Unified Contributing Guide (Draft for PR Submission)

This document establishes the mandated technical and governance standards for all internal and external contributions to organizational repositories, specifically targeting the Python ecosystem projects such as ScanCode.io, VulnerableCode, and DejaCode. The policy codified herein resolves historical fragmentation in tooling and procedure by mandating a unified approach centered on proactive local quality enforcement and architectural risk mitigation.[1, 2]

The primary objective of this unified guide is to standardize core development practices and tooling, eliminating inefficiencies related to fragmented linter and formatter configurations across repositories. This standardization is achieved through the adoption of high-efficiency tooling and mandatory local enforcement mechanisms, ensuring consistency, enhancing overall code quality, and streamlining the Continuous Integration (CI) and code review processes.[2]

Part I: Foundational Governance and Workflow

1 Legal Mandate: Attribution and the Developer Certificate of Origin (DCO)

1.1 The Mandatory DCO Requirement

Legal clarity concerning Intellectual Property (IP) is a non-negotiable standard for all code submissions. All contributors must agree to the mandatory **Developer Certificate of Origin (DCO)** when contributing to projects like ScanCode.io, VulnerableCode, and DejaCode.[2, 3] This agreement is technically enforced by requiring contributors to include a `Signed-off-by:` trailer, containing the contributor's name and email, in every commit message.[2, 1]

1.2 The Shift-Left Legal Governance Strategy

The traditional reliance on reactive checks within the CI environment to validate DCO compliance is inefficient, consuming CI resources and developer time for easily preventable administrative errors.[1, 2] To mitigate this systemic inefficiency, the organization mandated the integration of universal `pre-commit` hooks. This critical architectural decision implements a "Shift-Left" legal governance strategy, dictating that DCO validation must be performed locally via a mandatory hook before the commit is finalized or pushed.[1, 2] This proactive enforcement guarantees that legal attribution is confirmed automatically on every commit, reinforcing the organizational IP governance structure and eliminating CI failures related to missing sign-offs, thereby maximizing overall CI efficiency.[2, 1]

2 The Issue-First Policy: Scoping Contributions and Rationale

2.1 Mandatory Pre-Discussion for Focused Contributions

Code must be contributed using Pull Requests (PRs). A foundational rule of PR governance is the mandatory **Issue-First Policy**: a PR must always be attached to a pre-existing, pre-discussed issue.[2, 3] This procedural prerequisite dictates that when no existing issue corresponds to the work, the contributor is explicitly instructed to start by creating one to discuss potential solutions and implementation details before writing any code.[2, 1]

2.2 Functional Control of Scope

The Issue-First policy serves as the primary organizational mechanism for controlling contribution scope.[1] By requiring up-front definition and discussion of the problem and proposed solution within the issue tracker, the policy prevents the sudden introduction of excessively large, unfocused, or surprise changesets. This procedural prerequisite links governance directly to the development workflow, significantly reducing the cognitive load on maintainers by ensuring that changes are presented as focused, isolated units. Forcing scope discussion upfront aligns with best practices recommending that large changesets should be avoided to maintain high defect detection rates.[2] Furthermore, the policy supports the mandatory rationale requirement, as contributors are encouraged to "explain a solution [which] is always a good start," establishing technical justification as a mandatory precursor to coding.[2, 1]

3 Mandatory Local Tooling Setup: The Foundation of Compliance

The organization's entire quality and compliance strategy hinges on the mandatory adoption of local enforcement tooling, shifting resource-intensive checks from the CI environment to the developer workstation.[2]

3.1 The Critical First Step

New contributors are mandated to install the local enforcement mechanism immediately after cloning the repository. The process begins with installing `pre-commit` (a multi-language package manager for Git hooks) and activating the hooks in the local repository.[2] The success of the unified contribution flow relies on the mandatory execution of the hook installation command immediately after cloning the repository.[1]

The required actions are:

1. `pip install pre-commit` (or environment equivalent)
2. `pre-commit install`

3.2 Rationale for Universal `pre-commit` Adoption

Universal adoption of `pre-commit` is mandated because it automatically executes compliance checks on staged files before the `git commit` is finalized.[2] This is a critical architectural decision that operates as a fundamental element of a modern DevSecOps strategy, moving quality assurance and compliance checks "left".[2] This workflow optimization ensures that simple, autofixable style errors (like those handled by Ruff) and critical compliance violations (such as DCO status) are addressed immediately by the developer. This prevents the majority of easily correctable errors from ever reaching the CI pipeline, which drastically reduces CI failure rates and minimizes the organizational cost associated with repeated pipeline runs and unnecessary reviewer feedback cycles.[2] If any check fails, the commit is blocked, forcing local correction, thereby guaranteeing the integrity of code pushed to the remote repository.[2]

Part II: Mechanized Quality, Legal, and Security Enforcement

4 The Unified Code Quality Standard (Ruff Mandate)

4.1 Resolution of Python Tooling Fragmentation

The organization previously faced significant friction, technical debt, and CI/CD complexity due to fragmentation across Python tooling, necessitating the coordinating of multiple separate quality tools—each requiring distinct configuration, dependency management, and version updates.[2] The policies identified this situation as imposing an unnecessary organizational cost, or "tooling maintenance tax".[2] The policy formally mandates **Ruff** as the unified, organization-wide choice for Python linting, formatting, and import sorting, resolving this fragmentation.[2]

4.2 Strategic Justification and Performance

The decision to mandate Ruff is a strategic economic choice designed to eliminate the tooling maintenance tax.[2] Ruff acts as a consolidated layer, offering superior performance with "fast linting even in large codebases".[2] This performance advantage translates directly into reduced waiting times during local development and a significant acceleration of Continuous Integration (CI) pipeline execution, thus maximizing developer velocity.[2] Ruff strategically consolidates the functionality of multiple legacy tools, including Flake8, Pylint, and isort, into a single, high-performance binary.[2]

4.3 Seamless Migration and PEP8 Compliance

Ruff is engineered for consistency, specifically designed as a drop-in replacement for Black, achieving **greater than** 99.9% **formatting parity** when tested over Black-formatted code.[2] This technical consistency ensures that switching the underlying formatter used by the universal `make valid` command will result in minimal disruptive code changes, securing a stable and unified toolchain that efficiently enforces strict adherence to the **PEP8 conventions**.[2, 1] While a legacy exception exists allowing repositories historically using Black to continue doing so for formatting only, prioritized migration to Ruff is strongly recommended to achieve full standardization.[2]

4.4 Configuration Nuance (E501)

To prevent duplicate checks and simplify the quality profile, Ruff must be configured to defer line-length enforcement (PEP8 rule E501) entirely to its integrated formatter (`ruff format`).[2] The standard configuration explicitly specifies a consistent line-length (e.g., 99 characters) and configures the linter to ignore E501 violations (`ignore = ["E501"]`), allowing the integrated Ruff formatter to manage line wrapping deterministically.[2]

5 The Local Enforcement Configuration (Mandatory Hooks)

To ensure the local development environment enforces the same quality and compliance standards as the remote CI system, a definitive, standardized `.pre-commit-config.yaml` template must be deployed universally across all repositories.[1, 2]

5.1 Mandatory Pre-commit Hook Configuration

The standardized template must include four mandatory hooks covering code quality, legal compliance, workflow structure, and security assurance.[1] This configuration is the primary architectural anchor for the unified contribution strategy.

Mandatory Pre-commit Hook Configuration

Hook Function	**Policy Enforced**	**Tooling Mandate**	**Gover**
Formatting, Linting, Sorting	Unified Coding Standard (PEP8)	`Ruff` (Consolidated Tool)	Achiev
DCO Validation	Consistent Commit Convention (Legal)	`DCO Check Hook`	Proact
Branch Naming Validation	Unified Branch Naming Rules	`validate-branch-name` Hook	Enforc
Secrets Detection	DevSecOps / Security Assurance	`detect-secrets` Hook	Preven

5.2 Draft Configuration File (`.pre-commit-config.yaml`)

This template provides the required configuration structure to mechanize local enforcement, leveraging automated maintenance to sustain toolchain currency [1, 4]:

```
#.pre-commit-config.yaml
# Universal Enforcement Template - Mandated by Unified Contribution Policy

# Standardized minimum version requirement for pre-commit
minimum_pre_commit_version: '3.2.0'
fail_fast: true # Stops execution after the first hook failure to speed up feedback

# Automated hook version updates required by policy (Section 11.1)
ci:
  autoupdate_schedule: weekly # Mandated automated updates to prevent version drift [4]

repos:
  # 1. Mandatory Ruff Toolchain (Linting, Formatting, Import Sorting)
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.4.0 # Pinned revision/tag for stability
    hooks:
      # Mandatory Ruff Formatting Hook: Auto-formats staged files
      - id: ruff-format
        name: Ruff Formatting (Mandatory)
        types_or: [python, pyi, jupyter]
        args: [--line-length, '99']

      # Mandatory Ruff Linting Hook: Checks for quality/style violations
      - id: ruff
        name: Ruff Linting (Mandatory)
        types_or: [python, pyi, jupyter]
        args: [--fix, --line-length, '99']
        # Enforce E501 ignore rule, deferring line length to the formatter (Section 4.4)
        exclude: '__init__.py$'

  # 2. Mandatory DCO and Commit Message Validation (Legal Compliance)
  - repo: https://github.com/pre-commit/pre-commit-hooks # Or organization-specific utility re
    rev: v4.6.0
    hooks:
      # Mandatory DCO Check (using a generic commit-msg check for illustration)
      - id: check-commit-msg
        name: DCO Legal Attribution Check
        description: Ensures all commits contain the 'Signed-off-by:' trailer (requires custom
        stages: [commit-msg]

  # 3. Mandatory Security Checks (Secrets Detection - DevSecOps)
  - repo: https://github.com/Yelp/detect-secrets
    rev: v1.4.0
    hooks:
      # Mandatory Secrets Detection Hook (Shift-Left Security)
      - id: detect-secrets
        name: Secrets/Credential Leakage Detection
        stages: [commit]
  # 4. Mandatory Branch Naming Validation (Workflow Structure)
  - repo: https://github.com/pre-commit/validate-branch-name # Placeholder for validation tool
    rev: v1.0.0
    hooks:
      - id: validate-branch-name
        name: Unified Branch Naming Policy Check
        stages: [pre-push]
        args:
          # Regex enforces categories (feature/123/name, fix/123/name, chore/name, hotfix/name
          - --pattern=^(feature\/[0-9]+\/[a-z0-9]+(-[a-z0-9]+)*)|(fix\/[0-9]+\/[a-z0-9]+(-[a-z
          - --error-message="Branch name must adhere to the Lightweight Semantic Branching Pol
```

6 Lightweight Semantic Branching Policy (Git Flow Standardization)

6.1 Rationale for Standardization

The strategic analysis identified the absence of documented standards for Git branch structure as a **HIGH-priority policy void**.[1, 1] This lack of standardization fundamentally degrades the quality and utility of the repository's Git history, making it difficult to audit code and maintain consistency.[1] Furthermore, it prevents the seamless automation of key processes, such as semantic release note generation, because automated tools rely on predictable branch prefixes and commit messages.[1] The new standard ensures the Git workflow is clean, predictable, and machine-readable, which is a necessary prerequisite for high-maturity automation.[1]

6.2 Mandated Convention

The organization mandates a lightweight adaptation of Git Flow principles, requiring the use of explicit semantic prefixes to categorize the functional purpose of the branch.[1] This policy is immediately and mechanically enforced via the dedicated branch name validator hook integrated into the mandatory `pre-commit` system (Section 5.2).[1, 1]

Standardized Git Branch Naming Conventions

Purpose Category	**Naming Convention Pattern**
Feature	`feature/<issue-number>/short-description`
Bug Fix	`fix/<issue-number>/short-description`
Refactor/Chore	`chore/task-description`
Hotfix	`hotfix/release-version-bug`
Naming Rule	All descriptions must use lowercase and kebab-case (e.g., `fix/101/invalid-regex-logi`

Part III: High-Risk Policy Voids and Review Gates

7 Mandatory Testing and Code Integrity

7.1 The Universal Testing Mandate

The requirement for code integrity is supported by the institutional philosophy that contributors must "write tests, a lot of tests, thousands of tests".[2, 3] Testing is universally mandatory.[1] All contributors must add new or updated unit or integration tests when submitting fixes or implementing new features.[2, 1] Local verification is required using the standardized command `make test` (or equivalent) before seeking peer review.[1, 2]

7.2 The CI Quality Gate

The Continuous Integration (CI) environment serves as the definitive primary quality gate.[2] The policies explicitly state that pull requests that are **not passing the automated integration tests are unlikely to be reviewed.**[2, 3] The mandatory local enforcement via `pre-commit` (Section 5) supports the CI gate by preventing stylistic and compliance failures locally, thereby ensuring maintainer resources are reserved exclusively for the architectural and functional review of the code logic.[2]

8 Dependency Vetting Procedure (DVP) – Mitigating Architectural Risk

8.1 Identifying the Policy Void

The analysis confirmed a complete absence of any documented policy, process, or required checklist governing the evaluation and introduction of new external dependencies.[1, 1] This omission was classified as a **HIGH-priority policy void** because it exposed the organization to unacceptable legal, security, and maintenance risks.[1, 1] For an organization rooted in IP and licensing clarity (as evidenced by the DCO mandate and Apache-2.0 licensing commitments [1]), allowing unvetted dependencies to enter the codebase represents a critical failure in risk governance.[1]

8.2 Mandatory DVP Requirements

A formal **Dependency Vetting Procedure (DVP)** is mandated immediately. This policy requires that any proposal for a new external dependency must undergo formal review documented within the Issue/PR workflow.[1, 1] This review requires specific documentation addressing three critical risk vectors: Legal Risk (License Vetting), Security Risk (Assessment), and Maintenance Risk (Justification).[1] Enforcement of this policy is achieved by integrating a designated DVP sign-off field directly into the Pull Request template (Section 9).[1, 1]

Mandatory Requirements for New Dependency Introduction (Dependency Vetting Procedure - DVP)

Requirement	**Risk Vector Addressed**	**Required Artifact in PR**
Justification	Maintenance Risk	Clear documentation articulating necessity and why exis
License Vetting	Legal Risk	Confirmation that the new dependency's license is fully c
Security Assessment	Security Risk	Evidence of due diligence, including confirmation of activ

9 The Pull Request Quality Gate (Contributor Checklist)

9.1 Codifying the Implicit Review Checklist

The previous reliance on unstructured textual guidance created procedural inconsistency, forcing maintainers to manually verify compliance against an unwritten list of prerequisites.[1, 1] The organizational policy now mandates the deployment of a universal `PULL_REQUEST_TEMPLATE.md` to transform this implicit review checklist into a structured, enforceable format.[1]

9.2 Template Structure as a Formal Contract

The PR template serves as the mandatory **Contributor Quality Gate Checklist**, requiring explicit self-attestation from the developer regarding compliance with DCO, testing, formatting, and rationale standards.[1] This structured deployment approach ensures reviewers only engage with code that has already met the mechanical quality standard, converting the maintainer's checklist into required fields for the contributor. This maximizes the efficiency of high-value maintainer review time by shifting the verification burden left.[1]

9.3 Draft Pull Request Template (.github/PULL_REQUEST_TEMPLATE.md)

This template formalizes the mandatory requirements for all contributions:

```
# PULL REQUEST:/[Issue Number] - Brief Description

**Mandatory Pre-merge Checks (Contributor Quality Gate Checklist)**

By submitting this Pull Request (PR), the contributor confirms that all mandatory
checks listed below have been completed. Failure to complete these checks will
result in the PR being considered incomplete and unlikely to be reviewed until
all requirements are met.

***

### 1. Contribution Scope and Rationale (Issue-First Policy)

- [ ] This Pull Request is directly attached to an existing, pre-scoped Issue (Link: #____).
- [ ] The PR description clearly explains the logical approach and solution rationale,
      consistent with the initial issue discussion.

**DVP Vetting Required?** (Check if this PR introduces any new external library/dependency)
- [ ] NO, no new dependencies were added.
- [ ] YES, new dependencies were added. The Dependency Vetting Procedure (DVP)
      documentation (Section 4) below has been fully completed.

***

### 2. Code Quality, Style, and Attestation

| Check Required | Policy Mandate | Enforcement / Artifact | Status |
| :--- | :--- | :--- | :--- |
| **Code Style Adherence** | PEP8 Conventions enforced by Ruff (Section 4). | Automated ‘pre-c
| **Mandatory Testing** | Universal Testing Mandate (Section 7). | New or updated unit tests a
| **DCO Compliance** | Mandatory Legal Attribution (Section 1). | **ALL** commits contain the
| **Branch Naming** | Lightweight Semantic Branching Policy (Section 6). | The branch name adh

***

### 3. Documentation and Project Integrity

- [ ] Documentation (user docs, API guides, tutorials) has been updated to reflect the changes
- [ ] If applicable, a descriptive entry has been added to ‘CHANGELOG.rst‘.
- [ ] If I am a new contributor, my name and affiliation have been added to ‘AUTHORS.rst‘.

***

### 4. Dependency Vetting Procedure (DVP) Documentation

**(If Section 1 indicated YES, complete this section thoroughly.)**

| Requirement | Documentation Provided |
| :--- | :--- |
| **Justification** | |
| **License Vetting** | Confirmed compatibility with {Project’s License, e.g., Apache-2.0}. |
| **Security Assessment** | |
```

Part IV: Documentation and Sustaining Governance

10 Documentation as Code (DaaC)

10.1 Documentation is a Core Deliverable

Documentation is not viewed as a secondary task but is explicitly **”treated like code”** and considered a critical, integral aspect of the project delivery.[2, 3] This Documentation as Code (DaaC) model mandates that improvements to features, APIs, or user interfaces must be accompanied by corresponding updates to the documentation to maintain high standards of usability and clarity.[2, 3]

10.2 Changelog and Attribution Management

In addition to technical documentation updates, contributors must ensure proper attribution and release tracking. This requires noting changes in the project’s `CHANGELOG.rst` and, for new contributors or significant feature work, ensuring their name is added to `AUTHORS.rst`.[2, 3]

11 Sustaining Standards and Governance Maintenance

11.1 Preventing Toolchain Drift

To protect the economic benefits derived from standardizing on high-performance tooling like Ruff, automated governance mechanisms are required. The organization mandates automated dependency updating systems, such as `pre-commit.ci autoupdate`, to keep hook versions current and stable.[1, 4] This active maintenance prevents the recurrence of fragmentation and reduces the risk of environment/tool version drift between local developer machines and the centralized CI environment.[1]

11.2 Active Policy Audit

Governance must be actively managed to maintain relevance. A mandatory **12-month review cycle** is instituted for the newly created high-risk policies: the **Dependency Vetting Procedure (DVP)** and the **Lightweight Semantic Branching Policy**.[1] This auditing cycle is necessary to ensure these critical policies remain aligned with evolving architectural standards, legal compliance requirements, and security best practices, enabling the organization to move from passive adherence to active, dynamic governance.[1]

Conclusions and Recommendations

The strategic analysis confirms that robust foundational principles for contribution (DCO, testing, PEP8) were already entrenched but their effectiveness was hampered by technical fragmentation and procedural inconsistency.[1, 2] The Unified Contributing Guide addresses these structural weaknesses by mandating a foundational architectural shift.

The core policy recommendations are centered on automation and enforcement:

1. **Tooling Consolidation and Efficiency:** The strategic mandate of **Ruff** resolves the Python tooling conflict, providing a superior, consolidated, and high-performance standard that drastically reduces technical debt and CI overhead.[2]

2. **Proactive Enforcement:** The universal mandate for **`pre-commit` hooks** institutionalizes a shift-left DevSecOps strategy. This mechanism guarantees that mandatory standards for code style, security checks, and, critically, the legal DCO attribution, are met consistently before code enters the review cycle.[1, 2]

3. **Risk Mitigation:** The immediate enactment of the **Dependency Vetting Procedure (DVP)** and the **Lightweight Semantic Branching Policy** mitigates previously identified high-risk governance voids. These policies, enforced through structured PR templates and branch validation hooks, ensure architectural integrity and workflow predictability.[1, 1]

It is recommended that this unified guide, along with the standardized configuration file drafts, be immediately deployed across all core repositories. The updated contribution flow, anchored by the mandatory execution of `pre-commit install` as the crucial first step, establishes local enforcement as the organization's default mode of operation.[1]