Low-Level Design Proposal: Rule-Based Misconfiguration Detection Module

1. Introduction and Architectural Context

The project *"Automated Misconfiguration & Threat Detection in Public Cloud Storage"* is designed according to a layered (N-Tier) architecture, separating presentation, service/control, logic (business), and data responsibilities. This separation follows the architectural principles discussed in the course tutorials, where core business rules are isolated from user interface, infrastructure, and persistence concerns.

This document focuses on the **Rule-Based Misconfiguration Detection Module**, which is part of the **Logic/Business layer** of the system. The purpose of this module is to encapsulate the domain rules that define what constitutes an insecure or misconfigured cloud storage resource.

The document proposes a **low-level design (LLD)** for this single module and justifies its selection for detailed design and implementation.

2. Justification for Module Selection

The Rule-Based Misconfiguration Detection module was selected for low-level design because it represents the **core business logic** of the system. In accordance with the layered (N-Tier) architecture presented in the course material, essential business rules and domain logic are implemented within a dedicated logic layer, clearly separated from presentation, infrastructure, and data persistence concerns.

The selection of this module is justified by the following reasons:

1) **Central system functionality** - The module implements the primary security detection capability of the system by evaluating cloud storage configurations against predefined misconfiguration rules. Without this logic, the system cannot fulfill its main purpose.
2) **Non-trivial internal logic -** Rule evaluation requires normalization of heterogeneous cloud configurations, independent execution of multiple rules per resource, handling of missing or partial data, and consistent severity assignment. These aspects justify a dedicated low-level design.
3) **High impact on correctness and reliability** - Errors in this module directly lead to false positives or undetected security risks. A well-defined low-level design is therefore critical to ensure deterministic behavior and predictable results.
4) **Clear separation of concerns and testability** - The module is fully isolated from user interface, cloud API access, scheduling, and persistence logic. This strict separation aligns with the layered architecture principles and enables independent unit-level validation using synthetic configurations.

3. Module Responsibilities and Scope

3.1 Responsibilities

- Evaluating cloud storage configurations against predefined security rules.
- Detecting known misconfiguration patterns.
- Producing structured, explainable findings with severity classification.

3.2 Explicit Non-Responsibilities

To preserve layered architecture boundaries, this module does **not**:

- Retrieve configurations from cloud providers.
- Persist raw configuration data or findings.
- Handle scheduling, orchestration, or reporting.
- Interact with UI components.

These responsibilities are handled by other layers and services in the system, consistent with the tutorial's separation of concerns principles.

4. Low-Level Design

4.1 Internal Structure

The module consists of two main components:

1) **Rule Engine**
   - Coordinates rule execution.
   - Applies all registered rules to each storage resource configuration.
   - Aggregates detected findings.
2) **Rule**
   - Represents a single business rule.
   - Encapsulates the logic for detecting one specific misconfiguration.

4.2 Interfaces

The following interfaces exist:

1) **Rule Interface**
   - getRuleId()
   - getDescription()
   - getDefaultSeverity()
   - evaluate(resourceConfig) → Optional<Finding>

   Each rule evaluates exactly one condition and produces at most one finding.
2) **RuleEngine Interface**
   - registerRule(rule)
   - evaluateResource(resourceConfig) → List<Finding>
   - evaluateAll(resourceConfigs) → List<Finding>

4.3 Input Data

The RuleEngine operates on normalized, provider-agnostic configuration objects, prepared by higher layers.

These objects include:

- Resource identifiers and metadata.
- Public access indicators.
- Encryption configuration.
- Logging configuration.
- Simplified access policy attributes.

All input data is processed **in memory only**, in accordance with system security constraints.

4.4 Output Data

The module outputs structured **Finding** objects containing:

- Finding type (misconfiguration).
- Severity (Low/Medium/High).
- Title and description.
- Remediation recommendation.
- Rule identifier and non-sensitive metadata.

5. Control Flow and Error Handling

For each storage resource:

1) The RuleEngine iterates over all registered rules.
2) Each rule evaluates the resource configuration independently.
3) If a rule detects a violation, it returns a Finding.
4) The RuleEngine aggregates all findings and returns them to the orchestration layer.

Error Handling

- Rule evaluation errors (e.g., missing fields) are handled locally.
- A failure in one rule does not stop evaluation of other rules.
- Conservative behavior is preferred to avoid false positives when data is incomplete.

6. Testability and Validation Considerations

The module design supports unit-level validation by:

- Injecting synthetic configurations with known misconfigurations.
- Verifying correct detection and severity classification.
- Ensuring no findings are produced for correctly configured resources.

This approach aligns with the validation strategy recommended in the tutorial and supports deterministic, repeatable testing.

7. Conclusion

The Rule-Based Misconfiguration Detection module encapsulates the system's core business logic and fits naturally within the Business/Logic layer of a layered architecture. Its clear responsibilities and interfaces make it an appropriate candidate for low-level design, while preserving modularity, testability, and separation of concerns in accordance with the course tutorials.