Appendix X – Low-Level Design: Rule-Based Misconfiguration Detection Module

X.1 Purpose and Scope

The Rule-Based Misconfiguration Detection module evaluates **normalized cloud storage configurations** against a predefined set of best-practice rules and produces **structured, explainable findings** with severity (Low/Medium/High) and remediation guidance. This module belongs to the **Logic / Analysis tier** and is intentionally isolated from cloud API access, persistence, scheduling, reporting, and UI concerns.

In scope

- Apply predefined rules to each storage resource configuration.
- Produce findings with severity and short remediation suggestions.
- Handle missing/invalid configuration data gracefully.

Out of scope

- Retrieving configurations from providers.
- Storing raw configurations or credentials.
- Scheduling, reporting/export, UI interaction.

X.2 Module Responsibilities

The module's responsibilities are:

1. Evaluate each ResourceConfig against all registered Rules.
2. Detect misconfiguration categories including public exposure, missing/incorrect encryption, excessive permissions, and disabled/insufficient logging.
3. Generate a Finding per violated rule with: type=misconfiguration, severity (Low/Medium/High), description, remediation, and non-sensitive metadata.
4. Ensure deterministic, explainable results suitable for validation on synthetic configurations.

X.3 Internal Components

Component	Responsibility
RuleEngine	Coordinates rule execution per resource and aggregates findings.
Rule (interface/abstract)	Encapsulates detection logic for exactly one misconfiguration condition; returns ≤1 finding.
ResourceConfig	Provider-agnostic normalized representation of storage configuration used by rules.
Finding	Output object describing a detected misconfiguration, including severity and remediation
Severity	Enum: **LOW, MEDIUM, HIGH** (as required by SRS/SDD).

X.4 Data Structures (Contracts)

X.4.1 ResourceConfig (input contract)

ResourceConfig represents a normalized, provider-agnostic cloud storage configuration.
It is produced by higher layers after provider-specific retrieval and normalization. The object must not contain credentials or secrets; all data is processed in memory only.

Fields:

- resource_id: str - Unique identifier of the storage resource (e.g., bucket name with account context)
- provider: str - Cloud provider identifier (e.g., AWS, Azure, GCP)
- resource_type: str - Type of storage resource (e.g., bucket, container)
- public_access: bool | None - Indicates whether the resource is publicly accessible; None represents unknown or unavailable data
- encryption_enabled: bool | None - Indicates whether encryption at rest is enabled
- logging_enabled: bool | None - Indicates whether access logging is enabled
- policy_summary: dict[str, object] - Simplified representation of access policy attributes required for permission analysis
- metadata: dict[str, str] - Non-sensitive contextual information (e.g., region, account alias)

Invariant:
Missing or unknown values must be represented explicitly (e.g., None) to allow rules to apply conservative and predictable evaluation behavior.

X.4.2 Finding (Output Contract)

Finding represents a **detected misconfiguration** produced by the rule-based evaluation process.
Each finding is structured to support investigation, reporting, and remediation workflows.

Fields:

- finding_type: str - Fixed value: "misconfiguration".
- rule_id: str - Identifier of the rule that detected the issue.
- title: str - Short summary of the misconfiguration.
- description: str - Detailed explanation of the detected issue.
- severity: Severity - Severity classification (LOW, MEDIUM, HIGH).
- resource_id: str - Identifier of the affected resource.
- evidence: dict[str, object] - Non-sensitive facts supporting the finding (e.g., "public_access": true).
- remediation: str - Short recommendation describing how to resolve the misconfiguration.
- timestamp_utc: str *(optional)* - Time of detection; may be assigned by higher-level orchestration components.

X.5 Interfaces

X.5.1 Rule Interface

The Rule interface defines the contract for a single misconfiguration detection rule.
Each rule evaluates one specific condition on a given resource configuration and returns at most one finding.

Interface methods:

- getRuleId() -> str - Returns a unique identifier for the rule.
- getDescription() -> str - Returns a short, human-readable description of the misconfiguration being detected.
- getDefaultSeverity() -> Severity - Returns the default severity level assigned to findings produced by this rule.
- evaluate(resource: ResourceConfig) -> Finding | None - Evaluates the rule against the provided resource configuration. Returns a Finding if a violation is detected; otherwise returns None.

Rule constraints:

- Rules should be implemented as **stateless components** to ensure deterministic behavior and facilitate unit-level testing.
- Rules must handle missing or incomplete input data gracefully and must not propagate unhandled exceptions to the caller.

X.5.2 RuleEngine Interface

The RuleEngine interface defines the coordination logic for executing registered rules against storage resource configurations and aggregating the resulting findings.

Interface methods:

- registerRule(rule: Rule) -> None - Registers a rule instance with the engine.
- evaluateResource(resource: ResourceConfig) -> list[Finding] - Applies all registered rules to a single resource configuration and returns the list of detected findings.
- evaluateAll(resources: list[ResourceConfig]) -> list[Finding] - Applies rule evaluation to a collection of resource configurations and returns the aggregated list of findings.

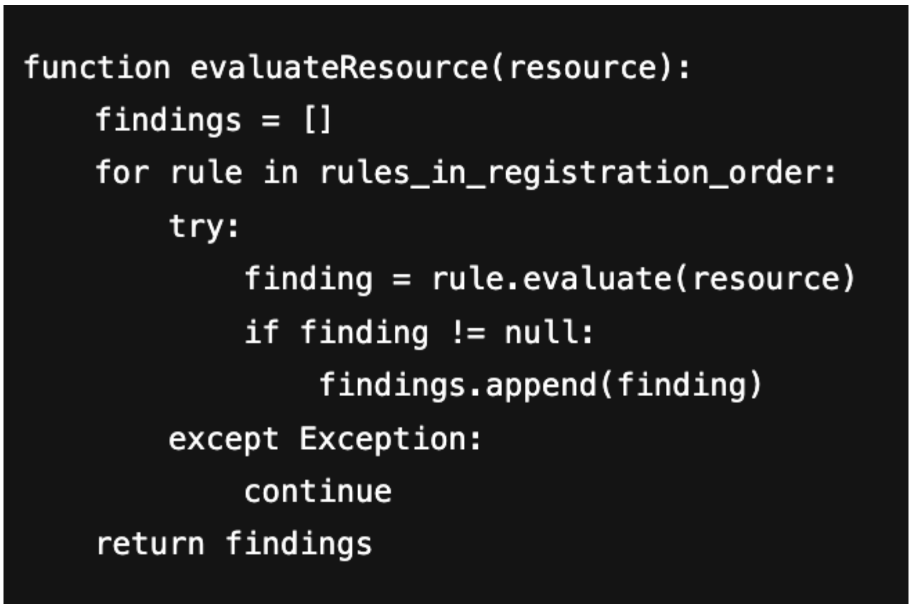
Engine constraints:

- Failure or exception in one rule evaluation must not interrupt the execution of other rules.
- Rule execution order must be deterministic to ensure repeatable and predictable evaluation results.

X.6 Control Flow (Pseudo-Code)

X.6.1 Evaluation of a Single Resource

The following pseudo-code describes the evaluation of a single normalized storage resource against all registered rules. Rules are executed sequentially in a deterministic order, and findings are aggregated per resource.



```
function evaluateResource(resource):
    findings = []
    for rule in rules_in_registration_order:
        try:
            finding = rule.evaluate(resource)
            if finding != null:
                findings.append(finding)
        except Exception:
            continue
    return findings
```

Rule evaluation failures are isolated to the individual rule and do not interrupt the overall evaluation process.

<u>X.6.2 Evaluation of Multiple Resources</u>

The evaluation of multiple resources applies the single-resource evaluation logic iteratively and aggregates all produced findings.

```
function evaluateAll(resources):
    allFindings = []
    for resource in resources:
        allFindings.extend(evaluateResource(resource))
    return allFindings
```

<u>X.6.3 Typical Rule Evaluation Pattern</u>

Each rule follows a common evaluation pattern: it validates the availability of required input fields, applies a specific misconfiguration condition, and produces a finding only when a violation is detected.

```
function evaluate(resource):
    if requiredFieldsMissing(resource):
        return null
    if violationDetected(resource):
        return Finding(
            rule_id = ...,
            severity = defaultSeverity,
            description = ...,
            remediation = ...,
            evidence = ...
        )
    return null
```

Rules are designed to behave conservatively when configuration data is incomplete, avoiding false positives while ensuring predictable and deterministic behavior.

X.7 Error Handling and Edge Cases

The Rule-Based Misconfiguration Detection module applies a defensive and deterministic error-handling policy to ensure robustness and predictable behavior during evaluation.

- **Missing or invalid fields:** If required configuration fields are missing or invalid, the rule evaluation returns no finding.
  Rules must not cause the evaluation process to fail due to incomplete input data.
- **Rule evaluation failure:** Any exception raised during the evaluation of a specific rule is caught at the engine level.
  The failure of one rule does not interrupt or affect the evaluation of other rules.
- **Empty rule set:** If no rules are registered in the engine, evaluation returns an empty list of findings.
- **Empty resource set:** If the input resource list is empty, evaluation returns an empty list of findings.
- **Incomplete configurations:** When configuration data is partial or inconclusive, rules apply conservative behavior to avoid generating false-positive findings.

This policy ensures that the module remains stable under imperfect input conditions while maintaining reliable and repeatable evaluation results.

X.8 Extensibility

The Rule-Based Misconfiguration Detection module is designed to support straightforward extension through the addition of new rules.
Adding a new misconfiguration rule requires the following steps:

1. Implement the Rule interface and assign a unique rule_id.
2. Define the rule’s default severity level and remediation description.
3. Register the rule with the RuleEngine using registerRule.

No changes are required to existing rules or to the rule engine when introducing new rules. This design enables modular evolution of the detection logic while preserving isolation, maintainability, and predictable behavior.

X.9 Testability

The module design supports **unit-level validation** through the use of synthetic resource configurations with controlled misconfiguration scenarios. Each rule and the rule engine can be tested independently to ensure correctness, determinism, and robustness.
Testing focuses on verifying:

- Detection of injected misconfigurations.
- Correct severity classification (LOW, MEDIUM, HIGH).
- Absence of findings for correctly configured resources (false-positive control).
- Stable behavior in the presence of incomplete or invalid input data.

Minimum Test Set

The minimum test set includes the following scenarios:

- **Public access detection:** A publicly readable or writable storage resource produces a finding.
- **Encryption detection:** A resource with missing or incorrectly configured encryption produces a finding.
- **Permission analysis:** A resource with overly permissive access policies produces a finding.
- **Logging configuration:** A resource with disabled or missing access logging produces a finding.
- **Missing fields:** A resource configuration with missing required fields produces no finding, and evaluation continues without failure.

These tests ensure that the module behaves predictably under both valid and imperfect input conditions and that individual rule failures do not affect overall evaluation.

X.10 Traceability to SRS Requirements

SRS Requirement	Design Element
FR4 – Predefined best-practice rules	**Rule** interface and rule registration mechanism in **RuleEngine**
FR5 – Public access detection	**PublicAccessRule** implementing **Rule**
FR6 – Missing or incorrect encryption detection	**EncryptionRule** implementing **Rule**
FR7 – Excessive permissions detection	**ExcessivePermissionsRule** implementing **Rule**
FR8 – Severity classification (Low / Medium / High)	**Severity** enumeration and **Rule.getDefaultSeverity()**
FR13 – Short remediation suggestions	**Finding.remediation** field
NFR8 – Graceful handling of incomplete or invalid data	Exception isolation in **RuleEngine** and conservative rule behavior