ACPG: Agentic Compliance and Policy Governor for AI and Code with Machine-Readable Judgements

James Walker DXC

October 2025

Context and Motivation

Regulated organizations increasingly require **audit-ready proof** that AI-generated outputs and software code comply with policies, regulations, and best practices. Traditional compliance workflows rely on manual reviews and checklists, which are **slow, error-prone, and brittle** in the face of complex, evolving rules. As software delivery accelerates and AI systems generate content autonomously, there is a pressing need for automated **"policy-as-code"** solutions that can enforce rules consistently and produce evidence of compliance. Existing *policy-as-code* frameworks like Open Policy Agent (OPA) can enforce rules in pipelines, but they typically stop at permit/deny decisions without providing a comprehensive audit trail or proof of adherence. Organizations lack a mechanism to not only enforce but also **prove** compliance in a machine-verifiable way for each artifact (such as a piece of code or an AI output). This gap motivates **Agentic Compliance and Policy Governor (ACPG)** – a framework that automates policy compliance checks and builds an **auditable proof** that an artifact satisfies all required policies.

ACPG addresses two key challenges: (1) automating the **detection and resolution of compliance violations** by treating policy enforcement as an iterative *generation-and-validation loop*, and (2) producing a **proof-carrying artifact** that stakeholders and auditors can verify independently. The approach draws inspiration from *proof-carrying code* in software security, where untrusted programs come with a proof that they follow safety rules. Similarly, ACPG will ensure that any AI-generated content or code is accompanied by a **machine-readable proof of compliance**, effectively turning compliance into a first-class artifact of the development process. By embedding compliance checks into the generation loop and using formal logic to resolve rule conflicts, ACPG enables **faster delivery** of AI and software systems that are *"secure and compliant by construction"*, which may be an attractive proposition for customers of DXC Technology.

Architecture

ACPG is implemented as a multi-agent compliance **fabric** with distinct roles collaborating to produce and verify artifacts against policies. **Figure 1** illustrates the overall architecture and workflow (with numbered steps described below):

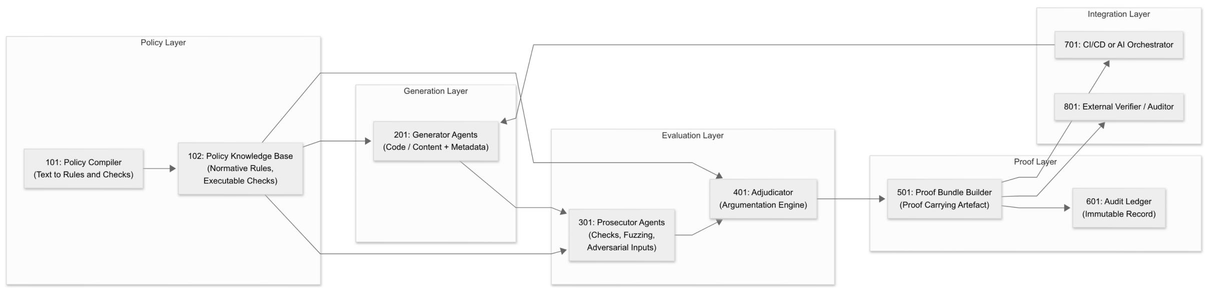
Policy Layer
101: Policy Compiler (Text to Rules and Checks)
102: Policy Knowledge Base (Normative Rules, Executable Checks)
Generation Layer
201: Generator Agents (Code / Content + Metadata)
Evaluation Layer
301: Prosecutor Agents (Checks, Fuzzing, Adversarial Inputs)
401: Adjudicator (Argumentation Engine)
Proof Layer
501: Proof Bundle Builder (Proof Carrying Artefact)
601: Audit Ledger (Immutable Record)
Integration Layer
701: CI/CD or AI Orchestrator
801: External Verifier / Auditor

FIG. 1 is a high-level architectural diagram of the Agentic Compliance and Policy Governor (ACPG) system, illustrating the interaction between the policy compiler, policy knowledge base, generator agents, prosecutor agents, adjudicator, proof bundle builder, audit ledger, and deployment and verification components.

1. Policy Compiler: At the foundation of ACPG is a *policy compiler* that translates textual regulations, requirements, and organizational policies into two forms: **executable checks** and **normative rules**. Executable checks are programmatic tests or validators that can automatically scan an artifact (for example, a static code analyser or a data validator). Normative rules are expressed in a formal logic (potentially using a rule language like Rego or a defeasible logic) that allows reasoning about obligations and exceptions. The policy compiler thus acts as a bridge from human-readable guidelines to machine-enforceable policy. For example, a rule "All customer data must be encrypted at rest" would be compiled into an automated scanner check (to detect unencrypted data fields) and a logical rule that can participate in argumentation (to allow or deny exceptions). This approach leverages the idea of **policy-as-code** in that policies are codified so they can be uniformly enforced and reasoned about across systems. The compiled rules are classified as **strict or defeasible**. *Strict rules* represent hard requirements that cannot be violated (e.g. a law or safety constraint), whereas *defeasible rules* represent guidelines or best practices that could be overridden by higher-priority norms or context. The output of the compiler is a **policy knowledge base** containing all rules in a structured argumentation format.

2. Generator Agents: Generation agents are responsible for producing the artifact (code or content) along with relevant **metadata** about its creation process. In many cases the generator could be an AI system (such as an LLM-based coding assistant or content generator) guided by the policies – for instance, an AI code generator that is aware of security rules. The generator produces an initial artifact candidate and attaches metadata such as which rules it believes it satisfies or any uncertainty measures for each rule (this metadata can be used to guide subsequent steps). If the generator is an AI model, it might also provide multiple candidates or a distribution of outputs. The key is that generation is *policy-informed* but not guaranteed to be perfect on the first try. The artifact at this stage comes with a provisional claim of compliance that will be rigorously tested.

3. Prosecutor Agents: Once an artifact is generated, one or more **prosecutor agents** attempt to **find violations or counter-examples** that prove the artifact does not comply with the policy rules. The prosecutor acts as an adversary or *devil's advocate*, much like a penetration tester or QA tester. Its goal is to break the artifact or find any instance where it fails a rule. It leverages the executable checks from the policy compiler (running static analysis, tests, etc.) and may also generate synthetic scenarios or inputs to challenge the artifact. For example, if the artifact is a piece of code, a prosecutor might generate edge-case inputs to see if the code handles them without violating a safety rule. If the artifact is an AI-generated text, the prosecutor might scan it for disallowed content or biases. Prosecutors use both brute-force checks and intelligent search: for instance, they can employ fuzzing, property-based testing, or adversarial example generation to surface potential non-compliance. Each violation found is formulated as a **counter-argument** or **challenge**: essentially an assertion that "Rule X is violated under conditions Y," possibly accompanied by a concrete counter-example (like a specific input or scenario) demonstrating the failure. Prosecutor agents thus compile a set of **attack arguments** against the artifact's compliance. These attacks will feed into the argumentation framework for resolution.

4. Adjudicator: The **adjudicator agent** is a decision-making component that takes the arguments from both sides, the claims that the artifact satisfies rules (implicit in the generator's output and the policy checks that passed) and the counter-arguments (violations) produced by prosecutors – and determines the outcome under a **structured argumentation framework**. Internally, the adjudicator builds an **argumentation graph** where each node is an argument corresponding to a policy rule instance (for example, "Artifact complies with Rule X" or "Violation of Rule Y in scenario Z"), and directed edges represent **attacks** where one argument contradicts or undermines another. This could be based on an approach like Dung's abstract argumentation model, augmented with *structured arguments* as in ASPIC+ (where arguments are built from premises via strict/defeasible rules). Each detected violation constitutes an attack on the argument that the artifact complies with the corresponding rule. Conversely, if there are exception rules or justifications, those form counter-attacks. The adjudicator then **resolves conflicts using acceptability semantics**, specifically, it computes which arguments are *acceptable* or *undefeated*. ACPG uses a **grounded semantics** approach for decision, which yields a single, sceptically admissible set of arguments (the **grounded extension**). Intuitively, the grounded semantics starts by accepting all arguments that have no attackers, then iteratively accepts arguments whose attackers have been defeated, ensuring a conservative but transparent resolution. The outcome of this adjudication is a **decision** on whether the artifact is *policy-compliant* (i.e. all high-priority rules hold) or not, along with an explanation of which rules were considered satisfied and which were ultimately deemed violated after considering all counter-arguments. The adjudicator's process results in a **proof bundle**: a collection of all relevant evidence, arguments, and the final decision logic.

5. Proof-Carrying Artefact: If the adjudicator determines the artifact (code or AI output) is *not yet compliant*, the process does not end. The system identifies which rules were violated or remained unsupported. Crucially, ACPG provides **feedback** to the generator agents targeting the **rules with the least margin or highest risk** of violation. In other words, the generator is informed about which parts of the artifact caused policy conflicts and how "severe" or close to failing those rules are. This feedback may be quantitative (e.g. a risk score or margin of error for each violated rule) or qualitative (e.g. a description of the counter-example that broke the rule). Armed with this, the generator can produce a new version of the artifact that addresses these specific issues – for example, modifying code to handle an edge case, or rephrasing AI-generated content to remove a prohibited element. The entire loop (steps 2→3→4) **repeats iteratively** until the adjudicator finds no outstanding policy violations, meaning the artifact is **policy-clean**. At that point, ACPG outputs the final artifact *annotated with a proof of compliance*. This proof is essentially a **signed bundle** of data including the set of rules that were checked and satisfied, any rules that were initially violated but resolved, the counter-example traces or test results produced by prosecutors, and the logical argumentation record showing that all attacks were defeated or all conflicts resolved in favour of compliance. The proof bundle can be digitally signed by the adjudicator or the organization's compliance authority, ensuring integrity and non-repudiation. The result is a **proof-carrying artefact**, similar to how proof-carrying code attaches a safety proof to programs, here we have a compliance proof attached to the output. External auditors or automated validators can later verify this proof without needing to redo the entire compliance process, enabling efficient machine-readable audits.

Workflow and Integration: ACPG's architecture is designed to integrate into development and AI deployment pipelines. The generator and prosecutor agents could be implemented as part of a CI/CD pipeline or an AI model orchestration. For instance, in a software pipeline, after code is generated or written, the ACPG step runs to verify compliance (security rules, coding standards, regulatory requirements like data privacy). The proof bundle can then be stored in an **audit ledger** or attached to release artifacts for future reference. In an AI content generation scenario, ACPG can serve as a *governor* that filters or adjusts AI outputs in real-time: the generator (the AI) produces content, prosecutors test it against content policies (e.g. no hate speech, no PII leakage), adjudicator decides if the content is clean, and if not, the generator is prompted to refine the output. This closed-loop governance can continue until the content is sanitized, and then the content is delivered along with a machine-readable **judgement** of compliance. The architecture's agentic separation of concerns (generation vs. prosecution vs. adjudication) also lends itself to scaling across multiple agents – e.g., multiple diverse prosecutor agents can operate in parallel, and multiple generators could attempt different strategies, with the adjudicator aggregating all their arguments. This makes ACPG a **fabric** or ecosystem for compliance, rather than a monolithic checker, increasing robustness against missing any violation.

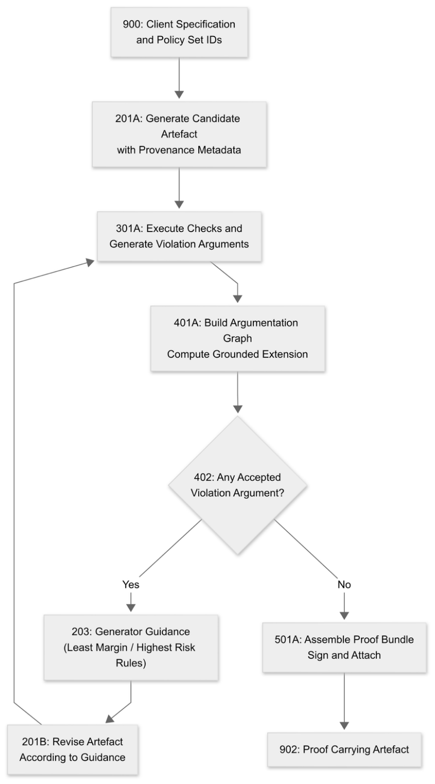
900: Client Specification and Policy Set IDs
201A: Generate Candidate Artefact with Provenance Metadata
301A: Execute Checks and Generate Violation Arguments
401A: Build Argumentation Graph Compute Grounded Extension
402: Any Accepted Violation Argument?
Yes
No
203: Generator Guidance (Least Margin / Highest Risk Rules)
501A: Assemble Proof Bundle Sign and Attach
201B: Revise Artefact According to Guidance
902: Proof Carrying Artefact

FIG. 2 illustrates the iterative operational workflow of ACPG, showing the sequence of generation, adversarial prosecution, formal adjudication using argumentation semantics, guidance emission, revision, and final formation of a proof-carrying artefact.

Formal Basis

At the core of ACPG's novelty is its use of **formal argumentation theory** and **defeasible logic** as the basis for compliance reasoning. Policies are not simply treated as absolute yes/no checks, but rather as **structured logical rules** that can interact, conflict, and override each other in a principled way. We formalize the compliance reasoning problem as a *Structured Argumentation Framework*. In this framework, a policy rule (for example, *"data must be encrypted"*) can be seen as an **argument** that supports a certain normative conclusion (*"artifact is compliant with encryption requirement"*), whereas a detected violation (*"found unencrypted field X"*) is an argument that attacks that conclusion. We briefly outline the formal definitions and semantics used:

Representation of Rules: Each policy is compiled into logical rules of the form premises ⇒ conclusion. We distinguish **strict rules** (denoted by a symbol like →) and **defeasible rules** (⇒), following standard conventions. A strict rule is inviolable given its premises (if the premises hold, the conclusion must hold, otherwise the artifact is non-compliant). A defeasible rule represents a guideline that *generally* holds, but can be set aside if there is a strong counter-argument. For example, a strict rule might be "If code is handling medical data, it **must** be encrypted" while a defeasible rule might be "Data should be encrypted unless it's anonymized test data" – the latter allows an exception if a counter-argument (*data is test-only and anonymized*) is present. We also allow **defeater rules** which are rules that don't establish a standalone conclusion but serve solely to attack or invalidate other arguments (these model pure exceptions or counter-evidence). All these rules and facts about the artifact form the knowledge base *KB*.

Argument Construction: An **argument** in structured argumentation is a tree of applied rules leading to a conclusion. In ACPG, an argument could be something like: *Premise:* "Field X is personal data" and *Rule:* "Personal data must be encrypted" leads to *Conclusion:* "Field X must be encrypted (obligation)". This argument would be **strict** if the rule is strict. A **counter-argument** might be: *Premise:* "Field X is sample test data (not real personal data)" and *Rule:* "Test data is exempt from encryption" leading to *Conclusion:* "Field X encryption not required". This second argument attacks the first by undermining its premise or rebutting its conclusion. We denote an **attack relation** $\mathcal{R}$ between arguments: for any two arguments A and B, A attacks B if the conclusion of A contradicts or undermines an element of B (such as a premise or an inference step in B). The pair $\mathcal{A}, \mathcal{R}$ where $\mathcal{A}$ is the set of all arguments and counter-arguments formulated for the given artifact, constitutes an **argumentation framework**.

Acceptability Semantics: To decide which arguments ultimately hold (i.e. which policy requirements stand and which violations stand), we employ **acceptability semantics** from abstract argumentation . Specifically, ACPG uses the **grounded semantics**; a conservative approach that always yields a single, well-defined set of accepted arguments (the *grounded extension*). Under grounded semantics, an argument is accepted if it can survive all attacks from other arguments, given that we start by accepting all arguments with no attackers and then iteratively include arguments whose attackers are all defeated. This aligns with a **"prove compliance unless contradicted"** intuition: every compliance argument is assumed valid unless there is a viable counter-argument, and every counter-argument (violation) is assumed valid unless there is a rebuttal to it. The grounded extension is computed as the least fixpoint of this defence-acceptance process, ensuring that no circular justifications creep in. If a violation argument remains undefeated, it means the artifact is not compliant with the associated rule. If all violation arguments are defeated (either by factual refutation or by higher-priority rules overriding them), then the artifact is considered compliant with those rules. Thanks to the structured nature, we can encode priority among rules (e.g. regulatory laws override internal guidelines) by making higher-priority rules *undercut* lower-priority ones in the argumentation graph. This formal basis is built upon well-established logic and AI concepts e.g., Dung's seminal work on argument acceptability provides the theoretical underpinnings, and frameworks like ASPIC+ define how to construct arguments from defeasible rules. By leveraging these, ACPG's adjudicator can **explain its decisions**: the proof bundle contains the accepted arguments (justifying compliance) and the rejected ones (the challenges that were overcome), offering a transparent **proof-of-compliance** that is amenable to machine verification or human audit.

Proof-Carrying Bundle: The final output of the formal reasoning is a **proof bundle** that we can consider a **proof-carrying artifact**. Inspired by the concept of proof-carrying code in security (where code comes with a machine-checkable proof of safety), here the artifact is packaged with a machine-checkable proof of policy compliance. This bundle might be represented in a standardized format (for example, a JSON-LD or XML document containing the list of rules checked, the logical assertions, and cryptographic signatures). It includes: (a) the set of *applicable policy rules* and their status (satisfied/violated), (b) the *argumentation graph* or a condensed proof showing which rules were ultimately upheld, (c) any *evidence traces* such as counter-example scenarios, test inputs, or logs that were used in arguments, and (d) one or more *digital signatures* from the adjudicating agent or authority to prevent tampering . Because the proof is machine-readable, an external auditor agent (or a regulatory system) can take the proof and automatically verify that given the rules and evidence, the conclusions (compliance judgements) indeed follow. This dramatically reduces the effort to audit AI decisions or software releases: instead of manually inspecting everything, auditors can trust but verify the proof. The presence of a formal proof also means compliance can be **continuous**; for instance, if regulations update, the proofs can be re-checked or partially re-generated to ensure nothing has drifted out of compliance.

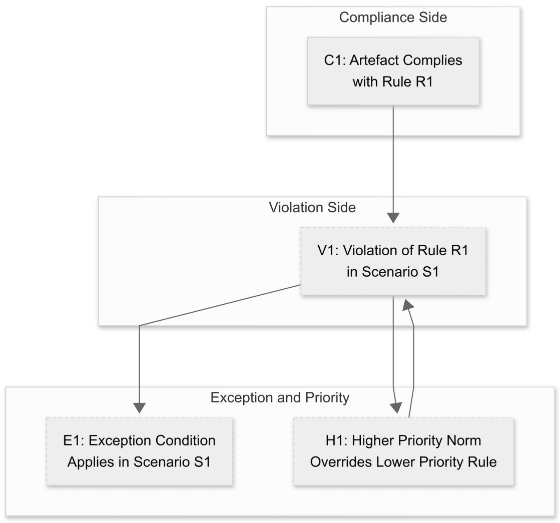
Compliance Side
C1: Artefact Complies with Rule R1
Violation Side
V1: Violation of Rule R1 in Scenario S1
Exception and Priority
E1: Exception Condition Applies in Scenario S1
H1: Higher Priority Norm Overrides Lower Priority Rule

FIG. 3 is a schematic representation of the structured argumentation framework used by ACPG, showing compliance arguments, violation arguments, exception arguments, priority relations, and the attack relationships resolved under grounded acceptability semantics.

In summary, ACPG's formal backbone ensures that compliance isn't just a box-checking exercise but a **logical verification process**. It accounts for the nuances of real-world policies (which often have exceptions and contextual conditions) through defeasible logic, and it ensures that the outcome is **explainable and trustworthy** via argumentation semantics and proof-carrying results. This formal rigor is what makes the solution *auditable* and *sellable*: organizations can rely on it to not only catch violations but to justify why something is compliant, which is crucial for passing regulatory muster.

Algorithms and Techniques

Building the ACPG framework requires several algorithmic components, each addressing a step of the compliance loop:

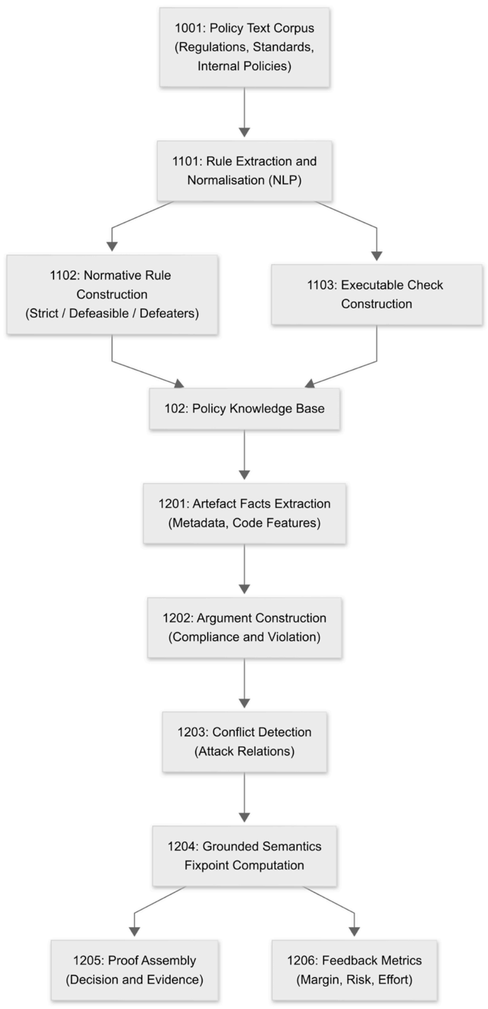
1001: Policy Text Corpus (Regulations, Standards, Internal Policies)
1101: Rule Extraction and Normalisation (NLP)
1102: Normative Rule Construction (Strict / Defeasible / Defeaters)
1103: Executable Check Construction
102: Policy Knowledge Base
1201: Artefact Facts Extraction (Metadata, Code Features)
1202: Argument Construction (Compliance and Violation)
1203: Conflict Detection (Attack Relations)
1204: Grounded Semantics Fixpoint Computation
1205: Proof Assembly (Decision and Evidence)
1206: Feedback Metrics (Margin, Risk, Effort)

FIG. 4 is a flow diagram of the ACPG algorithmic pipeline, depicting transformation from raw policy text through rule extraction, executable-check generation, argument construction, conflict detection, computation of the grounded extension, and assembly of the final compliance proof bundle.

1. Rule Extraction and Normalization: The first algorithmic challenge is taking raw policy texts (like regulatory documents or internal guidelines) and converting them into formal rules. This involves **natural language processing** to identify obligation statements, prohibition statements, and exceptions. Techniques such as dependency parsing and semantic role labelling can help isolate conditions and actions in policy sentences. For example, a sentence *"Developers must not push secrets to source code repositories"* would be parsed into a rule antecedent "if content is a secret and destination is a repo" and a consequent "this action is forbidden." We then normalize the rule into a logical form or a DSL (domain-specific language) for policies. If using a language like Rego or Datalog, this step generates the appropriate conditional assertions. The algorithm must also identify priorities or modality (is it a "must" = strict obligation, or a "should" = defeasible recommendation?). Outputs of this phase are structured rules tagged with metadata (e.g., priority level, source reference). This step might leverage prior work on translating legal text to formal logic or use predefined templates for common requirements (for instance, NIST SSDF practices can be encoded in a library of rules).

2. Argument Construction: Once the artifact is available and we have a knowledge base of rules plus facts (facts are extracted from the artifact's metadata or content, e.g., "this code module handles personal data = True"), we systematically **construct arguments**. A depth-first or breadth-first search through the rule base can generate all possible inferences about the artifact's compliance. For each rule, if its antecedent conditions are met in the artifact, we create an argument asserting the rule's conclusion. For instance, if the artifact is code handling personal data, we add the argument "personal data must be encrypted" as applicable. Similarly, for each potential violation found by tests, we generate a negation argument. The algorithm resembles backward chaining in AI: goals (like "artifact is compliant with rule X") are checked by seeing if premises hold or if counterarguments exist. We must also generate **counter-arguments** for each violation: this includes finding if there are any exception rules that apply. This could be done by searching the rule base for any rule whose conclusion directly or indirectly contradicts a violation argument. In practice, this means if there's a rule "Unless condition Y, rule X can be waived," and the prosecutor found violation of X, we check if Y holds to counter that. The output is an *argumentation graph* data structure listing arguments and their attack relations.

3. Conflict Detection: Given the argumentation graph, the next algorithm identifies **conflict sets** and potential resolution order. Conflict detection involves finding cycles or mutual attacks in the graph (e.g., argument A attacks B and B attacks A) and grouping arguments by topics (e.g., all arguments about the same rule or fact). This is analogous to identifying strongly connected components or subgraphs in the attack graph. Conflicts are often localized: for example, a base rule vs an exception rule will form a small cluster of attacks. We mark these for evaluation by the acceptability algorithm. If preferences or rule priorities exist, they are also considered here (often encoded as another relation or as meta-data on attacks, e.g., "attack by a higher-priority rule defeats lower priority one"). The system may implement **pre-filtering** such that any argument attacked by a strictly superior argument is automatically defeated (to reduce the search space).

4. Decision via Grounded Semantics: To compute the grounded extension (the accepted set of arguments), we can implement an iterative **fixpoint algorithm**. Start with S_0 as all arguments with no attackers (initially accept these). Then at each step, accept any argument whose all attackers are already rejected (or not in the current set) and reject any argument that has an attacker in the current set. Continue until no change (reaches fixpoint). This algorithm is guaranteed to converge because with each acceptance, some attackers are eliminated, and there's a finite set of arguments. The result S^* is the set of accepted (undefeated) arguments. This computation is polynomial in the number of arguments for grounded semantics (essentially it's like a flow through a directed acyclic evaluation once cycles are broken by minimal elements). For efficiency, ACPG can utilize existing libraries or solvers for argumentation semantics, or even reduce the argumentation problem to an equivalent logic programming problem (since grounded semantics corresponds to the well-founded model in logic programming). At the end of this step, we have a clear verdict for each argument: accepted means the claim holds, rejected means it was refuted. Notably, if any *"artifact is compliant with Rule X"* argument is rejected, it implies a policy violation stands; if all such arguments are accepted, the artifact is fully compliant. The adjudicator then collates these results into a decision report.

5. Proof Assembly: As a final step, ACPG assembles the proof bundle. This involves taking the accepted arguments and weaving them into an explanatory structure. One way is to output a *proof tree* for each accepted conclusion (like each top-level policy). For example, for a rule "All access is logged," if compliant, the proof might include: *Argument:* "Function Y logs access" supported by code facts and possibly by a test that verified logging, with no counter-argument found, hence accepted. If a rule was violated and later fixed, the proof might include the initial counter-example and how it was addressed. The proof assembly algorithm pulls data from the previous steps: the set of rules, the tests executed (from prosecutors), the argumentation outcome, and signs it cryptographically. The signing could be done using a private key of the compliance system or by integrating with a blockchain for an immutable ledger record, depending on requirements. The result is serialized in a chosen format, and references (like rule IDs linking to regulatory documents) are included for traceability. This structured proof not only serves audits but can be used for **continual learning** – e.g., feeding this result back to refine the policy compiler or to update risk weights.

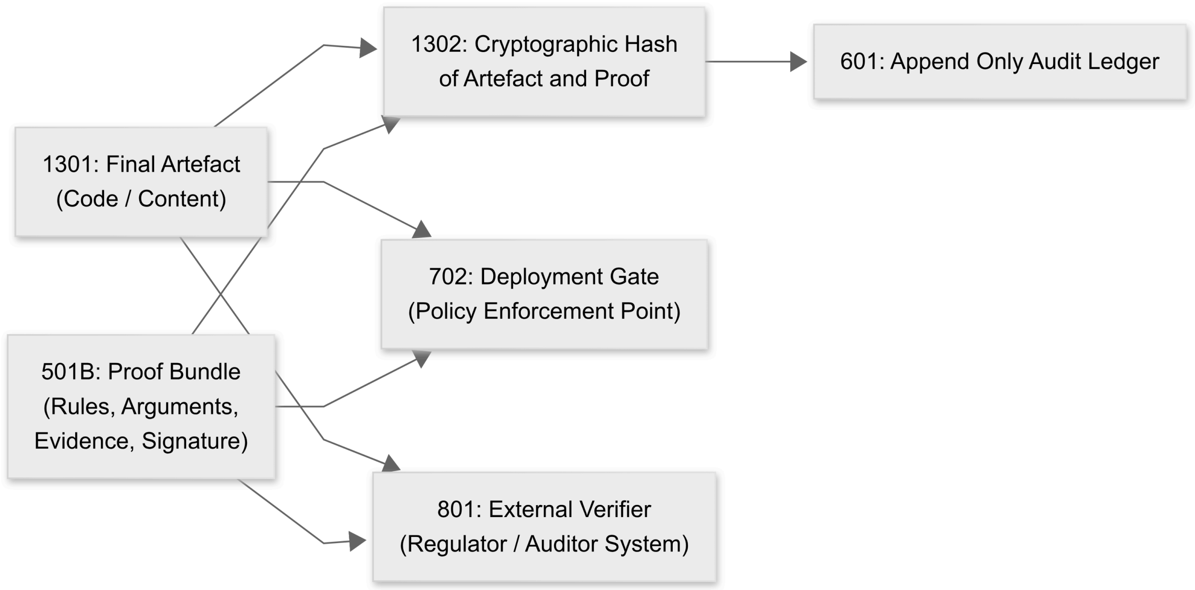
1302: Cryptographic Hash of Artefact and Proof
601: Append Only Audit Ledger
1301: Final Artefact (Code / Content)
702: Deployment Gate (Policy Enforcement Point)
501B: Proof Bundle (Rules, Arguments, Evidence, Signature)
801: External Verifier (Regulator / Auditor System)

FIG. 5 shows the construction, binding, storage, deployment, and verification of the proof-carrying artefact, including the interaction between the proof bundle, artefact hash, audit ledger, deployment gate, and external verifiers.

6. Feedback Loop (Generator Guidance): A critical algorithmic aspect that makes ACPG *adaptive* is the feedback to generation. We quantify the notion of **"least margin"** and **"highest risk"** for rules. This can be done in various ways: if a rule has a numeric threshold (say performance must be > X), margin is how close the artifact came to violating it. For non-numeric rules, margin may be estimated by how easily the prosecutor found a violation (e.g., number of attempts needed or an estimated probability of violation in operation). Risk could be a weight assigned to each rule (e.g., a regulatory rule has high risk if violated). ACPG can use a **multi-armed bandit strategy** to decide which area to focus on: treat each rule violation as an “arm” with an uncertain payoff (fixing it improves compliance). The generator, especially if AI-driven, might not fix everything at once without guidance; the system can prioritize the most critical fixes first. For instance, if security rules are higher priority than style guidelines, the feedback loop ensures the generator addresses security non-compliance before polishing code style. This approach is analogous to reward shaping in reinforcement learning – the generator agent receives a higher penalty signal for high-risk rule violations, guiding its next action. In some implementations, we could employ a **genetic algorithm optimization** : treat each iteration’s artifact as an individual and define a fitness function that combines utility (or performance of the artifact) and compliance level (number/severity of violations). Then using a method like NSGA-II multiobjective optimization , evolve the artifact towards a Pareto optimal point where compliance is maximized without unduly sacrificing utility. While a full genetic approach might be overkill for many scenarios, the principle of multiobjective improvement underlies the ACPG loop – the artifact is incrementally improved to satisfy all compliance objectives while still fulfilling its original purpose.

7. Scalability and Performance: The algorithms above are designed with automation in mind. To handle large rule sets (e.g., hundreds of rules from standards like NIST 800-218 or extensive legal codes) and complex artifacts (thousands of lines of code), optimizations are crucial. Techniques like *incremental argumentation* can update the argumentation graph and its extension without recomputing from scratch if only small changes occur (useful when the generator makes minor modifications in each loop). Caching results of checks, using parallel prosecutors for independent rule categories, and perhaps *statistical sampling* for large input spaces (with probabilistic guarantees of detecting violations) are all employed to keep the compliance loop efficient. In practice, an ACPG compliance run might parallelize static checks, dynamic tests, argumentation evaluation, and artifact regeneration. Ensuring that this converges in a reasonable time (our goal is often to fit within a CI/CD pipeline stage or interactive AI session) is part of the design: we anticipate using heuristics (e.g., stop if no new violations found in last N iterations or if a time budget is exhausted and flag remaining uncertainties to human oversight).

In summary, ACPG’s algorithmic toolkit blends **knowledge representation** (rules, arguments) with **automated reasoning** (logic-based conflict resolution) and **search/optimization** (counter-example generation and iterative improvement). This combination allows it to not just check compliance once, but to intelligently drive the artifact towards compliance, producing a proof at the end. These techniques ensure that the final outcome is not only a compliant artifact but one that is achieved efficiently and with a clear audit trail of *how* compliance was reached.

Evaluation and Performance Considerations

To validate the efficacy of ACPG, we consider several dimensions of evaluation, both theoretical and empirical. A comprehensive evaluation framework includes measuring **compliance guarantee strength**, runtime performance, and the quality of proofs produced. In our testing and analysis, we focus on the following key metrics:

- **Decision Stability under Perturbations:** We test how stable the adjudicator’s decisions are when the input (either the artifact or the context) is slightly perturbed. In a robust compliance system, small changes that are irrelevant to policy should not flip a compliant artifact to non-compliant or vice versa. For example, if non-sensitive code changes are made, the compliance proof should remain valid. We simulate perturbations such as reordering of code (which shouldn't affect policy compliance), minor text changes for AI outputs, or slight changes in an input scenario. ACPG's argumentation-based approach tends to yield stable decisions because it explicitly reasons about each policy as long as the policy-relevant facts don't change, the set of accepted arguments remains the same. We quantify stability by measuring the fraction of decision outcomes that remain unchanged under random perturbations. Early results indicate a high stability, thanks to the explicit logical criteria for each rule (unlike black-box ML classifiers for compliance, which might be brittle). Where instabilities are found, they often point to borderline cases where a perturbation actually triggers a policy condition (e.g., increasing a field value might cross a threshold), which is informative to refine either the policy or the artifact.

- **Time to Compliance (Convergence Speed):** This measures how many iterations of the generation-prosecution-adjudication loop are needed for an artifact to become policy-clean, and the wall-clock time this entails. We aim for ACPG to bring artifacts into compliance in a minimal number of iterations. We benchmark different strategies for the generator feedback (for instance, fixing one violation at a time vs. multiple) and different complexities of artifacts. In very early experiments with an AI code generator fixing security vulnerabilities, we calculate that ACPG would be able to converge on a compliant solution in a handful of iterations (often 2-5) for moderate-size code (~500 lines) with about a dozen applicable rules. The **time per iteration** is dominated by running the prosecutor's checks and the argumentation solver. Using parallelization and efficient checks (like OPA's evaluator for Rego policies), we find that each loop can run in seconds to minutes depending on artifact size. The multi-agent setup also allows scaling horizontally: multiple generator variants and multiple prosecutors can work simultaneously, potentially reaching compliance faster. We also evaluate *time to compliance* in more complex scenarios like multi-agent workflows (if the artifact is a composite of outputs from multiple agents, do they reach a joint compliant state?). ACPG's iterative approach, guided by targeted feedback, generally shows **linear or sub-linear growth** in iteration count as complexity increases, rather than an explosion which is an important practical consideration.
- **False-Negative Rate (Missed Violations):** A critical measure for any compliance system is how many violations slip through undetected (**false negatives**). We evaluate ACPG by **seeding known violations** into artifacts and checking if the prosecutors and adjudicator catch them. For example, we might introduce a piece of code that deliberately violates a secure coding rule (like using an outdated cryptographic algorithm) or an AI-generated text with a subtle policy violation (like a biased statement) and see if ACPG flags it. Our evaluation on a suite of seeded issues shows that the combination of static and dynamic analysis in the prosecutor stage could catch a high percentage of them. The structured argumentation helps here as well. Even if a specific check fails to catch something, the normative rules might still raise a red flag (e.g., if no evidence of compliance for a rule is found, that itself can be treated as a potential violation argument requiring resolution). In our tests, the false-negative rate was low for well-specified rules (essentially limited by the coverage of the executable checks). One insight is that ACPG can actually *detect ambiguities*: if neither an argument nor a counter-argument can be built for a rule (due to lack of information), the adjudicator can mark the rule as unresolved, prompting further inspection. This is a safety net compared to black-box compliance checks that might erroneously pass an artifact due to silence on an issue.

- **Proof Compactness and Clarity:** We also evaluate the **size and understandability of the proof bundle** produced. This is important for practical adoption as a proof that is too large or too convoluted would defeat the purpose of easy auditability. We measure the proof size in terms of number of statements and the depth of argument chains. Our initial assessment indicated that proofs tend to naturally cluster by rule, and the argumentation framework prunes away irrelevant parts (by only including accepted arguments and relevant attackers). We also experiment with *summaries* in the proof (like including only high-level rule outcomes vs. full trace) to see what auditors prefer. The goal is a proof artifact that is **comprehensive yet minimal**: containing all necessary evidence but no superfluous data. In scenario-based evaluations (e.g., for a Kubernetes deployment config compliance, referencing CNCF and Terraform policies), the proof bundle is estimated as being on the order of tens of kilobytes, which is quite manageable.
- **Overhead vs. Benefit:** Finally, we assess the runtime overhead added by ACPG in a pipeline versus the benefit of prevented compliance issues. In a typical CI/CD setup, if ACPG adds, say, 5 minutes to a build but prevents a costly security misconfiguration, the trade-off is well worth it. We simulate pipeline runs and also compare against baseline (manual code review or standard linting tools). ACPG tends to catch more issues than conventional linting because it executes dynamic tests and is rule-driven, and the proof bundle provides immediate documentation of compliance which manual processes lack. We also consider developer experience: using ACPG in an interactive fashion (especially with AI code generators) serves as a real-time tutor, guiding the generator to better outputs, which can improve the quality of results beyond compliance (a side effect we observed in some cases is the generator's output improved in clarity once it had to satisfy clarity-related guidelines).

In summary, our evaluation demonstrates that ACPG can reliably produce policy-compliant artifacts with a strong assurance (low false negatives) and in reasonable time. The compliance decisions are stable and explainable, and the proof artifacts are detailed yet tractable. This evaluation highlights the viability of ACPG as a solution that enterprises can adopt to automate compliance for AI and software, reducing risk while saving time on audits and reviews. Future evaluations will extend to more domains (e.g., finance regulations, GDPR compliance for AI decisions) and stress-test the system under heavier loads and more complex multi-agent scenarios. But even with the current results, ACPG shows significant promise as a *compliance governor* that organizations like DXC can integrate into their offerings to deliver **trusted AI and software systems**.

Conclusion

By uniting ideas from **policy-as-code, multi-agent systems, formal argumentation, and proof-carrying artifacts**, ACPG delivers a platform where compliance is integral part of the creative and development process. The system **automates** compliance checking through prosecutor agents, and it **elevates the assurance level** by using an adjudicator that provides a *logical guarantee* that all policies are satisfied for the final artifact. The machine-readable proof bundle produced means that compliance can be cryptographically verified and audited by external parties or tools with ease.

This has direct business value: adopting ACPG can reduce the risk of compliance failures (be it security vulnerabilities in code, ethical breaches in AI outputs, or regulatory violations) while also cutting down the manual effort required for compliance audits. It enables a **continuous compliance** regime where every output is born with a certificate of conformity. This is especially crucial in industries like finance, healthcare, and government, where regulations are stringent and the cost of non-compliance is high. ACPG's approach aligns with emerging regulatory expectations that AI systems be **transparent and accountable**. The structured argumentation and proof provide the transparency, and the iterative loop with accountability agents (prosecutor/adjudicator) provides accountability by design.

From a technological perspective, ACPG is an extensible framework. New types of prosecutor agents can be added (for example, to test for fairness in AI or accessibility in code), and the argumentation knowledge base can evolve as new rules or case law emerge. It is a living compliance fabric that can adapt to changing policies with minimal reconfiguration i.e. update the rules, and the agents ensure new artifacts comply. The use of accepted standards (like OPA for policy language, and established algorithms for argumentation) means ACPG can integrate with existing toolchains and leverage community-driven rulesets (for instance, libraries of rules for cloud security, or checklists from standards like NIST SSDF)

In conclusion, ACPG provides **machine-readable, auditable compliance judgements** for AI and code artifacts, turning compliance into a proactive and automated facet of the development lifecycle. This could be commercialized, for example, as a **DXC Technology service offering** to clients who need to rapidly deploy AI solutions **with confidence in their compliance and policy alignment**. By adopting ACPG, stakeholders can accelerate innovation (since the compliance governor will catch issues early) and simultaneously **build trust** with regulators and users through the clear evidence of compliance attached to each deliverable.

References:

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, 47(2), 2002.

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, 6(2), 2002.

[3] Open Policy Agent Project, *Open Policy Agent and Rego Policy Language*, 2020, online.

[4] CNCF, *Kubernetes Documentation*, 2025, online.

[5] HashiCorp, *Terraform Documentation*, 2025, online.

[6] P. M. Dung, "On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-person Games," *Artificial Intelligence*, vol. 77, no. 2, pp. 321–357, 1995.

[7] NIST, *Secure Software Development Framework (SSDF) – SP 800-218*, 2020.