

MOSAICO

D4.3: Rule-based Runtime Engine

September, 2026

DOCUMENT STATUS: SUBMITTED (Pending review and approval from the European Commission)

Funded by the European Union

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however 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 European Union nor the granting authority can be held responsible for them.

D4.3 Rule-based Runtime Engine

Work package	WP4 AI Community Governance and Supervision
Task	T4.3 Rule-based Runtime decision engine to monitor, drive and finalise the consensus process
Task contributors	LIST
Due date	30/09/2026
Submission date	30/09/2026
Type of deliverable	Demonstrator
Dissemination Level	Public
Deliverable lead	LIST
Version	1.0.0
Authors	LIST
Reviewers	IMT, UY, COLLINS
Keywords	governance, policy, domain-specific language, decision engine

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1.0	31/08/2026	Draft for review	Gwendal JOUNEAUX
v1.0.0	30/09/2026	Final version	Marcos GOMEZ VAZQUEZ

Partners

























Funding



Funded by
the European Union

Call	Digital and emerging technologies for competitiveness and fit for the Green Deal (HORIZON-CL4-2024-DIGITAL-EMERGING-01)
Topic	HORIZON-CL4-2024-DIGITAL-EMERGING-01-22: Fundamentals of Software Engineering
Type of action	Research and Innovation Action (RIA)

Disclaimer

Funded by the European Union under the Grant Agreement No 101189664. Views and opinions expressed are however 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 European Union nor the granting authority can be held responsible for them.

Copyright Notice

© MOSAICO Consortium, 2026

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both. Reproduction is authorised provided the source is acknowledged.



Executive Summary

This deliverable presents the Rule-based Runtime Engine developed in Task 4.3 to enforce governance policies defined using the MOSAICO governance language. The decision engine plays a central role in the MOSAICO platform: it bridges the gap between governance policy specification and their actual enforcement in collaborative environments involving both human and AI participants, thereby enabling accountable and auditable decision-making across the platform. The engine monitors interactions occurring in collaborative environments, maintains a runtime representation of decision-making processes, and automatically applies the relevant governance rules. The engine has been evaluated on GitHub-based collaborations using governance policies from Open-Source Software projects representing different decision-making models, including voting, consensus, and leader-driven approaches. The results demonstrate the engine's ability to correctly enforce specified policies and identify governance violations. A live human-agent playground further illustrates its applicability to collaborative scenarios involving AI agents. The deliverable also provides guidance for deploying and using the decision engine and establishes a foundation for its future integration into the MOSAICO platform.

Table of Contents

1 Introduction 7

2 The Governance Language 7
2.1 Abstract Syntax 8
2.2 Concrete syntax 9

3 The Runtime Metamodel 10

4 Decision Engine Implementation 12
4.1 The Governance Agent 12
4.2 Example of Policy Enforcement 14

5 Engine Evaluation 16
5.1 Experimental setup. 17
5.2 Results and discussion. 17
5.3 Live Deployment and Human-Agent Playground 18

6 How to use the decision engine 19
6.1 Preliminaries 19
6.2 Launching the engine 19
6.3 Linking the engine to the repository 20

7 Conclusion 20

Annex 23
Governance Policies used in the Engine Evaluation 23

List of Figures

1 Governance metamodel. For the sake of clarity, some associations are represented as class attributes. 8
2 Runtime metamodel. For the sake of clarity, method parameters are omitted from the diagram, and some attributes are shown with union types. 10
3 Governance agent state-machine. 13
4 Collaboration diagram of the example of policy enforcement. 14
5 Runtime model of the example. 16
6 Configuration of the webhook. 20
7 Selected events for the webhook. 21

Abbreviations

OSS	Open-Source Software
LLM	Large Language Model
PaC	Policy-as-Code
PR	Pull Request

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, 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 (Cánovas Izquierdo et al., 2023). Governance policies formalize decision rights and procedures so that responsibilities are transparent to all participants.

In the context of MOSAICO, the contributors to a software project are not only the humans developing it, but also the communities of agents helping in the requirements, implementation, and documentation of the software. The governance language presented in the deliverable 4.1 allows the formalization of governance policies defining the roles and responsibility of this ecosystem as well as the processes to manage contributions and make decisions. However, this governance language only describe the policies. With this deliverable, we propose a decision engine allowing the enforcement of the described policies.

The decision engine is a cornerstone of the MOSAICO platform. As MOSAICO operates at the intersection of human collaboration and AI agency, having a mechanism to systematically enforce governance policies is essential to ensure that all interactions—whether initiated by humans or AI agents—remain within the boundaries defined by the community. By automating the application of governance rules, the decision engine reduces the burden of manual oversight, promotes consistent and transparent policy enforcement, and provides an auditable record of decisions. This is particularly valuable in MOSAICO, where AI agents participate in decision-making alongside humans, making it critical to balance the efficiency of automation with the accountability and trust required in collaborative software development.

To illustrate this decision engine, we use it to enforce governance policies in Open-Source Software (OSS) projects. Indeed, OSS projects increasingly rely on automation (a.k.a. bots) (Santhanam et al., 2022; Wessel et al., 2018) and AI-powered agents that participate in development workflows, introducing new governance challenges around transparency, accountability, and human-agent collaboration (Rasheed et al., 2024). Thanks to advancements in AI, agents, powered by Large Language Models (LLMs), are becoming more capable of performing complex tasks, and even participating in decision-making (Xi et al., 2025).

The rest of the deliverable is structured as follows: Section 2 provides a brief reminder on the governance language, Section 3 details the runtime metamodel at the core of the engine, Section 4 presents the agent implementation and its use of the runtime model, Section 5 evaluates the decision engine, Section 6 details the steps to launch and use the decision engine, and finally Section 7 concludes this deliverable.

2 The Governance Language

In this section, we provide a brief reminder of the governance language. In particular, we present the metamodel defining its abstract syntax, and an example of policy to illustrate its concrete syntax.

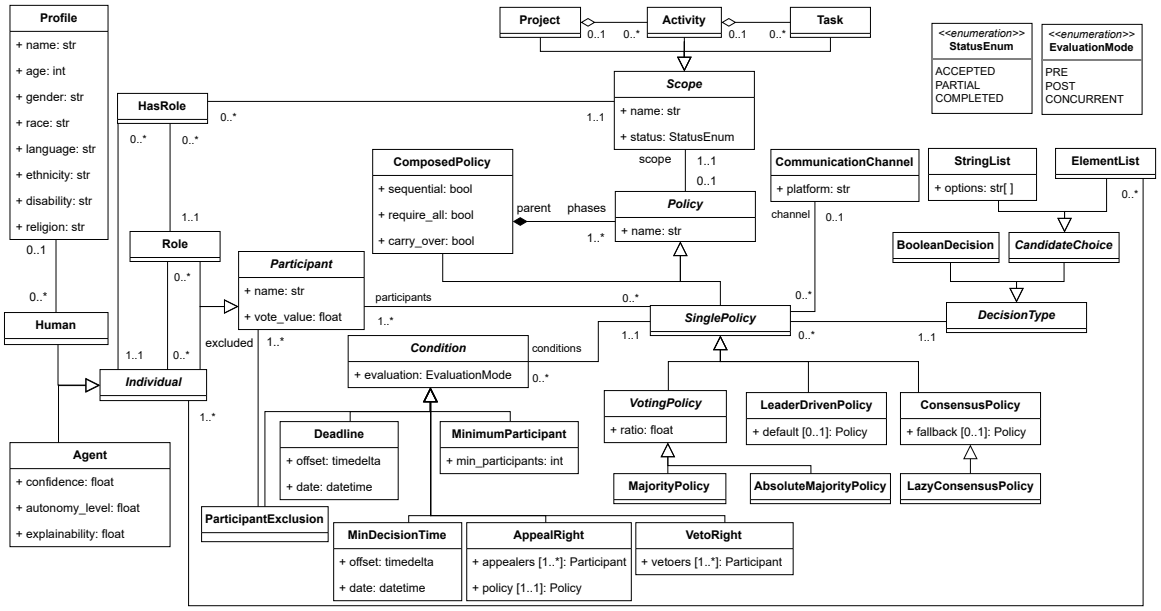


Figure 1: Governance metamodel. For the sake of clarity, some associations are represented as class attributes.

2.1 Abstract Syntax

The abstract syntax of our DSL is defined via the metamodel shown in Figure 1. 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)
4. conditions (i.e., when it should be applied)

Scope.
The scope identifies the context in which a policy is enforced. It is structured hierarchically into Projects, Activities, and Tasks, allowing policies to be defined at different levels of granularity. It can also capture the decision-making status and the relevant communication channel.

Participants.
Participants represent the entities involved in decision-making. They can be defined as Roles (groups of individuals sharing a responsibility) or Individuals, including both humans and AI agents. The DSL supports the definition of additional characteristics for roles or individuals such as voting weight, human profiles, agent capabilities and reliability indicators, and context-dependent roles within particular scopes.

Policies.
The policy hierarchy defines the decision-making mechanism. Policies can use voting, consensus, or leader-driven approaches and can specify different decision types, such as accepting/rejecting a proposal or choosing among candidates. Policies can also be composed

to create more complex decision processes, with mechanisms controlling sequencing, combination of results, and the transfer of outcomes between sub-policies.

Conditions.
Conditions define the requirements and constraints governing policy execution. They include evaluable conditions, such as a minimum number of participants, as well as structural properties such as deadlines, minimum decision times, participant exclusions, veto rights, and appeal mechanisms. These conditions determine when a policy can be evaluated and how its decision process is constrained.

2.2 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 a 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.

```
1  Scopes:
2      Projects:
3          GhRepo from GitHub : BESSER-PEARL/governanceDSL {
4              Activities: development {
5                  Tasks: codeReview : PullRequest {
6                          Action : review
7                          Labels : unresolved, needs-review
8                      }
9              }
10         }
11 Participants:
12     Roles : maintainer
13 MajorityPolicy CRApproval {
14     Scope: codeReview
15     DecisionType as BooleanDecision
16     Participant list : maintainer
17     Conditions:
18         CheckCiCd : true
19         LabelCondition pre : unresolved, needs-review
20 }
```

Listing 1: Running example: Early days in GitHub.

Listing 1 instantiates a simple majority policy with GitHub-specific conditions.[1] 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 should 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.

[1]The governance language has been extended with GitHub-specific component for early validation of the language (See Deliverable 4.1)

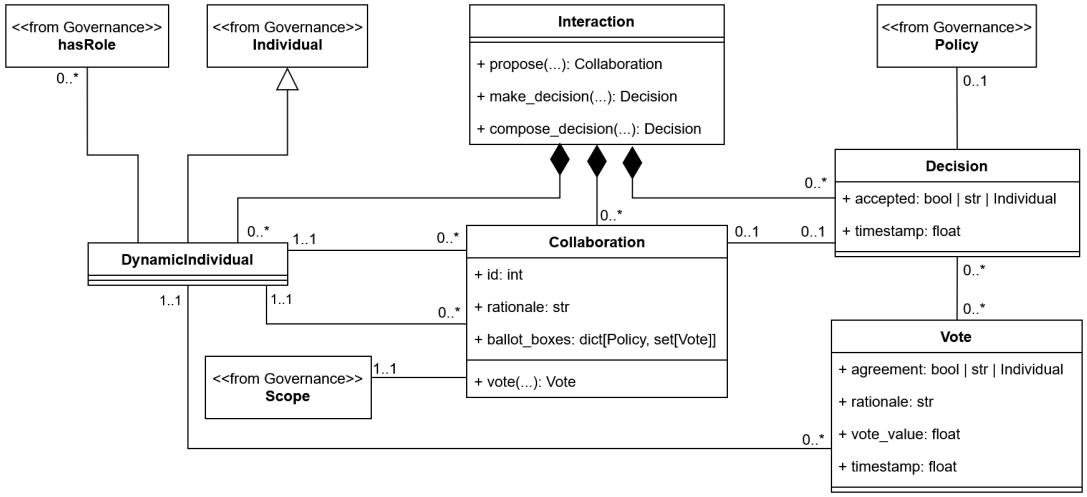


Figure 2: Runtime metamodel. For the sake of clarity, method parameters are omitted from the diagram, and some attributes are shown with union types.

3 The Runtime Metamodel

To enforce governance policies, the decision engine 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.

The runtime metamodel, depicted in Figure 2, 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 2.1), 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 2.1). 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 previous survey (see deliverable 4.1); 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.

4 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 as collaboration platform, using their webhook mechanism and REST API to monitor interaction and enact decisions. The complete implementation is available at the tool repository[2].

4.1 The Governance Agent

The governance agent was built using the BESSER Agentic Framework[3]. This framework allows defining agents using a state-machine formalism in Python. Figure 3 presents the overview of the implemented state-machine.

While the current implementation uses GitHub as the collaboration platform, the governance agent is designed to act as the enforcement layer of the broader MOSAICO ecosystem. In MOSAICO, it will receive interaction events from the collaboration platform (WP3), consume policies written in the governance language (D4.1), initiate consensus and manage debates (D4.4) for deliberation-based decisions, and treat AI agents from the community (WP2/WP3) as first-class governance participants whose attributes inform their voting weight.

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`) 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.

[2] https://github.com/BESSER-PEARL/GovernanceDecisionEngine
[3] https://github.com/BESSER-PEARL/BESSER-Agentic-Framework

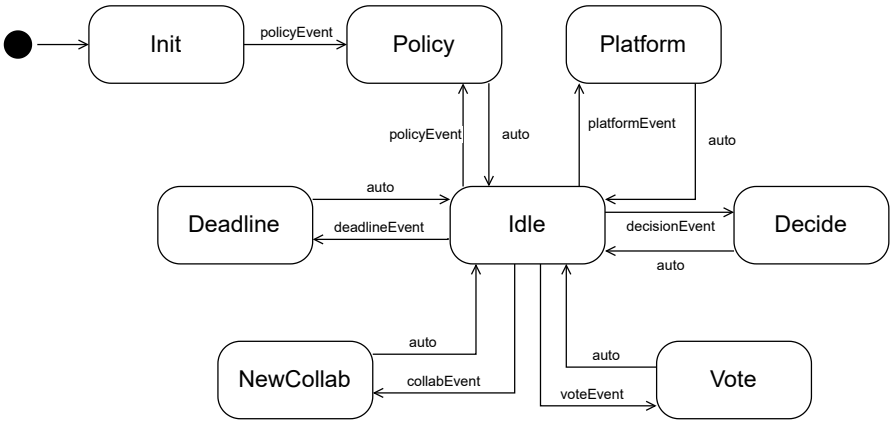


Figure 3: Governance agent state-machine.

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 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, 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 run-time 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. 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.

4.2 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 4 presents the collaboration diagram of the example. Each action of the example is represented with a link and numbered in the diagram.

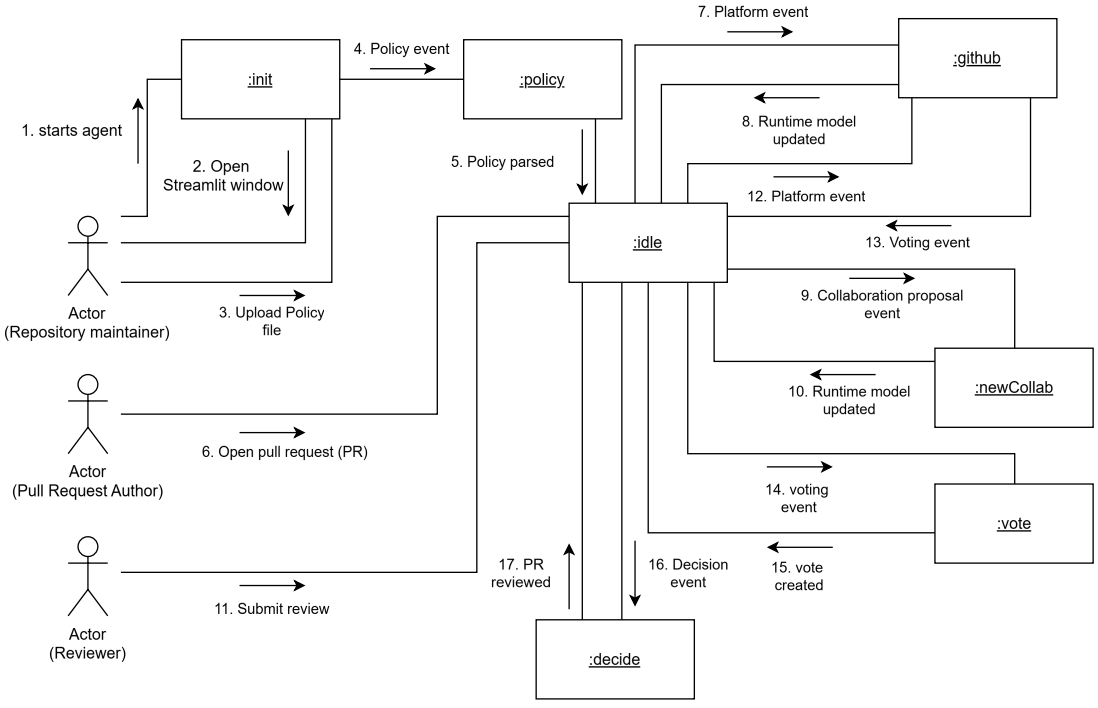


Figure 4: Collaboration diagram of the example of policy enforcement.

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[4] 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 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 1. 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 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 5 presents the resulting runtime model after the enforcement of the example. For the sake of clarity, we omitted the `rationale`, `timestamp` and `ballot_boxes`

[4] https://streamlit.io/

attributes, as well as `vote_value` attribute since all participants have its default value (1.0).

5 Engine Evaluation

In this subsection, we illustrate how our engine is capable of executing governance policies 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 (Ait et al., 2025), 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, 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. (Yan et al., 2023) 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)[5]. 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.

[5]Policies: 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.

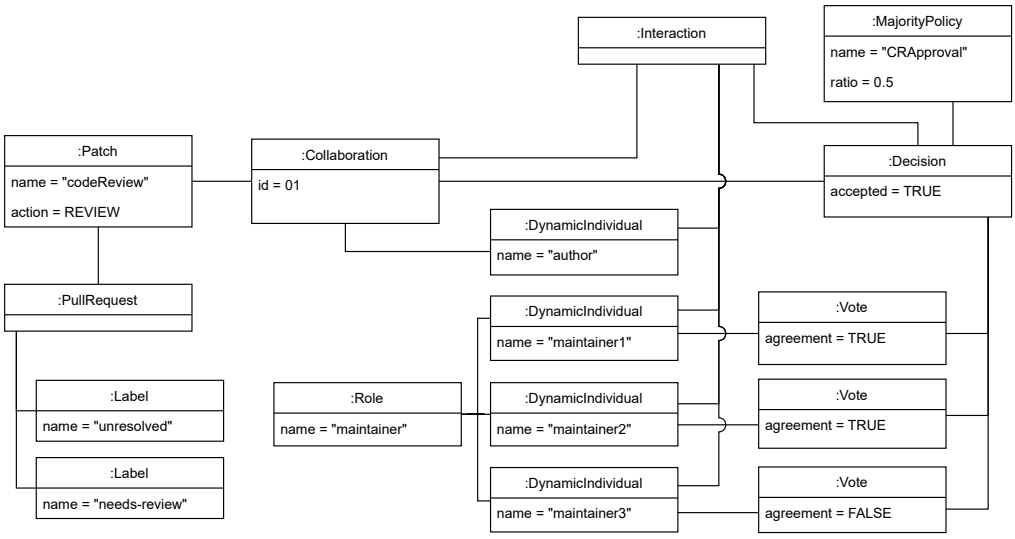


Figure 5: Runtime model of the example.

5.1 Experimental setup.

To reproduce the review and approval process, we first formalized the governance policies of these project with our DSL. The formalizations can be found in Annex. 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 votes 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 the 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 Technical Steering Committee.

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.

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.

5.2 Results and discussion.

The results of the experiment showed that our governance engine was able to successfully execute the governance policy of the three repositories, 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 re-

quests, 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` rejects 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 closes 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 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.

5.3 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[6] 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 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.

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.

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

6 How to use the decision engine

In this demonstration, we will use GitHub as the main platform for collaboration and decision making. Collaborations take the form of a pull request, while pull request reviews serve as votes. The decision will be enacted as merging the pull request or not depending on the policy applied and votes received.

6.1 Preliminaries

To use the decision engine, you first have to clone, in a folder called *DecisionEngine*, the two repositories containing the language definition and the engine itself, respectively available here https://github.com/BESSER-PEARL/governanceDSL and here https://github.com/BESSER-PEARL/Governance-DecisionEngine. Then, create and activate a Python virtual environment in the *DecisionEngine* folder, and install the dependencies with the following commands:

```
python3 -m venv .
source ./bin/activate
pip install -r GovernanceDecisionEngine/requirements.txt
```

Since the engine relies on webhooks to receive inputs from GitHub, it needs to be reachable by GitHub. This means being reachable on the public internet. If you want to run the engine locally, you will have to open the path for the webhooks to your engine. This can be done using a tool called ngrok[7] that you need to install and configure as explained in their "Get started" page[8] on sharing localhost.

6.2 Launching the engine

To launch the engine, you first have to make the governance language implementation available to your decision engine. This can be done by adding it to the PYTHONPATH environment variable with: **`export PYTHONPATH="$PYTHONPATH:/path/to/GovernanceDSL"`**.

Then, you need to configure the agent by updating the `config.yml` file, filling it with your GitHub personal token, and choosing a validation token for webhooks and the port used by the engine.

```
platforms:
  github:
    personal_token: "your-personal-token"
    webhook_token: "secret_test_webhook"
    webhook_port: 8901
```

After these steps, the engine is ready to start. When started, the engine will open a chat to configure the policies. You can simply drag and drop a governance file to update the governance policies that can be applied. You can launch the engine with this command:

```
python -m path.to.decision_engine
```

[7] https://ngrok.com/
[8] https://dashboard.ngrok.com/get-started/share-localhost

If you started the engine locally, you need to make it reachable using ngrok. the following command should setup a public url connected to the decision engine, keep this url for the next step.

```
ngrok http http://localhost:8901
```

6.3 Linking the engine to the repository

To link the engine to a repository, you will have to setup a webhook on the GitHub repository settings. For that, go on your repository settings in the "webooks" section, and add a new webhook. As shown in Figure 6, put the address (with port) of the decision engine as the payload URL. If you use ngrok, this is the public url you got in the previous step. Then set content type to JSON and copy your `webhook_token` as the secret. Depending on where you have deployed the engine, you might have to disable SSL verification. Then, you can select the event that will be received by the decision engine. Figure 7 details all the relevant events. Finally, tick the activation box and save the webhook. The decision engine will now receive all the information to take decisions. You can try it by opening a pull request to create a collaboration, and use the PR reviews as votes.

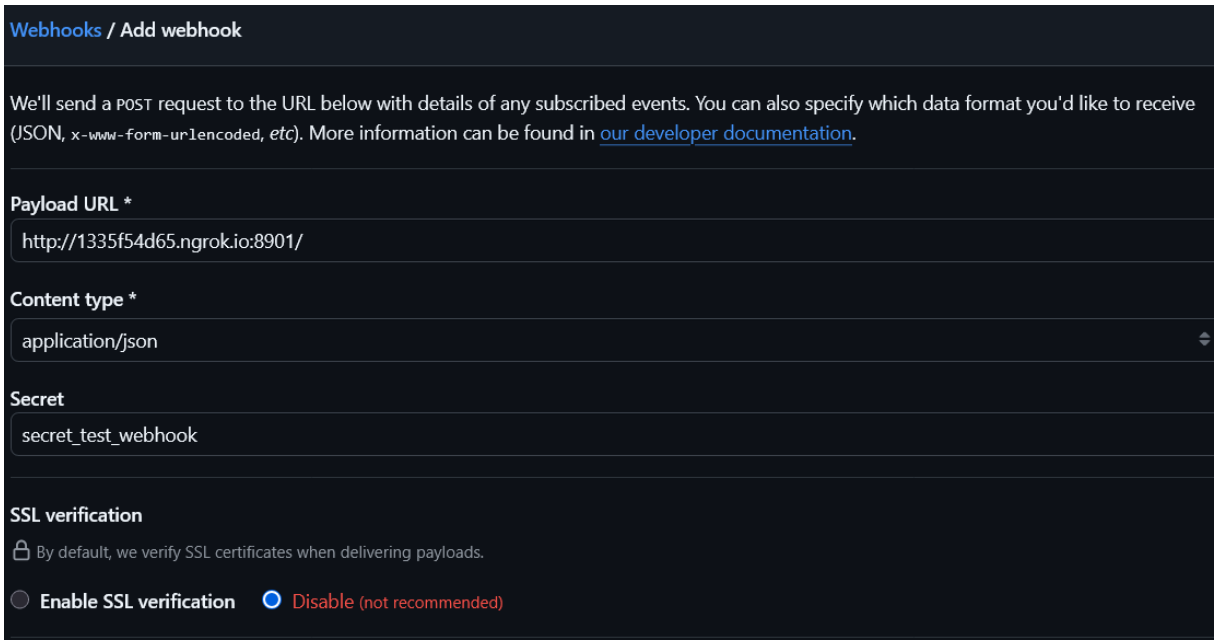


Figure 6: Configuration of the webhook.

7 Conclusion

In this deliverable, we have presented a rule-based decision engine allowing to enforce governance policies written with the MOSAICO governance language. This engine take the form of an agent receiving inputs from a platform (*e.g.,* MOSAICO, GitHub), and updating its representation of the decision process state (runtime model) based on concrete governance

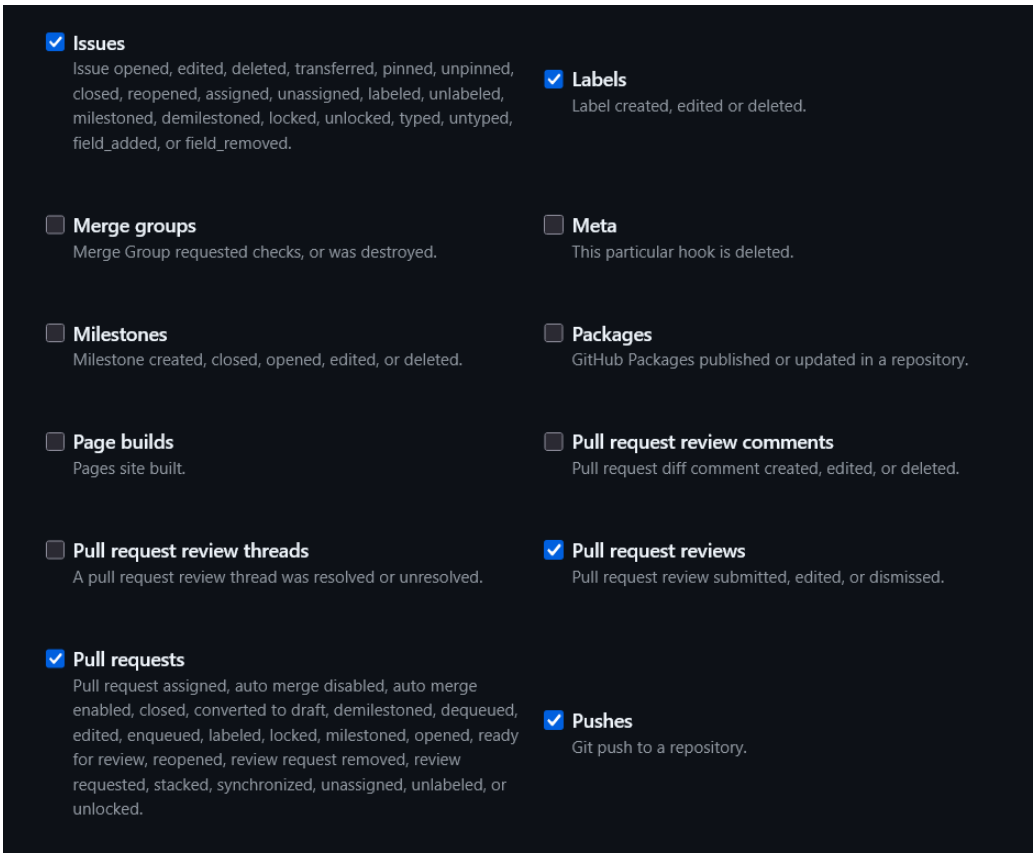


Figure 7: Selected events for the webhook.

policies. We validated the decision engine implementation by running an experiment on OSS projects, showing a correct and automatic enforcement of the specified governance policies. Finally, we provided a guide to start and use the decision engine using GitHub as platform. In a near future, this engine will be integrated to the MOSAICO platform after alignment with WP3 on communication between collaboration agents and the decision engine. While the current prototype relies on GitHub-specific mechanisms—webhooks for receiving events and REST API calls for enacting decisions—the MOSAICO integration will differ in how events are delivered and how decisions are enacted. The key to bridging this gap is the AISP interface: the governance agent will expose and implement an AISP interface, enabling standardized communication with the collaboration agents and the observability platform of the MOSAICO ecosystem. This interface will replace the current GitHub-specific adapter, allowing the engine's core enforcement logic to remain unchanged while connecting natively to the MOSAICO platform.

Then, this engine will be enhanced as part of task 4.4 to handle debate between agents to manage consensus-based policies.

References

Ait, Adem, Gwendal Jouneaux, Javier Luis Cánovas Izquierdo, and Jordi Cabot (2025), "Towards Automated Governance: A DSL for Human-Agent Collaboration in Software Projects", *in*: *CoRR* abs/2510.14465. [Cited on page 16]

Cánovas Izquierdo, Javier Luis and Jordi Cabot (2023), “For a More Transparent Governance of Open Source”, *in*: *Commun. ACM* 66.8, pp. 32–34. [Cited on page 7]

Rasheed, Zeeshan, 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*, pp. 15–23. [Cited on page 7]

Santhanam, Sivasurya, Tobias Hecking, Andreas Schreiber, and Stefan Wagner (2022), “Bots in software engineering: a systematic mapping study”, *in*: *PeerJ Comput. Sci.* 8, e866. [Cited on page 7]

Wessel, Mairieli Santos, 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”, *in*: *Hum. Comput. Interact.* 2.CSCW, 182:1–182:19. [Cited on page 7]

Xi, Zhiheng, 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”, *in*: *Sci. China Inf. Sci.* 68.2. [Cited on page 7]

Yan, Yibo, Seth Frey, Amy X. Zhang, Vladimir Filkov, and Likang Yin (2023), “GitHub OSS Governance File Dataset”, *in*: *Int. Conf. on Mining Software Repositories*, pp. 630–634. [Cited on page 16]

Annex

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 2: 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 3: 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 4: Governance policy for the `ethicalsource` repository.