ACPG Prototype Development Guide

System Architecture Overview

The **Agentic Compliance and Policy Governor (ACPG)** system is designed as a multi-agent, iterative compliance loop (see **Figure 1** below) that produces a software artifact (code) along with a machine-readable proof of its compliance. The architecture separates concerns into distinct components that collaborate to enforce policies and generate proofs of adherence. At a high level, the workflow involves the following steps (numbered as in Figure 1):

- **1. Policy Compiler:** Compiles human-readable policies into machine-enforceable rules.
- **2. Generator Agent:** Produces an initial code artifact guided by the policies (e.g. an AI coding assistant using the LLM).
- **3. Prosecutor Agent(s):** Analyze the artifact (statically and dynamically) to find any policy violations or counter-examples.
- **4. Adjudicator:** Uses an argumentation engine to weigh the artifact's compliance claims against violations and decide which rules hold or are broken.
- **5. Proof Assembler:** If compliant, packages the results into a signed proof bundle; if not, feeds back guidance to the generator for another iteration.

Figure 1: ACPG architecture and workflow (conceptual diagram of components and data flow):

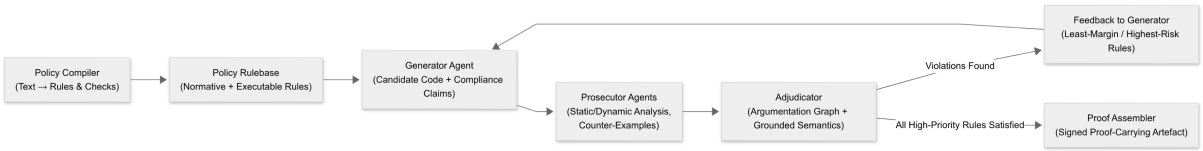

FIG. 1 illustrates the ACPG iterative compliance loop. The Policy Compiler produces a machine-readable rulebase consumed by the Generator Agent. The Generator produces a candidate artefact with provisional compliance claims. Prosecutor Agents perform static and dynamic analysis to surface violations, which are passed as counter-arguments to the Adjudicator. The Adjudicator constructs an argumentation graph and applies grounded reasoning to determine whether all high-priority policies are satisfied. If violations remain, targeted feedback is returned to the Generator to rectify the artefact. Once all required rules are satisfied, the Proof Assembler outputs a signed proof-carrying artefact.

Components and Their Interactions

- **Policy Compiler:** This module ingests policies written in natural language (e.g. regulatory guidelines, security standards, organizational coding policies) and compiles them into two forms: (a) **executable checks**, which are programmatic tests or static analyzers that can automatically scan code, and (b) **normative rules**, which are logical statements used for reasoning. For example, a policy *"No customer data shall be logged in plaintext"* might be compiled into an executable check (a static code scan for logging calls that handle customer data) and a logical rule for the

argumentation engine (e.g. *if code logs customer data without encryption, it violates policy X*). The compiler classifies rules as **strict** (hard requirements that cannot be violated) or **defeasible** (guidelines or best practices that might have exceptions). The output is a structured **Policy Rulebase** (e.g. a JSON or knowledge-base file) containing all rules, conditions, and metadata (like rule IDs, descriptions, severity, and whether they are strict or defeasible).

- **Generator Agent:** The generator is responsible for producing the initial **artifact**, in this case, source code, while being aware of the policy requirements. In our prototype, the Generator Agent will be implemented via an AI coding assistant (the LLM through the OpenAI API). We supply the agent with the relevant policies (or a summary thereof) and prompt it to produce code that adheres to those rules. The generator creates a candidate code artifact along with metadata about its compliance claims. For instance, the agent might output a JSON structure containing the code and a self-assessment of which rules it believes are satisfied or any potential concerns. (This metadata can be as simple as comments or a separate JSON list of rules it tried to follow.) **Note:** The initial code may not be perfect, the generator is *policy-informed* but might still violate some rules, so the process is iterative. The generator's output is then passed along for verification.
- **Prosecutor Agents:** These are one or more analysis modules that act as "attackers" or testers against the code artifact. Each Prosecutor Agent attempts to find policy violations, bugs, or security issues in the code. They use both **static analysis** and **dynamic testing** techniques. For static analysis, the prototype will integrate existing linters and security scanners; for example, **Bandit** for Python code can detect common security issues (like use of eval, hardcoded passwords, etc.) by analyzing the abstract syntax tree of the code. Bandit will automatically scan the Python source files and generate a report of any issues (violations) it finds. We can also incorporate linters like ESLint (for JavaScript) or **Semgrep** (a multi-language code scanning tool) with rules corresponding to our policies. For dynamic analysis, we can use tools like **PyTest** with **Hypothesis** (property-based testing) to generate inputs and test runtime behavior. Hypothesis will systematically try a range of inputs (including edge cases) to see if the code ever violates an invariant or rule. For example, if a policy says "the function must sanitize all log outputs," a prosecutor test might feed the function malicious input and check if any unsanitized data is printed. Prosecutors may also perform fuzz testing or runtime checks to find issues like uncaught exceptions, security vulnerabilities, or performance policy violations. Each violation found by a prosecutor is formulated as a **counter-argument**: essentially an assertion that *"Rule X is violated under conditions Y"*, possibly with a concrete example (e.g., "input '; DROP TABLE users;--' causes an SQL injection vulnerability, violating the DB safety policy"). These findings are collected as a set of **violation reports** to be passed to the Adjudicator. (In implementation, the prosecutors can return a JSON list of issues, each with a rule ID, description, and evidence like a stack trace or input that caused the issue.)
- **Adjudicator:** The adjudicator is the "judge" agent that takes the compliance claims (from the Generator and any passing checks) and the counter-arguments (violations from Prosecutors) and determines the overall compliance outcome. It uses a structured **argumentation framework** to logically resolve which rules are satisfied and which are defeated by counter-examples. Internally, the adjudicator builds an **argument graph** where each node is an argument about the artifact's compliance: e.g. an argument might be *"Artifact complies with Rule X"* supported by evidence, or *"Artifact violates Rule X under scenario Y"* supported by a prosecutor's test. Directed

edges between nodes represent **attacks** (a violation node attacks the compliance claim node for that rule, or an exception rule could attack a violation). We will leverage an argumentation logic approach such as **Dung's abstract argumentation** with **ASPIC+** for structured arguments, as described in the research. Each policy rule becomes an argument that the artifact *should* comply; each detected violation becomes an argument that attacks that compliance. The adjudicator then computes which arguments are **acceptable** (undefeated) under a chosen semantics. ACPG uses **grounded semantics**, starting from arguments that have no attackers (i.e. all rules are assumed satisfied until proven otherwise), then iteratively accepting arguments whose attackers have been defeated. This yields a single, conservative set of accepted arguments (the *grounded extension*), meaning any rule that still has an undefeated violation is considered broken. In practice, the adjudicator algorithm will take the JSON outputs from prosecutors (violations) and the list of all rules, and mark each rule as *complied* or *violated* by evaluating the attack relations. We can implement this reasoning using an existing argumentation library or a custom algorithm:

  - For the **prototype**, a straightforward approach is to implement the grounded semantics loop: start with all compliance arguments (rules) as tentatively accepted, then eliminate any that have an active violation. If a rule is strict and has any violation, it's immediately considered defeated. If a rule is defeasible and an exception argument exists, that might counter the violation. This process can be coded, but to reduce custom logic, we can use a library like **pygarg** (Python argumentation engine). The pygarg library can take a set of arguments and attacks and compute Dung-style semantics (including grounded semantics) by reducing the problem to SAT solving. We could formulate each argument and attack in pygarg's input format (e.g. as an Abstract Argumentation Framework in a .apx file) and use its solver to get the accepted set. Either way, the adjudicator will produce a decision: for each policy rule, whether it is ultimately satisfied or not, along with the justification. If any high-priority (strict) rule is not satisfied, the artifact is non-compliant overall. The adjudicator also prepares information for the next step: a **summary of which rules failed and why** (e.g. rule IDs with violation reasons), and which rules passed.

- **Proof Assembler:** If the adjudicator finds the artifact fully compliant (all strict rules satisfied, and any defeasible rule violations properly countered by exceptions), the process can conclude. The artifact is declared policy-compliant. The Proof Assembler then generates a **Proof-Carrying Artifact** package. This **proof bundle** is a structured data package containing: (a) the list of all policies checked, with indications of satisfied vs. waived vs. violated, (b) the full argumentation graph or at least a trace of which arguments were accepted (for transparency), and (c) evidence supporting each accepted argument (for example, links to test results or static analysis reports that back up compliance). Crucially, the proof bundle is then **digitally signed** by the adjudicator component or by a corporate private key to ensure integrity and non-repudiation. By signing the proof, any later auditor can verify that the proof has not been altered and indeed originates from the compliance system. If the artifact was not compliant, the Proof Assembler may still package a partial bundle (with the discovered violations and evidence) for audit purposes, but the main action is to feed back to the generator for another iteration instead of finalizing. In iterative mode, the adjudicator's findings about unmet rules are sent back to the Generator Agent (this feedback loop is depicted by the red arrow in Fig.1). The generator can then be

prompted to **fix the code** specifically to address those issues. This cycle repeats until no active violations remain or a set number of iterations is reached.

Integration into Development Workflow: The ACPG system is designed to be modular and integrable into various scenarios. For the prototype, our primary interface will be a **web UI** for interactive use, but the backend architecture allows running the compliance checks in other contexts as well (CLI tools, CI/CD pipelines, etc.). For example, in a CI/CD pipeline (such as a GitHub Action), a commit could trigger the ACPG compliance check: the code (artifact) is passed to the ACPG service, it runs through the steps above (likely non-interactively, possibly without the iterative regeneration for speed), and produces a pass/fail compliance result along with a proof artifact. This proof bundle can be stored as part of the build artifacts or an audit log. In an interactive setting (our web UI), a developer might paste or upload code and get back highlighted violations and suggestions for fixes (with the option for the AI to automatically apply fixes). The modular design, with distinct services for generation, analysis, adjudication, etc., means we could also scale out the system: e.g. run multiple prosecutor agents in parallel (for different categories of tests) or use different generator strategies concurrently, then aggregate results. The architecture ensures that compliance is not just a one-time check but a feedback-driven loop aiming to *"build compliance in"* to the code through iterative improvement.

API and Microservice Layout

To implement the above architecture in a robust, extensible way, we will structure the prototype as a set of **microservices** (or at least logically separated modules) with a clear API for each component. We will use **FastAPI** (Python) on the backend to define RESTful endpoints (and background tasks) and **React** for the frontend UI. Each major component of ACPG will correspond to one or more endpoints (and could be separate services in a production deployment):

- **Policy Service (Policy Compiler API):** Responsible for managing policies and rules. This service might provide endpoints such as:
    - POST /policies/compile, accepts a policy document or set of rules (perhaps in JSON or plain text) and produces the compiled rule set. In the prototype, we might load a predefined policy file (e.g., a JSON file of rules) at startup to avoid needing NLP on the fly, but the structure is in place for future expansion. The compiled rules (with their IDs, descriptions, conditions, etc.) are stored in a Policy Knowledge Base (could be in-memory or a database). If we implement on-the-fly compilation (NLP), this endpoint would call an LLM or use regex templates to translate text policies into our rule schema.
    - GET /policies, retrieve the current set of compiled rules (for the UI or other services to use).
    - (Optionally, POST /policies to add a new policy rule via API, etc.)
- **Generation Service (Generator Agent API):** This service handles interactions with the LLM (OpenAI the LLM). Endpoints might include:
    - POST /generate, Generate a new code artifact from a given specification and policy context. The request could include a prompt or spec for the code (e.g., "implement a function to do X") and perhaps a selection of relevant policy rules. The service will construct a prompt for the LLM that includes instructions about the policies (see Prompt Engineering below) and ask it to produce code. The response will be a JSON containing the code (and possibly

the model's own notes on policy compliance). For example, the JSON might look like: { "code": "<base64 or plain code here>", "analysis": { "likely_compliant": ["rule1", "rule2"], "needs_attention": ["rule3"] } }. We can refine this format as needed.

- POST /generate/fix, This endpoint will be used when we have an existing code artifact that failed compliance. It sends the artifact plus the list of violations back to the LLM, asking for a revised version of the code that fixes the issues. The request will include the code and a summary of what rules were broken or what the problems were. The service then returns a new version of the code in JSON (similar structure as above).
- These endpoints will internally call the OpenAI API (likely the chat/completions endpoint with the the LLM model) using an API key. The service will enforce timeouts and handle errors (e.g., if the model fails to produce valid JSON, we may need to catch that and possibly retry or correct format).

- **Analysis Service (Prosecutor Agents API):** We can design this as a collection of endpoints or a single endpoint that runs all checks:
  - POST /analyze/static, run static analysis on the given code artifact. The request contains the code (or a reference to it). The service then invokes tools like **Bandit** for Python (via its Python API or subprocess). We may also run additional static checkers or custom regex rules here (for example, a simple grep or AST parse to detect hardcoded secrets or banned functions). The result is a JSON list of findings. For instance:

```
[
  {"rule_id": "SEC-001", "description": "Hardcoded password found in code",
"line": 42, "severity": "HIGH"},
  {"rule_id": "SEC-003", "description": "Use of eval() detected", "line":
10, "severity": "MEDIUM"}
]
```

  - Each finding should map to a policy rule (the rule_id corresponds to the compiled policy rules from the Policy service).
  - POST /analyze/dynamic, run dynamic tests on the code. For Python, this might involve generating and executing tests. We could auto-generate simple tests for certain conditions: e.g., if a rule says *"function must throw an exception on invalid input"*, we attempt to call the function with invalid inputs to see if it throws. Using **Hypothesis** can automate generating inputs within certain strategies. Another approach is fuzzing: if the artifact is a runnable program, the service could run it with random or boundary inputs to try to break it. In the prototype, we might implement a limited dynamic test harness (since full program analysis can be complex). The output here would also be a JSON list of any violations found at runtime (e.g., "for input X, the program crashed or produced unsafe output Y, violating rule Z"). If no issues are found, it could return an empty list or success status.
  - POST /analyze, (comprehensive) alternatively, for simplicity, we can have a single endpoint that orchestrates both static and dynamic analysis and returns a combined report. It would call the static checks and dynamic checks internally and merge the results.
  - **Note:** If we support multiple languages, the Analysis service might route the request to language-specific analyzers. For example, if the artifact is

JavaScript, call ESLint or other linters; if Python, call Bandit/Pylint, etc. In a multi-service architecture, we might have separate microservices for each language's analysis (Python Analyzer Service, JS Analyzer Service, etc.), but in the prototype a single service with conditional logic is sufficient.

- **Adjudication Service (Adjudicator API):** This service encapsulates the argumentation logic and decision making.
    - POST /adjudicate, Determine compliance given the rule set and analysis results. The request would contain: the list of all relevant rules (maybe fetched from the Policy service, or a reference to a policy set ID) and the list of violations found (from the Analysis service), plus optionally any claims of compliance from the Generator. The service will then construct the argumentation graph: each rule that applies to this artifact is an argument "artifact complies with Rule_i". For each violation found, it creates an argument "Rule_i is violated under scenario S". It then marks attacks: the violation obviously attacks the compliance argument for that rule. If there are any exception or override rules (defeasible logic), those are also considered here (e.g., if Rule_i is defeasible and has an exception clause that matches scenario S, an argument for the exception would attack the violation argument). The Adjudicator then computes the outcome. In the prototype, since our policies will be relatively straightforward (mostly strict rules), the logic may reduce to: any rule with a violation = not compliant. But we will structure it to allow defeasible reasoning (for example, a rule "Sensitive data should be encrypted, **unless** it is an example dataset", here an exception could override a violation if the code is handling an example dataset). We can implement a simple algorithm for grounded semantics in Python (iteratively remove defeated arguments). However, as noted earlier, we can also leverage libraries:
        - Using **pygarg**: We can feed the arguments and attacks to pygarg's solver to directly compute the grounded extension. For example, create a temporary .apx file listing all arguments and attacks in the format pygarg expects (it supports .apx format for argumentation frameworks). Then call pygarg.dung.solver functions to get the set of acceptable arguments under grounded semantics. Pygarg is capable of solving acceptability and credulous/skeptical queries via SAT solving. In our case, we might simply ask: for each compliance argument "Rule_i_complied", is it skeptically accepted (meaning it holds in the grounded extension)? If yes, rule is satisfied; if not (i.e., a violation was undefeated), rule is not satisfied.
        - If not using a library, a custom logic: iterate over each violation; mark its rule as defeated (unless an override argument defeats the violation, which could be checked by seeing if any exception conditions apply). This is manageable if our policy logic is not too complex initially.
    - The response from /adjudicate will be a JSON decision object. For example:

```
{
  "compliant": false,
  "unsatisfied_rules": ["SEC-003", "INPUT-1"],
  "satisfied_rules": ["SEC-001", "SEC-002", "..."],
  "reasoning": [
     {"rule": "SEC-003", "status": "violated", "attacker": "Detected use of
eval() on untrusted input"},
```

```
    {"rule": "INPUT-1", "status": "violated", "attacker": "No input
validation on field 'username' (fuzzer found SQL injection)"},
    {"rule": "SEC-001", "status": "satisfied"},
    ...
  ]
}
```

    - The reasoning array can include a brief justification per rule, indicating either that it was satisfied (no attackers) or why it was not (listing the violation that defeated it). This information is crucial for both the user (to understand what to fix) and for constructing the proof bundle.
- **Proof Service (Proof Assembler and Signing):** This component finalizes the proof-carrying artifact.
    - POST /proof, Generate and (optionally) verify proof bundle. This endpoint takes the adjudication result, the original artifact, and possibly the full set of evidence (like static analysis reports, test logs) and creates the **proof bundle**. The proof is essentially a JSON document (or a set of files) containing:
        - **Metadata:** e.g. artifact identifier (maybe a hash of the code or a timestamp), the policy set ID or version used, and the adjudicator's decision.
        - **Rule outcomes:** a list of all rules with their final status (pass/fail/waived) and references to evidence.
        - **Evidence:** This could include the static analysis findings for each rule (for passes, maybe just "no issues found" or specific tests that passed; for fails, the details of the violation). It might include code annotations or excerpts where issues were found, to aid human auditors.
        - **Argumentation trace:** optionally, a representation of the argument graph or a simpler proof tree for each accepted rule. For example, for each rule that passed, we can include something like: "Rule X: PASSED, evidence: [test_case_12.log shows encryption working]". For each rule that was initially violated but then fixed in subsequent iteration, the proof might include both the original violation and the resolution (e.g., "initial violation found (secret key hardcoded) → code updated to remove hardcoded key → rule now satisfied").
        - **Signatures:** Once assembled, the service will cryptographically sign the JSON. We can use a library like Python's cryptography to create a digital signature. For simplicity, we might use an RSA or ECDSA key pair generated for the prototype. The signature (and the public key or a certificate) is attached to the proof bundle. This ensures that anyone later can verify that the proof came from our ACPG system and hasn't been tampered with.
    - The response from /proof could be the proof bundle itself (perhaps base64-encoded or as a JSON download) or an ID where the proof is stored. In a web UI, we might prompt the user to download the proof file for their records. In a CI/CD context, this could be automatically archived. Storing proofs in an **audit log database** or even a blockchain for immutability can be considered in a real deployment (to guarantee an auditor can always retrieve the exact proof produced).
- **Front-End (Web UI):** The React front-end will present an interface for users to:
    - Submit source code (or even request code generation by specifying a function they want).

- Choose or upload a set of policies (for the prototype, maybe a preloaded dropdown like “OWASP Top 10 checks” or “Custom Policy”).
- Trigger the compliance check (which calls the backend API).
- See the results: The UI can highlight lines in the code with issues (from the static analysis results), show a list of violations and their descriptions, and show an overall pass/fail summary. If the system can auto-fix issues, the UI may offer a “Auto-Fix Code” button which calls the Generator fix endpoint and then refreshes the analysis results.
- Download the proof bundle or expand it to see details of each rule’s proof. For example, the UI might display a summary like “**Compliance Proof:** 10 rules checked, 10 passed. Digitally signed by ACPG on 2025-11-20.” with an option to view details.
- The web UI will communicate with the backend via REST calls (and possibly WebSocket if we want real-time updates during the iterative process). However, given that LLM calls and analyses might take a few seconds, a simple approach is to have the frontend poll for status or wait for the final response (FastAPI can stream or send intermediate updates, but that can be an enhancement).

Microservice considerations: While the prototype might implement all these components within a single FastAPI application for simplicity, the design keeps them modular. In future, each could be a separate service (e.g., a dedicated microservice for the Generator (LLM calls) so that it can be scaled or secured separately, a separate service for heavy static analysis that might be resource-intensive, etc.). They would then communicate over HTTP or a message queue. We should design clean interfaces (e.g., the data schemas for requests/responses of each step as described) to allow this separation. This also allows headless use: for CLI or CI/CD, one could directly call the Analysis and Adjudication services with code from a repo without using the web UI.

Module Details

Now we break down the development specifics for each module in the ACPG prototype: how to implement them, what libraries to use, and how they interact.

Policy Compiler Module

Role: Ingest compliance policies (security rules, best practices, regulatory requirements) in a human-friendly format and output a structured, machine-checkable format. Initially, to simplify, we will assume policies are provided in a structured form (e.g., a JSON file with rules) or as simple textual statements that we manually codify. The long-term vision is to use NLP to parse policy text, but for a working prototype, focus on the format and enforcement.

Policy Representation: We will define a JSON schema for policy rules. For example:

```
{
  "policy_id": "SEC-001",
  "description": "No hardcoded credentials (passwords, API keys) in code",
  "type": "strict",
  "checks": {
      "pattern": "(?i)(password\\s*=|API_KEY|SECRET)",
      "match_type": "regex",
```

```
      "languages": ["python", "javascript"]
  },
  "fix_suggestion": "Use environment variables or secure vault for
credentials"
}
```

Each rule entry can have:

- policy_id: a unique identifier.
- description: human-readable description.
- type: either "strict" or "defeasible" (or perhaps "mandatory"/"recommendation"). Strict means any violation is unacceptable; defeasible means it could be overridden by context.
- checks: This can include information for the **Prosecutor** on how to detect violations:
    - We might have a simple pattern (regex or AST condition) for static analysis, or a reference to a function that implements the check.
    - If using OPA/Rego, this could even be a snippet of Rego policy code or a reference to a rule in a Rego module.
    - Could include which languages it applies to (so the analysis service knows which rules to run for a given artifact).
- (Optionally, normative logic metadata: e.g., if defeasible, what condition would defeat it, but that gets complicated; we can handle exceptions in code for now.)
- fix_suggestion: a short suggestion on how to fix if violated (useful to feed to the generator or display to user).

For the prototype, we can create a **seed set of rules** focusing on common issues:

- Security: No hardcoded secrets (like passwords) , no use of weak cryptography, input validation on external inputs, etc.
- Quality: functions must have logging for key actions, no large functions (enforce modularity), etc.
- We can incorporate some **OWASP Top 10** related code practices. For instance: *"Validate all inputs"*, ensure any input from user is validated , *"Avoid OS command injection"*, flag any use of os.system or backticks that run shell commands.
- Each of these becomes a rule in our JSON.

The **Policy Compiler** in our case will mostly involve loading this JSON. If we had textual policies (like *"Passwords must not be hardcoded"*), an NLP approach would involve identifying the obligation ("must not"), the subject ("passwords"), and object ("hardcoded in code") to create a rule. For now, we skip the NLP and assume rules are predefined or manually added via JSON.

However, to demonstrate the idea, we might implement a very basic "compiler" that can parse a minimal domain-specific language or structured text. For example, we could support policy definitions like:

```
RULE "NoHardcodedSecrets":
    IF code contains pattern /password=|api_key|secret/i
    THEN violation "Hardcoded secret found"
    STRICT
```

This could be parsed by a simple Python script into the JSON schema above. We'd have to write a parser or use a parsing library (like ANTLR or simple regex) if we go that route. Given time constraints, likely we will just define the rules in JSON directly.

Output: The compiled policy rulebase will be stored perhaps as a Python dictionary or a JSON file loaded at runtime. The other modules (especially the Prosecutor and Adjudicator) will use this. For instance, the Prosecutor uses the checks info to run appropriate tests, and the Adjudicator uses policy_id, type (strict/defeasible) to apply logic.

*(We should also consider **policy priorities**: e.g., a regulatory rule might trump a guideline. In argumentation, this would be modeled by an attack from the higher priority rule on the lower. For now, if we include any such, we can simply hardcode priority values and let the adjudicator prefer higher ones.)*

Generator Agent Module

Role: Use AI (LLM) to generate or refactor code that complies with the policies. This module wraps the OpenAI the LLM API.

Implementation Plan: We will use the official OpenAI Python library (openai package) to communicate with the LLM. The generator module will encapsulate prompt construction and API calls, providing a simple interface like generate_code(spec, policies) and fix_code(code, violations).

Prompt Engineering: To ensure the LLM is effective and returns structured output, we need carefully crafted prompts:

- We will use a **system message** that establishes the assistant's role: e.g., *"You are a coding assistant that outputs code following given compliance policies. Provide only the code (and minimal commentary if required) in your response."*
- We will provide the **policies or rules** in the prompt. If the set is large, we may give a summary or only the relevant ones for the given task. For example, if the user wants a function to parse user input, relevant policies might be input validation and logging rules.
- Use an example or explicit format instruction to get JSON output. OpenAI models often obey format if instructed clearly and given an example. We might say: *"Output a JSON object with keys code and analysis. Under code, put the full source code. Under analysis, list any potential issues or uncertainties you foresee."* By *showing* the desired format in the prompt (or a few-shot example), we increase the chance the model returns valid JSON. We can also use triple backticks in the prompt to indicate where code should go.

For **code generation** (/generate): The prompt might look like:

```
System:
"You are an expert Python developer and code assistant. Your task is to
generate code that meets a set of compliance policies. You will be given a
description of a function to implement and a list of policies. You must
produce the function code in Python that adheres to all policies. If you
think the initial solution might violate a policy, adjust the code to
comply."
```

```
User:
"Function spec: Implement a function `process_login(username, password)`
that checks a user's credentials and logs the attempt.
Policies:
1. No hardcoded passwords or secrets in code.
2. Input validation: All inputs must be validated (e.g., check for SQL
injection or script tags in username).
3. Logging: All login attempts must be logged, but sensitive data (like
passwords) should not be logged.
4. Error handling: Do not reveal sensitive info in error messages.
Output format: JSON with keys 'code' and 'analysis'."
```

We may also include a *few-shot example* of a simpler policy and code to guide format. However, the LLM is usually able to follow format instructions if clearly given, especially if we say *"output JSON only, no extra commentary outside the JSON"*.

For **code fixing** (/generate/fix): The prompt will include the current code and the list of violations:

```
System:
"You are an AI developer assistant. You will be given some source code and
a list of policy violations found in it. Your job is to suggest fixes by
providing a revised version of the code that resolves all listed
violations, while preserving the original functionality."

User:
"Here is the code that needs fixes:
```python
<code here>
```

Violations:

- SEC-001: Hardcoded password found on line 10. The code has DB_PASSWORD = "P@ssw0rd" which is not allowed.
- INPUT-003: Missing input validation. The function process_input does not sanitize its user_input argument, leading to potential XSS.

  Please return a JSON with the fixed code."

```
We instruct the model to *not* include the violation list in the output,
just the fixed code. We'll parse the JSON to get the `code` field.

**OpenAI API settings:** We will use `model="gpt-4"`. We should set
`temperature` relatively low (maybe 0.2-0.5) for deterministic compliance
(less creative, more focused on rules). Also, to ensure JSON, we could use
the function calling feature: define a fake function with a schema for
`code` and `analysis` and let the model fill it. If we use that, the OpenAI
API can directly return structured data (which reduces parsing errors).
However, using function calling requires us to define the schema ahead. For
simplicity, we might try just prompt-based formatting first.

**Handling output:** The generator module will attempt to parse the model's
response as JSON. If parsing fails (the model sometimes deviates), we can
implement a simple retry or corrective action: e.g., if the response is not
valid JSON, we can reprompt with something like *"The output was not in the
required JSON format. Please output only JSON."*. Another strategy is to
```
```

post-process the output (e.g., extract a code block), but since the requirement is JSON, we prefer to enforce that format from the start.

Verification of generator's output: After generation or fixing, the new code will be sent again to the Analysis service to verify compliance. This closes the loop.

Prosecutor Agent Module (Static & Dynamic Analysis)

Role: Automate the detection of compliance violations in the code artifact through static and dynamic techniques. We will implement this by leveraging existing tools to avoid reinventing static analysis.

Static Analysis Implementation:
- **Python:** We will use **Bandit** (from PyCQA) for security linting. Bandit can be used via CLI or as a library. As a library, we can import Bandit's `BanditTester` or simply call `subprocess.run(["bandit", "-f", "json", "-"], input=code)` to get a JSON report (Bandit supports JSON output). We'll need to feed the code (or file path). If our code is in-memory, we might save it to a temp file for Bandit to scan. Bandit comes with a set of tests (like B105 for hardcoded passwords, B108 for insecure usage of `mktemp`, etc.) which align with many common policies [oai_citation:30‡dev.to](https://dev.to/angelvargasgutierrez/bandit-python-static-application-security-testing-guide-47l0#:~:text=Bandit%3A%20Python%20Static%20Application%20Security,Bandit). For any rule not covered by Bandit, we can add custom checks:
  - e.g., For a policy like "no `print()` statements (use logging instead)", we could do a simple AST parse (Python's `ast` module) to find `Print` nodes.
  - Or use **Semgrep**, which allows writing rules in a YAML for patterns in code. Semgrep also supports multi-language, but adding it is optional.
- **JavaScript (if needed):** We could use **ESLint** with appropriate plugins (ESLint has rules for detecting things like eval usage, or we can write a custom rule for disallowed patterns). For the prototype focusing on Python, we may skip JS, but designing for extension, we mention it.
- **Secrets detection:** Even outside Bandit, we might integrate a tool like **Detect Secrets** (by Yelp) or **TruffleHog** to scan code for API keys, passwords, etc. These tools scan for high-entropy strings or known patterns (like AWS key format). Since Bandit already flags some hardcoded secrets, this might be redundant. But if needed, a simple regex search for common secret patterns can be done (as in the policy example regex for "password|secret").
- **Coding standards:** If we had style policies (like naming conventions), we could incorporate **flake8** or **pylint** checks, but that might be beyond compliance (more quality than compliance). Focus on security and correctness rules first.

The static analyzer will produce a list of findings. We will map these to our policy rules via rule IDs. For example, if Bandit outputs an issue "Hardcoded password - MEDIUM - location: file.py:10", we know that corresponds to our rule SEC-001. We can maintain a mapping of Bandit's test IDs to our policy IDs if needed, or more simply, search the issue description for keywords to infer the rule.

Dynamic Analysis Implementation:
- We will create a small testing harness for the code. This is tricky because arbitrary code might need context or might not run in isolation. However, for prototype, we might target a specific kind of code (like a single function or module) and we can execute it in a sandbox (perhaps using `exec` safely or writing it to a file and importing). **Important:** running arbitrary generated code has security implications. In a real

system, you'd run this in an isolated container or sandbox with limited permissions. For our prototype, we can assume we trust our environment or simulate dynamic tests without truly running untrusted code (maybe just symbolically evaluate, or only run if it's safe).
- **Hypothesis:** We can write parameterized tests using Hypothesis if we know the function signature. For example, if we have a function `process_login(username, password)`, we can use Hypothesis to generate many strings for `username` (including SQL injection attempts, very long strings, etc.) and ensure the function behaves (doesn't crash, doesn't log the password, etc.). We might integrate this only for some known patterns rather than arbitrarily for all code.
- **Fuzzing:** There are Python fuzzing frameworks (like `afl` or using Hypothesis as fuzzer). Given time, a simpler approach: if the code reads from input or arguments, we call it with some representative bad values to see if it violates something. For instance, if there's a database query, try a `' OR '1'='1` input to see if the returned result suggests injection (though detecting that automatically is non-trivial).
- **Runtime monitors:** If we run the code, we can instrument it to catch certain things:
  - For example, if policy says "no network calls", we could monkey-patch `socket` or `requests` and see if they get called.
  - If policy says "all errors must be handled", we could see if any exception propagates out of the function when fuzzing.
  - Use Python's `logging` capture to see if sensitive info is printed.

Dynamic analysis is the most challenging to automate generally. For the prototype, we may demonstrate it with a simple example test. For instance, if the code includes a logging of user input, we can simulate an attack string and verify that the log output is sanitized (or that the code removed dangerous characters). If the check fails, we record a violation.

Output from Prosecutor module: We define a schema for violations as mentioned: list of objects with `rule_id`, description, location (if applicable), and maybe a `evidence` field (like the offending line of code or the input that caused it). This output goes to Adjudicator.

Adjudicator Module (Argumentation & Decision Logic)

Role: Take the rules and violations and determine the final compliance status, with explanations. This is the core of proving compliance, ensuring that for every rule either it holds or there is a justified exception.

Implementing Argumentation Logic: We will implement a simplified structured argumentation approach:
- We prepare two sets of statements:
  1. Compliance claims: e.g. "Rule X is satisfied". We create one for each rule in the policy set that applies to this artifact. Initially, assume all are true.
  2. Violation claims: for each violation found, "Rule X is violated under scenario Y".
- We then establish attack relationships:
  - Each violation claim attacks the corresponding rule's compliance claim.
  - If there are any rules that provide exceptions or counter-arguments, include those. For example, consider a defeasible rule: *"Data should be encrypted unless it is public data."* If a prosecutor finds "Data not encrypted in module A" (violation of encryption rule), but we know module A deals only with public data (perhaps from metadata), then an argument "Module A deals with public data (exception)" would attack the violation argument. We would mark the exception as higher priority (since "unless" indicates an override).

  - In our initial prototype, we might not implement many complex exceptions unless we deliberately include one policy with an exception to demonstrate. If we do, we need to encode that logic, perhaps as part of the policy JSON (like `exception_condition: ...`).
- Once the graph is built, we compute the **grounded extension**. This can be done iteratively:
  1. Start with **Accepted = {}**, **Rejected = {}**.
  2. Find any argument with no attackers (or all its attackers are already rejected), accept it.
  3. If an argument is accepted, then all arguments it attacks are rejected.
  4. Repeat until no change.
  5. Any remaining arguments not accepted or rejected at the end can be considered rejected (grounded extension is a complete set of accepted ones).
  6. In practice, compliance claims start unattacked *unless* a violation is present. So initially, all compliance claims for which no violation exists are accepted. Any compliance claim that has a violation will have an attacker, so we don't accept it initially. If that violation has an exception attacker, there is a cycle that grounded semantics resolves by priority (we might need to enforce priority by saying the exception is a stronger argument that defeats the violation).
  7. The outcome: all accepted compliance claims correspond to rules that hold; any compliance claim not accepted means that rule is considered broken.

Using **pygarg** or another solver can verify this logic. Pygarg could let us simply ask: is the argument "RuleX_satisfied" skeptically accepted (true in all extensions) or credulously (true in at least one extension)? Under grounded (which is a single extension), we just need that extension. Since grounded extension is unique, we can just compute it directly.

Decision and Proof Prep: The Adjudicator will produce:
- A boolean overall compliance (`compliant` True/False).
- A list of violated rules (if any).
- Potentially, a list of *waived* rules (if an exception applied, e.g., "Rule Y would normally apply but context Z exempted it"). This could be shown to the user as "Rule Y was not enforced due to exception Z."
- Explanations per rule: for each rule, indicate why it passed (no violations, or violations countered) or why it failed (no counter to violation). This is essentially the *proof rationale* for each policy.

This module also is where we integrate **policy priorities**. If we have priority levels, we implement that as: a higher priority rule's violation might automatically make the artifact non-compliant even if a lower priority rule would have allowed something. But formalizing that might be complex. As a simpler approach, we ensure strict rules (likely high priority) must pass. Defeasible ones (lower priority) can be waived if needed.

Logging: The adjudicator should log the decision outcome, including which arguments were considered. This helps in debugging and audit. For instance, we log: "Accepted arguments: {Rule1_complied, Rule2_complied, Exception3_applies, ...}. Rejected arguments: {Violation_rule2_x}."

Proof Assembler Module

Role: Construct the final proof artifact and handle digital signing and logging. This is the final step once an artifact is deemed compliant.

Proof Bundle Structure: We decide on a format, likely JSON for ease of machine parsing (and it can be extended or converted to JSON-LD/RDF for more semantic use if needed). Example structure:

```json
{
  "artifact": {
    "name": "process_login.py",
    "hash": "SHA256:abcd1234...",
    "generated_by": "the LLM",
    "timestamp": "2025-11-20T15:02:00Z"
  },
  "policies": [
    {"id": "SEC-001", "description": "No hardcoded secrets", "result": "satisfied"},
    {"id": "SEC-002", "description": "Use parameterized SQL queries", "result": "satisfied"},
    {"id": "INPUT-001", "description": "Validate all inputs", "result": "satisfied"},
    {"id": "LOG-001", "description": "No sensitive data in logs", "result": "satisfied"}
  ],
  "evidence": [
    {"rule_id": "SEC-001", "type": "static_analysis", "tool": "Bandit",
      "output": "No hardcoded secrets found."},
    {"rule_id": "INPUT-001", "type": "dynamic_test", "test": "fuzz_special_chars",
      "output": "All inputs were properly sanitized (no XSS or SQLi triggered)."}
  ],
  "argumentation": {
    "accepted": ["SEC-001_complied", "SEC-002_complied", "INPUT-001_complied", "LOG-001_complied"],
    "rejected": ["(none)"],
    "exemptions": []
  },
  "decision": "Compliant",
  "signed": {
    "signature": "<base64_signature>",
    "signer": "ACPG-Adjudicator",
    "algorithm": "ECDSA-SHA256"
  }
}
```

Fields explained:

- artifact: identifies the artifact (could include the entire code or a hash of it to tie the proof to the exact code version).
- policies: each policy rule with its result. Could also have a field if a rule was waived due to an exemption (we could list "result": "waived" and maybe an exemption_reason).
- evidence: a list of evidence items. These can be cross-referenced by rule. Evidence might be outputs from tools (truncated or summarized), or a reference (like a file path or log ID). For a real system, storing full logs might be too bulky; we can store a summary or a hash of logs for integrity.
- argumentation: (optional in output) the internal reasoning result. For transparency, we can include which arguments were accepted or rejected. If needed, also the attack

relations, but that might be overkill for the guide. A condensed form is fine, just to illustrate that we have internally validated the compliance via logic.
- decision: overall decision, e.g. "Compliant" or "Non-compliant".
- signed: contains the digital signature information. We will generate a signature by serializing the above sections (except the signature itself) in a canonical form (to ensure consistency, we might use JSON canonicalization or simply sort keys and then sign). The signature is the base64 of the cryptographic signature. signer could be an identifier or certificate of the signing entity. algorithm notes what algorithm was used.

Digital Signing: We will create an RSA or ECDSA key pair for the prototype. For example, use ECDSA (secp256r1 curve) for compact signatures. In Python, using cryptography library:

```
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.asymmetric.utils import Prehashed
from cryptography.hazmat.primitives import serialization

# Generate key (in practice, use a fixed key saved securely)
private_key = ec.generate_private_key(ec.SECP256R1())
data = json.dumps(proof_dict, sort_keys=True).encode()  # canonical JSON
signature = private_key.sign(data, ec.ECDSA(hashes.SHA256()))
```

We then base64 encode signature. The public key (or certificate) corresponding to private_key should be distributed to allow verification. We can include the public key in the proof (or have it known out-of-band). For simplicity, we might skip including the actual key in the JSON, but note that in a real scenario the verifier needs the public key.

Verification: A verifier can take the artifact.hash, independently compute the hash of the code, check that it matches, then use the public key on the signature to verify the proof data. Because our proof includes the important details (like which rules were satisfied), a verifier could also re-run any needed analysis if they distrust the results. But the point is they don't have to re-run everything, they can trust the proof if the signature is valid.

Logging & Audit: We will log each proof generation in a ledger (could be a simple log file or database). Each entry might include the artifact hash, timestamp, and decision. This provides an audit trail. In future, this could integrate with an immutable log or blockchain if desired, but for now, a local log file suffices.

Output Delivery: The proof bundle JSON can be offered to the user as a downloadable file (e.g., artifact_proof.json). Also, we might present a human-readable summary in the UI (like a table of rules and pass/fail).

Prompt Engineering Instructions (for the AI Assistant)

Using the AI assistant (the LLM) effectively is key to this prototype. We outline how to craft prompts and control outputs for our needs:

- **Aim for Structured JSON Output:** Since we want JSON from the model, always **instruct the model explicitly about the output format**. For example, prepend the user prompt with: *"IMPORTANT: Respond only with a JSON object following this exact format: ..."* and describe the keys. Models like the LLM usually follow this, but

adding an example of a small JSON in the prompt can further increase reliability. Avoid giving the model free-form output opportunities; whenever possible, constrain it to the format (especially for the generator which might be inclined to explain code in English, we must suppress that tendency by instruction).

- **Few-Shot Examples:** If the model still produces extraneous text, consider a few-shot approach:
    - Example: Provide a dummy policy and code generation example in the prompt. E.g., *"Policy: All print statements are banned.\nUser asks for a hello world.\nAssistant output: {"code": "...", "analysis": "..."}"*. Then say *"Now given the following actual request and policies, output in the same format."* This shows the model exactly how to format the answer.
- **System vs User Message:** Utilize the system message to set the stage. For generation, system message should include the role and high-level guidance ("You are a code generator that must adhere to policies."). The user message will contain the specifics (function spec and list of rules). For fixing, system message says ("You are a code reviewer and fixer…"), user message contains code and violations.
- **Avoiding Unwanted Content:** Since compliance often means *not* doing certain things (e.g. not logging passwords), the prompt should clarify that. But we must be careful: just saying "Do not include X" might lead the model to omit needed functionality. Instead, frame it as *"Ensure the code does not violate the following rules: ..."*. If the model might be tempted to explain its changes, instruct: *"Do not write any explanation or reasoning outside of the JSON. If you need to comment, do it inside the code as comments."*
- **Token Limits:** Our prompts with multiple rules and possibly code can get long. the LLM supports large contexts (8K or even 32K tokens in some versions), but we should still be concise in what we send. For example, if the policies list is huge, perhaps filter to those relevant to the current code or summarize them. If the code is large and we're asking for fixes, consider focusing on the sections with issues (though risk missing context). In a prototype, we can assume manageable sizes (a few hundred lines at most of code, and maybe a dozen policies).
- **OpenAI Function Calling:** An advanced option is to use the function-calling API. We can define a function schema like:

```
functions=[{
  "name": "return_code",
  "description": "Return the generated code and analysis",
  "parameters": {
     "type": "object",
     "properties": {
        "code": {"type": "string", "description": "The source code"},
        "analysis": {"type": "array", "items": {"type": "string"},
"description": "List of any issues or important compliance notes"}
     },
     "required": ["code"]
  }
}]
```

- Then call the chat with this. The model, if it follows, will respond with a JSON that the OpenAI library can directly interpret as a function call, giving us structured data. This guarantees valid JSON structure as defined. If time permits, implementing this is ideal for robustness. Otherwise, stick to prompt-only.

- **Testing Prompts:** We should test a few prompts with the LLM by hand (in the playground or via a small script) to see if it outputs correctly. Adjust until it consistently does.
- **Handling Model Hallucinations or Non-compliance:** If the model output violates a policy (ironically it might do so, e.g., suggest insecure code), our loop will catch it via the Prosecutor. We then rely on the feedback to correct. But to minimize that, include the major policies in the generation prompt so the model is aware. For example: *"Policy: no hardcoded credentials. Therefore, do not put any password or key in the code, use a parameter or config."* By stating the rationale, the model is more likely to follow (models do well when they understand *why* in context).
- **Clarity and Specificity:** When describing violations to fix, be explicit. Instead of *"Fix the security issues"*, list them specifically as shown in the example. The model should address each one. If it misses one, our analysis will find it again and we can reiterate (maybe emphasizing the missed one more strongly in a second round prompt).

Example Prompt for Generation: (combining pieces discussed)

```
[System Message]
You are an AI coding assistant. You output code that strictly follows given
policies. Output only JSON as specified.
Format: {"code": "<code here>", "analysis": ["<issue1>", "<issue2>", ...]}

[User Message]
Task: Write a Python function called `authenticate_user(user_input)` that
checks if the input matches a stored password and prints either "Welcome"
or "Access Denied".
Policies:
- No hardcoded secrets: do not hardcode passwords or API keys in code (SEC-
001).
- Validate all inputs: treat user_input as untrusted, avoid risky functions
(INPUT-001).
- No cleartext passwords in logs or outputs (SEC-002).
- Use constant-time comparison for secrets (SEC-003).
Remember to apply all policies. If something is unclear, make safe
assumptions.
```

We expect the LLM to return JSON like:

```
{"code": "def authenticate_user(user_input):\n    #
...implementation...\n", "analysis": ["Followed all policies: no hardcoded
secret, input validated, etc."]}
```

The analysis might be redundant since we will analyze anyway, but we include it to see the model's thought or to ensure it considered the policies. We could exclude analysis to get just code.

Example Prompt for Fixing:

```
[System]
You are an AI code fixer. You will receive code and a list of policy
violations. Modify the code to fix all violations. Output JSON {"code":
"..."} with only the fixed code.

[User]
```

```
Code to fix:
```python
def authenticate_user(user_input):
    stored_password = "P@ssw0rd"  # hardcoded password
    if user_input == stored_password:
        print(f"Welcome, user {user_input}!")
    else:
        print("Access Denied")
```

Violations:

- SEC-001: Hardcoded secret (stored_password).
- SEC-002: Sensitive data printed (printing password in welcome message).
- INPUT-001: No input validation or sanitization on user_input.

Please fix these issues.

```
We expect the assistant to output JSON:
```json
{"code": "def authenticate_user(user_input):\n    import os\n
stored_password = os.environ.get('STORED_PW')\n    # Validate input...\n
if not isinstance(user_input, str):\n        return \"Access Denied\"\n
# ... constant-time comparison ...\n    if hmac.compare_digest(user_input,
stored_password):\n        print(\"Welcome!\")\n    else:\n
print(\"Access Denied\")\n"}
```

(plus maybe some comments). Our analysis then would find no hardcoded password (SEC-001 fixed), no secret in print (SEC-002 fixed), input type check (partial fix for validation). If something still not perfect, it goes another round.

By following these prompt guidelines, we harness the LLM's capabilities while keeping outputs predictable and easy to integrate into the system.

Data Formats and Schemas

Standardizing data formats between modules is crucial. Below are the key schemas and examples:

Policy Rule Schema

We define policies in a JSON format (could be a file policies.json). For example:

```
{
  "policies": [
    {
      "id": "SEC-001",
      "description": "No hardcoded credentials (passwords, API keys) in
code",
      "type": "strict",
      "severity": "high",
```

```
      "check": {
        "type": "regex",
        "pattern": "(?i)(password\\s*=|api[_-]?key\\s*=|secret\\s*=)",
        "message": "Hardcoded secret suspected"
      }
    },
    {
      "id": "SEC-002",
      "description": "Do not log or print sensitive information (like
passwords)",
      "type": "strict",
      "severity": "medium",
      "check": {
        "type": "ast",
        "function": "find_print_of_variable",
        "target": "password"
      }
    },
    {
      "id": "INPUT-001",
      "description": "All user inputs must be validated or sanitized",
      "type": "defeasible",
      "severity": "high",
      "check": {
        "type": "manual",
        "message": "No direct check - violation if user input is used in
dangerous context without validation"
      }
    }
  ]
}
```

- Each rule has an id and description.
- type (strict/defeasible) as explained.
- severity (informational, low, medium, high, critical) to help prioritize fixes.
- check: We allow multiple types:
    - "regex": means the prosecutor can simply search the code text for the given pattern. Here, case-insensitive password= or apiKey= would flag a violation of SEC-001. Regex checks are quick for patterns like hardcoded secrets.
    - "ast": means we'll do an AST-based analysis. For SEC-002, the idea is an AST function that finds any print/log statement that includes a variable named 'password' or likely containing sensitive data. We might implement find_print_of_variable(code_ast, var_name="password") to detect that.
    - "manual": indicates no automatic check; perhaps it requires human judgement or a complex analysis. For instance, input validation (INPUT-001) might be too broad for a simple automated pattern, we might need to see if the code calls any validation function or escapes input. For now, we could mark it violated if, say, the code uses the input in a SQL query or an eval without prior checks. Alternatively, we rely on dynamic tests to expose if unvalidated input can cause issues. A "manual" check might always pass unless a dynamic test fails or a heuristic triggers.

Policy Knowledge Base: Once compiled, we may transform this into internal objects. For argumentation, we might create a structure:

```
policy_rules = {
```

```
    "SEC-001": {"type": "strict", "depends_on": None},
    "SEC-002": {"type": "strict", "depends_on": None},
    "INPUT-001": {"type": "defeasible", "exception": "INPUT-001-exc"},
    "INPUT-001-exc": {"type": "defeater", "attacks": "INPUT-001"}
}
```

This shows how a defeasible rule might have a linked exception rule. But unless we have concrete exceptions defined, we may not need that complexity for now.

Artifact and Violation Data

When the code artifact is submitted or generated, we might internally represent it as:

```
{
  "artifact_id": "session123_artifact1",
  "language": "python",
  "code": "def foo(): ...",
  "metadata": {
      "generator": "gpt-4",
      "generation_time": "2025-11-20T15:02:00Z"
  }
}
```

This can be stored or passed between calls (though often we just pass code directly to analyzers).

Violations (output of analysis):

```
{
  "artifact_id": "session123_artifact1",
  "violations": [
    {
      "rule_id": "SEC-001",
      "description": "Hardcoded secret found: variable DB_PASSWORD on line
5",
      "line": 5,
      "evidence": "DB_PASSWORD = 'P@ssw0rd'",
      "detector": "Bandit B105",
      "severity": "HIGH"
    },
    {
      "rule_id": "INPUT-001",
      "description": "User input `username` is used in SQL query without
validation",
      "line": 20,
      "evidence": "f\"SELECT * FROM users WHERE name = '{username}'\"",
      "detector": "grep 'SELECT'",
      "severity": "HIGH"
    }
  ]
}
```

Here we include where possible:

- rule_id to tie to policy.
- description explaining the issue.

- line number in code (if applicable).
- evidence snippet.
- detector (which tool or method found it).
- severity if the tool provided one or map from policy severity.

This JSON can be directly used by the Adjudicator.

Argumentation Graph Schema

If we wanted to serialize the argumentation graph:

```
{
  "arguments": [
    {"id": "SEC-001_complied", "rule_id": "SEC-001", "type": "compliance"},
    {"id": "SEC-001_violated_1", "rule_id": "SEC-001", "type": "violation",
"evidence": "line 5 hardcoded secret"},
    {"id": "INPUT-001_complied", "rule_id": "INPUT-001", "type":
"compliance"},
    {"id": "INPUT-001_violated_1", "rule_id": "INPUT-001", "type":
"violation", "evidence": "no input validation on username"},
    {"id": "INPUT-001_exc_1", "rule_id": "INPUT-001", "type": "exception",
"details": "username is not used in dangerous context"}
  ],
  "attacks": [
    {"attacker": "SEC-001_violated_1", "target": "SEC-001_complied"},
    {"attacker": "INPUT-001_violated_1", "target": "INPUT-001_complied"},
    {"attacker": "INPUT-001_exc_1", "target": "INPUT-001_violated_1"}
  ]
}
```

This shows each compliance claim and each violation as an argument node, and an exception (if any). The attacks array defines the edges. This structure could be used with an argumentation solver or for debugging. For output in the proof, we probably simplify it.

Proof Bundle Schema

As discussed earlier, the proof bundle is essentially a combination of the above data plus the signature. A formal schema (in e.g. JSON Schema terms) might be:

```
{
  "type": "object",
  "properties": {
    "artifact": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},
        "hash": {"type": "string"},
        "language": {"type": "string"},
        "generator": {"type": "string"},
        "timestamp": {"type": "string"}
      },
      "required": ["hash", "timestamp"]
    },
    "policies": {
      "type": "array",
      "items": {
```

```
        "type": "object",
        "properties": {
          "id": {"type": "string"},
          "description": {"type": "string"},
          "result": {"type": "string", "enum": ["satisfied", "violated",
"waived"]}
        },
        "required": ["id", "result"]
      }
    },
    "evidence": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "rule_id": {"type": "string"},
          "type": {"type": "string"},
          "output": {"type": "string"}
        }
      }
    },
    "decision": {"type": "string", "enum": ["Compliant", "Non-compliant"]},
    "signed": {
      "type": "object",
      "properties": {
        "signature": {"type": "string"},
        "signer": {"type": "string"},
        "algorithm": {"type": "string"}
      },
      "required": ["signature"]
    }
  },
  "required": ["artifact", "policies", "decision", "signed"]
}
```

Not every field is required (like evidence can be optional or partial), but this is the general expectation.

In implementation, we will populate this structure after adjudication:

- Fill artifact info (we can compute hash easily using SHA256 of the code string).
- For each policy, mark result. If a rule was violated, result “violated”; if it was not (and not applicable or something) then “satisfied”; if we explicitly waived it, “waived”.
- Evidence: we collate from prosecutor outputs and possibly dynamic tests. We might include evidence only for violated or fixed rules to keep it concise (for fully satisfied rules, “Code scanned by Bandit: no issues found” can be evidence).
- Decision: straightforward.
- Then sign it.

We also consider making the proof **tamper-evident** beyond the signature: including the code’s hash means the proof is bound to that exact artifact version. If someone changes the

code, the hash breaks and the proof no longer applies (which is what we want, you'd need to re-run compliance if code changes).

Logging, Audit, and Signing Mechanisms

Logging: Throughout the system, we will implement detailed logging for traceability:

- The Policy Compiler logs what rules were loaded/compiled (and any errors in rules).
- The Generator logs prompts sent to the AI (perhaps sanitized if containing sensitive data) and responses (this is useful to debug why the AI produced something). Caution: API keys or secrets in prompts should be redacted in logs.
- The Prosecutor logs each tool's output (e.g., log the Bandit report, Hypothesis test results). If a tool fails (like syntax error in code prevents analysis), log that too.
- The Adjudicator logs the final set of accepted/rejected arguments and the compliance decision.
- The Proof Assembler logs the creation of proof and the signature details (e.g., first 8 chars of signature for reference).

We can use Python's logging module to output to console or file. Potentially structure logs in JSON for machine parsing, but not necessary for the prototype.

We should also ensure **error logging**: if the OpenAI API call fails or times out, log the exception; if Bandit crashes, log it, etc., so that troubleshooting is possible.

Audit Trail: The ultimate audit artifact is the proof bundle. In addition, we might maintain an **audit log file or database** that records each compliance run:

e.g., a simple CSV or database table with columns: run_id, timestamp, artifact_hash, compliant (Y/N), proof_hash, policies_used. The proof itself can be stored on disk (named by run_id or artifact hash). This allows later retrieval and verification. For instance, an auditor could pick a random artifact, compute its hash, look up the proof by that hash, and verify signature and content.

For a high-assurance system, we might integrate with an immutable ledger:

- e.g., append the proof hash or entire proof to a blockchain or a service like AWS QLDB, so that even the system operators cannot alter past proofs without detection.
- In the prototype, we can simulate immutability by simply not deleting or altering logs, and maybe computing a running hash of the log (like a simple Merkle chain of entries).

Digital Signing Implementation: As described, we'll use a cryptographic library for signing. Concretely:

- Generate a key pair once (when system starts). Possibly load from config if we want a persistent identity.
- Use ECDSA or RSA. ECDSA with a NIST P-256 curve and SHA256 is FIPS-approved and gives a shorter signature (~64 bytes) which base64 to ~88 chars, which is fine.

- After assembling the proof JSON (except the signature part), serialize it in a stable way. Sorting keys in JSON is one way (the JSON spec doesn't guarantee order, but if we sort, then the same data will produce the same string every time).
- Compute a SHA-256 hash of that JSON string and then sign it with the private key.
- Attach the signature (and possibly the signer's public key or an identifier). If including the raw public key, we can use e.g. PEM format or JWK format. Alternatively, just document that the public key is stored in config/public_key.pem and provide that to auditors separately.

We also ensure the signature covers everything except itself. In our schema above, the signed.signature is outside of what's signed obviously. But signed.signer or algorithm could either be included or not in the signed payload. Probably safe to include algorithm and signer so that those can't be tampered either (though if someone changes algorithm to something else, the signature wouldn't verify because we would verify with the real algorithm; anyway, minor detail).

Signature Verification: We will provide (maybe not in the prototype UI, but as part of documentation) instructions to verify:

- Use the known public key to verify the signature on the proof JSON.
- Optionally, we could write a small verification script as part of the system for internal use. For example, an endpoint /verify that accepts a proof JSON and returns true/false if signature is valid and artifact hash matches provided code. That could be a future extension or used in CI (e.g., an independent verifier microservice that clients can use to double-check proofs).

Confidentiality Consideration: Our proof contains potentially sensitive info (like evidence strings, which might include code snippets or test input). If code is proprietary, the proof is essentially an audit artifact that may be shared with regulators or customers. We should consider if any piece of evidence should be sanitized. For instance, if the code had API keys (which it shouldn't, by policy!), they would appear in evidence — but since policy forbids them, ideally no secrets are in the proof. Log sanitization is itself a policy requirement (no sensitive data in logs), which our system enforces on the code, and by extension we should enforce on our logs and proofs. We can assert that *the ACPG system will not log or include any real secrets*, because any found would cause a violation that stops the process.

Time-stamping: The proof should include a timestamp. For stronger audit, one might want an external timestamp (to prove the proof was created at or before a certain time, e.g., via Time Stamp Authority). We won't implement that, but including system time is straightforward. If needed, one could later notarize the proof by putting a hash on a blockchain or similar.

In summary, the Logging, Audit, and Signing mechanisms ensure that every compliance check is recorded and that the final result is verifiable. The digital signature, in particular, turns the proof bundle into a **tamper-evident compliance certificate** for the artifact , which is the key deliverable of ACPG.

Example Compliance Rules and Tests

To validate the system, we will start with a few concrete policy rules and run sample code against them. Below are some example rules (drawn from common security best practices) and how the ACPG prototype would enforce them:

1. **No Hardcoded Secrets (Secrets Management)**, *"Credentials and secrets must not be hardcoded in source code."* (This corresponds to OWASP recommendations against hardcoded passwords and is a top security risk).
    - **Policy ID:** SEC-001 (strict).
    - **Enforcement:** The static analysis (Bandit) has a built-in check for hardcoded passwords (e.g., **Bandit test B105** flags the use of literal strings that look like passwords). We also use regex to catch common patterns like "password=" or API keys.
    - **Example Violation:** In code: db_password = "P@ssw0rd". Bandit or our regex will catch this as a violation of SEC-001.
    - **Auto-fix:** The Generator, when prompted to fix, might replace the hardcoded password with a call to os.getenv("DB_PASSWORD") or move it to a config file, thereby resolving the issue.
    - **Proof:** The proof will note SEC-001 was initially violated at line X and then satisfied after fix, including evidence of removal of the literal.
2. **Input Validation**, *"All external inputs must be validated or sanitized before use."*
    - **Policy ID:** INPUT-001 (defeasible, because there might be contexts where input is inherently safe, but generally high priority).
    - **Enforcement:** Static analysis could try to detect if input is used unsafely. For example, if user input flows into a database query or an eval, that's a red flag. We might not catch all via static means, so dynamic testing comes into play. For instance, we fuzz the input with a SQL injection attempt and see if the code is vulnerable (e.g., does it not escape input and directly uses it in a query?). If a test demonstrates a successful injection or XSS, that is concrete evidence of violation .
    - **Example Violation:** Code that builds an SQL statement by concatenation: query = "SELECT * FROM users where name = '"+ user_input +"'". Our static check could search for pattern SELECT * FROM plus use of an unsanitized variable. Or we run a dynamic test with user_input = "' OR '1'='1" and see if the output indicates a likely injection (maybe the code returns more data than it should, etc.). Another example: passing user input to os.system without cleaning.
    - **Auto-fix:** The AI could fix such code by using parameterized queries (e.g., using database library parameter substitution) or at least escaping the input. Or it might implement a validation function (like allow only alphanumeric in user_input for the query).
    - **Proof:** The proof will show INPUT-001 satisfied, possibly referencing a test: *"Tested with special characters and SQL meta-characters, no SQL injection achieved."* If an exception was made (say the code is a trivial script not interacting with external systems), we might mark the rule as waived, but more likely we enforce it regardless for demonstration.
3. **Logging Policy**, *"Do not log sensitive information."*
    - **Policy ID:** SEC-002 (strict).

    - **Enforcement:** Static check can look for logging.info or print statements that include variables named like password, secret, etc., or large data dumps. We might implement a heuristic: if a variable name contains "password" and is used in a log, flag it.
    - **Example Violation:** print(f"User logged in with password {pwd}"). This obviously prints a password. The static analyzer can catch that string formatting includes pwd (assuming pwd holds a password).
    - **Auto-fix:** The AI would remove or mask the password in the log. E.g., print("User logged in") or print("Password provided length:", len(pwd)) instead of the actual password.
    - **Proof:** Show that originally the password was logged, then changed to not reveal it. Provide evidence that no sensitive data appears in logs (maybe by scanning the code again for such patterns and finding none).
4. **Error Handling**, *"Exceptions should be caught and handled; do not expose stack traces or sensitive info in errors."*
    - **Policy ID:** ERR-001 (strict).
    - **Enforcement:** Static: look for usage of generic exception catches vs none. If a function calls something that can throw but there's no try/except, we might consider it a violation (depending on context). Dynamic: we could deliberately trigger an error (e.g., passing bad data) to see if it crashes or if it returns a controlled error. If it crashes with a stack trace, that's a violation (in a web app context; in a script, maybe not as severe).
    - **Example Violation:** Code that does not catch a ValueError from converting input to int, so it just raises and prints a traceback.
    - **Auto-fix:** The AI can add try/except around that portion and handle it (maybe returning an error message without internal details).
    - **Proof:** Indicate that exceptions are now caught; no raw stack trace will be exposed (could show the new except block in evidence).
5. **Use Safe Functions**, *"Certain dangerous functions are disallowed."*
    - **Policy ID:** SEC-003 (strict).
    - **Enforcement:** Static analysis will simply look for these patterns. E.g., eval() usage, or exec in Python, or System.out.exec in Java, etc., depending on language. These are straightforward to grep/AST for.
    - **Example Violation:** result = eval(user_input). This is a big no-no. Our static check flags eval(as violation.
    - **Auto-fix:** The AI might replace eval with a safer parser logic, or if it's not needed, remove it, or use literal ast.literal_eval (if appropriate).
    - **Proof:** Show that no eval remains in code (evidence by static scan showing the function is gone).

These example rules cover a broad range: secrets management, input validation, output sanitization, error handling, dangerous functions. We will test the system with code snippets that intentionally violate these to see the loop in action:

Sample Test Scenario:

- Take a piece of code (perhaps a small web login handler) that has multiple issues:

```
# sample vulnerable code
import os
SECRET_KEY = "ABC123SECRET"  # hardcoded secret
def login(user, pwd):
    print(f"Logging in user {user} with password {pwd}")  # logs sensitive
info
    stored_pwd = get_password_from_db(user)
    if pwd == stored_pwd:
        os.system("echo 'User logged in'")  # not dangerous but just an
example usage
        return "Welcome!"
    else:
        return "Access Denied: " + str(Exception("Invalid credentials"))
```

- Violations: Hardcoded SECRET_KEY (SEC-001), logging password (SEC-002), returning exception string potentially (ERR-001, it's showing internals), maybe use of os.system (not too unsafe here but if we had a policy against any os.system usage that could be flagged too as SEC-003).
- The ACPG process would:
    - Static analysis: flag SECRET_KEY, flag the print of pwd. Possibly flag returning Exception text (not sure with simple tools, but we can custom check for concatenating Exception to string).
    - Adjudicator: sees violations, marks non-compliant.
    - Generator fix: likely remove the SECRET_KEY (or use os.environ.get), remove or mask password in print, and change the error handling (maybe not show exception text).
    - New code might look like:

```
import os
def login(user, pwd):
    # Removed logging of password for compliance
    print(f"Logging in user {user}")
    stored_pwd = get_password_from_db(user)
    if pwd == stored_pwd:
        os.system("echo 'User logged in'")  # this might remain if not
flagged
        return "Welcome!"
    else:
        # handle error without revealing details
        return "Access Denied"
SECRET_KEY = os.environ.get("APP_SECRET")  # moved to env at end, or maybe
eliminated
```

- (It might do things in a slightly different order, but that's fine.)
- Static analysis again: now SECRET_KEY line is not a literal (assuming env var or maybe gone), print line is safe, error message doesn't leak exception. If os.system was a concern, it might remain unless we had a policy specifically. Let's say we did have SEC-003 banning system commands: then that usage would be flagged and the AI might remove it or mark as safe. Possibly, the AI might not know to remove os.system unless told by policy, so if we had a rule like "No OS command execution", it would catch and then fix (maybe remove or use a Python function instead).
- Adjudicator: all issues resolved, declares compliant.
- Proof: lists how each of those rules got satisfied in the final code, with references to the changes.

We will prepare such a scenario to manually verify the prototype's flow.

Additionally, we test edge cases:

- Code that is already compliant (to ensure no false positives and system correctly outputs a proof without needing fixes).
- Code that cannot be made compliant automatically: for example, if a rule says "All cryptography must use AES-256" and the code uses MD5 hashing which is fundamentally wrong for that purpose, the AI might struggle to refactor to a completely different approach without more context. The system should then either flag it as non-compliant or require human intervention. For the prototype, we might not tackle such complex cases, but it's something to note (in such a case, the loop might iterate and the AI might keep doing minimal changes or confuse MD5 with something else, eventually we may have to stop and say manual fix needed).

By walking through these examples, we ensure that:

- The static checks and dynamic tests we wrote actually trigger (and that our integration of Bandit/Hypothesis works).
- The generator receives actionable feedback (the violations list) and that our prompt makes it address them.
- The adjudicator correctly interprets the results.
- The final proof bundle accurately reflects the state.

We should verify that in the final proof for a test snippet, if we intentionally leave one issue unfixed (say we limit iterations to 1 for a test), the proof would show non-compliance on that rule. That is also a valid outcome, the system should be able to produce a proof bundle even if not compliant (essentially documenting which rules failed, which can be used as a report for developers).

Development Roadmap

Implement one module at a time and gradually integrate:

Milestone 1: Basic Static Compliance Checking Pipeline

- **Goal:** Input code, run static checks, output a compliance report.
- **Tasks:**
  - Define a few sample policy rules in JSON (e.g., SEC-001 and SEC-003 from above).
  - Implement the **Prosecutor (Static)** module to scan code using Bandit or simple regex checks based on the policy.
  - Implement a minimal **Adjudicator** that simply marks a rule failed if any violation for it is found (since initially we handle only strict rules without exceptions).
  - Create a CLI or simple FastAPI endpoint /check that accepts code (or file) and returns JSON of violations and pass/fail for each rule.
- **Test:** Use a Python file with a known issue (like a hardcoded password) and verify that the system catches it and reports non-compliance. No AI involved yet.

Milestone 2: Integrate the LLM for Code Generation/Fixing

- **Goal:** Enable the system to automatically fix code violations using the LLM.
- **Tasks:**
  - Set up the **Generator Agent** module with OpenAI API calls. (Obtain API key, etc.)
  - Implement the prompt templates for generation and fix as planned. Test these prompts in isolation with some examples to fine-tune instructions.
  - Connect the pipeline: after Milestone 1's check, if violations exist, call the generator to fix them. Possibly loop once or twice.
  - FastAPI endpoints: maybe add /generate_code (given spec, return code) and /fix_code (given code + violations, return fixed code).
  - Integration: Extend the /check (or another endpoint) to, on request, attempt auto-fix. For instance, POST /enforce that takes code, runs check, if violations then calls fix and re-checks, up to N iterations.
- **Test:** Use the same sample code from Milestone 1 that had a violation. Now call the enforce function and see if it returns fixed code with the issue resolved. Evaluate the output for correctness and compliance by running check again. Adjust prompt if the AI didn't do the expected fix.

Milestone 3: Adjudicator with Argumentation and Exceptions

- **Goal:** Implement structured argumentation for decisions, allowing defeasible rules and producing detailed explanations.
- **Tasks:**
  - Introduce an exception rule example (like the encryption unless public data scenario) to test defeasible logic.
  - Pick an argumentation approach: easiest might be to integrate pygarg. Install and familiarize with how to supply arguments and get results.
  - Alternatively, implement the custom grounded semantics loop for our context (which may be simpler given limited scope). Ensure it handles a basic attack graph.
  - Modify the Adjudicator to not just do a trivial pass/fail, but to use the logic: e.g., consider a rule satisfied if no undefeated violation exists. This matters once we have an exception rule or multiple violations attacking each other, etc.
  - Ensure the adjudicator can produce an explanation structure. Perhaps have it generate a text or JSON snippet: e.g., "Rule X satisfied because no violations." or "Rule Y not satisfied because violation Z remained." This will feed into proof and possibly UI.
- **Test:** Create a scenario where a defeasible rule is in effect. For instance, policy: "Data should be encrypted (defeasible) unless it's sample data (exception)". Provide code where data is not encrypted but it's marked as sample. The static check flags "not encrypted" but we also feed in knowledge "this is sample data". The adjudicator should conclude compliance (the exception defeats the violation). We'll likely simulate the exception input to the adjudicator (since automating detection of "sample data" context might be beyond scope). This is more of a logic test than an AI one.

Milestone 4: Proof Bundle Assembly and Signing

- **Goal:** Generate the final proof artifact for a compliance run.
- **Tasks:**
    - Gather outputs from previous modules (final code, list of rules and statuses, evidence of tests).
    - Compose the proof JSON as per schema.
    - Use the cryptography library to generate a key and sign the JSON. Decide how to store the key (for prototype, maybe generate each run or have a static one for all runs).
    - Implement an endpoint /prove that, given an artifact ID or the context from a completed compliance check, returns the proof bundle.
    - Also possibly a /verify for internal use, that can verify a given proof (this could be a CLI tool instead).
- **Test:** For a known compliant artifact, generate the proof and then attempt to verify the signature using the public key. Alter the proof JSON slightly and confirm the signature verification fails (to test integrity). Also ensure the proof contains all expected info (like rule list, etc.).

Milestone 5: Web UI Integration

- **Goal:** Provide a user-friendly interface to input code and view results.
- **Tasks:**
    - Develop a React application with a form or text editor to input code (or select a sample code snippet).
    - Provide options to run compliance check or auto-fix.
    - When run, call the backend (FastAPI endpoints from earlier milestones). Possibly do this in steps: first get initial report then ask if user wants auto-fix.
    - Display results: highlight code issues. We can get line numbers of violations from the analysis JSON and use that to mark the code (e.g., using a code editor component or just text area with line numbers).
    - Show a summary of each rule (pass or fail). We could color-code passes green, fails red.
    - If auto-fix was run and code changed, update the code display to the new version, and show which issues were fixed (maybe mark those issues as resolved).
    - Provide a button to download proof bundle JSON. Or display the proof summary directly (like a modal with proof details).
    - Ensure the UI is modular to later allow file upload or larger text input (maybe use a monospaced font area).
- **Test:** Through the browser, try the same scenarios: code with issues -> see issues in UI -> click “Auto Fix” -> see updated code and now mostly green results. Download proof and inspect it.

Milestone 6: CLI and CI/CD Prep (Optional)

- **Goal:** Ensure the system can run headlessly (without UI) for CI usage.
- **Tasks:**
    - Write a small CLI script (maybe acpg_cli.py) that takes a file path and prints out compliance result or generates proof. It can call the FastAPI internally or directly use the same logic.

    - Document how to use it in a CI environment (e.g., "run acpg_cli.py --input code.py --policies policies.json --output proof.json; exit code 0 if compliant, 1 if not").
    - Possibly create a GitHub Action YAML snippet or reference to how one would integrate (though implementing an actual action is beyond scope).
- **Test:** Run the CLI on the sample files from earlier and ensure it outputs expected info and proper exit codes.

Throughout these milestones, after each, we integrate and refactor:

- By Milestone 3 or 4, we likely have all core pieces and can do an end-to-end dry run.
- Adjust the number of iterations the loop will attempt. Possibly default to, say, 3 tries to fix, to avoid infinite loops if the AI oscillates or fails. That number can be config.
- Implement a timeout for the whole process if needed, especially since the LLM calls can be slow. In CI, we wouldn't want it stuck. FastAPI can handle async timeouts or we manage it in code (maybe skip auto-fix if it takes too long).

Future Enhancements (beyond prototype):

Although not required now, we note them:

- More sophisticated dynamic analysis (spinning up the code in a real environment or using instrumentation).
- Support for multiple programming languages artifacts in one go (e.g., checking a whole project with front-end JS and back-end Python).
- Integration with version control (e.g., a pre-commit hook or a PR check using ACPG).
- Knowledge base of common fixes to assist the AI or even avoid calling AI for trivial fixes (like a hardcoded string can be fixed by a deterministic find-replace, which might be quicker and less costly than prompting GPT).
- Learning from fixes: when the AI produces a fix, perhaps adding that to a rule's known fixes so next time it can be suggested more directly.
- UI enhancements: show diff of code changes made by auto-fix, etc.
- Role-based usage: maybe an auditor view where you only upload code + proof and verify it (to simulate an external party verifying someone else's proof).

Each module can be developed and tested in isolation before integrating, which aligns well with using an AI coding assistant, one could, for example, prompt the assistant: *"Implement a function to run Bandit on given code and return issues as JSON"* once the requirements are clear, and so on, module by module.

Testing the System

To ensure the ACPG prototype works as intended, we will perform a series of tests:

1. **Unit Tests for Modules:**
    - Write unit tests for the Policy Compiler (e.g., give it a sample policy text or JSON and check that the internal representation is correct).
    - Tests for the Static Analysis wrapper: feed known code snippets to the analysis module. For example, a snippet with a hardcoded password should

yield a violation SEC-001. A snippet without that pattern should yield no SEC-001 violation. If using Bandit, test that our parsing of Bandit's JSON output to our schema works.

- Tests for the Generator: these would be more like integration tests since they call external API. Possibly use mock for OpenAI API in a test to simulate a known response. Or run it with a trivial prompt to see if we get JSON (not focusing on code quality, just format).
- Tests for Adjudicator logic: feed a small argument graph manually (like 1 compliance arg, 1 violation) and see that it marks compliance as defeated. Test a case with an exception: compliance + violation + exception attacker, see that compliance is accepted at end (violation defeated).
- Tests for Proof Assembler: given a decision and some dummy evidence, ensure it produces a JSON and that verifying the signature with the known public key returns true.

2. **End-to-End Scenario Tests:**
   - **Scenario A (Compliant code):** Provide a simple piece of code that already meets all policies. For example:

```
def add(a, b):
    # simple addition with type check
    if not (isinstance(a, (int,float)) and isinstance(b, (int,float))):
        raise TypeError("Inputs must be numbers")
    result = a + b
    print("Addition result:", result)  # Logging only non-sensitive info
    return result
```

   - Assuming our policies are focusing on security (no secrets, etc.), this code might be fully compliant (though it does print a result, we assume that's not sensitive). The system should analyze it and find no violations. Adjudicator declares compliant. The proof bundle should list all policies as satisfied. We verify the proof is signed. This tests the "do nothing" path to ensure no false positives.
   - **Scenario B (Simple violation and fix):** A code with one obvious issue:

```
API_KEY = "ABCD1234"  # This is a secret
def connect(api_key=API_KEY):
    print("Connecting with API_KEY:", api_key)
```

   - Here we violate "no hardcoded secret" and "no printing secret". The analysis should catch both (two violations for essentially the same rule or two rules). The AI fix should remove the hardcoded key (maybe require it as param or use env var) and stop printing it (maybe print "Connecting" without the key). Then the second analysis finds no issues. We check the final code manually that the issues are indeed gone. This tests static analysis and single iteration fix.
   - **Scenario C (Multiple iterations):** A more complex snippet where the first AI fix might not get everything:

```
def process(data):
    # Suppose policy: must encrypt sensitive data and no eval
    enc = False
    if not enc:
        eval(data)  # using eval unsafely
    return data
```

- Policies: no eval (strict), and say "encrypt if not enc" (defeasible because if enc flag is true maybe it's fine). The code violates both: uses eval and does not encrypt. The AI might fix eval by removing or replacing it with a safe parse. It might not know what to do about encryption flag; maybe it just sets enc=True or implements some encryption (unlikely without detailed instructions). Let's say first fix, it removes eval but leaves encryption unaddressed. Then analysis still flags encryption rule not satisfied (we detect that sensitive data data is returned without encryption while enc=False). We feed back "data not encrypted" to AI. It then maybe adds a step to encrypt data (or sets enc True to indicate it's encrypted). That second iteration yields compliance. This would test that the loop can run twice. (It also tests a bit of reasoning since encryption is abstract here, might need to guide the assistant in prompt about how to "fix" that, otherwise it might be confused.)
- **Scenario D (Defeasible rule with exception):** If we included a policy like *"Data must be encrypted unless it's non-sensitive"*, we can test code that doesn't encrypt data that is clearly non-sensitive. E.g.

```
# Mark data as public
data = "This is public info"
def transmit(info):
    send(info)  # some function sending data out
transmit(data)
```

Policy: all data sent out must be encrypted unless it's public. Here data is public info. Static check might say "unencrypted send" violation, but if we had a way to tag data as public, the adjudicator could waive it. This is a tricky test because it involves semantic context. Perhaps we simulate by saying the violation is attacked by an argument "public data exception applies". We might not have automated detection of that, but we could manually flag it in this scenario for demonstration. Then adjudicator should accept the exception and mark compliance. This tests our argumentation framework with an override.

- **Scenario E (Integration in CI manner):** Run the CLI or API on a repository file or a set of files. If time, maybe integrate on a small open-source project file to see if it catches something (for fun). But primarily ensure that if we call our main function with a file path, it goes through entire flow and exits properly.

3. **Performance and Stability:**
   - Test with slightly larger code (few hundred lines) to check performance. The biggest delay will be the LLM API calls (a few seconds each). Bandit on a few hundred lines is quick. So the loop might take, say, 10-20 seconds per iteration

with the LLM. That's acceptable for interactive use (though borderline; maybe use GPT-3.5 for faster testing, but quality might drop). In CI, 20-30 seconds per check is fine for moderate usage.
    - Ensure no crashes on unexpected input. E.g., test what happens if code has syntax errors (our static analysis might fail). Perhaps add a pre-check: attempt to parse the code into AST to ensure it's syntactically valid. If not, report to user that code must be valid or skip dynamic tests that can't run broken code.
    - Test how the system behaves if OpenAI API limit is reached or fails, our generator should handle exceptions (maybe try again or return an error status to the user).

By thoroughly testing each component and then the system as a whole with these scenarios, we can be confident in the prototype's functionality. The tests double as demonstration of the ACPG capabilities on real examples, showing how the system detects issues, fixes code, and produces a proof of compliance.

Seed Policy Rule Sets and LLM Prompt Recommendations

As an optional resource, we can provide some ready-made policy sets and suggestions on how to convert established guidelines into ACPG rules:

- **OWASP Top 10 / ASVS:** The OWASP Application Security Verification Standard has many requirements that can translate to code policies. For example: *"Verify that all password fields do not echo the password"*, if our code deals with password input, ensure input type is password (for a web app) or that it's not printed. For ACPG, we focus on code-level, so a rule could be "Functions handling passwords should not print them or return them in responses" (which we essentially did with logging policy). Another: *"Verify that all SQL queries use parameterized queries."* That can be a rule: detect string concatenation with SQL keywords and user input (with static analysis or dynamic test). **Prompt to codify**: *"Policy: Use parameterized queries instead of string concatenation for SQL in code."* We could ask GPT to generate a static analysis rule for it. Possibly have the LLM generate a regex to catch patterns of SELECT ... + variable or even generate a Semgrep rule in YAML. This is a potential use of GPT: to help create new rules from natural description. We might instruct it like: *"You are to help create a static code check. The policy is: no SQL queries built via string concatenation. Provide a Python function or pseudocode that detects this."* But for now, we can manually codify some.
- **NIST 800-218 (Secure Software Development Framework):** It includes practices like "implement continuous security testing" and "protect all forms of sensitive data". These can yield rules such as encryption of sensitive data at rest and in transit. Code-level: *"Cryptographic modules must be FIPS 140-2 certified"*, we could have a rule checking that if hashlib.md5 is used, it's a violation (since MD5 is not secure), etc. LLM prompt: *"Given the rule 'No use of MD5 or SHA1 for cryptographic purposes', suggest a way to detect if Python code violates this."* The LLM might answer: search for .md5(or .sha1(in code.
- **CERT Secure Coding Standards:** These are fine-grained rules for C, C++, Java, etc. For Python, there's not an official CERT, but we can glean some: e.g., *"Thou shalt not use os.system with unsanitized input"*, similar to our dangerous function rule. Or

"Close file descriptors after use", not easily checkable by static unless we simulate resource flow.

- **Cloud Security (if code interacts with cloud):** For instance, AWS keys should not be in code (already covered by secrets rule). Or all S3 buckets created must have encryption flag true (harder to check at code level, but if using SDK, could check parameters).

Providing **seed rules**: We can bundle, say, 5-10 core rules as default:

1. No hardcoded secrets.
2. Input validation on external inputs.
3. No use of dangerous functions (eval/exec, etc.).
4. Use secure communication (e.g., if code uses HTTP library, ensure it's HTTPS, we could search for http:// in requests).
5. Error handling (no sensitive info in error messages).
6. Authentication handling (e.g., *"Passwords must be hashed, not stored plaintext"*, static check for usage of a secure hash function on passwords, dynamic check if possible).
7. Authorization checks (harder to do generically, maybe skip for now).
8. Logging practices (no secrets in logs, and use proper logging framework instead of prints in prod code, maybe style choice).
9. Resource cleanup (like file handles), could be optional, depending on language.
10. Code style guidelines (if any, probably out of scope for compliance, but could include one like "No functions longer than 50 lines" to demonstrate style rule; static check just counts lines per func).

Recommended LLM Prompts to Codify Policies:

If one has a batch of natural language rules and wants to create checks, one approach is to ask the LLM to draft either code for checks or test cases:

- For static: *"You are an AI specializing in code analysis. Convert the following policy into a static analysis plan or pattern: 'All user inputs must be range-checked to prevent overflow.'"* The model might suggest checking for use of input in array indexing or memory allocation without prior validation. This might be more relevant for C, but shows how to use LLM in writing detectors.
- We could also use the LLM to generate Rego policies for OPA if we decided to incorporate OPA. For example: *"Write a Rego policy that denies deployment if any container has privileged=true."* That's more infra-as-code style, but shows the idea.
- Another use: generating test cases. For a given function and rule, prompt the LLM: *"Generate a test input for function X that would violate the no-SQL-injection policy if the function is vulnerable."* Essentially the LLM can act as a smart fuzzer by using its knowledge of attack patterns. This can complement Hypothesis. We could integrate that by asking GPT to produce an input string that's likely to cause issues (like ' OR '1'='1 for SQL or <script>alert(1)</script> for XSS) and then run that.

Given the scope, we will not implement an automatic policy generator with LLM, but it's good to mention these as future directions.

For now, we provide the seed rule set in the code (or config) so the user can start with those and add more. The developer using this guide can extend the policies easily by editing the JSON and possibly adding corresponding detection logic if needed.