LLM Code Failure Observatory

Technical Specification (v1)2

0. Executive Summary

This project defines a Code Evaluation System that assesses AI-generated code against explicit, user-defined success contracts, declared in a YAML file.

The system does not generate code.
It evaluates code outputs and produces explainable failure reports, including functional, behavioral, performance, and security failures.

The core value is to answer:

> *"Can this AI-generated code be accepted for production, and if not, exactly why?"*

1. Problem Definition

1.1 Problem Statement

Large Language Models (LLMs) increasingly generate executable code.
However:

- Code may run but be incorrect
- Code may pass basic tests but violate constraints
- Code may introduce security vulnerabilities
- Code may fail silently or degrade performance
- Developers must manually inspect, debug, and reason about failures

This human-in-the-loop validation is:

- Time-consuming
- Non-systematic
- Hard to reproduce
- Poorly documented

1.2 Problem This System Solves

This system provides:

- Explicit success definitions (via YAML contracts)
- Automated evaluation of AI-generated code (Compile + run in a dockerized environment)
- Explainable failure classification
- Evidence-backed diagnostics
- Security and static analysis integration (Through static code analysis tools)

It allows teams to gate AI-generated code before production.

1.3 Explicit Non-Goals (Out of Scope)

This system does NOT:

- Optimize prompts
- Generate or modify code
- Fine-tune models
- Rank LLMs globally
- Provide UI dashboards (v1)
- Replace human review

2. Technical specification

2.1 User defined contract (YAML file)

```json
{
  "version": "1.0",
  "project_path": ".",
  "entrypoint": "src.app",
  "runtime": {
    "python": "3.11"
  },
  "sandbox": {
    "network": false,
    "filesystem": "read_only",
    "allowed_write_paths": [
      ".artifacts"
    ]
  },
  "deps": {
    "requirements": "requirements.txt"
  },
  "rules": {
    "build_import": {
      "enabled": true,
      "import_timeout_seconds": 3
    },
    "unit_tests": {
      "enabled": true,
      "path": "tests/",
      "coverage": {
        "enabled": true,
        "min_line_coverage": 0.7
      }
    },
    "security_sast": {
      "enabled": true
    },
    "security_deps": {
      "enabled": true
    },
    "policy": {
      "enabled": true,
      "forbidden_modules": [
        "yaml",
        "Crypto",
        "sklearn"
      ],
      "forbidden_packages": [
        "pyyaml",
        "pycryptodome",
        "scikit-learn"
      ],
      "forbidden_apis": [
        "eval",
        "exec",
        "os.system",
        "subprocess.Popen",
        "importlib.import_module"
      ]
    },
    "quality": {
      "enabled": true
    }
  }
}
```

Purpose and Scope: This document specifies the Job YAML format used to evaluate AI-generated Python projects against explicit, user-defined constraints. The system is an evaluator, not a code generator: it executes deterministic checks and emits evidence artifacts allowing a user to decide acceptability without manual code review.

In scope (v1.0):

- Python-only evaluation
- Dependency installation from `requirements.txt`
- Sandbox controls: network disablement and filesystem write restrictions
- Rule-based evaluation: build/import, unit tests, static security analysis, dependency vulnerability scanning, policy checks, and quality checks
- Generate a Report

YAML Format

1. **Versioning:**
   This is a general rule for YAML files

2. **Project_path:** MUST point to the root directory of the project to be evaluated.The evaluator MUST treat paths as relative to the location of the job YAML file unless otherwise documented.

3. **Entrypoint:** MUST be a Python module import path.The Build/Import rule MUST use this entrypoint as an import target.The evaluator MUST reject entrypoints that are file paths (e.g., src/app.py) in v1.0 unless explicitly extended later.

4. **Runtime**: MUST be a string indicating the requested Python major.minor version. In v1.0, the evaluator MUST provision an execution environment matching this version (typically via Docker base image selection such as `python:3.11-slim` or equivalent).
5. **Sandbox**:
    a. sandbox.network: false means outbound and inbound network access MUST be disabled during evaluation runs.
    b. If the runner is Docker-based, the evaluator MUST start containers with network disabled (e.g., --network none).
    c. sandbox.filesystem MUST be one of:
        i. Read_only
        ii. project_write
        iii. Custom
    d. read_only:The project directory MUST be mounted read-only.Only the artifacts directory MUST be writable.

e. project_write:The project directory MAY be writable. The evaluator SHOULD still ensure that writes outside the mounted project path are not possible.
f. Custom: Write permissions MUST be restricted to the paths listed in allowed_write_paths.
g. allowed_write_paths MUST be specified if sandbox.filesystem == "custom".
h. Paths are interpreted relative to project_path unless absolute.

6. **Deps**: MUST reference a requirements file path relative to `project_path.` The evaluator MUST install dependencies from this file prior to executing any rule that depends on imports, tests, or analysis tools.

7. **Rules Section:** Rules are declared as a mapping of rule blocks. Each rule block MUST include:
    a. Enabled: true | false
    b. optional rule-specific parameters

    Rules MAY be executed sequentially in the order listed in the file, but the evaluator MUST ensure that environment provisioning and dependency installation occur before rules that require them**.**

Rules high level description:

1. **Build/Import** Can modules be installed/imported and compiled
2. **Unit tests** (pass/fail + logs + coverage)
3. **Security** (SAST + dependency scan + sandbox policy)
4. **Policy** (allowed libs, forbidden APIs, file/network/subprocess rules)
5. **Quality** (lint/type/complexity budgets)

Rules Section:

1. Build_import:
    a. Objective:
        i. The evaluator can install and verify the declared dependencies (from requirements.txt) and then import.
            1. *This catches: bad pins, broken wheels, platform-specific issues, missing system libs, dependency conflicts.*
        ii. Validate that the project can compile.
            1. *This catches: syntax errors, indentation errors, broken f-strings, incomplete code.*
        iii. Validate that Python can import the module declared as entrypoint without crashing.
            1. *This catches: missing modules, circular import errors, missing shared libraries, wrong package structure, unresolved imports.*

    b. How it should work in the evaluator:

The evaluator validates project loadability by constructing an isolated Docker-based Python environment. A temporary dependency image is built from the specified Python version, where all dependencies from requirements.txt are installed. This image is reused for evaluation to ensure reproducibility and avoid re-installation.

Using this dependency image, the evaluator runs a sandboxed container and mounts the project source code. It first compiles all Python files to detect syntax errors, then imports the declared entrypoint module to verify that the project is structurally importable and that dependencies are correctly resolved. No application logic is executed beyond module import. If network access is disabled, dependency installation is performed in a separate build stage, and compilation and import are executed in a container with network access fully disabled.

c. **Technical spec:**
    i. Phase 1: Dependency install
        1. Install dependencies from *deps.requirements* using *python -m pip install --upgrade pip setuptools wheel*
            a. runs pip as a module to guarantee the pip bound to the selected Python interpreter.
            b. --*upgrade* ensures modern resolver/build tooling to reduce install variance.
            c. *pip/setuptools/wheel* are upgraded to support building and installing wheels reliably
        2. *python -m pip install --no-input --disable-pip-version-check -r requirements.txt*
    ii. Phase 2: Compilation
        1. *python -m compileall -q /workspace*
            a. *compileall compiles all .py files it finds*
    iii. Phase 3: Entrypoint import
        1. *PYTHONPATH=/workspace python -c "import importlib; importlib.import_module('src.app')"*
            a. PYTHONPATH=/workspace ensures the project root is on the import path without requiring packaging
            b. python -c executes a small one-liner without starting the application
            c. importlib.import_module imports the entrypoint deterministically and surfaces missing dependencies, bad packaging, and import-time side effects

Note: If sandbox.network is false, pip cannot download from PyPI inside the sandboxed evaluation container, so the evaluator MUST perform Phase 1 in a separate “deps build” stage where network is temporarily enabled (build a deps image layer containing the installed requirements), then run Phases 2–3 in a fresh container created from that deps image with --network none and the filesystem restrictions applied, ensuring that compilation/import evidence reflects the requested sandbox.

2. **unit_test:**
    a. **Objective:**
        i. Execute developer-provided unit tests in an isolated runtime to validate functional correctness of the evaluated code.

    b. **How it should work in the evaluator**:
    The evaluator runs unit tests inside the same isolated Docker-based environment used for build_import (i.e., a container created from the dependency image built from runtime.python + installed requirements.txt). The project source is mounted into the container and tests are executed using pytest. If coverage is enabled, the evaluator collects line coverage using pytest-cov (coverage.py) during the same pytest run, and compares the resulting coverage percentage to min_line_coverage. The evaluator never starts the application directly; it only imports and executes tests. Sandbox constraints (network/filesystem) MUST remain enforced during test execution to ensure tests do not silently rely on external resources.
    c. **How users provide unit tests:**
    users place their tests under the configured directory (e.g., tests/) and the job YAML points to it via rules.unit_tests.path.
    d. **Technical spec:**
        i. Tooling:The evaluator environment MUST ensure these tools are available (either included in a dev requirements file or installed by the evaluator).
            1. Test runner: pytest
            2. Coverage: pytest-cov
        ii. Phase 1: Unit test execution (pytest)
            1. Command: *pytest tests/ -q*
            2. Pass/Fail:
                a. The rule MUST FAIL if pytest exits non-zero (any failing/erroring tests).
                b. The evaluator MUST capture stdout/stderr to pytest.log.
        iii. Phase 2: Coverage collection (if enabled)
            1. Command : *pytest tests/ -q --cov=/workspace --cov-report=term*

--cov-report=json:.artifacts/unit_tests/evidence/coverage.json

a. --cov=/workspace measures line coverage across the project code mounted at /workspace (broad, works without packaging assumptions).
b. --cov-report=term prints a readable coverage summary into the pytest log for quick debugging.
c. --cov-report=json:<path> writes a machine-readable coverage report to a deterministic location for parsing and reporting.

2. Pass/Fail:
   a. If coverage.enabled == true, the evaluator MUST compute overall line coverage from the JSON report and compare it to min_line_coverage.
   b. The rule MUST FAIL if measured line coverage is below min_line_coverage even if all tests pass.

e. Evidence outputs:

i. For unit_tests, the evaluator MUST write:
   1. .artifacts/unit_tests/status.json containing:
      a. status (PASS|FAIL|ERROR|SKIP)
      b. reason_code (e.g., TESTS_FAILED, COVERAGE_BELOW_THRESHOLD, TOOL_EXECUTION_ERROR)
      c. message (templated)
      d. Metrics:
         i. Duration_seconds
         ii. tests_collected
         iii. tests_failed
         iv. Tests_succeded
         v. coverage_line (0–100%, only if enabled)
   2. .artifacts/unit_tests/evidence/pytest.log containing full pytest output
   3. .artifacts/unit_tests/evidence/coverage.json (only if coverage enabled)

3. security_sast

a. Objective:
   i. Run a deterministic, reproducible static application security test (SAST) on the submitted Python project using Bandit, and fail the evaluation if Bandit reports any security findings above a fixed policy threshold.
   ii. This catches (within Bandit's scope):

- use of dangerous APIs (eval, exec, pickle, insecure YAML load, weak crypto, hardcoded credentials patterns)
- risky subprocess usage / shell injection patterns

- insecure temporary files and permissions patterns
- other known Python security anti-patterns covered by Bandit's built-in ruleset

b. How it should work in the evaluator

The evaluator executes Bandit inside the same isolated Docker-based environment used for other checks. The project source is mounted read-only, and no application runtime execution is performed. Bandit runs purely as static analysis and produces machine-readable evidence artifacts (JSON) that the evaluator converts into a contract decision (PASS/FAIL/ERROR).

c. Technical spec

i. Phase 1: Bandit scan (recursive, JSON evidence)
   1. Run Bandit against the mounted workspace, excluding standard non-source directories, and output JSON.
   2. Command: *bandit -r /workspace -f json -o /artifacts/security_sast/bandit.json -q*
      (-q reduces console noise)

ii. Phase 2: Deterministic policy gate (fixed thresholds)
   1. Parse bandit.json and apply a fixed v1 decision policy
   2. Fail if there exists any finding where:
      a. issue_severity >= LOW
      b. issue_confidence >= LOW
      c. issue_severity= *How bad would this be if it's real?*
      d. issue_confidence=How likely is this to be a real security issue vs a false positive?
   3. Otherwise PASS

iii. **Outcome mapping:**
   1. Bandit executed successfully + gated findings == 0 → PASS
   2. Bandit executed successfully + gated findings > 0 → FAIL with reason_code SAST_FINDINGS_DETECTED
   3. Bandit tool error / malformed output / crash → ERROR with reason_code TOOL_EXECUTION_ERROR

d. Evidence produced:

i. Bandit will generate a raw .json file as following:


```
{
 "results": [
  {
   "code": "subprocess.run(\"ls \" + user_input, shell=True)",
   "filename": "/workspace/src/app.py",
   "issue_confidence": "HIGH",
   "issue_severity": "HIGH",
   "issue_text": "subprocess call with shell=True identified, security issue.
   "line_number": 4,
   "test_id": "B602",
   "test_name": "subprocess_popen_with_shell_equals_true"
  },
```

ii. Evaluator status for this MUST write:
    1. .artifacts/security_sast/status.json containing:
        a. status (PASS|FAIL|ERROR|SKIP)
        b. reason_code (e.g., SAST_FINDINGS_DETECTED)
        c. Findings for each failure:
            i. test_id (in bandit)
            ii. test_name
            iii. issue_confidence
            iv. issue_severity
            v. issue_text
            vi. line_number
            vii. filename
            viii. code
        d. Metrics:
            i. total_findings
            ii. Breakdown
                1. HIGH:
                2. MEDIUM:
                3. LOW
        e. Reference to evidence files

4. **Security_deps**
    a. **Objective**: The evaluator detects known, publicly disclosed vulnerabilities in the project's Python dependencies, using a deterministic, pinned dependency-audit tool, and fails the evaluation if any vulnerable packages are present in the resolved dependency set.
    b. This catches:
        i. dependencies with known CVEs / advisories (direct + transitive)
        ii. vulnerable versions introduced by dependency resolution
        iii. "it installs fine but it's insecure" cases (common with AI-generated requirements)
    c. **How it should work in the evaluator:**
    The evaluator performs dependency vulnerability auditing in a Docker-isolated environment. It reuses the same "dependency image" concept you already defined for build_import: dependencies are installed once (network allowed during build stage), then evaluation runs in a container with network disabled for reproducibility and safety.
    Two key principles:
        i. Audit what will actually run → audit the resolved installed set (not just the declared requirements)
        ii. Keep evidence complete → store raw tool output + evaluator-normalized summary + final status
    d. Technical spec:

i. Phase 1: Produce a resolved dependency snapshot
python -m pip freeze --all > /artifacts/security_deps/resolved_requirements.txt
Why?
1. requirements.txt can be incomplete (unpinned or indirect deps)
2. *pip freeze* gives you the exact installed versions → best audit target
3. stored as evidence for reproducibility/debugging

ii. Phase 3: Dependency vulnerability audit
1. Run the audit against the resolved set and emit JSON:
pip-audit -r /artifacts/security_deps/resolved_requirements.txt -f json -o /artifacts/security_deps/pip_audit.json
2. We need to fetch an advisory source that will tell us about the potential vulnerabilities, this should be done in the build stage if we have the network disabled, rightly after we install dependencies.
    a. In build if network is disabled:
        i. Install deps from requirements.txt
        ii. If security_deps is enabled run pip_audit rule
        iii. Safe container's image as a base image

iii. Outcome mapping:
1. Fail if **any vulnerability** is reported for any resolved dependency.
    a. Audit ran successfully + 0 vulns → **PASS**
    b. Audit ran successfully + ≥1 vuln → FAIL with reason_code DEPS_VULNERABILITIES_DETECTED
    c. Tool execution error / invalid output → ERROR with reason_code TOOL_EXECUTION_ERROR

e. Evidence output

i. Evaluator status for this MUST write::
1. /artifacts/security_deps/status.json
    a. status: PASS|FAIL|ERROR
    b. reason_code:DEPS_VULNERABILITIES_DETECTED | TOOL_EXECUTION_ERROR
    c. For each failure:
        i. Name of the package
        ii. Version
        iii. Vulnerability
            1. Id

2. Description
3. Fix versions
4. Aliases (CVE…)

d. Summary:
   i. summary counts (packages audited, vulns found, unique vuln IDs)

e. Reference to evidence files

5. Policy

a. **Objective**: The evaluator enforces a project policy by rejecting code that uses forbidden APIs (specific functions/methods) and/or forbidden libraries (imports / dependencies), according to an explicit denylist.

b. This catches:

i. use of dangerous Python primitives (e.g., eval, exec)
ii. use of disallowed frameworks or libraries (which can be preset in resolved deps)
iii. use of banned system interaction modules (subprocess, os.system, socket)

c. What users must specify (and where to get the correct names):

i. Import/module name (best for code scanning)
   1. This is what you can detect via:
      a. AST parsing of import / from ... import …
      b. (optionally) regex for dynamic imports like importlib.import_module("x")
   2. Where users get it:
      a. from the library docs (“import name” shown in usage)
      b. from the package itself (common knowledge)
      c. or by checking in Python: *python -c "import <name>; print(<name>.__name__)*"

ii. Package/distribution name:
   1. What appears in pip freeze
   2. Where users get it:
      a. PyPI project name (the canonical distribution name)
      b. What appears in requirements.txt

d. **How it should work in the evaluator:**
The evaluator runs in a sandboxed container with the project mounted. It executes code to to detect imports, and the use of forbidden api and the forbidden packages in resolved requirements.txt

e. Technical specs:

i. Phase 1: Build an import index (AST-based)
   1. Parse all .py files and extract:
      a. import X -> X
      b. import X as Y -> X
      c. from X import Y -> X

d. Also detect dynamic imports:
    i. importlib.import_module("X") -> X
    ii. __import__("X") -> X
2. Output:
    a. */artifacts/policy/imports_index.json*
3. This catches "forbidden libraries" as they appear in code.

ii. Phase 2: Read resolved deps:
1. Reuse: */artifacts/security_deps/resolved_requirements.txt*
2. Parse normalized distribution names (casefold, normalize - vs _).
3. This catches "forbidden libraries" even if not imported directly in scanned files (e.g., imported in generated code paths, or imported conditionally).

iii. Phase 3: Forbidden API detector (AST call pattern)
1. Scan AST for forbidden calls, with apis in user inputs
2. Output:
    a. */artifacts/policy/api_violations.json*

iv. Phase 4: Decision policy
1. Fail if any violation is found in:
    a. forbidden_modules ∩ imports_index
    b. forbidden_packages ∩ resolved_requirements
    c. forbidden_apis ∩ detected_calls
2. Outcome mapping:
    a. violations found → FAIL with POLICY_VIOLATION_DETECTED
    b. none → PASS
    c. scan error (parse failure etc.) → ERROR with TOOL_EXECUTION_ERROR (or ANALYSIS_ERROR)

f. What users should write in YAML:

1. policy:
    a. forbidden_modules:
    b. - yaml
    c. - Crypto
    d. - sklearn
    e. forbidden_packages:
        i. - pyyaml
        ii. - pycryptodome
        iii. - scikit-learn
    f. forbidden_apis:
        i. - eval
        ii. - exec
        iii. - os.system
        iv. - subprocess.Popen

v. - importlib.import_module

g. Output:

i. /artifacts/policy/imports_index.json

```
{
  "modules_imported": [
    {"module": "yaml", "file": "src/app.py", "line": 1},
    {"module": "os", "file": "src/app.py", "line": 2, "as": "o"},
    {"module": "subprocess", "file": "src/app.py", "line": 3, "imported_names": [{"name": "Popen", "as": "Proc"}]}
  ],
  "dynamic_imports": []
}
```

ii. /artifacts/policy/policy_violations.json

```
{
  "violations": [
    {
      "type": "FORBIDDEN_MODULE",
      "match": "yaml",
      "evidence": {"file": "src/app.py", "line": 1, "snippet": "import yaml"}
    },
    {
      "type": "FORBIDDEN_API",
      "match": "os.system",
      "evidence": {"file": "src/app.py", "line": 6, "snippet": "o.system(cmd)", "note": "alias o -> os"}
    },
    {
      "type": "FORBIDDEN_API",
      "match": "subprocess.Popen",
      "evidence": {"file": "src/app.py", "line": 7, "snippet": "Proc(cmd, shell=True)", "note": "alias Proc -> subprocess.Popen"}
    },
    {
      "type": "FORBIDDEN_PACKAGE",
      "match": "pyyaml",
      "evidence": {"source": "resolved_requirements.txt", "line": "pyyaml==6.0.1"}
    }
  ]
}
```

h. Evidence output:

i. Evaluator status for this MUST write:

1. /artifacts/policy/status.json
    a. status: PASS|FAIL|ERROR
    b. reason_code:DEPS_VULNERABILITIES_DETECTED | TOOL_EXECUTION_ERROR
    c. Summary:
        i. total_violations
        ii. by_type:
            1. FORBIDDEN_MODULE
            2. FORBIDDEN_PACKAGE
            3. FORBIDDEN_API
        iii. Artifacts:
            1. Imports_index: "policy/imports_index.json"
            2. Violations : …
            3. Resolved_requirements: …

6. Quality:

a. Objective:
   The evaluator assesses whether the generated code meets a minimal maintainability and hygiene baseline by running deterministic static quality checks in an isolated environment. It fails the rule if the code violates fixed quality constraints that are strong signals of low maintainability or likely defects.
b. This catches:
    i. obvious style/format issues that reduce readability and reviewability
    ii. unused imports / unreachable code / ambiguous constructs (lint-level defects)
c. How it should work in the evaluator:
   The evaluator runs quality checks inside a sandboxed Docker container with the project mounted. No app logic is executed. Tools are pinned in the evaluator image for reproducibility. Output is captured as JSON evidence where possible, and normalized into a stable evaluator schema, then converted into PASS/FAIL/ERROR.
d. Technical spec:
    i. Phase 0: Tool Provisioning
        1. Ruff : primary quality gate
        2. Black
    ii. Phase 1: Lint
        1. Run Ruff over the project and emit machine-readable output.
            a. *ruff check /workspace --output-format json > /artifacts/quality/ruff.json*
        2. FAIL if any Ruff violations exist
    iii. Phase 2 : Format check (Black)

1. Black doesn't tell you quality it tells you format conformity, its useful for reviewability
2. *black --check --diff /workspace > /artifacts/quality/black.diff*

iv. Gating opinions:

v. Ruff rules are tagged (e.g. F=pyflakes errors, E/W=pycodestyle, etc.). For v1 we can gate on a minimal set:

1. Fail if any finding is in:
    a. F* (Pyflakes: undefined names, unused imports, etc.)
    b. E9* (syntax-like issues surfaced by lint)

e. Evidence output:

i. Evaluator status for this MUST write:

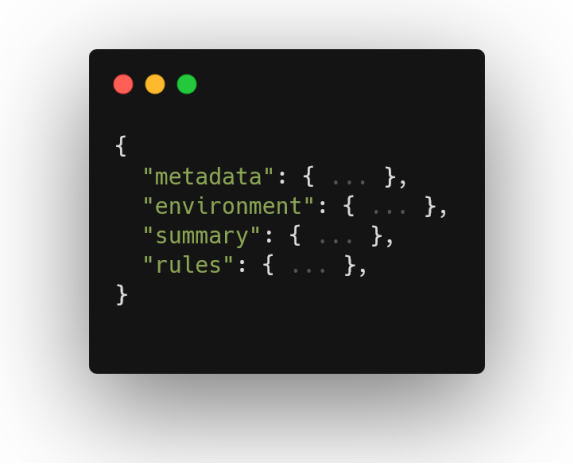
1. /artifacts/quality/status.json
    a. status: PASS|FAIL|ERROR
    b. reason_code:QUALITY_VIOLATIONS_DETECTED
    c. Summary:
        i. Findings_total
        ii. Findings_families
    d. Artifacts:
        i. quality/ruff.json
        ii. quality/black.diff
        iii. quality/ruff.log

Final Report Format

Canonical format: JSON

.artifacts/final_report.json

High-level Structure of the Reports



Report Sections (Detailed)

1. **metadata:**
    a. job_id : A simple auto-generated hash or id to track runs
    b. timestamp: Timestamp of the run
    c. duration_seconds: Total time duration of the run
    d. evaluator_version: Current Version of the evaluator (0.1.0)
2. **environment** :
    a. runtime: Specifies the runtime version (in 0.1.0 python)
    b. sandbox: The sandbox field in the Yaml
    c. Dependencies:
        i. requirements_file: The requirements.txt file
        ii. dependency_image (optional if network is false): The hash of the image used for dependencies install
3. **summary**:
    a. status: (PASS | FAIL | ERROR | SKIP)
    b. failed_rules: list of rules that failed
    c. passed_rules: list of rules that passed
    d. skipped_rules: list of rules that weren't specified
    e. error_rules: list of rules that had an internal error (due to an internal failure)
    f. primary_failure:
        i. rule
        ii. reason_code
        iii. message
4. **rules**: Detailed, per-rule results + evidence pointers.
    a. enabled (True|False): enabled field from yaml
    b. status: Status of the rule
    c. severity: Severity type of the rule
    d. reason_code:
    e. message:
    f. Metrics (dependent on rule):
        i. Build_import:
            1. Install_duration_seconds: Dependencies
            2. compiled_files
        ii. Unit_tests:
            1. Test_runs
            2. coverage
        iii. security_sast:
        iv. security_deps:
        v. policy:
        vi. quality:
    g. evidence:
        i. logs: logs of compilation and pip install
    h. Artifacts: List of files generated

Reasons_code and message:

reasons_code are deterministic codes defined dependent on the situation and what is written in the YAML

Each code has a message mapped to it:

- **"DEP_INSTALL_FAILED":**"Dependency installation failed while installing {requirements_file}.",
- **"COMPILE_FAILED":** Python compilation failed due to syntax errors.",
- **"IMPORT_FAILED":** "Failed to import entrypoint module '{entrypoint}'.",
- **"IMPORT_TIMEOUT":**"Import of entrypoint module '{entrypoint}' exceeded timeout ({timeout_seconds}s).",
- **"TESTS_FAILED":** "{failed_tests} test(s) failed during unit test execution.",
- **"COVERAGE_BELOW_THRESHOLD":** "Line coverage {coverage:.2f} is below required threshold {threshold:.2f}.",
- **"SAST_FINDINGS_DETECTED":** "Static security analysis detected {findings_total} issue(s).",
- **"DEPS_VULNERABILITIES_DETECTED":** "Dependency scan detected {vulns_total} vulnerable package(s).",
- **"FORBIDDEN_API_USED":** "Forbidden API usage detected ({violations_total} occurrence(s)).",
- **"SKIPPED_DUE_TO_PREVIOUS_FAILURE":** "Rule skipped because a prerequisite rule failed.",
- **"RULE_DISABLED_OR_NOT_IMPLEMENTED":** "Rule is disabled or not executed in this version."