (such as NGOs). This could enable the study of how different governance structures impact organizational effectiveness and inclusivity. Finally, we plan to explore the ethical implications, risks and potential misuse that may arise (e.g., over-reliance on agents, gaming confidence scores, or reinforcing existing power structures) and to investigate mitigation strategies, such as templates that partially reduce these risks by assigning greater weight to specific stakeholder groups.

ACKNOWLEDGMENTS

This work has been funded by the European Union (EU) under the Grant Agreement No 101189664 (MOSAICO project); SE4GenAI (PID2023-147592OB-I00) financed by MCIU/AEI/10.13039/501100011033/FEDER, UE; and BESSER, funded by the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475. Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the EU nor the granting authority can be held responsible for them.

REFERENCES

[1] Adem Ait, Gwendal Jouneaux, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2025. Towards Automated Governance: A DSL for Human-Agent Collaboration in Software Projects. In *Int. Conf. on Automated Software Engineering*. 3891–3895.

[2] Alejandro Barredo Arrieta, Natalia Díaz Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. 2020. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Inf. Fusion* 58 (2020), 82–115.

[3] Pernille Bjørn, Maria Menendez-Blanco, and Valeria Borsotti. 2023. *Diversity in computer science: Design artefacts for equity and inclusion.* Springer Nature.

[4] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition.* Morgan & Claypool Publishers.

[5] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2015. Enabling the Definition and Enforcement of Governance Rules in Open Source Systems. In *Int. Conf. on Software Engineering*. 505–514.

[6] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2023. For a More Transparent Governance of Open Source. *Commun. ACM* 66, 8 (2023), 32–34.

[7] Gemma Catolino, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Filomena Ferrucci. 2019. Gender diversity and women in software teams: how do they affect community smells?. In *Int. Conf. on Software Engineering: Software Engineering in Society*. 11–20.

[8] Mahasweta Chakraborti, Curtis Atkisson, Stefan Stanciulescu, Vladimir Filkov, and Seth Frey. 2024. Do We Run How We Say We Run? Formalization and Practice of Governance in OSS Communities. In *Conf. on Human Factors in Computing Systems*. 923:1–923:26.

[9] Simon Chesterman, Yuting Gao, Jungpil Hahn, and Valerie Sticher. 2024. The Evolution of AI Governance. *Computer* 57, 9 (2024), 80–92.

[10] Robert A Dahl. 2008. *Democracy and its Critics.* Yale university press.

[11] Ivan De Noni, Andrea Ganzaroli, and Luigi Orsi. 2013. The evolution of OSS governance: a dimensional comparative analysis. *Scand. J. Manag.* 29, 3 (2013), 247–263.

[12] Mattia Gasparini, Robert Clarisó, Marco Brambilla, and Jordi Cabot. 2020. Participation Inequality and the 90-9-1 Principle in Open Source. In *Int. Symposium on Open Collaboration*. 6:1–6:7.
[13] Matthew Gaughan, Kaylea Champion, Sohyeon Hwang, and Aaron Shaw. 2025. The Introduction of README and CONTRIBUTING Files in Open Source Software Development. In *Int. Conf. on Cooperative and Human Aspects of Software Engineering*. 191–202.
[14] Joan Giner-Miguelez, Sergio Morales, Sergio Cobos, Javier Luis Cánovas Izquierdo, Robert Clarisó, and Jordi Cabot. 2026. The Software Diversity Card: A framework for reporting diversity in software projects. *Inf. Softw. Technol.* 190 (2026), 107950.
[15] Hashini Gunatilake, John C. Grundy, Rashina Hoda, and Ingo Mueller. 2024. The impact of human aspects on the interactions between software developers and end-users in software engineering: A systematic literature review. *Inf. Softw. Technol.* 173 (2024), 107489.
[16] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model Based Multi-agents: A Survey of Progress and Challenges. In *Int. Joint Conf. on Artificial Intelligence*. 8048–8057.
[17] David Held. 2006. *Models of democracy*. Polity.
[18] M Lynne Markus. 2007. The governance of free/open source software projects: monolithic, multidimensional, or configurational? *J. Manag. Gov.* 11, 2 (2007), 151–163.
[19] Josianne Marsan, Mathieu Templier, Patrick Marois, Bram Adams, Kevin Carillo, and Georgia Leida Mopenza. 2019. Toward Solving Social and Technical Problems in Open Source Software Ecosystems: Using Cause-and-Effect Analysis to Disentangle the Causes of Complex Problems. *IEEE Softw.* 36, 1 (2019), 34–41.
[20] Henry Mintzberg. 1979. The structuring of organizations. In *Readings in strategic management*. Springer, 322–352.
[21] Iyad Rahwan, Manuel Cebrián, Nick Obradovich, Josh C. Bongard, Jean-François Bonnefon, Cynthia Breazeal, Jacob W. Crandall, Nicholas A. Christakis, Iain D. Couzin, Matthew O. Jackson, Nicholas R. Jennings, Ece Kamar, Isabel M. Kloumann, Hugo Larochelle, David Lazer, Richard McElreath, Alan Mislove, David C. Parkes, Alex 'Sandy' Pentland, Margaret E. Roberts, Azim Shariff, Joshua B. Tenenbaum, and Michael P. Wellman. 2019. Machine behaviour. *Nat.* 568, 7753 (2019), 477–486.
[22] Zeeshan Rasheed, Muhammad Waseem, Malik Abdul Sami, Kai-Kristian Kemell, Aakash Ahmad, Anh Nguyen-Duc, Kari Systä, and Pekka Abrahamsson. 2024. Autonomous Agents in Software Development: A Vision Paper. In *Int. Conf. on Agile Software Development*. 15–23.
[23] Gema Rodríguez-Pérez, Reza Nadri, and Meiyappan Nagappan. 2021. Perceived diversity in software engineering: a systematic literature review. *Empir. Softw. Eng.* 26, 5 (2021), 102.
[24] Sivasurya Santhanam, Tobias Hecking, Andreas Schreiber, and Stefan Wagner. 2022. Bots in software engineering: a systematic mapping study. *PeerJ Comput. Sci.* 8 (2022), e866.
[25] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark G. J. van den Brand, Alexander Serebrenik, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Gender and Tenure Diversity in GitHub Teams. In *Conf. on Human Factors in Computing Systems*. 3789–3798.
[26] Leijie Wang, Nicholas Vincent, Julija Rukanskaite, and Amy Xian Zhang. 2024. Pika: Empowering Non-Programmers to Author Executable Governance Policies in Online Communities. In *Conf. on Human Factors in Computing Systems*. 925:1–925:18.
[27] Andrzej Wasowski and Thorsten Berger. 2023. *Domain-Specific Languages - Effective Modeling, Automation, and Reuse*. Springer.
[28] Mairieli Santos Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor Scaliante Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco Aurélio Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Hum. Comput. Interact.* 2, CSCW (2018), 182:1–182:19.
[29] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. 2025. The rise and potential of large language model based agents: a survey. *Sci. China Inf. Sci.* 68, 2 (2025).
[30] Yibo Yan, Seth Frey, Amy X. Zhang, Vladimir Filkov, and Likang Yin. 2023. GitHub OSS Governance File Dataset. In *Int. Conf. on Mining Software Repositories*. 630–634.
[31] Yang Yang, Tanya Y. Tian, Teresa K. Woodruff, Benjamin F. Jones, and Brian Uzzi. 2022. Gender-diverse teams produce more novel and higher-impact scientific ideas. *Proc. Natl. Acad. Sci. U.S.A.* 119, 36 (2022), e2200841119.
[32] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. 2020. PolicyKit: Building Governance in Online Communities. In *ACM Symposium on User Interface Software and Technology*. 365–378.

A GOVERNANCE POLICIES USED IN THE ENGINE EVALUATION

```
Scopes:
    Projects:
        K8sProject from GitHub : kubernetes/kubernetes {
            Activities :
                TestActivity {
                    Tasks :
                        TestTask : Pull request {
```

```
                        Action : merge
                    }
                }
        }
Participants:
    Roles : Reviewers, Approvers
    Individuals : Author, (Agent) k8s-ci-robot
ComposedPolicy pr_merge {
    Scope: TestTask
    Order :
        Execution : sequential
        RequireAll : true
        CarryOver : true
    Phases {
        MajorityPolicy phase_1 {
            DecisionType as BooleanDecision
            Participant list : Reviewers
            Conditions:
                ParticipantExclusion : Author
                LabelCondition post : lgtm
            Parameters:
                ratio : 0.0
        }
        MajorityPolicy phase_2 {
            DecisionType as BooleanDecision
            Participant list : Approvers
            Conditions:
                LabelCondition post: approved
            Parameters:
                ratio : 1.0
        }
        LeaderDrivenPolicy phase_3 {
            DecisionType as BooleanDecision
            Participant list : k8s-ci-robot
            Conditions:
                LabelCondition pre: lgtm, approved
                LabelCondition pre not: do-not-merge/hold, needs-rebase
        }
    }
}
```

Listing 5. Governance policy for accepting pull requests in the kubernetes repository.

```
Scopes:
    Projects:
        Interpret from GitHub : interpretml/interpret {
            Activities : appealingIssue
        }
Participants:
    Roles: Contributors, Maintainers, TSC
    Individuals:
```

```
        paulbkoch { role: Maintainers },
        DerWeh { role: Maintainers }
LazyConsensusPolicy decisionMaking {
    Scope: Interpret
    DecisionType as BooleanDecision
    Participant list: Maintainers
    Conditions:
        Deadline deadline : 7 days
        AppealRight : {
            Appealers : Contributors, Maintainers, TSC
            Policy : ConsensusPolicy appealingPolicy {
                DecisionType as BooleanDecision
                Participant list: Maintainers
                Conditions:
                    AppealRight : {
                        Appealers : Contributors, Maintainers, TSC
                        Policy : ConsensusPolicy appealingPolicyTSC {
                            DecisionType as BooleanDecision
                            Participant list: TSC
                        }
                }
            }
        }
}
```

Listing 6. Governance policy for the `interpret` repository.

```
Scopes:
    Projects:
        ethicalsourcedev from GitHub: EthicalSource/ethicalsource
Participants:
    Roles: BenevolentDictator
    Individuals :
        CoralineAda { role: BenevolentDictator},
        tobie { role: BenevolentDictator },
        DonGoodman-Wilson { role: BenevolentDictator }
LeaderDrivenPolicy ethicalSourceDecisionMaking {
    Scope: ethicalsourcedev
    DecisionType as BooleanDecision
    Participant list: BenevolentDictator
}
```

Listing 7. Governance policy for the `ethicalsource` repository.