Behavior Rule Architecture: Rule-Based Governance of AI System Behavior

> From Normative Language to Executable Rule Sets

Author: Spark Tsai
ORCID: https://orcid.org/0009-0006-8847-4703
Email: spark.tsai@gmail.com
Date: March 2026

Abstract

The rapid adoption of AI coding assistants has introduced unprecedented challenges in controlling generated behavior. Current approaches—including prompt engineering, custom instructions, and repository-level rules files—rely on natural language that lacks semantic precision and enforcement capability. These methods conflate intent optimization with behavior enforcement, leading to semantic drift, context overload, and rule conflicts without systematic governance mechanisms.

This paper introduces Behavior Rule Architecture (BRA), a structured framework for defining and managing AI execution behavior through normative rules. BRA establishes a clear separation between Normative Natural Language (NNL) for intent expression and Rule Normative Language (RNL) for behavior enforcement using MUST/MUST NOT semantics. The architecture comprises three layers: (1) Rule Language Layer for semantic transformation from intent to enforceable rules, (2) Rule Architecture Layer for atomic rule definition with Policy (MUST) and Constraint (MUST NOT) classifications based on observability differences, and (3) Governance Asset Layer for organizing reusable rule libraries through Decision Behavior Normative Categories and composable Application Rulesets.

BRA addresses three fundamental challenges in AI behavior governance: formalization of behavioral constraints, auditability through structured metadata, and reusability via shared rule libraries. The framework enables the transition from ad-hoc prompt engineering to systematic behavior rule management, establishing the engineering foundation for controllable, auditable, and governable AI systems.

Keywords

- Behavior Rule Architecture
- AI Behavior Governance
- Rule-Based AI Systems
- Normative Natural Language
- Rule Normative Language
- Rule Library
- Decision Behavior Normative Category
- Category Ruleset
- Application Ruleset
- Ruleset Composition
- Task Boundary
- Behavior Scope
- Semantic Drift
- AI Auditability
- Guardrails for LLMs
- AI-Assisted Software Engineering

1. Introduction

The proliferation of AI coding assistants—such as GitHub Copilot, Cursor, and Claude Code—has fundamentally transformed software development practices. These systems now generate substantial portions of codebases, architectural decisions, and technical documentation with minimal human intervention. While this automation accelerates development velocity, it introduces a critical challenge: **how do we ensure AI-generated behavior aligns with organizational policies, security requirements, and engineering standards?**

Current approaches to controlling AI coding behavior suffer from fundamental limitations. Prompt engineering relies entirely on natural language instructions that lack enforcement semantics. Custom instructions and workspace configurations provide fragmented rules that cannot be systematically managed or audited. Repository-level rules files (such as AGENTS.md or Cursor Rules) represent an emerging trend toward structured governance, yet they remain confined to natural language expressions without normative force.

The core problem lies in **semantic conflation**: existing methods merge intent optimization with behavior enforcement. When developers write prompts like "ensure code quality" or "follow security best practices," they express high-level intent in Normative Natural Language (NNL). However, AI systems cannot reliably distinguish between descriptive guidance and prescriptive mandates. This ambiguity leads to three failure modes:

1. **Semantic Drift**: AI interprets non-enforceable suggestions as optional guidelines rather than mandatory constraints
2. **Context Overload**: Accumulated instructions overwhelm the model's attention mechanism, causing critical rules to be ignored
3. **Rule Conflicts**: Contradictory instructions across different configuration layers lack systematic resolution mechanisms

These failures stem from the absence of a **systematic rule architecture**. Current methods treat behavior constraints as ad-hoc natural language fragments scattered across prompt templates, configuration files, and documentation. There is no unified structure for defining atomic rules, composing rule sets, managing rule libraries, or governing rule lifecycles.

This paper introduces **Behavior Rule Architecture (BRA)**, a comprehensive framework for defining and managing AI execution behavior through structured normative rules. BRA establishes a fundamental separation between **Normative Natural Language (NNL)** for expressing human intent and **Rule Normative Language (RNL)** for defining enforceable behavior constraints using MUST/MUST NOT semantics.

BRA comprises three integrated layers:

- **Rule Language Layer**: Transforms intent expressions (NNL) into enforceable rules (RNL), separating what developers want (intent optimization) from what AI must do (behavior enforcement)
- **Rule Architecture Layer**: Defines atomic rules as structured governance artifacts, classifying rules into Policy (MUST) and Constraint (MUST NOT) based on their observability characteristics
- **Governance Asset Layer**: Organizes rules into reusable libraries through Decision Behavior Normative Categories and composable Application Rulesets, enabling cross-project and cross-organization governance asset accumulation

The architecture addresses three fundamental challenges:

1. **Formalization**: How do we transform natural language intent into structured, machine-interpretable rules?

2. **Auditability**: How do we ensure behavior constraints are traceable, versionable, and accountable?
3. **Reusability**: How do we build shared rule libraries that can be composed for different contexts?

This paper makes the following contributions:

1. **Semantic Separation**: We formalize the distinction between NNL (intent optimization) and RNL (behavior enforcement), providing clear guidelines for rule authoring
2. **Complete Behavior Control Chain**: We establish an end-to-end chain from prompt → constraint → rule → composition → scope → execution → evidence
3. **Governance as Asset**: We introduce Rule Library as a structured, versionable, shareable governance artifact, transforming rules from ephemeral prompt fragments to manageable assets
4. **Dynamic Composition Capability**: We define Category Ruleset (behavior-based classification) and Application Ruleset (principle-based composition), enabling flexible rule combinations
5. **Decision Behavior Normative-Based Classification**: We propose categorizing rules based on decision behavior normative types rather than application domains, ensuring precise correspondence between rules and AI decision constraints

The remainder of this paper is organized as follows: Section 2 reviews related work in AI behavior governance. Section 3 analyzes the limitations of current approaches. Section 4 presents the conceptual model of BRA, including its mathematical foundations and design philosophy. Sections 5, 6, and 7 detail the three architecture layers. Section 8 discusses external integration and the architecture ecosystem. Section 9 addresses limitations and future work. Section 10 concludes with a summary of contributions and research outlook.

2. Related Work

The governance of AI coding behavior has evolved through several generations of approaches, from manual prompt engineering to emerging structured rule systems. This section reviews the primary methods and their inherent limitations.

2.1 Prompt Engineering and System Prompts

Prompt engineering represents the earliest and most widespread approach to AI behavior control. Developers craft natural language instructions embedded within prompts to guide AI behavior, often employing techniques such as few-shot examples, chain-of-thought reasoning, and role-playing personas.

System prompts—persistent instructions injected at the conversation initialization layer—represent a refinement of this approach. Tools like Claude Code and Cursor allow workspace-level system prompts that persist across sessions. However, these methods suffer from fundamental issues:

- **Semantic Ambiguity**: Natural language prompts cannot reliably distinguish between descriptive intent and prescriptive mandates. An instruction like "prioritize code readability" lacks enforcement semantics—AI may interpret it as optional guidance.
- **Context Window Saturation**: As prompts accumulate instructions, they consume context window capacity and suffer from attention dilution, where critical rules receive insufficient model focus.
- **Lack of Structure**: Prompts are monolithic text blocks without modular decomposition, making it impossible to compose, version, or independently manage individual constraints.

2.2 Custom Instructions and Workspace Configurations

Modern AI coding tools have introduced custom instructions and workspace-level configurations. GitHub Copilot allows repository-specific instructions through configuration files. Cursor enables workspace instructions that apply to all AI interactions within a project. Claude Code supports persistent instructions through its settings system.

These approaches recognize the need for persistent, project-scoped behavior control. However, they remain confined to natural language expressions without normative semantics. Instructions like "always write tests" or "never use deprecated APIs" lack MUST/MUST NOT force—they represent aspirations rather than enforceable constraints.

Furthermore, these configurations are **fragmented**: they exist as isolated files without systematic organization, versioning, or governance metadata. There is no mechanism for:

- Composing instruction sets from different sources
- Tracking instruction provenance and ownership
- Managing instruction lifecycles (draft, active, deprecated)
- Auditing which instructions were applied to specific AI-generated artifacts

2.3 Repository-Level Rules Files

An emerging trend exemplified by AGENTS.md and Cursor Rules involves storing behavior guidelines as structured files within code repositories. These files use formats like Markdown or YAML to document AI behavior expectations, and some tools parse these files to extract structured rules.

This approach advances beyond ad-hoc prompts by:

- **Persisting rules in version control**, enabling change tracking and collaboration

- **Associating rules with specific codebases**, providing contextual relevance
- **Supporting limited structural organization** through headings and sections

However, repository-level rules files still suffer from critical limitations:

- **Natural Language Semantics**: Despite structured formats, the content remains natural language without normative keywords. "AI should include tests" is semantically weak compared to "AI MUST include at least one test case per function."
- **No Enforcement Mechanism**: These files serve as documentation rather than executable constraints. AI may read them but is not compelled to follow them.
- **Composition Scarcity**: Most tools support only one rules file per repository, preventing composable rule sets based on different principles (e.g., security rules, testing rules, domain-specific rules).
- **Governance Gaps**: Rules lack metadata for ownership, risk classification, applicability context, and lifecycle management.

2.4 Policy-as-Code and LLM Guardrails

Policy-as-Code approaches, inspired by infrastructure-as-code practices, encode policies in domain-specific languages (DSLs) or general-purpose programming languages. Frameworks like Open Policy Agent (OPA) and HashiCorp Sentinel allow defining policies that are evaluated against system states.

LLM guardrails represent a related approach focused on constraining LLM outputs. Tools like Guardrails AI and NeMo Guardrails provide mechanisms for validating, correcting, or rejecting LLM outputs based on predefined rules.

These methods offer stronger enforcement semantics but target **post-hoc verification** rather than **pre-execution constraint**:

- Policies are evaluated after AI generates output, not during the generation process
- They focus on output validation (e.g., ensuring JSON schema compliance, detecting toxic content) rather than shaping the generation process itself
- They lack integration with AI coding assistants' decision-making processes

Furthermore, policy-as-code systems typically require:

- **Custom DSLs or programming knowledge**, creating barriers for non-technical stakeholders
- **Separate execution environments**, disconnected from AI coding tools
- **Limited behavioral expressiveness**, focusing on structural validation rather than nuanced behavioral constraints

2.5 Agent Governance and Execution Control

Research on autonomous AI agents has explored execution control mechanisms, including:

- **Tool use restrictions** that limit which tools agents can invoke
- **Approval workflows** that require human confirmation for sensitive operations
- **Budget constraints** that cap resource consumption
- **Stopping conditions** that halt execution when specific criteria are met

These mechanisms operate at the **execution layer**, controlling what actions AI can perform. However, they do not address the **decision layer**—how AI reasons about which actions to take. A tool restriction can prevent AI from deleting files, but it cannot guide AI's decision-making about when file modification is appropriate.

BRA complements execution control by defining **behavioral norms** that shape AI's decision process before execution constraints are applied.

2.6 Current State and Gap Analysis

Current approaches exhibit a common pattern: they blur the line between **intent expression** and **behavior enforcement**. Natural language simultaneously serves both purposes, leading to semantic ambiguity. There is a growing recognition of the need for structured rules, as evidenced by repository-level rules files and custom instruction systems.

However, no existing approach provides:

1. **Semantic clarity** through explicit separation of intent (NNL) and enforcement (RNL)
2. **Atomic rule definition** with structured metadata for governance
3. **Composable rule sets** based on different organizational principles
4. **Shared rule libraries** that accumulate governance assets across projects and organizations
5. **Complete behavior control chain** from intent to execution to evidence

BRA addresses these gaps by establishing a systematic rule architecture that treats behavior constraints as first-class, manageable artifacts rather than ephemeral prompt fragments.

3. The Problem of AI Coding Behavior Governance

This section analyzes the fundamental limitations of current approaches and articulates why AI coding behavior governance requires a systematic architecture.

3.1 The Core Problem: Semantic Conflation

The fundamental failure mode of current approaches is **semantic conflation**: they merge three distinct concerns into natural language expressions:

1. **System Intent (What we want AI to accomplish)**: High-level goals like "build a secure login system" or "refactor this module for better maintainability"
2. **Boundary Constraints (Where AI can operate)**: Scope limitations like "only modify files in the authentication module" or "do not access production databases"
3. **Behavior Normatives (How AI must behave)**: Prescriptive rules like "all database queries must use parameterized statements" or "never commit sensitive data to version control"

When these concerns are expressed in natural language, AI cannot reliably distinguish their normative force. Consider the instruction:

> "Refactor the authentication module to improve security, but be careful not to break existing tests."

This single sentence conflates:

- Intent: "improve security" (goal to optimize for)
- Boundary: "authentication module" (scope constraint)
- Behavior: "not to break existing tests" (prescriptive mandate)

From AI's perspective, all three are natural language tokens with ambiguous priority and enforceability. There is no semantic marker distinguishing "optimize security" (soft goal) from "preserve test integrity" (hard constraint).

3.2 Failure Modes of Current Approaches

This semantic conflation manifests in three observable failure modes:

3.2.1 Semantic Drift

When behavior constraints are expressed as natural language suggestions, AI treats them as optimization targets rather than mandatory requirements. Over time, as context accumulates and attention dilutes, AI progressively relaxes these constraints.

Example: A repository's AGENTS.md file includes the instruction:

> "Prefer functional programming patterns."

Initially, AI may favor functional style. However, when faced with imperative codebases or performance constraints, AI interprets this as "prefer when convenient" rather than "always apply." The constraint drifts from mandatory to optional to ignored.

Root Cause: NNL lacks normative keywords that signal enforcement semantics. "Prefer" is semantically weaker than "MUST use."

3.2.2 Context Overload

AI coding assistants operate within finite context windows. As instruction files, prompts, and configurations accumulate, they consume available context, reducing the model's capacity for code understanding and generation.

Manifestations:

- Critical rules mentioned at the end of long instruction files receive insufficient attention
- Accumulated rules create contradictions that confuse the model
- Rules scattered across multiple files (AGENTS.md, .cursorrules, system prompts) lack coherent prioritization

Example: A project accumulates 47 custom instructions across workspace configuration, repository rules files, and system prompts. AI must allocate attention across all 47, inevitably deprioritizing some critical constraints.

Root Cause: No systematic mechanism for rule prioritization, conflict resolution, or contextual activation.

3.2.3 Rule Conflicts

When multiple rule sources exist, contradictions are inevitable. A security-focused instruction may mandate strict input validation, while a usability instruction may prioritize minimal friction. Natural language provides no mechanism for systematic conflict detection or resolution.

Example:

- Security rule: "All user inputs must be validated before processing"
- Performance rule: "Minimize preprocessing overhead for real-time interactions"

These rules conflict in scenarios requiring real-time processing of user inputs. AI cannot determine which takes precedence.

Root Cause: Rules lack metadata for expressing priority, applicability conditions, and conflict resolution strategies.

3.3 Three Key Challenges

These failure modes point to three fundamental challenges that any AI behavior governance system must address:

Challenge 1: Formalization

How do we transform natural language intent into structured, machine-interpretable rules?

Current methods require AI to parse natural language and infer normative force—a process prone to errors and inconsistencies. We need:

- **Semantic clarity**: Explicit distinction between intent optimization and behavior enforcement
- **Normative language**: Structured syntax for expressing MUST/MUST NOT constraints
- **Formal representation**: Machine-readable rule structures that AI can reliably interpret

Challenge 2: Auditability

How do we ensure behavior constraints are traceable, versionable, and accountable?

Current methods scatter rules across ephemeral prompts and unmanaged configuration files. We need:

- **Provenance tracking**: Know who created each rule, when, and why
- **Version control**: Manage rule evolution over time
- **Lifecycle management**: Support draft, active, deprecated states
- **Execution evidence**: Generate audit trails showing which rules were applied to which AI decisions

Challenge 3: Reusability

How do we build shared rule libraries that can be composed for different contexts?

Current methods require reauthoring rules for each new project or context. We need:

- **Atomic rule definition**: Rules as independent, composable units
- **Categorization mechanisms**: Organize rules by decision behavior normative types
- **Composition strategies**: Combine rules based on different principles (workflow, domain, risk, compliance)
- **Shared libraries**: Accumulate governance assets across projects and organizations

3.4 Case Analysis: Failure Mode Table

Table 1 summarizes common failure modes across different governance approaches:

Method Type	Failure Mode	Representative Case
Prompt Engineering	Semantic Drift	Security guideline "always sanitize inputs" interpreted as "when convenient," leading to SQL injection vulnerabilities
Custom Instructions	Context Overload	50+ workspace instructions dilute attention, causing critical rules about test coverage to be ignored
Repository Rules Files	Rule Conflicts	AGENTS.md mandates "maximize code reuse," while security rules require "minimize dependency surface"—no resolution mechanism
Policy-as-Code	Post-hoc Gap	Policies validated after code generation cannot prevent insecure code from being written, only detect it after creation
LLM Guardrails	Behavioral Scope Limitation	Output filtering catches profanity but cannot enforce architectural patterns like "use dependency injection"
Execution Control	Decision Layer Gap	Tool restrictions prevent file deletion but cannot guide AI's reasoning about when modification vs. deletion is appropriate

3.5 Convergence: The Need for Systematic Rule Architecture

These failure modes and challenges converge on a single conclusion: **AI execution behavior lacks a systematic rule architecture**. Current approaches are:

- **Fragmented**: Rules scattered across multiple sources without unification
- **Semantically Weak**: Natural language cannot express normative force
- **Ungoverned**: No mechanisms for versioning, ownership, or lifecycle management
- **Non-composable**: Cannot flexibly combine rules based on different principles

This leads to the central thesis of this paper:

> **We need a structured architecture that treats behavior rules as first-class artifacts—definable, auditable, composable, and reusable—rather than ephemeral prompt fragments.**

The Behavior Rule Architecture (BRA) introduced in the next section provides this systematic foundation.

4. Conceptual Model of Behavior Rule Architecture

This section presents the conceptual model of BRA, including its three-layer internal structure, mathematical foundations, design philosophy, and relationships with external execution mechanisms.

4.1 BRA Positioning and Scope

BRA defines the structure and governance of behavior rules, serving as the normative foundation for AI execution behavior. It occupies the **development stage** of the AI lifecycle, focusing on rule definition rather than rule execution.

What BRA Covers:

- Structured rule definition through three integrated layers
- Semantic transformation from natural language intent to enforceable rules
- Governance metadata for auditability and lifecycle management
- Rule organization through Decision Behavior Normative Categories
- Rule composition through Application Rulesets

What BRA Does NOT Cover (External Dependencies):

- **Execution**: Runtime enforcement of rules (handled by Execution Layer)
- **Boundary**: Scope constraints and context binding (handled by Boundary Architecture)
- **Risk Assessment**: Dynamic risk signal detection (handled by Behavior Control Model - BCM)
- **Evidence**: Execution evidence generation and storage (handled by BCM)

This separation follows a core design principle:

> **BRA should only be responsible for "defining behavior constraints," not "executing, judging, governing, or validating" them.**

4.2 Three-Layer Internal Architecture

BRA comprises three integrated layers that transform human intent into governable rule artifacts:

Layer 1: Rule Language Layer

Purpose: Input abstraction and normative constraint transformation

Components:

- **Normative Natural Language (NNL)**: Prompt optimization, human-readable intent, non-enforceable
- **Rule Normative Language (RNL)**: Behavior rule expression, MUST/MUST NOT normative rules, enforceable behavior normatives

Function: Transforms what developers want (intent optimization) into what AI must do (behavior enforcement), establishing semantic clarity.

Layer 2: Rule Architecture Layer

Purpose: Structured rule definition

Components:

- **Decision Behavior Normative Category**: Classification of AI decision behavior normatives; defines behavior constraint categories before deriving specific rules
- **Rule**: Structured governance artifact consisting of RNL + Metadata, classified as Policy (MUST) or Constraint (MUST NOT)

Function: Converts normative language into atomic, manageable rule units with governance metadata.

Layer 3: Governance Asset Layer

Purpose: Reusable shared rule repository

Components:

- **Rule Library**: Cross-industry / cross-enterprise / cross-project repository for accumulating governance assets
- **Category Ruleset**: Rule collections based on the same Decision Behavior Normative Category
- **Application Ruleset**: Rule compositions based on arbitrary principles (workflow-based, risk-based, compliance-based, etc.)

Function: Enables rule sharing, composition, and long-term governance asset accumulation.

4.3 Complete Architecture Chain

The refined architecture chain establishes a complete transformation pipeline:

```
NNL (Intent) → RNL (Normative) → Rule (Artifact) → Ruleset (Collection) → Decision Behavior Norr
```

This chain ensures:

1. **Semantic precision** at each transformation step
2. **Governance capability** through structured metadata
3. **Compositional flexibility** through multiple ruleset types
4. **Asset accumulation** through shared libraries

4.4 Mathematical Foundations

This section formalizes the relationships between rules, rulesets, and libraries.

4.4.1 Cardinality Relationships

Rule ↔ Category Ruleset (1:N):

```
Category_Ruleset : Rule = 1 : N

∀ rule ∈ Library:
  ∃! category ∈ Behavior_Category:
    rule ∈ Category_Ruleset(category)
```

Interpretation: Each rule belongs to exactly one Decision Behavior Normative Category; conversely, one Category Ruleset contains multiple rules.

Rule ↔ Application Ruleset (N:N):

```
Rule : Application_Ruleset = N : N

∀ rule ∈ Library:
  ∃ {P₁, P₂, ..., Pₖ} ⊆ Principle:
    rule ∈ Application_Ruleset(Pᵢ), ∀ᵢ

∀ Application_Ruleset:
  ∃ {rule₁, rule₂, ..., ruleₘ} ⊆ Library:
    ruleⱼ ∈ Application_Ruleset
```

Interpretation: One rule can be composed into multiple Application Rulesets; one Application Ruleset can combine multiple rules.

Ruleset Cardinality Constraint:

```
∀ Ruleset ⊆ Library:
  |Ruleset| ≥ 1
```

Interpretation: A ruleset contains at least one rule, with no upper limit.

4.4.2 Composition Relationships

Rule Library as Union of Category Rulesets:

```
Library = ⋃_{C ∈ Behavior_Category} Category_Ruleset(C)
```

Interpretation: The union of all Category Rulesets constitutes the complete Rule Library.

Category Ruleset Mutual Exclusivity:

```
∀ C₁ ≠ C₂ ∈ Behavior_Category:
  Category_Ruleset(C₁) ∩ Category_Ruleset(C₂) = ∅
```

Interpretation: Category Rulesets from different Decision Behavior Normative Categories do not overlap (mutual partitioning).

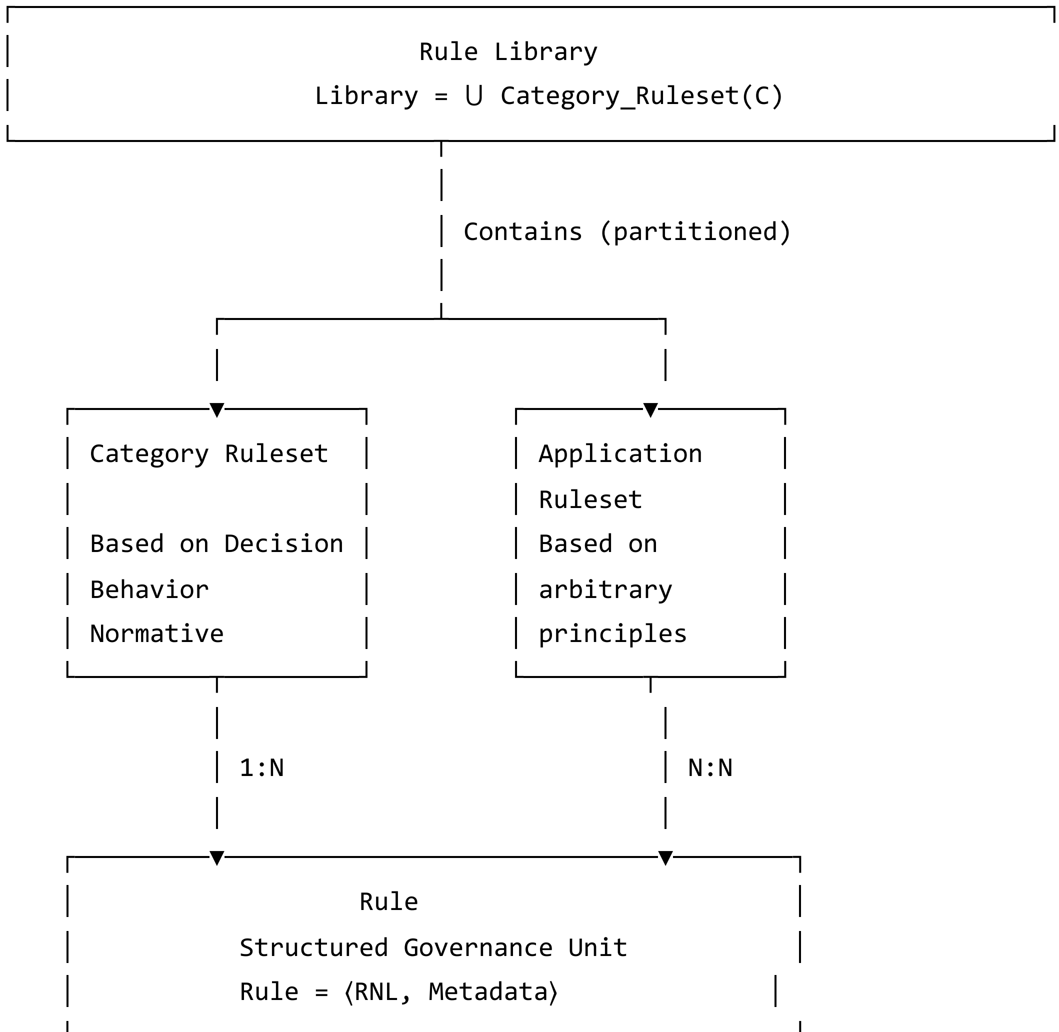
Application Ruleset and Category Ruleset Intersection:

```
∀ C ∈ Behavior_Category, ∀ P ∈ Principle:
  Category_Ruleset(C) ∩ Application_Ruleset(P) = ∅ or ≠ ∅
```

Interpretation: The same rule can belong to both a Category Ruleset and an Application Ruleset, but this intersection is not mandatory—it depends on the specific composition.

4.4.3 Entity-Relationship Model

The entity-relationship structure is visualized as:



4.4.4 Mapping Relationship Summary

Relationship	Type	Description
Rule → Category Ruleset	1:N	Each rule belongs to one Decision Behavior Normative Category
Category Ruleset → Rule	1:N	Each Category Ruleset contains multiple rules

Relationship	Type	Description
Rule → Application Ruleset	N:N	One rule can be composed into multiple Application Rulesets
Application Ruleset → Rule	N:N	One Application Ruleset contains multiple rules
Library ↔ Category Ruleset	Decomposition	Library = union of all Category Rulesets
Category Ruleset ∩ Category Ruleset	∅	Category Rulesets from different categories are mutually exclusive
Category Ruleset ∩ Application Ruleset	∅ or ∃	May intersect, but not necessarily

4.5 Design Philosophy

BRA's design is guided by core principles that ensure architectural purity and practical applicability.

4.5.1 Architecture Layering Principle

> **If it is injected, it is execution. If it is an attribute, it is governance.**

This principle distinguishes between:

- **Execution semantics**: Content injected into AI's active context that shapes generation behavior
- **Governance attributes**: Metadata that describes rules without affecting execution

Application: Rule's RNL (MUST/MUST NOT statements) is execution semantics; governance metadata (owner, version, intent) is governance attribute.

4.5.2 Declarative vs. Imperative Separation

> **Governance does not depend on what is described, but on whether it is declarative or imperative.**

BRA rules are **declarative**—they describe "what state must be maintained" rather than "how to act." This contrasts with imperative instructions that specify procedures.

Example:

- Imperative: "Before generating code, check if tests exist, and if not, create them first"
- Declarative: "Generated code MUST include corresponding test cases"

Declarative rules reduce AI's interpretation burden and increase enforceability.

4.5.3 Rule as Governance Atom

> **There is no governance without a rule. There is no rule without a policy or constraint.**

A rule is the **smallest unit that carries governance meaning**. It encapsulates:

- One explicit behavioral policy (MUST) or constraint (MUST NOT)
- Its applicable scope
- Its governance interpretation context
- Its accountability and audit expectations

Mental Model:

> A rule is a single constrained sentence, written so that AI has less room to misinterpret it, while humans and governance systems can still reason about it.

4.5.4 Struct Semantics Mechanism

> **LLMs are not "commanded" but confined in a smaller semantic room.**

Structured representations (YAML, JSON) constrain AI behavior not by increasing "command force" but by reducing "degrees of freedom":

Struct Semantic	Effect on LLM
WHY (intent)	Constrains "what this rule is for"
WHO (roles)	Constrains "who should obey"
WHAT (applies_to)	Constrains "which output types are relevant"
WHEN (context)	Constrains "in what situations this is relevant"
WHAT HAPPENS (effect)	Constrains "consequences of violation"

Modern LLMs (GPT-4.1+) treat structure as "semantic boundary lines" rather than mere data formats. YAML keys become semantic anchors; hierarchy indicates priority and scope.

4.5.5 Normative Core Exclusivity

> **Policy or Constraint defines behavior. Everything else explains context.**

Other fields in a rule specification MUST NOT:

- Impose mandatory behaviors
- Override or weaken the Policy or Constraint
- Introduce alternative interpretations

This exclusivity ensures semantic purity and prevents governance-execution confusion.

4.6 BRA Boundaries and External Integration

BRA operates within clearly defined boundaries, delegating execution concerns to external architectures:

BRA is responsible for:

- Defining rules (development stage)
- Defining NNL / RNL
- Defining Ruleset classification
- Defining Governance Metadata
- Defining Decision Behavior Normative Categories
- Risk Potential (a priori assessment)
- Rule-level Boundary (static declaration)

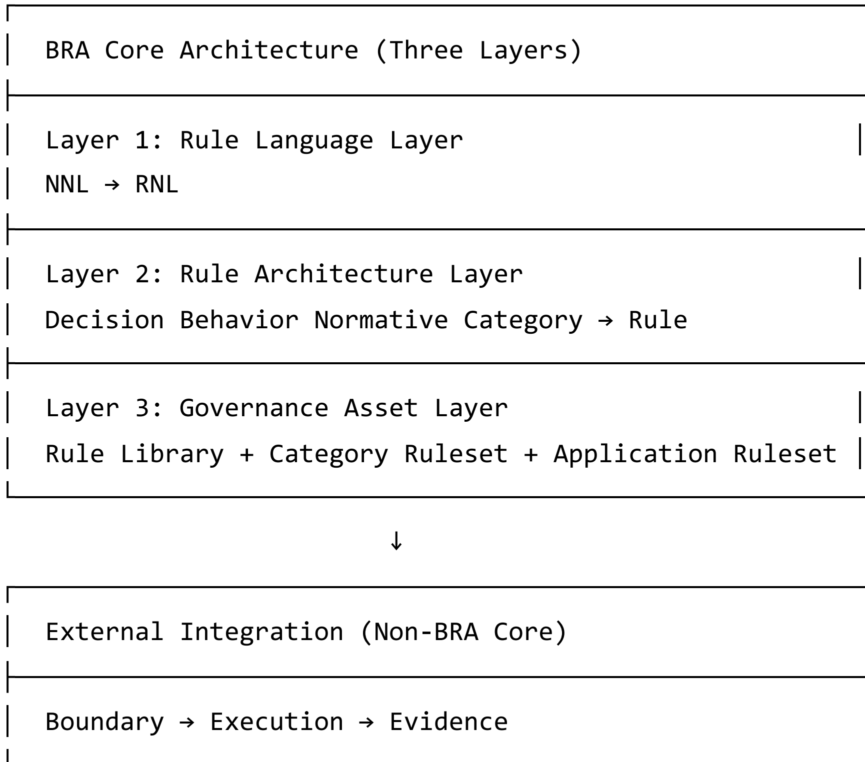
BRA is NOT responsible for:

- Runtime Enforcement (BCM)
- Risk Signal Detection (BCM)
- Evidence Generation (BCM)
- Execution Boundary Binding (Boundary Architecture)
- Validation (External Tool)
- Schema Management (External Tool)

This separation ensures BRA maintains architectural purity—it excels at rule definition without conflating definition with execution.

4.7 Layer Relationships and External Consumers

BRA's three layers feed external execution mechanisms:



Boundary Architecture consumes BRA's Rule-level Boundary declarations to perform dynamic Execution Boundary Binding.

Behavior Control Model (BCM) consumes BRA's rules to perform Runtime Enforcement, Risk Signal Detection, and Evidence Generation.

External Tools consume BRA's rule structures to perform Validation, Schema Management, and RNL processing.

This clear separation of concerns ensures each architectural layer can evolve independently while maintaining interoperability.

The following three sections (5, 6, and 7) detail each BRA layer, providing concrete specifications and design rationales.

5. Rule Language Layer

This section details the Rule Language Layer, which transforms natural language intent into enforceable normative rules through a clear semantic separation.

5.1 Semantic Separation: Intent Optimization ≠ Behavior Enforcement

The fundamental insight of BRA is that **natural language serves two incompatible purposes** in current AI systems:

1. **Intent Optimization**: Expressing what developers want AI to accomplish (goals, preferences, aspirations)
2. **Behavior Enforcement**: Defining what AI must or must not do (mandatory constraints, prohibitions)

When these purposes are conflated, AI cannot distinguish between "it would be nice if" and "it absolutely must." BRA resolves this by allocating different languages to each purpose:

- **Normative Natural Language (NNL)** for intent optimization
- **Rule Normative Language (RNL)** for behavior enforcement

5.2 Normative Natural Language (NNL)

Definition: NNL is human-readable natural language used to express task intent and operational descriptions.

Characteristics:

- **Non-enforceable**: Lacks normative force; AI treats content as guidance rather than mandate
- **High semantic flexibility**: Accommodates ambiguity, context-dependence, and subjective terms
- **Prone to semantic drift**: Interpretation varies across contexts and over time

Role: Serves exclusively as **prompt input**, not participating in behavior control.

Three Intent Types in NNL:

1. **Goal Clarification**: Clarifying what the developer truly wants
   - Example: "Help me implement login" → "I want a [login page] with username/password fields"
2. **Important Condition Signaling**: Emphasizing conditions that must not be ignored
   - Example: "Login must verify username and password"
3. **Expected Result or Behavior**: Describing what should happen on success or failure
   - Example: "If authentication fails, deny access"

Design Philosophy:

> **Important things should not be left for AI to guess.**

NNL reduces AI's guessing burden by making intent explicit, but it does not—and cannot—enforce behavior. That role belongs to RNL.

Comparison with Current Methods:

- **Prompt Engineering**: Relies entirely on NNL, lacking structured constraints
- **Custom Instructions**: NNL fragments without ensuring execution semantics
- **Rules Files**: Still express rules in NNL, without enforceability

5.3 Rule Normative Language (RNL)

Definition: RNL is a normative language for expressing executable behavior rules, using MUST/MUST NOT statements to define AI behavior policies or constraints.

Formal Specification:

- **MUST**: Mandatory policy (strong expectation)
- **MUST NOT**: Prohibitive constraint (absolute restriction)

RNL Design Principles:

1. **Keyword Restriction**: Only MUST / MUST NOT (no SHOULD / SHOULD NOT / MAY)
2. **Prohibition of Vague Terms**: No "may," "might," "try," "attempt," "possibly," "if possible"
3. **Sentence Structure**: `<Subject> <Normative Keyword> <Action> <Target> [Condition]`

Subject Selection Principle (Output-Oriented):

Rules SHOULD represent:

- Concrete artifacts (code, tests, documentation)
- Execution states (generation, validation, deployment)
- Observable results (logs, reports, metrics)

Rules SHOULD NOT represent:

- Conversational personas (helpful assistant, senior engineer)
- Implied speakers or listeners

Design Rationale:

> **Prefer describing what must exist or change, not who must perform the action.**

Preferred Examples:

```
Report MUST include a count of all warning-level findings.
Execution MUST stop when required input is missing.
Output MUST NOT contain data outside FILE_SCOPE.
```

Why RNL Works:

> **RNL does not make AI "more free" but makes AI "harder to deviate from boundaries."**

RNL constrains AI's interpretation space by reducing degrees of freedom. When AI encounters a MUST statement, it recognizes normative force; MUST NOT signals absolute prohibition.

5.4 NNL → RNL Semantic Convergence

The transformation from NNL to RNL represents **semantic convergence**—narrowing interpretation possibilities:

```
NNL (Broad Intent Space) → RNL (Narrow Normative Space)
```

Example Transformation:

NNL (Intent)	RNL (Enforcement)
"Prioritize code security"	"Generated code MUST use parameterized queries for all database operations"
"Be careful with user data"	"Output MUST NOT contain unmasked personally identifiable information"
"Write good tests"	"Each function MUST include at least one test case covering its primary execution path"

This convergence ensures:

1. **Explicit normative force**: MUST/MUST NOT leaves no ambiguity about mandate
2. **Behavioral specificity**: RNL describes concrete actions or states, not abstract qualities
3. **Machine interpretability**: Structured syntax reduces AI's interpretation burden

5.5 MUST vs. MUST NOT: Semantic Distinction and Design Evolution

Early Design (Historical Note): Initial BRA versions used "Constraint" as a top-level concept, not distinguishing MUST from MUST NOT.

Implementation Discovery: Actual deployment revealed fundamental differences between MUST and MUST NOT:

Characteristic	MUST (Policy)	MUST NOT (Constraint)
Trigger Observability	Not directly observable	Observable when triggered
Evidence Generation	No direct evidence	Generates violation evidence
AI Semantics	Expectation (what AI should achieve)	Restriction (what AI must avoid)
Enforcement Level	Medium (desired but not easily detectable if omitted)	High (immediately detectable if violated)
Execution Timing	AI should actively perform	AI absolutely must not perform
Audit Method	Check "whether achieved" (positive verification)	Check "whether violated" (negative verification)

Key Insight:

- **Policy (MUST)**: Defines behaviors AI should achieve. Triggering is implicit—if AI follows the policy, behavior aligns with expectations, but "compliance" itself is difficult to directly observe or produce evidence for. Enforcement level is medium.
- **Constraint (MUST NOT)**: Defines behaviors AI must avoid. Triggering is explicit—if AI violates, it can be detected, producing clear evidence (violation logs, behavior deviation records). Enforcement level is high.

Implications for Governance Systems:

This distinction enables governance systems to adopt different verification strategies:

Rule Type	Verification Strategy	Tool Support
Policy (MUST)	Output review, compliance check	Post-hoc verification tools
Constraint (MUST NOT)	Runtime detection, violation evidence	Runtime enforcement, evidence generation

Example:

- **Policy**: "AI MUST include tests with generated code" → Verify by checking if test files exist (post-generation)
- **Constraint**: "AI MUST NOT delete files without explicit user request" → Detect immediately if deletion attempted (runtime)

5.6 Language Characteristics

RNL exhibits three critical characteristics:

1. **Readability**: Human-readable for rule authoring and review
2. **Parsability**: Machine-parseable for validation and processing
3. **Composability**: Composable into rulesets without semantic conflicts

These characteristics ensure RNL serves both human governance needs and machine processing requirements.

5.7 Comparison with Current Methods

Method	Language Type	Normative Force	Enforceability
Prompt Engineering	NNL only	None	Low (guidance only)
Custom Instructions	NNL fragments	Weak	Low (aspirational)
Rules Files	NNL in structured format	Weak to moderate	Low to moderate
Policy-as-Code	DSL or code	Strong	High (but post-hoc)

Method	Language Type	Normative Force	Enforceability
LLM Guardrails	Validation rules	Strong	High (but output-focused)
BRA (RNL)	**Normative language**	**Strong (MUST/MUST NOT)**	**High (pre-execution)**

BRA's RNL combines the **accessibility of natural language** with the **enforceability of formal constraints**, positioning it uniquely for AI behavior governance.

5.8 NNL → RNL Transformation Guidelines

Rule Authoring Process:

1. **Express Intent in NNL**: "I want secure database access"
2. **Identify Normative Core**: What must be enforced? "All database queries must use parameterized statements"
3. **Classify as Policy or Constraint**: Is this MUST or MUST NOT? "MUST use parameterized queries" (Policy)
4. **Write in RNL**: "All generated database queries MUST use parameterized statements"
5. **Add Governance Metadata**: Owner, version, intent, risk potential

This systematic process ensures semantic clarity from intent to enforcement.

6. Rule Architecture Layer

This section elaborates on the Rule Architecture Layer, detailing the structure of Rules and Rulesets, including the Three Iron Laws governing Category Rulesets and Application Rulesets.

6.1 Rule as Structured Governance Artifact

Definition: A Rule is a structured governance unit consisting of RNL and governance metadata, representing the smallest manageable unit in BRA.

Formal Representation:

```
Rule = ⟨RNL, Metadata, Boundary_Declaration⟩
```

Classification:

- **Policy**: MUST (mandatory behavior policy)
- **Constraint**: MUST NOT (prohibitive behavior constraint)

6.2 Rule Structure Components

6.2.1 Normative Core (RNL)

The normative core is the **only enforceable component** of a rule, containing either a Policy or Constraint statement in RNL.

Policy (MUST) Example:

```
policy: >
  Generated code MUST include corresponding test cases
  for all new functions.
```

Constraint (MUST NOT) Example:

```
constraint: >
  AI MUST NOT execute commands that modify system configurations
  without explicit user approval.
```

Exclusivity Principle: The normative core defines behavior; all other components provide context.

6.2.2 Execution View (Optional)

Purpose: Defines the **interpretation viewpoint** the AI should adopt when processing this rule.

Characteristics:

- It IS: A perspective or lens (e.g., security, quality assurance, architecture)
- It IS NOT: A responsibility assignment, role-playing instruction, or human-in-the-loop declaration

Example:

```
execution_view: security_engineer
```

Design Rationale: Reduces rule interpretation ambiguity (e.g., security vs. QA perspective) without affecting execution control. It complements Decision Behavior Normative Categories without

overlapping.

6.2.3 Governance Metadata

Purpose: Provides descriptive, non-executable context for governance, auditability, and lifecycle management.

Core Metadata Fields:

1. **version**: Version number supporting rule evolution
2. **status**: Lifecycle state (draft / active / deprecated)
3. **owner**: Responsible party for the rule
4. **intent (WHY)**: Governance rationale for the rule's existence
5. **responsibilities (WHO)**: Stakeholder accountability (non-execution roles)
6. **risk_potential**: A priori risk assessment (BRA scope)
    - impact_score: Potential impact rating (1-5)
    - rationale: Impact reasoning
7. **applicability_context (WHEN)**: Governance-relevant applicability scenarios
    - Must not be interpreted as trigger conditions or execution logic
8. **governance_impact (WHAT HAPPENS)**: Expected governance-level outcomes and evidence expectations

Risk Potential (Three-Layer Risk Model):

BRA uses **risk_potential** for a priori assessment, distinguishing it from runtime risk signals and realized risks:

Layer	Owner	Description
Risk Potential	BRA (Rule metadata)	A priori assessment: if something goes wrong, how large is the potential impact?
Risk Signal	BCM	Execution stage: risk symptom detection
Realized Risk	Governance / Audit	Audit stage: actual consequences caused

Example:

```yaml
risk_potential:
  impact_score: 4
  rationale: affects_release_safety
```

Governance Metadata Design Principle:

> **Governance metadata exists to support future governance utilization but does not define governance workflows, evidence generation logic, or enforcement mechanisms.**

6.2.4 Boundary Declaration (Static)

Purpose: Declares the **decision scope** in which the rule is semantically applicable.

Characteristics:

- Declarative (describes applicability)
- Non-executive (does not trigger execution)
- Non-normative (does not enforce behavior)

Example:

```yaml
boundary:
  artifact_type: source_code    # Applicable artifact type
  source_ref: STS-JWT-001      # Referenced specification (optional)
```

BRA's Role: Rule-level Boundary provides **static semantic applicability declaration**, supporting rule self-containment for Library classification and query.

Important: This is static declaration within BRA's scope. **Execution Boundary Binding** is handled by **Boundary Architecture** through dynamic runtime scope control. This dual-layer design balances rule self-containment with execution flexibility:

- Rule-level Boundary (BRA): Static semantic applicability declaration
- Execution Boundary (Boundary Architecture): Dynamic execution scope binding

6.3 Rule Design Principles

Principle 1: Single Rule, Single Normative
Each rule contains exactly one normative statement (Policy or Constraint).

Principle 2: Normative Over Control

Rules define boundaries, not procedures.

Principle 3: Declarative, Not Procedural

Rules describe "what state must be maintained," not "how to act."

Principle 4: Governance-Aware but Execution-Neutral

Governance metadata supports auditability but must not influence execution behavior.

Principle 5: Representation-Agnostic

YAML/JSON schemas are reference representations, not the standard itself. Rules can be expressed in different formats while preserving semantic equivalence.

6.4 Rule as Governance Atom

Core Concept:

> **There is no governance without a rule. There is no rule without a policy or constraint.**

A rule is the **smallest unit that carries governance meaning**, encapsulating:

- One explicit behavioral policy (MUST) or constraint (MUST NOT)
- Its applicable scope
- Its governance interpretation context
- Its accountability and audit expectations

Minimal Rule Concept:

```yaml
rule:
  id: CODE-TEST-001
  policy: >
    AI MUST include at least one test case
    for each newly generated function.
```

This minimal rule already possesses:

- Valid AI constraint force
- Direct copyability into prompts
- Human understandability
- System interpretability

Practical Implication: Rules can exist independently without complex metadata, making BRA adoptable incrementally.

6.5 Ruleset: Pure Rule Collection

Definition: A Ruleset is a named collection of rule references that groups rules for governance purposes.

Mental Model:

> A Ruleset is a **governance envelope** that provides a **shared identity** for multiple rules without changing any rule's substantive meaning.

Ruleset Design Principles:

1. **Rules Remain Atomic**: Rulesets MUST NOT alter rule semantics
2. **Aggregation Without Interpretation**: Rulesets group rules without interpreting them
3. **Governance Over Structure**: Rulesets exist to support governance, audit, and lifecycle management
4. **Execution Neutrality**: Rulesets carry no execution semantics
5. **Reference Not Duplication**: Rules are referenced by ID, not embedded or rewritten

What Rulesets ARE:

- Named collections of rule references
- Constraint bundles within governance scopes
- Version control units

What Rulesets ARE NOT:

- New constraint definitions
- Rule semantic modifications
- Rule execution or evaluation engines
- Rule conflict resolvers

6.6 Two Types of Rulesets

BRA defines two distinct ruleset types with different purposes and constraints:

6.6.1 Category Ruleset

Definition: A collection of all rules based on the same Decision Behavior Normative Category.

Mathematical Representation:

```
Category_Ruleset : Behavior_Category → 𝒫(Rule)

Category_Ruleset(C) = {rule ∈ Library | constraint_type(rule) = C}

Cardinality: Category_Ruleset : Rule = 1:N
```

Characteristics:

- Different Category Rulesets are mutually exclusive (non-overlapping)
- The union of all Category Rulesets constitutes the complete Rule Library
- Category Rulesets only classify rules; they do not compose rules

Example:

```
C_tr = "Traceability"
Category_Ruleset(C_tr) = {
  rule_must_preserve_execution_trail,
  rule_must_not_break_audit_chain,
  rule_must_include_trace_metadata,
  ...
}
```

6.6.2 Application Ruleset

Definition: A rule collection designed based on arbitrary application principles, such as workflow-based, domain-based, risk-based, compliance-based, or any custom principles.

Mathematical Representation:

```
Application_Ruleset : Principle → 𝒫(Rule)

Application_Ruleset(P) = {rule ∈ Library | satisfies(rule, P)}

Cardinality: Rule : Application_Ruleset = N:N
```

Principle Types (Unlimited):

- **Workflow-based**: Composed by enterprise operation flows (development, testing, deployment)
- **Domain-based**: Integrated by application domains (security, testing, documentation)

- **Risk-based**: Filtered by risk levels (high, medium, low)
- **Compliance-based**: Composed by regulatory requirements (GDPR, HIPAA, SOX)
- **Custom**: Enterprise/industry-specific principles

Characteristics:

- One rule can be composed into multiple Application Rulesets
- One Application Ruleset can combine multiple rules
- May intersect with Category Rulesets, but not necessarily

Example:

```
P_dev = "Development Workflow"
Application_Ruleset(P_dev) = {
  rule_version_control,
  rule_code_review,
  rule_must_use_linter,      // From TR Category
  rule_must_include_tests,   // From VR Category
  ...
}
```

6.7 Three Iron Laws of Category vs. Application Rulesets

To ensure BRA's architectural clarity and flexibility, we define three iron laws:

Iron Law 1: Category MUST NOT Define Rule Composition

Implications:

- Categories only classify rules based on Decision Behavior Normatives
- Categories MUST NOT define or recommend any rule compositions
- Rule composition authority belongs exclusively to Application Rulesets

Example Violation: A "Security Category" that prescribes "always combine with Testing Category" violates this iron law.

Correct Usage: The "Security Category" simply contains all security-related rules; Application Rulesets decide which security rules to combine with which testing rules.

Iron Law 2: Rule Composition MUST Occur Only in Application Ruleset

Implications:

- Free rule composition occurs only at the Application Ruleset level
- Application Rulesets can combine rules based on arbitrary principles (workflow, domain, risk, compliance, etc.)
- One rule can be composed into multiple Application Rulesets

Example: A "Secure Code Review Application Ruleset" might combine:

- AR (Artifact Isolation) rules
- BD (Boundary & Stop) rules
- CN (Constraint Neutrality) rules
- TR (Traceability) rules

This cross-category composition is permitted—and encouraged—at the Application level.

Iron Law 3: Rules MUST Remain Atomic and Independent

Implications:

- Each rule must maintain atomicity and independence
- Rules MUST NOT know which Rulesets they belong to (unidirectional reference: Ruleset → Rule)
- Rules MUST NOT have mutual exclusion, dependency, or grouping relationships (such relationships, if needed, should be defined at the Application Ruleset level)

Example Independence:

- CODE-BD-01 (Prohibit file deletion)
- CODE-BD-02 (Prohibit system file modification)

These rules are independent; either can be used alone without requiring the other.

Consequence: If dependencies between rules are needed, they must be defined in Application Rulesets, not in rules themselves.

6.8 Key Conclusion: Responsibility Boundaries

The Three Iron Laws establish clear responsibility boundaries:

```
Categories describe Decision Behavior Normatives (classifying behavior constraints)
Rules enforce policies and constraints (executing mandates and prohibitions)
Applications decide composition (determining rule combinations)
```

Responsibility Boundary Table:

Level	Responsibility	Must Not Involve
Decision Behavior Normative Category	Behavior normative classification	Rule composition, application logic
Rule	Define single policy or constraint	Composition relationships, other rules
Application Ruleset	Compose rules based on arbitrary principles	Define new policies or constraints

6.9 Ruleset Versioning and Evolution

Rulesets support version control and lifecycle management, enabling:

1. **Version Tracking**: Changes to ruleset membership are versioned
2. **Lifecycle States**: Draft → Active → Deprecated transitions
3. **Backward Compatibility**: Application Rulesets can specify version constraints
4. **Change Auditing**: Who changed what, when, and why

Example Versioning:

```yaml
ruleset:
  id: APP-SECURE-DEV-001
  version: 2.1.0
  status: active
  rules:
    - CODE-AR-001@1.0.0
    - CODE-BD-001@2.0.0
    - SPEC-TR-003@1.2.0
```

This versioning ensures reproducible rule applications and supports governance audits.

6.10 Minimal Constraint Principle

Requirement: Every Ruleset (both Category and Application) MUST contain at least one Constraint (MUST NOT) rule.

Rationale:

- Constraints (MUST NOT) are directly observable and enforceable
- Policies (MUST) rely on post-hoc verification

- At least one observable constraint ensures each ruleset has a behavioral boundary

Example:
A "Code Quality Application Ruleset" might include:

- Policies: "MUST use meaningful variable names", "MUST document public APIs"
- Constraint: "MUST NOT commit code without passing tests" ← Ensures behavioral boundary

6.11 Comparison with Current Methods

Method	Rule Structure	Composition Mechanism	Governance Capability
Prompt Engineering	None (monolithic text)	None	Low (no metadata)
Custom Instructions	Fragmented text	None	Low (no versioning)
Rules Files	Semi-structured text	Single file per repo	Low to medium
Policy-as-Code	Code or DSL	Limited	Medium (requires technical expertise)
BRA	**Atomic rules with metadata**	**Category + Application Rulesets**	**High (versioning, ownership, lifecycle)**

BRA's Rule Architecture Layer provides systematic structure, composition flexibility, and governance capability unavailable in current approaches.

7. Governance Asset Layer

This section elaborates on the Governance Asset Layer, explaining how rules are organized into classifiable, shareable, governable assets through Decision Behavior Normative Categories and Application Rulesets.

7.1 Layer Purpose

The Governance Asset Layer transforms rules from ephemeral artifacts into **accumulating governance assets**. It provides mechanisms for:

1. **Classification**: Organizing rules by Decision Behavior Normative Categories
2. **Composition**: Combining rules through Application Rulesets
3. **Sharing**: Building cross-project and cross-organization rule libraries
4. **Governance**: Managing rule lifecycles, ownership, and accountability

7.2 Decision Behavior Normative Category

7.2.1 Core Concept

Definition: A Decision Behavior Normative Category is a classification mechanism for **AI decision behavior normatives**, not a classification of AI behaviors themselves. It defines **decision behavior normative types** first, then derives corresponding rules.

Key Characteristics:

- **Classification Core**: Based on "types of normatives on decision behaviors," not on application domains or simple behavior types
- **Category-Rule Relationship**: One Decision Behavior Normative Category corresponds to one Category Ruleset
- **Category-Rule Cardinality**: Category Ruleset : Rule = 1:N (each rule belongs to only one Category)
- **Constraint Requirement**: Each Category MUST contain at least one Constraint (MUST NOT) rule, ensuring observable behavioral boundaries

7.2.2 Design Philosophy

Decision Behavior Normative-Based vs. Domain-Based:

Approach	Sequence	Classification Basis
Decision Behavior Normative-Based (BRA)	First define decision behavior normative types, then derive rules	AI decision behavior normative types (e.g., artifact isolation, boundary control)

Approach	Sequence	Classification Basis
Domain-Based (Traditional)	First design rules, then determine applicable domains	Application domains (e.g., healthcare, finance, manufacturing)

Why Decision Behavior Normative-Based?:

1. **Semantic Precision**: Rules precisely correspond to AI decision constraint points
2. **Reusability**: Same decision behavior normative applies across multiple domains
3. **Governance Clarity**: Clear mapping between governance intent and rule enforcement
4. **Composition Flexibility**: Application Rulesets can combine rules from different categories without semantic conflicts

7.2.3 Decision Behavior Normative Category Examples

BRA defines the following primary categories, each representing a fundamental type of AI decision behavior normative:

AR (Artifact Isolation) - Artifact Isolation Decision Normative

Focus: Normatives on AI's decisions regarding artifacts (code, specifications, documents)

Core Constraints:

- Artifacts MUST maintain independence
- Artifacts MUST NOT cross-boundary contaminate
- Artifact modifications MUST be traceable

Example Rules:

- `CODE-AR-001` : Generated code MUST NOT directly modify files outside the designated scope
- `SPEC-AR-001` : Specification documents MUST be independently versionable
- `VIEWPOINT-AR-001` : Viewpoint models MUST NOT implicitly depend on undocumented artifacts

When to Use: When governance requires strict artifact boundaries, preventing cascading modifications across unrelated components.

BD (Boundary & Stop) - Boundary and Stop Decision Normative

Focus: Normatives on AI's decisions regarding execution boundaries and stopping conditions

Core Constraints:

- AI MUST operate within defined boundaries
- AI MUST stop when stop conditions trigger
- AI MUST NOT proceed when required inputs are missing

Example Rules:

- `CODE-BD-001` : Execution MUST stop when encountering unresolvable type errors
- `SPEC-BD-001` : Specification generation MUST halt when requirements are contradictory
- `CODE-BD-002` : AI MUST NOT execute commands modifying system configurations without explicit approval

When to Use: When governance requires clear execution boundaries and fail-safe stopping mechanisms.

CN (Constraint Neutrality) - Constraint Neutrality Decision Normative

Focus: Normatives ensuring AI's decision constraints are unambiguous and value-neutral

Core Constraints:

- Constraints MUST be unambiguous
- Constraints MUST NOT contain implicit assumptions
- Constraints MUST NOT embed hidden value judgments

Example Rules:

- `CODE-CN-001` : Generated validation logic MUST explicitly state all acceptance criteria
- `SPEC-CN-001` : Requirements MUST NOT use subjective terms without operational definitions
- `VIEWPOINT-CN-001` : Architecture decisions MUST document rationale without prescribing implementation details

When to Use: When governance requires transparent, interpretable, and bias-free constraints.

TR (Traceability) - Traceability Decision Normative

Focus: Normatives on AI's decisions regarding execution trails and audit chains

Core Constraints:

- AI MUST preserve execution trails
- AI MUST NOT break audit chains
- AI MUST include trace metadata in outputs

Example Rules:

- `CODE-TR-001` : Generated code MUST include comments linking to requirement IDs
- `SPEC-TR-001` : Specification documents MUST trace to source requirements
- `CODE-TR-002` : AI MUST log all file modification operations with timestamps and rationale

When to Use: When governance requires comprehensive auditability and change tracking.

LG (Logging & Report) - Logging and Report Decision Normative

Focus: Normatives on AI's decisions regarding logging behavior and report generation

Core Constraints:

- AI MUST log critical operations
- AI MUST NOT omit required reports
- Logs MUST be structured and queryable

Example Rules:

- `CODE-LG-001` : Generated code MUST log all exception cases with context
- `SPEC-LG-001` : Specification changes MUST be accompanied by change logs
- `CODE-LG-002` : AI MUST NOT suppress warning messages without explicit justification

When to Use: When governance requires operational visibility and reporting compliance.

ST (Structural Change) - Structural Change Decision Normative

Focus: Normatives on AI's decisions regarding system structure modifications

Core Constraints:

- Structural changes MUST undergo review
- AI MUST NOT perform destructive refactoring
- Structural changes MUST be reversible

Example Rules:

- `CODE-ST-001` : Refactoring operations MUST preserve existing interfaces
- `SPEC-ST-001` : Architecture changes MUST be approved before implementation
- `CODE-ST-002` : AI MUST NOT delete modules without explicit migration plans

When to Use: When governance requires controlled architectural evolution and change risk management.

TI (Test Integrity) - Test Integrity Decision Normative

Focus: Normatives on AI's decisions regarding test environment integrity

Core Constraints:

- Tests MUST maintain independence
- AI MUST NOT introduce inter-test dependencies
- Test environments MUST be isolated

Example Rules:

- `CODE-TI-001` : Generated test cases MUST be independently executable
- `SPEC-TI-001` : Test specifications MUST NOT assume execution order
- `CODE-TI-002` : AI MUST NOT use shared mutable state across test cases

When to Use: When governance requires reliable, reproducible, and maintainable test suites.

7.2.4 Naming Conventions and Cross-Domain Consistency

Format: `<DOMAIN>-<CATEGORY>-<SEQUENCE>`

Examples:

- `CODE-AR-001` : Code domain, Artifact Isolation category, rule 001
- `SPEC-BD-001` : Specification domain, Boundary & Stop category, rule 001
- `VIEWPOINT-CN-001` : Viewpoint domain, Constraint Neutrality category, rule 001

Cross-Domain Consistency: The same Decision Behavior Normative Category (e.g., BD) applies consistently across different domains (CODE, SPEC, VIEWPOINT), ensuring semantic uniformity.

Governance Benefit: Organizations can govern specific decision behavior normative types (e.g., all Boundary & Stop rules) across all domains, or govern all rules within a specific domain.

7.2.5 Category Ruleset Structure

Each Decision Behavior Normative Category corresponds to a Category Ruleset:

```
Category_Ruleset(AR) = {
  CODE-AR-001: Prohibit cross-scope file modifications,
  SPEC-AR-001: Ensure specification independence,
  VIEWPOINT-AR-001: Prevent undocumented dependencies,
  ...
}
```

Mutual Exclusivity: Category Rulesets are mutually exclusive—a rule belongs to exactly one Category.

Union Property: The union of all Category Rulesets forms the complete Rule Library.

7.3 Rule Library

7.3.1 Definition

Definition: A Rule Library is a centralized storage and management system for all rules, serving as an accumulatable governance asset.

Mathematical Representation:

```
Library = ∪_{C ∈ Behavior_Category} Category_Ruleset(C)
```

7.3.2 Characteristics

1. **All Rules Collection**: The library contains all defined rules
2. **Cross-Boundary Sharing**: Can span industries, enterprises, and projects
3. **Long-Term Accumulation**: Supports gradual governance asset accumulation
4. **Version Management**: Tracks rule evolution over time

7.3.3 Evolution Path

Rule Libraries evolve through three stages:

Stage 1: Rule Pool

- **Scope**: Single project or team
- **Sharing**: Limited to immediate stakeholders
- **Governance**: Informal, ad-hoc management

Stage 2: Repository

- **Scope**: Organization-wide
- **Sharing**: Cross-team and cross-project
- **Governance**: Formal version control, ownership, lifecycle management

Stage 3: Marketplace

- **Scope**: Cross-organization
- **Sharing**: Public or industry consortium
- **Governance**: Standardized schemas, quality metrics, certification processes

Example Evolution:

- A team develops security rules for their project (Pool)
- The organization adopts these rules across all projects (Repository)
- The organization publishes rules to an industry consortium (Marketplace)

7.3.4 Library Structure

Organization: Rule Libraries are organized by Decision Behavior Normative Categories as primary classification:

```
Rule_Library/
├── AR_Artifact_Isolation/
│   ├── CODE-AR-001.yaml
│   ├── SPEC-AR-001.yaml
│   └── VIEWPOINT-AR-001.yaml
├── BD_Boundary_Stop/
│   ├── CODE-BD-001.yaml
│   ├── SPEC-BD-001.yaml
│   └── CODE-BD-002.yaml
├── CN_Constraint_Neutrality/
│   └── ...
├── TR_Traceability/
│   └── ...
├── LG_Logging_Report/
│   └── ...
├── ST_Structural_Change/
│   └── ...
└── TI_Test_Integrity/
    └── ...
```

Access Patterns:

- Query by Category: "Get all AR rules"
- Query by Domain: "Get all CODE rules"
- Query by Risk: "Get all high-risk potential rules"
- Query by Owner: "Get all rules owned by Security Team"

7.4 Application Ruleset

7.4.1 Definition

Definition: An Application Ruleset is a rule collection based on arbitrary application principles, such as workflow-based, risk-based, compliance-based, or any custom principles.

Mathematical Representation:

```
Application_Ruleset : Principle → 𝒫(Rule)

Application_Ruleset(P) = {rule ∈ Library | satisfies(rule, P)}

Cardinality: Rule : Application_Ruleset = N:N
```

7.4.2 Composition Principle Examples

Workflow-Based Application Rulesets:

Rules composed according to development workflow stages:

```yaml
ruleset:
  id: APP-DEV-WORKFLOW-001
  name: Development Workflow
  principle: workflow-based
  rules:
    # Planning stage
    - SPEC-BD-001@1.0.0  # Requirements must be contradictory-free
    - SPEC-TR-001@1.0.0  # Trace to source requirements
    # Coding stage
    - CODE-AR-001@1.0.0  # No cross-scope modifications
    - CODE-TI-001@1.0.0  # Independent test cases
    # Review stage
    - CODE-TR-001@1.0.0  # Trace to requirement IDs
    - CODE-LG-001@1.0.0  # Log exception cases
```

Risk-Based Application Rulesets:

Rules filtered by risk levels:

```yaml
ruleset:
  id: APP-HIGH-RISK-001
  name: High-Risk Operations
  principle: risk-based
  risk_level: high
  rules:
    - CODE-BD-002@1.0.0  # No system config modifications without approval
    - CODE-ST-002@1.0.0  # No module deletion without migration plans
    - SPEC-ST-001@1.0.0  # Architecture changes require approval
```

Compliance-Based Application Rulesets:

Rules composed by regulatory requirements:

```yaml
ruleset:
  id: APP-GDPR-COMPLIANCE-001
  name: GDPR Compliance
  principle: compliance-based
  regulation: GDPR
  rules:
    - CODE-AR-001@1.0.0  # Data isolation constraints
    - CODE-TR-001@1.0.0  # Data processing traceability
    - CODE-LG-001@1.0.0  # Data access logging
    - SPEC-CN-001@1.0.0  # Unambiguous consent criteria
```

Domain-Specific Application Rulesets:

Rules tailored for specific domains:

```yaml
ruleset:
  id: APP-SECURE-CODE-001
  name: Secure Code Generation
  principle: domain-based
  domain: security
  rules:
    - CODE-AR-001@1.0.0  # Artifact isolation
    - CODE-BD-001@1.0.0  # Boundary constraints
    - CODE-CN-001@1.0.0  # Constraint neutrality
    - CODE-TR-001@1.0.0  # Traceability
```

7.4.3 Relationship with Category Rulesets

Intersection Possibility:

- The same rule can belong to a Category Ruleset AND multiple Application Rulesets
- This intersection is not mandatory—it depends on specific composition needs

Example:

- `CODE-TR-001` (traceability rule) belongs to **TR Category Ruleset**
- The same rule is composed into:
  - **Development Workflow Application Ruleset** (workflow principle)
  - **GDPR Compliance Application Ruleset** (compliance principle)
  - **Secure Code Generation Application Ruleset** (domain principle)

Cross-Category Composition:
Application Rulesets can combine rules from different Categories:

```
APP-SECURE-DEV-001 contains:
- From AR Category: CODE-AR-001
- From BD Category: CODE-BD-001
- From CN Category: CODE-CN-001
- From TR Category: CODE-TR-001
```

This cross-category composition is the core strength of Application Rulesets—enabling flexible rule combinations without violating Category boundaries.

7.5 Governance Mechanisms

7.5.1 Policy vs. Constraint Semantic Distinction

Implementation Validation: The distinction between Policy (MUST) and Constraint (MUST NOT) is not merely theoretical—it is validated through implementation:

Characteristic	Policy (MUST)	Constraint (MUST NOT)
Observability	Low (compliance is implicit)	High (violations are explicit)
Evidence Generation	Requires post-hoc collection	Generates violation evidence automatically
Enforcement Strategy	Output review, compliance check	Runtime detection, immediate enforcement
Audit Approach	Positive verification (check if achieved)	Negative verification (check if violated)

Governance Implication:

- Constraints are **immediately governable** through runtime enforcement
- Policies require **governance workflows** for post-hoc verification

7.5.2 Rule Lifecycle Management

Lifecycle States:

1. **Draft**: Rule is under development, not yet enforceable
2. **Active**: Rule is enforced in production environments
3. **Deprecated**: Rule is phased out but retained for audit purposes
4. **Retired**: Rule is archived and no longer appears in active rulesets

State Transitions:

```
Draft → Active → Deprecated → Retired
```

Governance Controls:

- State transitions require approval from designated owners
- Active rules cannot be deleted (only deprecated)
- Deprecated rules trigger warnings in ruleset composition

7.5.3 Versioning

Semantic Versioning: Rules use semantic versioning (MAJOR.MINOR.PATCH)

- **MAJOR**: Breaking semantic changes (e.g., changing MUST to MUST NOT)
- **MINOR**: Backward-compatible enhancements (e.g., adding metadata)
- **PATCH**: Bug fixes, clarifications

Ruleset Versioning: Rulesets track rule versions:

```
ruleset:
  id: APP-SECURE-DEV-001
  version: 2.1.0
  rules:
    - CODE-AR-001@1.2.0
    - CODE-BD-001@2.0.0
    - CODE-CN-001@1.0.0
```

Locking: Application Rulesets can lock to specific rule versions, ensuring reproducibility.

7.5.4 Sharing Modes

Intra-Organization:

- Teams share rules through centralized repositories
- Governance: Internal ownership, version control, lifecycle management
- Access: Role-based permissions (author, reviewer, administrator)

Cross-Organization:

- Organizations exchange rules through federated repositories
- Governance: Consortium agreements, certification processes
- Access: Licensing models, attribution requirements

Platform Ecosystem:

- Public marketplaces for rule sharing
- Governance: Community ratings, quality metrics, security audits
- Access: Open source, commercial licensing, or hybrid models

Evolution Trajectory:

```
Team Ruleset → Organization Repository → Industry Marketplace → Global Standard
```

7.6 Comparison with Current Methods

Method	Library Concept	Sharing Mechanism	Governance Capability
Prompt Engineering	None	None	None
Custom Instructions	Fragmented configs	Manual copy-paste	Low (no versioning)
Repository Rules Files	Single-repo rules	Git-based sharing	Low to medium
Policy-as-Code	Shared policy repos	Git, package managers	Medium
BRA	**Rule Library with Category + Application Rulesets**	**Multi-level sharing (team → org → marketplace)**	**High (lifecycle, versioning, ownership)**

7.7 Summary: Governance Asset Layer Benefits

The Governance Asset Layer transforms rules into manageable assets:

1. **Classifiability**: Rules organized by Decision Behavior Normative Categories for semantic clarity
2. **Composability**: Application Rulesets enable flexible rule combinations
3. **Shareability**: Rule Libraries support cross-project and cross-organization reuse
4. **Governability**: Lifecycle management, versioning, ownership ensure accountability
5. **Accumulability**: Rules become long-term governance assets, not ephemeral fragments

This layer establishes the foundation for **governance as asset**—treating behavior rules as valuable, manageable, shareable resources that appreciate over time through refinement and reuse.

8. External Integration and Architecture Ecosystem

This section discusses how BRA integrates with external execution mechanisms and the broader architecture ecosystem's responsibility separation and extension possibilities.

8.1 Architecture Ecosystem Responsibility Separation

BRA operates within a larger AI behavior governance ecosystem, collaborating with complementary architectures:

Core Design Philosophy:

> **BRA should only be responsible for "defining behavior constraints," not "executing, judging, governing, or validating" them.**

Architecture Division of Labor:

```
BRA: Define rules (development stage)
Boundary Architecture: Bind scopes (application stage)
BCM: Execute rules (runtime stage)
External Tools: Validate and manage (tool stage)
```

8.2 Architecture Ecosystem Responsibility Separation

8.2.1 BRA's Responsibility

Scope: Development-stage rule definition

Core Functions:

- Define Rule Normative Language (RNL) and Normative Natural Language (NNL)
- Define atomic rules with governance metadata
- Define Decision Behavior Normative Categories
- Define Category Rulesets and Application Rulesets
- Define Rule Libraries for asset accumulation

Key Deliverable: Structured, governable rule definitions ready for consumption by execution mechanisms.

8.2.2 Boundary Architecture

Core Responsibility:

- Execution Boundary Binding (runtime scope binding)
- Runtime Context Binding (execution environment binding)
- Behavior Scope Binding (behavior range binding)
- Dynamic scope management

Relationship with BRA:

- BRA provides Rule-level Boundary (static declaration)
- Boundary Architecture provides Execution Boundary (dynamic binding)
- Dual-layer design separates concerns

Dual-Layer Boundary Model:

1. Rule-level Boundary (Static Declaration - BRA Scope):

```yaml
boundary:
  artifact_type: source_code
  source_ref: STS-JWT-001
```

- **Purpose**: Rule self-containment for Library classification and querying
- **Semantics**: Semantic applicability (where the rule is semantically relevant)

2. Execution Boundary (Dynamic Binding - Boundary Architecture):

- **Location**: External Boundary controller in Layer 4
- **Purpose**: Execution flexibility for different scenarios
- **Semantics**: Runtime scope (which rules apply to this execution)
- **Functions**:
  - Specify applicable Rulesets for this execution (single or multiple)
  - Runtime Context Binding (environment, user, project context)
  - Behavior Scope constraints (which operations are permitted)

Key Definition:

```
Rule boundary defines semantic applicability
Execution boundary defines runtime binding
```

Design Rationale:

- Balances rule self-containment with execution flexibility
- Rules can be reused across different projects and scenarios
- Static declaration supports governance; dynamic binding supports execution

8.2.3 Behavior Control Model (BCM)

Core Responsibility:

- Runtime Enforcement (runtime-stage forced execution)
- Risk Signal Detection (risk symptom detection)
- Evidence Generation (execution evidence generation)
- Constraint (MUST NOT) immediate execution
- Policy (MUST) post-hoc verification

Three-Layer Risk Model:

```
Risk Potential (BRA) → Risk Signal (BCM) → Realized Risk (Governance)
A priori assessment   → Runtime detection → Actual consequences
```

Policy vs. Constraint Execution Differences:

Type	Execution Approach	Evidence	BCM Responsibility
Policy (MUST)	Expect execution, post-hoc verification	No direct evidence	Post-hoc verification
Constraint (MUST NOT)	Immediate execution, violation detection	Generates violation evidence	Runtime enforcement

Example:

- **Policy**: "AI MUST include tests with generated code" → BCM verifies test existence after generation
- **Constraint**: "AI MUST NOT delete files without explicit user request" → BCM detects deletion attempts immediately and blocks execution

Relationship with BRA:

- BRA defines rules with risk potential (a priori assessment)

- BCM detects risk signals during execution (runtime detection)
- BCM enforces constraints and verifies policies

8.2.4 External Tools

Core Responsibility:

- Validation (validation tools)
- Schema Management (schema management)
- RNL Tools (RNL parsing and validation)
- Evidence Structure Definition - GES (evidence structure definition)

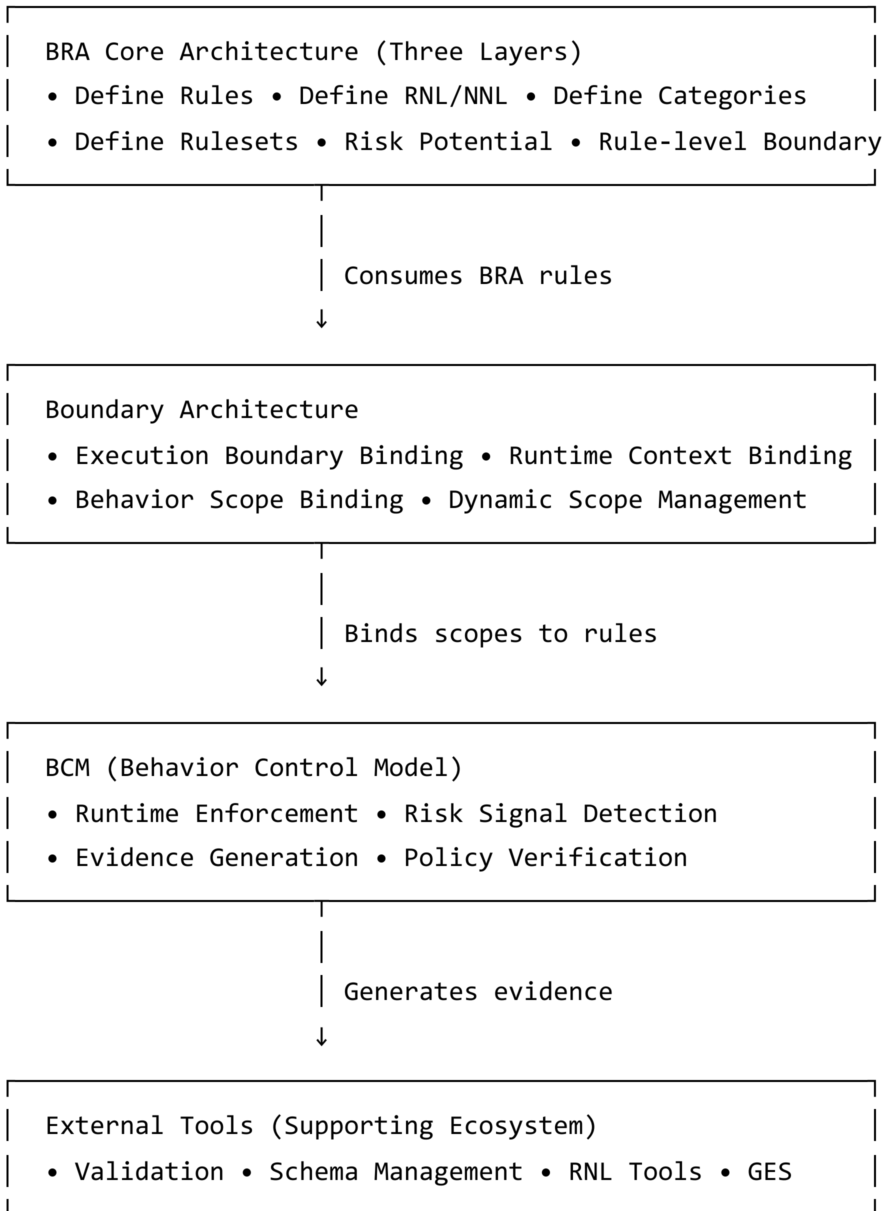
Validation Three-Layer Model:

1. **Artifact Structure Validation**:
   - Check YAML/JSON basic structure
   - Use JSON Schema validators
   - Result: pass/fail
   - Does not touch RNL or semantics
2. **RNL Conformance Validation**:
   - Check if `rule.policy` or `rule.constraint` conforms to RNL representation schema
   - Result: warning or fail (depending on mode)
   - Not a rule prerequisite, but a quality gate
   - Must support configurable strict/advisory modes
3. **Authoring Quality Advisory**:
   - Help human authors understand field alignment, old/new format conversion, expression clarity
   - Result: never fail, only produce warnings/info
   - Best position for AI as "assistant tool"

Relationship with BRA:

- These tools assist BRA development and management
- But they are not part of BRA core architecture
- Can evolve and be replaced independently

8.3 Responsibility Boundary Diagram



8.4 Tool Integration and Ecosystem

8.4.1 AI Coding Assistant Integration

Integration Points:

- AI assistants consume BRA rules through standardized interfaces
- Rulesets can be bound to specific assistant sessions or projects
- Execution evidence flows back to governance systems

Example Integrations:

- **GitHub Copilot**: Consume BRA rules from repository libraries
- **Cursor**: Load workspace-specific Application Rulesets
- **Claude Code**: Reference shared Rule Libraries for consistent behavior

8.4.2 Rule Library Sharing Mechanisms

Sharing Levels:

1. **Team-Level**: Rules shared within a development team
2. **Organization-Level**: Cross-team sharing through internal repositories
3. **Industry-Level**: Cross-organization sharing through consortia or marketplaces
4. **Global-Level**: Open-source or standardized rule libraries

Sharing Infrastructure:

- Git-based repositories for version control
- Package managers for rule distribution
- Quality metrics and community ratings for trust establishment

8.4.3 Extension to Software Lifecycle

BRA initially targets code generation but can extend across the software lifecycle:

Code Generation → Specification → Viewpoint Domain:

1. **Code Generation**: Current BRA focus—governing AI coding behavior
2. **Specification Generation**: Rules for requirements, specifications, documentation
3. **Viewpoint Domain**: Rules for architectural viewpoints, domain models, system perspectives

Example Extension:

- CODE-AR-001: Code artifact isolation
- SPEC-AR-001: Specification artifact isolation
- VIEWPOINT-AR-001: Viewpoint artifact isolation

Same Decision Behavior Normative Category, different domains.

8.5 Future Research Directions

8.5.1 Automated Ruleset Recommendation

Research Question: How can we automatically recommend Application Rulesets based on project characteristics?

Approaches:

- Machine learning models trained on successful rule applications
- Context-aware recommendation based on project metadata
- Collaborative filtering from similar projects

8.5.2 Rule Mining from Existing Practices

Research Question: Can we extract rules from existing codebases, prompts, and documentation?

Approaches:

- NLP analysis of existing instructions and guidelines
- Pattern mining from code reviews and quality gates
- Reverse engineering from successful projects

8.5.3 Risk-Aware Rule Activation

Research Question: How can we dynamically adjust rule activation based on risk signals?

Approaches:

- Real-time risk assessment during AI execution
- Adaptive ruleset composition based on risk levels
- Escalation mechanisms for high-risk scenarios

8.5.4 Cross-Domain Rule Transfer Learning

Research Question: Can rules learned in one domain be transferred to related domains?

Approaches:

- Meta-learning across different application domains
- Generalization of domain-specific rules to abstract patterns
- Transfer learning for rule effectiveness prediction

8.6 Standards and Interoperability

Alignment with Standards:

- **ISO/IEC 42001**: BRA serves as a supporting governance artifact within an AI Management System (AIMS)
- **IEEE Standards**: Potential alignment with emerging AI governance standards

- **Industry Consortia**: Interoperability with industry-specific governance frameworks

Interoperability Mechanisms:

- Standardized rule schema formats (YAML/JSON)
- Common vocabularies for Decision Behavior Normative Categories
- Export/import capabilities for rule libraries

8.7 Summary: Architecture Ecosystem Benefits

The architecture ecosystem separation provides:

1. **Modularity**: Each architecture component evolves independently
2. **Interoperability**: Standardized interfaces enable integration
3. **Scalability**: Components scale according to their specific responsibilities
4. **Maintainability**: Clear boundaries reduce complexity
5. **Extensibility**: New components integrate without disrupting existing ones

BRA's focused responsibility—rule definition—enables it to excel at governance asset management while delegating execution, boundary control, and validation to specialized architectures.

9. Discussion

This section discusses BRA's limitations, practical challenges, and positioning within the broader AI ethics and governance landscape.

9.1 Limitations

9.1.1 Rule Design Complexity

Challenge: Authoring effective BRA rules requires expertise in both domain knowledge and normative language design.

Specific Issues:

- **RNL Proficiency**: Developers must learn to express constraints using MUST/MUST NOT without ambiguity
- **Semantic Precision**: Rules must be specific enough to be enforceable yet general enough to be reusable

- **Balance Between Policy and Constraint**: Determining when to use MUST (policy) vs. MUST NOT (constraint) requires understanding observability and enforcement implications

Mitigation Strategies:

- **Rule Authoring Tools**: AI-assisted rule authoring with RNL validation
- **Rule Templates**: Predefined templates for common decision behavior normative types
- **Community Libraries**: Shared rule libraries reduce need for custom authoring

9.1.2 RNL Expressiveness Boundaries

Challenge: RNL's constrained syntax (MUST/MUST NOT only) may be insufficient for complex behavioral requirements.

Specific Limitations:

- **Conditional Constraints**: Rules with complex conditions (e.g., "if X then MUST Y, else MUST Z") require decomposition into multiple simpler rules
- **Temporal Constraints**: Time-dependent rules (e.g., "must complete within 30 seconds") are difficult to express purely in RNL
- **Probabilistic Requirements**: Uncertainty-based constraints (e.g., "must achieve 95% confidence") exceed RNL's current expressiveness

Potential Extensions:

- **Extended RNL**: Introduce limited conditional constructs while preserving normative clarity
- **Hybrid Approaches**: Combine RNL with external configuration languages for complex scenarios
- **Domain-Specific RNL Variants**: Specialized RNL dialects for specific domains (security, performance, compliance)

9.1.3 Creativity Constraints

Challenge: Overly prescriptive rules may constrain AI's creative problem-solving capabilities.

Specific Concerns:

- **Over-Constraining**: Excessive MUST rules may prevent AI from exploring novel solutions
- **Rule Conflicts**: Multiple active rulesets may create conflicting directives that paralyze AI's decision-making
- **Context Inflexibility**: Rules authored in one context may be inappropriately applied in different contexts

Mitigation Strategies:

- **Constraint Minimalism**: Follow the "Minimal Constraint Principle"—include only necessary constraints
- **Context-Aware Activation**: Activate rulesets based on specific contexts rather than applying all rules universally
- **Override Mechanisms**: Provide controlled mechanisms for temporarily relaxing rules in exceptional circumstances

9.2 Practical Challenges and Mitigation Strategies

9.2.1 Adoption Barriers

Challenge: Organizations face barriers when adopting BRA.

Barriers:

- **Learning Curve**: Teams must learn new concepts (NNL, RNL, Decision Behavior Normative Categories)
- **Legacy Integration**: Existing prompt libraries and configurations require migration
- **Tool Support**: Limited tooling for rule authoring, validation, and management initially

Adoption Strategies:

- **Incremental Adoption**: Start with high-risk domains, gradually expand coverage
- **Hybrid Operation**: BRA rules coexist with existing methods during transition
- **Community Resources**: Leverage shared rule libraries to bootstrap adoption

9.2.2 Rule Maintenance Overhead

Challenge: Rule libraries require ongoing maintenance, versioning, and governance.

Maintenance Burdens:

- **Version Synchronization**: Rules and rulesets must evolve together
- **Deprecation Management**: Deprecated rules must be phased out without breaking existing rulesets
- **Conflict Resolution**: As rule libraries grow, potential conflicts increase

Mitigation Strategies:

- **Automated Testing**: Test rulesets against representative scenarios to detect conflicts
- **Impact Analysis**: Tools to assess which rulesets are affected by rule changes
- **Governance Workflows**: Formal review and approval processes for rule modifications

9.2.3 Inter-Team Coordination

Challenge: Cross-team rule sharing requires coordination and standardization.

Coordination Issues:

- **Naming Conflicts**: Different teams may create rules with overlapping IDs
- **Semantic Divergence**: Same category names may have different interpretations across teams
- **Quality Variance**: Rule quality may vary across authoring teams

Coordination Mechanisms:

- **Centralized Registries**: Shared rule ID namespaces and metadata registries
- **Standardized Vocabularies**: Common definitions for Decision Behavior Normative Categories
- **Quality Metrics**: Community-driven quality assessments and ratings for shared rules

9.3 AI Ethics and Governance Positioning

9.3.1 BRA's Role in AI Ethics

Alignment with Ethical Principles:

1. **Transparency**:
    - BRA rules explicitly state behavioral constraints
    - Governance metadata provides intent and rationale
    - Execution evidence enables auditability
2. **Accountability**:
    - Rule ownership is explicit
    - Decision traceability through execution evidence
    - Clear responsibility boundaries between BRA, BCM, and Boundary Architecture
3. **Fairness**:
    - Constraint Neutrality (CN) category ensures unambiguous, bias-free constraints
    - Rules can encode fairness requirements (e.g., "generated code MUST NOT use protected attributes in decision logic")
4. **Safety**:
    - Boundary & Stop (BD) category enforces fail-safe mechanisms
    - Constraints (MUST NOT) provide immediate violation detection
    - Risk Potential assessment prioritizes high-impact rules

9.3.2 Regulatory Compliance

Alignment with Emerging Regulations:

1. **EU AI Act**:
   - BRA provides documentation and traceability required for high-risk AI systems
   - Rule libraries serve as evidence of systematic risk mitigation
2. **NIST AI Risk Management Framework**:
   - BRA's governance metadata aligns with risk documentation requirements
   - Three-layer risk model (Risk Potential → Risk Signal → Realized Risk) supports risk lifecycle management
3. **ISO/IEC 42001 (AI Management System)**:
   - BRA serves as a supporting governance artifact
   - Rule lifecycle management aligns with AIMS requirements

9.3.3 Governance as Code Paradigm

BRA as Governance-as-Code:

- Rules are version-controlled artifacts
- Governance policies are executable and testable
- Governance evolution is traceable and auditable

Advantages:

- **Reproducibility**: Same ruleset produces consistent governance outcomes
- **Testability**: Rules can be tested against scenarios before deployment
- **Collaboration**: Git-based workflows enable collaborative governance authoring

Limitations:

- **Not All Governance Is Codeable**: Some governance requirements remain tacit or context-dependent
- **Governance Technical Debt**: Poorly designed rules accumulate as technical debt
- **Governance Complexity Explosion**: Overly complex rule libraries become unmanageable

9.4 Open Questions and Future Research

9.4.1 Dynamic Rule Adaptation

Question: Should rules adapt dynamically based on context, or remain static?

Trade-offs:

- **Static Rules**: Simplicity, predictability, auditability
- **Dynamic Rules**: Flexibility, context-sensitivity, but complexity and opacity

Research Direction: Hybrid models where static rules define boundaries and dynamic policies optimize within those boundaries.

9.4.2 Rule Effectiveness Measurement

Question: How do we measure whether rules achieve their intended governance goals?

Challenges:

- **Counterfactual Problem**: Cannot observe what would have happened without the rule
- **Long-Term Effects**: Rule benefits may emerge over time
- **Interaction Effects**: Multiple rules interact in complex ways

Research Direction: Empirical studies comparing outcomes with and without specific rules, controlling for confounding factors.

9.4.3 Rule Explainability

Question: How do we explain why a specific rule was triggered (or not) in a given context?

Challenges:

- **AI's Internal Reasoning**: AI's decision process may be opaque
- **Rule Interaction**: Multiple rules may jointly influence behavior
- **Context Dependency**: Same rule may trigger differently in different contexts

Research Direction: Explainability mechanisms that trace rule activation through AI's reasoning process.

9.5 Summary: Balancing Control and Flexibility

BRA navigates a fundamental tension in AI governance:

Too Much Control:

- Over-constrains AI capabilities
- Stifles creativity and exploration
- Creates maintenance burden

Too Little Control:

- AI behavior becomes unpredictable
- Governance gaps lead to risks
- Accountability becomes impossible

BRA's Balance:

- **Strong Constraints (MUST NOT)** for behavioral boundaries
- **Guidance Policies (MUST)** for desired outcomes
- **Compositional Flexibility** through Application Rulesets
- **Governance Metadata** for auditability without over-specification

This balance enables AI systems to operate autonomously within well-defined boundaries, achieving both controllability and capability.

10. Conclusion

This paper introduced Behavior Rule Architecture (BRA), a systematic framework for defining and managing AI execution behavior through structured normative rules. BRA addresses a fundamental gap in current AI coding assistant governance: the lack of a systematic rule architecture that treats behavior constraints as first-class, manageable artifacts rather than ephemeral prompt fragments.

10.1 Core Contributions

1. Semantic Separation (NNL ≠ RNL)

BRA establishes a clear distinction between **Normative Natural Language (NNL)** for intent optimization and **Rule Normative Language (RNL)** for behavior enforcement. This separation resolves the semantic conflation problem where current methods merge goals, boundaries, and behavioral mandates into ambiguous natural language expressions.

Impact: Developers express intent in human-friendly NNL while defining enforceable constraints in structured RNL using MUST/MUST NOT semantics, eliminating interpretation ambiguity.

2. Complete Behavior Control Chain

BRA completes the governance chain from intent to execution:

```
Intent (NNL) → Constraint (RNL) → Rule (Artifact) → Composition (Ruleset) → Scope (Boundary) → E
```

Impact: Unlike current methods that address fragments of this chain, BRA provides end-to-end traceability from human intent to AI execution evidence.

3. Governance as Asset

BRA transforms rules from ephemeral prompt fragments into **structured, versionable, shareable governance assets** through:

- **Atomic Rule Definition**: Rules as independent, auditable units with governance metadata
- **Rule Libraries**: Cross-project and cross-organization repositories for accumulating governance assets
- **Lifecycle Management**: Version control, ownership tracking, state transitions (draft → active → deprecated)

Impact: Organizations accumulate governance knowledge over time, amortizing rule authoring costs and enabling knowledge transfer.

4. Dynamic Composition Capability

BRA introduces two complementary ruleset types:

- **Category Rulesets**: Behavior-based classification using Decision Behavior Normative Categories (AR, BD, CN, TR, LG, ST, TI)
- **Application Rulesets**: Principle-based composition (workflow, risk, compliance, domain-specific)

Three Iron Laws ensure architectural clarity:

1. Category MUST NOT define rule composition
2. Rule composition MUST occur only in Application Rulesets
3. Rules MUST remain atomic and independent

Impact: Rules can be flexibly combined for different contexts without semantic conflicts or categorization ambiguity.

5. Decision Behavior Normative-Based Classification

BRA classifies rules based on **AI decision behavior normative types** rather than application domains. This approach ensures:

- **Semantic Precision**: Rules precisely correspond to AI decision constraint points

- **Cross-Domain Reusability**: Same decision behavior normative applies across multiple domains
- **Governance Clarity**: Clear mapping between governance intent and rule enforcement

Impact: Organizations govern specific decision behavior normative types (e.g., all Boundary & Stop rules) consistently across all domains, or govern all rules within specific domains—providing flexible governance dimensions.

10.2 Value Proposition

BRA delivers three fundamental values for AI behavior governance:

1. Controllability

- Strong constraints (MUST NOT) provide behavioral boundaries
- Guidance policies (MUST) steer AI toward desired outcomes
- Execution boundaries limit operational scopes

2. Auditability

- Structured metadata enables traceability and accountability
- Execution evidence links AI decisions to governing rules
- Lifecycle management tracks rule evolution

3. Governance Readiness

- Rules align with regulatory requirements (EU AI Act, NIST AI RMF, ISO/IEC 42001)
- Governance-as-code paradigm enables systematic governance
- Shared rule libraries support industry best practices

10.3 Broader Implications

For Software Engineering Practice:

BRA introduces a new discipline: **AI Behavior Governance Engineering**. Just as test engineering, security engineering, and reliability engineering emerged as specialized practices, AI behavior governance engineering will become essential as AI coding assistants proliferate.

For AI Research:

BRA opens research directions:

- **Rule-Aware AI Models**: AI architectures that natively understand and reason about normative constraints

- **Governance Learning**: Machine learning techniques that infer rules from successful governance outcomes
- **Cross-Modal Governance**: Extending BRA beyond code generation to specification, design, and testing

For Industry Standards:

BRA provides a foundation for industry standards:

- **Standardized Vocabularies**: Common Decision Behavior Normative Categories across tools and platforms
- **Interoperability Formats**: Standardized rule schema formats for cross-platform rule sharing
- **Certification Frameworks**: Rule quality metrics and certification processes for trusted rule libraries

10.4 Community Call to Action

We invite the AI and software engineering communities to:

1. **Contribute Rule Libraries**: Share domain-specific rules through open-source libraries
2. **Develop Tooling**: Build authoring, validation, and management tools for BRA ecosystems
3. **Conduct Empirical Studies**: Measure BRA's effectiveness in real-world deployments
4. **Extend the Framework**: Explore extensions for new domains and AI capabilities
5. **Establish Standards**: Collaborate on industry standards for rule vocabularies and formats

10.5 Limitations and Future Work

BRA represents a foundational framework with known limitations:

Current Limitations:

- Rule design complexity requires expertise
- RNL expressiveness boundaries limit complex constraints
- Creativity constraints require careful balance

Future Directions:

- AI-assisted rule authoring tools
- Extended RNL dialects for specialized domains
- Dynamic rule adaptation mechanisms
- Rule effectiveness measurement methodologies

- Explainability for rule activation

We envision BRA as an evolving framework that will mature through community adoption, empirical validation, and iterative refinement.

10.6 Closing Perspective

The proliferation of AI coding assistants represents a paradigm shift in software development. This shift demands a corresponding paradigm shift in governance—from ad-hoc prompt engineering to systematic rule architecture.

The Central Thesis:

> **Behavior rules should be first-class artifacts—definable, auditable, composable, and reusable—rather than ephemeral prompt fragments.**

BRA provides the architectural foundation for this shift, establishing the engineering basis for **controllable, auditable, and governable AI systems**.

As AI capabilities continue to advance, the need for systematic governance frameworks will only intensify. BRA offers a path forward: transforming governance from an afterthought into a core engineering discipline, ensuring AI systems operate within intentional, explicit, and accountable boundaries.

The future of AI-assisted software development depends not just on what AI can do, but on ensuring what AI does aligns with human intent, organizational values, and societal norms. BRA provides the structural foundation for that alignment.

Reference

AI & Foundation Models

1. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. (2020).
   Language models are few-shot learners.
   Advances in Neural Information Processing Systems, 33, 1877–1901.
2. Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., et al. (2021).
   On the opportunities and risks of foundation models.
   arXiv:2108.07258.

3. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., et al. (2022).
   Training language models to follow instructions with human feedback.
   Advances in Neural Information Processing Systems, 35, 27730–27744.
4. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., et al. (2022).
   Emergent abilities of large language models.
   arXiv:2206.07682.

Large Language Models in Software Engineering

5. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., et al. (2019).
   Software engineering for machine learning: A case study.
   Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 291–300.
   https://doi.org/10.1109/ICSE-SEIP.2019.00042
6. Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., et al. (2021).
   Evaluating large language models trained on code.
   arXiv:2107.03374.
7. Rasheed, Z., Waseem, M., Sami, M. A., Kemell, K. K., Ahmad, A., Duc, A. N., et al. (2023).
   Large language models in software engineering: A systematic literature review.
   arXiv:2308.10620.
8. Tian, H., Lu, W., Li, T. O., Tang, X., Cheung, S., Klein, J., & Bissyande, J. F. (2023).
   Is ChatGPT the ultimate programming assistant — How far is it?
   arXiv:2304.11938.
9. Zhao, T. Z., Zhang, M., Lee, D., & Chen, X. (2023).
   Large language model for software engineering: A survey of recent advances.
   arXiv:2312.11970.

AI Governance and Risk Frameworks

10. European Union. (2024).
    Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act).
    Official Journal of the European Union.
11. ISO/IEC. (2023).
    ISO/IEC 42001: Information technology — Artificial intelligence — Management system.

International Organization for Standardization.

12. National Institute of Standards and Technology (NIST). (2023).
    Artificial Intelligence Risk Management Framework (AI RMF 1.0).
    https://doi.org/10.6028/NIST.AI.100-1
13. Hendrycks, D., Mazeika, M., & Woodside, T. (2023).
    An overview of catastrophic AI risks.
    arXiv:2306.12001.

Decision Behavior Governance (EDBG)

14. Tsai, S. (2026).
    Toward Decision Behavior Governance: Governance Existence, Invocation, and Decision Formation.
    SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6105226
    https://doi.org/10.5281/zenodo.18876165
15. Tsai, S. (2026).
    Four Stages of AI Governance: A Stage-Based Framing of Contemporary Practices.
    SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146666
    https://doi.org/10.5281/zenodo.18871083
16. Tsai, S. (2026).
    Runtime Governance vs. Development Governance.
    SSRN (submitted).
    https://doi.org/10.5281/zenodo.18876913
17. Tsai, S. (2026).
    Realizing ISO/IEC 42001 through Decision Behavior Governance.
    SSRN (submitted).
    https://doi.org/10.5281/zenodo.18877215
18. Tsai, S. (2026).
    Decision Risk: A Structural Governance Framework for AI-Assisted Software Development.
    SSRN (submitted).
    https://doi.org/10.5281/zenodo.19025533

Decision Phenomena

19. Tsai, S. (2026).
    From Inference Creep to Risk Acceleration Pipelines.
    SSRN. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6146686
    https://doi.org/10.5281/zenodo.18872172
20. Tsai, S. (2026).
    Structural Degradation: From Stateless Generation to Layered Software Decay in AI-Generated Code.
    SSRN (submitted).
    https://doi.org/10.5281/zenodo.19043086
21. Tsai, S. (2026).
    Ghost Intent: An Effect of Traceability Collapse in GenAI-Assisted SDLCs.
    SSRN (submitted).
    https://doi.org/10.5281/zenodo.18872540

Engineering Decision Behavior Framework (EDBF)

Before Decision Behavior

22. Tsai, S. (2026).
    Anchor Architecture: Structural Traceability for AI-Generated Artifacts.
    engrXiv. https://engrxiv.org/preprint/view/6580
    https://doi.org/10.5281/zenodo.18856781
23. Tsai, S. (2026).
    Viewpoint-Structured Specification (VSS).
    engrXiv. https://engrxiv.org/preprint/view/6612
    https://doi.org/10.5281/zenodo.18930951

During Decision Behavior

24. Tsai, S. (2026).
    Boundary as an Execution-Time Primitive for AI-Assisted Software Development Governance.
    engrXiv. https://engrxiv.org/preprint/view/6583
    https://doi.org/10.5281/zenodo.18883242

After Decision Behavior

25. Tsai, S. (2026). *Decision Analysis: Effect-Oriented Structural Scope Audit for AI-Assisted Software Development.* engrXiv. https://engrxiv.org/preprint/view/6616 https://doi.org/10.5281/zenodo.18934765