Automated Misconfiguration & Threat Detection in Public Cloud Storage

Project Report

by Jamal Tannous, Asya Persan and Dan Ydov

Link to Codebase

Link to Demo

1.Problem Description

Public cloud storage services such as Amazon S3, Google Cloud Storage, and Azure Blob Storage are widely used to store application data, logs, backups, and sensitive information due to their scalability and availability. As cloud adoption increases, storage configurations are often managed by multiple teams and automated pipelines, leading to growing system complexity.

A persistent security risk in public cloud environments is **misconfiguration**. Storage resources are frequently exposed due to incorrect access policies, missing encryption, disabled logging, or overly permissive permissions. These issues are not caused by vulnerabilities in the cloud providers themselves, but by configuration errors made by users. Numerous data breaches have demonstrated that misconfigured storage can lead to severe data exposure even when underlying cloud infrastructure is secure.

Manual auditing of cloud storage configurations is time-consuming, error-prone, and does not scale with modern cloud environments. As configurations evolve continuously, previously secure assumptions may become invalid without clear visibility. Consequently, manual reviews and ad-hoc scripts are insufficient for maintaining a reliable security posture.

This project addresses the need for an **automated approach** to detecting security-relevant misconfigurations in public cloud storage systems. The proposed solution analyzes storage configurations, identifies common misconfiguration patterns, and produces structured, explainable findings with severity classification. Emphasis is placed on modularity, accuracy, and validation using synthetic configurations, enabling extensibility and predictable behavior as system complexity grows.

2. Technological aspect

2.1 Technological Survey

Public cloud storage technologies provide scalable and highly available mechanisms for storing data without direct management of physical infrastructure. Object-based storage services offered by major cloud providers are configured through access policies, encryption settings, logging mechanisms, and lifecycle rules. While this flexibility enables diverse use cases, it also introduces a broad configuration surface that directly impacts security.

Cloud storage security relies primarily on **configuration-based controls**. Access control is enforced through identity and access management (IAM) policies and resource-level permissions, while data protection depends on encryption at rest and in transit. Visibility and auditability are achieved through logging and monitoring mechanisms. As a result, the security posture of cloud storage systems is largely determined by the correctness of these configurations rather than by underlying platform vulnerabilities.

Technological approaches to detecting misconfigurations can be broadly divided into **rule-based detection** and **anomaly-based detection**. Rule-based approaches apply predefined best-practice rules to configuration data in order to identify known insecure patterns, such as public access, missing encryption, or overly permissive permissions. These methods are deterministic, explainable, and well-suited for validating compliance with established security requirements.

Anomaly-based approaches, often implemented using machine learning techniques, focus on identifying deviations from normal behavior rather than explicitly defined configuration errors. In cloud storage environments, this typically involves analyzing access logs to detect unusual usage patterns. While such methods can reveal subtle or previously unknown threats, they generally require more complex validation and offer lower explainability.

In practice, effective cloud security solutions often combine these approaches. Rule-based detection provides reliable coverage for well-understood risks, while anomaly detection can complement it by identifying unexpected behavior. This project primarily focuses on rule-based misconfiguration detection, prioritizing explainability, deterministic behavior, and systematic validation.

2.2 Current Solutions Survey

Existing approaches to securing public cloud storage configurations can be grouped into **cloud-provider native tools**, **commercial cloud security posture management (CSPM) solutions**, and **academic or research-based approaches**.

Cloud providers offer built-in security services that analyze storage configurations and flag common misconfigurations such as public access or missing encryption. These tools benefit from deep integration with the provider's ecosystem but are typically **provider-specific** and offer limited transparency or customization of the underlying detection logic.

Commercial CSPM platforms aim to provide unified visibility across multiple cloud environments using predefined rule sets aligned with security benchmarks and compliance standards. While these solutions offer broad coverage and operational maturity, they are often **complex, costly**, and largely opaque, making it difficult to validate or adapt their detection logic.

Academic and open-source approaches explore both **rule-based** and **machine learning–based** techniques for cloud security analysis. Rule-based methods provide deterministic and explainable detection of known misconfiguration patterns, while machine learning approaches focus on anomaly detection using behavioral data such as access logs. Although research solutions demonstrate valuable concepts, they are often evaluated in controlled environments and may lack emphasis on modularity, maintainability, or systematic validation.

Across these categories, common limitations include limited explainability, restricted multi-cloud portability, and insufficient validation transparency. These gaps motivate the need for solutions that prioritize clarity, extensibility, and verifiable behavior alongside detection accuracy.

This project addresses these limitations by focusing on a **modular, rule-based detection framework** that emphasizes deterministic behavior, explainability, and explicit validation, rather than attempting to replicate full-scale commercial platforms.

2.3 Parameters for Solution Comparison

In order to evaluate and compare different approaches for detecting misconfigurations in public cloud storage systems, a set of comparison parameters is required. These parameters are chosen to reflect both **security effectiveness** and **software engineering quality**, in line with the goals of this project and the course principles.

Detection accuracy is a primary parameter, as the system must correctly identify genuine misconfigurations while avoiding false positives. Excessive false alarms reduce trust in the system and increase operational overhead, particularly in large-scale cloud environments.

Explainability is a key requirement for security-related systems. Detected issues must be accompanied by clear explanations that describe why a configuration is considered insecure and how it can be remediated. Explainable results enable security engineers and developers to understand findings, verify their correctness, and take appropriate corrective action.

Determinism and predictability are essential for reliable validation and maintenance. Given the same input configuration, a solution should always produce the same output. Deterministic behavior simplifies testing, debugging, and regression analysis, and is particularly important in rule-based security validation.

Extensibility and maintainability measure how easily new detection logic can be added or existing logic modified as security best practices evolve. Cloud environments change rapidly, and a viable solution must support incremental extension without requiring major architectural changes.

Validation feasibility is another important parameter. The solution should support systematic validation using synthetic configurations with known properties. This enables controlled testing of detection logic and supports reproducible evaluation, which is essential in an academic and engineering context.

Finally, **portability and modularity** are considered to assess how tightly a solution is coupled to specific cloud providers or execution environments. A modular, provider-agnostic design

allows detection logic to be reused across different systems and integrated into larger security workflows.

These parameters collectively provide a structured basis for comparing existing solutions and for justifying the design decisions made in this project. They ensure that the selected approach balances security effectiveness with clarity, reliability, and engineering rigor.

2.4 Decisions Justifications

This section justifies the key architectural and methodological decisions made in the project using the comparison parameters defined in Section 2.3. The goal was to design a solution that balances security effectiveness with explainability, determinism, extensibility, and systematic validation, in accordance with both practical security needs and software engineering principles.

Decision 1: Selecting a rule-based core for misconfiguration detection
A rule-based approach was chosen as the primary mechanism for detecting storage misconfigurations such as public exposure, missing encryption, excessive permissions, and disabled logging. This decision directly supports **explainability**, as each detected issue can be clearly associated with a specific violated security rule and accompanied by an explicit remediation recommendation.

In addition, rule-based detection provides **deterministic and predictable behavior**. Given the same configuration input, the system produces consistent results, which simplifies debugging and enables repeatable validation. This property is particularly important for systematic testing using synthetic configurations, where expected outcomes must be clearly defined in advance.

Finally, rule-based detection aligns well with established cloud security best practices, which are often expressed as explicit configuration requirements rather than probabilistic patterns.

Decision 2: Adopting a layered, modular system architecture
The system was designed using a layered architecture that separates configuration retrieval, analysis logic, and result handling. This decision supports **modularity and portability**, as detection logic remains independent of cloud-provider-specific APIs and execution environments.

From a maintainability perspective, clear separation of concerns allows individual components to evolve independently. New detection rules can be added without modifying orchestration logic, and support for additional cloud providers can be introduced without altering the core analysis layer. This structure also enables focused testing of individual layers, improving overall system reliability.

Decision 3: Introducing a normalized, provider-agnostic configuration representation
Cloud providers expose storage configuration data using different schemas and semantics. To enable consistent analysis, the system relies on a normalized, provider-agnostic representation of configuration attributes supplied by higher layers.

This design choice improves **portability and maintainability** by decoupling detection logic from provider-specific formats. It also enhances **determinism**, as missing or unknown

attributes can be handled consistently, enabling predictable and conservative evaluation behavior across different environments.

Decision 4: Conservative evaluation and fault isolation
The system adopts conservative behavior when configuration data is incomplete or ambiguous, preferring to avoid false positives rather than flag uncertain violations. This decision directly supports **detection accuracy** and reduces unnecessary noise in security findings.

In addition, the evaluation process isolates failures at the individual rule level, ensuring that an error in one rule does not interrupt the overall analysis. This approach improves **reliability** and aligns with non-functional requirements related to robustness and graceful error handling.

Decision 5: Complementing rule-based detection with anomaly-based analysis
While rule-based detection provides reliable coverage for known misconfiguration patterns, it does not address all potential security risks. To extend coverage, the project includes an anomaly-based analysis component operating on access behavior data.

This component is designed as a complementary mechanism rather than a replacement for rule-based checks. It targets unusual or suspicious behavior patterns that may indicate misuse or compromise, while remaining lightweight and suitable for validation using synthetic data. This balance supports broader detection coverage without sacrificing explainability or validation feasibility.

Decision 6: Producing structured findings with severity classification
Detected issues are reported as structured findings that include severity classification and remediation guidance. This decision supports **usability and prioritization**, allowing security-relevant issues to be addressed efficiently based on risk level.

Severity classification also enables consistent interpretation of results across different system components and aligns with both functional and non-functional requirements defined earlier in the project.

3. Stakeholders' Description

This section identifies the main stakeholders of the system and describes their roles and expectations. The stakeholder analysis is derived from the project scope, the defined use cases, and the functional and non-functional requirements established in earlier project phases. Understanding these stakeholders is essential for ensuring that the system addresses real operational needs and aligns with its intended usage context.

Security Engineer / Security Analyst

Role: Responsible for maintaining the organization's cloud security posture, identifying security risks, and investigating potential exposures related to cloud storage resources.

Primary needs:

- Ability to analyze cloud storage configurations and detect high-impact misconfigurations such as public access, missing encryption, excessive permissions, and disabled logging.
- Clear, explainable findings that indicate why a configuration is considered insecure and how it can be remediated.
- Severity classification to support prioritization of risks and efficient triage.
- Sufficient non-sensitive evidence and metadata to support investigation without exposing raw configuration data.

DevOps / Cloud Engineer

Role: Designs, deploys, and operates cloud infrastructure, including storage resources, often through automated pipelines and infrastructure-as-code tools.

Primary needs:

- Early detection of risky configuration changes, ideally before or immediately after deployment.
- Actionable remediation guidance that maps directly to concrete configuration fixes.
- Deterministic and predictable scan results that can be trusted and integrated into operational workflows.
- Minimal false positives to avoid alert fatigue and reduce friction in fast-paced development environments.

Compliance Officer / Project Manager

Role: Oversees organizational compliance, risk management, and reporting, typically without requiring deep technical involvement in system configuration.

Primary needs:

- High-level visibility into the security posture of cloud storage resources.
- Structured reports grouped by severity that are suitable for audits, reviews, and stakeholder communication.
- The ability to consume findings without direct exposure to sensitive technical details.

Security Administrator

Role: Responsible for system setup and governance, including onboarding cloud environments, configuring access permissions, and managing operational settings such as scan scheduling.
Primary needs:

- Secure onboarding mechanisms that rely on read-only access and avoid unnecessary privilege escalation.
- Predictable and configurable scan execution (manual or scheduled).
- Assurance that sensitive data handling constraints are respected, particularly regarding credentials and raw configuration data.

Indirect Stakeholders (Organization and Data Owners)

Role: Business units and data owners whose information is stored in cloud storage systems. These stakeholders do not interact directly with the system but are affected by its effectiveness.
Primary needs:

- Reduced risk of unintended data exposure.
- Faster identification and mitigation of security-relevant configuration issues.
- Improved overall organizational security posture without increased operational burden.

The identified stakeholders collectively represent security, operations, compliance, governance, and business interests. The system is designed to address these perspectives by combining explainable detection logic, severity-based reporting, and modular architecture, while maintaining clear separation of responsibilities across system layers.

4. Requirements

This section defines the functional and non-functional requirements of the system. Functional requirements are first presented as an overview table to establish terminology and traceability, and are then mapped to stakeholders and system behavior.

4.0 Functional Requirements Overview

The following table summarizes all functional requirements referenced throughout the project. Each requirement is assigned a unique identifier and a concise description. These identifiers are used consistently in subsequent sections for traceability.

Table 1 – Functional Requirements Overview

FR ID	Short Name	Description
FR1	**Cloud connection**	**The system shall securely connect to a cloud provider using read-only credentials.**
FR2	**Configuration retrieval**	**The system shall retrieve cloud storage configuration attributes required for analysis (permissions, encryption, logging, policies).**
FR3	**Configuration normalization**	**The system shall normalize provider-specific configuration data into a unified, provider-agnostic representation.**
FR4	**Rule-based evaluation**	**The system shall evaluate storage configurations using predefined security rules.**
FR5	**Public access detection**	**The system shall detect publicly accessible storage resources.**
FR6	**Encryption detection**	**The system shall detect missing or incorrectly configured encryption.**
FR7	**Permission analysis**	**The system shall detect overly permissive access policies.**
FR8	**Severity classification**	**The system shall assign a severity level (Low/Medium/High) to each detected issue.**
FR9	**Log ingestion**	**The system shall ingest access logs for anomaly analysis.**
FR10	**Anomaly detection**	**The system shall apply unsupervised anomaly detection techniques to access logs.**

FR11	**Anomaly scoring**	**The system shall assign anomaly scores to suspicious access behavior.**
FR12	**Report generation**	**The system shall generate a human-readable scan report.**
FR13	**Remediation guidance**	**The system shall provide remediation suggestions for detected issues.**
FR14	**Export results**	**The system shall support exporting scan results in a structured format.**
FR15	**Scan history**	**The system shall maintain scan metadata for historical reference.**
FR16	**On-demand scanning**	**The system shall allow manual execution of scans.**
FR17	**Scheduled scanning**	**The system shall support periodic, scheduled scan execution.**

4.1 Functional Requirements Mapped to Stakeholders

This section maps the functional requirements defined in Section 4.0 to the system's stakeholders. The purpose of this mapping is to demonstrate traceability between stakeholder needs and system functionality, ensuring that all core requirements address concrete usage scenarios and responsibilities.

Security Engineer / Security Analyst

Stakeholder needs: Identify security-relevant misconfigurations, investigate findings, and prioritize risks based on severity.

Mapped functional requirements:

- **FR1–FR2:** Enable secure connection to cloud providers and retrieval of storage configuration data required for analysis.
- **FR4–FR8:** Evaluate storage configurations using predefined security rules, including detection of public access, missing encryption, and excessive permissions, and assign severity levels to detected issues.
- **FR9–FR11:** Support ingestion of access logs and application of anomaly detection to identify suspicious behavioral patterns.
- **FR12–FR13:** Generate readable reports with clear explanations and remediation guidance to support investigation and response.
- **FR16–FR17:** Allow both on-demand and scheduled scans to support reactive investigation and continuous monitoring.

DevOps / Cloud Engineer

Stakeholder needs: Validate cloud storage configurations during deployment and operation, receive actionable feedback, and avoid excessive false positives.
Mapped functional requirements:

- **FR2:** Retrieve relevant configuration attributes needed for validation of storage resources.
- **FR4–FR8:** Apply rule-based checks and severity classification to identify and prioritize risky configuration changes.
- **FR12–FR13:** Provide remediation guidance that maps directly to concrete configuration fixes.
- **FR16:** Support manual scan execution to enable pre-deployment and post-change validation workflows.

Compliance Officer / Project Manager

Stakeholder needs: Obtain high-level visibility into cloud storage security posture and support audit and review processes.
Mapped functional requirements:

- **FR12–FR13:** Produce structured, human-readable reports grouped by severity.
- **FR14:** Support export of scan results in a structured format suitable for reporting and review.
- **FR17:** Enable scheduled scans to support periodic compliance assessments.

Security Administrator

Stakeholder needs: Onboard cloud environments securely, configure scan execution, and ensure safe system operation.
Mapped functional requirements:

- **FR1:** Establish secure, read-only connections to cloud providers.
- **FR2:** Retrieve configuration data required for analysis without modifying cloud resources.
- **FR16–FR17:** Configure and manage manual and scheduled scan execution.

Indirect Stakeholders (Organization and Data Owners)

Stakeholder needs: Reduce the risk of unintended data exposure and improve overall security posture.
Mapped functional requirements:

- **FR4–FR8:** Detect high-impact misconfigurations and classify their severity.
- **FR12–FR13:** Present findings in a clear and understandable manner to support timely remediation.
- **FR16–FR17:** Support continuous detection through recurring scans.

The mapping above demonstrates that each functional requirement is directly linked to one or more stakeholder needs. This ensures that system functionality is not defined in isolation, but instead reflects real operational, security, and governance concerns.

4.2 General Functional Requirements

This section describes the system's general functional behavior at a system level, independent of specific stakeholders. It summarizes the main end-to-end capabilities of the system.

Cloud Connectivity and Configuration Retrieval

- The system shall support secure onboarding of cloud environments using read-only access.
- The system shall retrieve cloud storage configuration attributes required for analysis, including access policies, encryption settings, and logging status.
- Raw configuration data and credentials shall be handled in memory only during scan execution and shall not be persisted.

Rule-Based Misconfiguration Detection

- The system shall evaluate storage configurations against predefined best-practice rules to detect security-relevant misconfigurations such as public exposure, missing encryption, and overly permissive permissions.
- Each detected issue shall be assigned a severity level and accompanied by a brief explanation and remediation guidance.

Anomaly Detection on Access Logs

- The system shall support analysis of synthetic access logs using unsupervised anomaly detection techniques.
- Suspicious activity patterns shall be identified and represented as structured anomaly findings with associated scores.

Results Aggregation and Reporting

- The system shall aggregate misconfiguration and anomaly findings into a unified results set.
- Scan results shall be presented in a human-readable format, grouped by severity, and support export in a structured format.

Scan Execution Modes

- The system shall support both on-demand scan execution and scheduled, periodic scans.

Overall, the system provides an end-to-end workflow for onboarding cloud environments, retrieving storage configurations, applying rule-based misconfiguration analysis, optionally performing anomaly detection on synthetic logs, and presenting/exporting consolidated findings with severity and remediation guidance.

4.3 Non-Functional Requirements

This section defines the non-functional requirements (NFRs) of the system. These requirements specify quality attributes and operational constraints that apply across all system functions, independently of specific stakeholders or use cases.

Performance Requirements

- **NFR1:** The system shall retrieve and analyze cloud storage configuration data within a reasonable response time suitable for interactive use.
- **NFR2:** The anomaly detection component shall process access logs efficiently so as not to significantly delay overall scan execution.

Security Requirements

- **NFR3:** All communication with cloud providers shall use secure communication protocols (e.g., HTTPS/TLS).
- **NFR4:** The system shall not persist cloud credentials or raw cloud configuration data beyond the duration of a scan.
- **NFR5:** Access to scan execution and scan results shall be restricted to authorized users only.

Usability Requirements

- **NFR6:** The system shall be usable without requiring advanced security or cloud-specific expertise.
- **NFR7:** Scan results shall be presented in a clear and understandable format, enabling straightforward interpretation and action.

Reliability Requirements

- **NFR8:** The system shall handle missing, incomplete, or invalid configuration data gracefully without terminating the scan process.
- **NFR9:** Temporary failures in communication with cloud provider APIs shall not cause system failure and shall be handled robustly.

Portability Requirements

- **NFR10:** The system shall be portable and runnable on common operating systems, including Windows, macOS, and Linux.

5. Architecture and High-Level Design

This section presents the overall architecture of the system and describes the responsibilities and interactions of its main components. The architecture is designed to ensure modularity, explainability, and extensibility, while supporting secure handling of cloud configuration data and systematic validation of detection logic.

5.1 Architectural Overview

The system adopts a **layered and modular architecture** that separates cloud interaction, orchestration, analysis, and reporting into distinct components. This separation ensures that cloud-provider-specific details are isolated from detection logic and that core analysis functionality can be developed and tested independently.
At a high level, the system consists of:

- A **Cloud Integration layer**, responsible for retrieving cloud storage configurations.
- A **Control and Orchestration layer**, responsible for managing scan execution.
- An **Analysis layer**, responsible for rule-based misconfiguration detection and anomaly detection.
- A **Reporting and Persistence layer**, responsible for aggregating and presenting results.

This architecture supports both on-demand and scheduled scans and enables independent evolution of individual components without affecting the overall system.

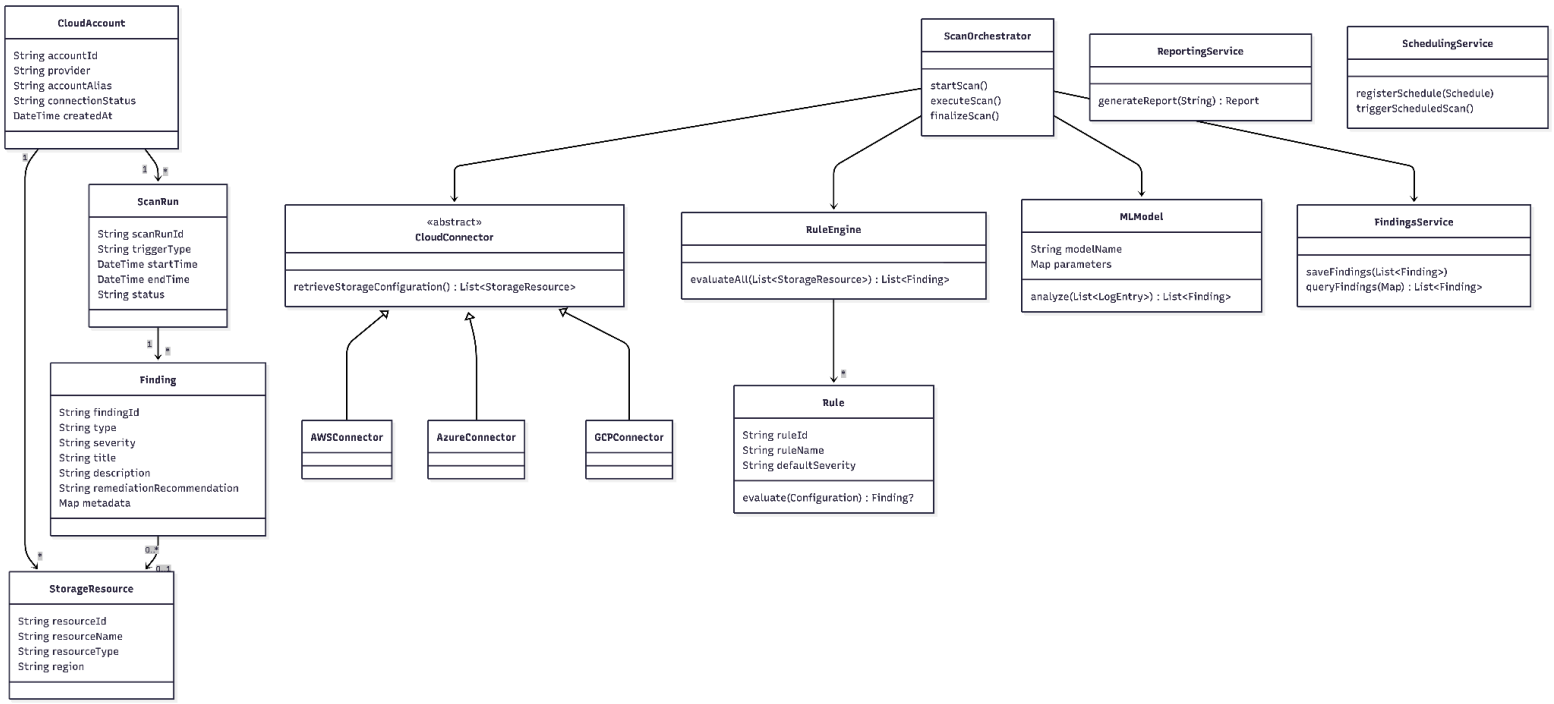

Figure 5.1 – High-Level System Architecture

Figure 5.1 presents the high-level architecture of the system, illustrating the separation between cloud integration, orchestration, analysis, and reporting components, as well as the abstraction of cloud-provider-specific connectors.

5.2 Component Responsibilities

Cloud Integration Components

Cloud integration is implemented through an abstract **CloudConnector** interface, which defines a uniform mechanism for retrieving storage configuration data. Concrete implementations (e.g., AWS, Azure, and GCP connectors) encapsulate provider-specific APIs and data formats.
Key characteristics:

- Read-only access to cloud environments.
- No modification of cloud resources.
- No persistence of raw configuration data or credentials beyond scan execution.

This abstraction ensures portability and allows support for additional cloud providers without changes to the analysis logic.

Control and Orchestration Components

The **ScanOrchestrator** coordinates the overall scan lifecycle, including scan initiation, execution, and finalization. It is responsible for invoking cloud integration and analysis components in the correct order and managing scan state.
A separate **SchedulingService** supports periodic scan execution, enabling automated, recurring security assessments.
These components provide a clear execution boundary and ensure predictable and repeatable scan behavior.

Analysis Components

The Analysis layer consists of two complementary components:

- The **RuleEngine**, which evaluates storage configurations against predefined security rules to detect known misconfiguration patterns.
- The **MLModel**, which performs anomaly detection on synthetic access logs to identify unusual behavior patterns.

Analysis components are designed to be independent of cloud-provider specifics and operate on normalized input data. Individual rule failures do not interrupt overall scan execution, supporting robust and fault-tolerant behavior.

Reporting and Persistence Components

The **FindingsService** is responsible for storing and querying detected findings, while the **ReportingService** generates structured, human-readable reports based on aggregated results.
These components focus exclusively on result handling and presentation, ensuring that reporting logic remains decoupled from detection and orchestration logic.

5.3 High-Level Data Flow

A typical scan proceeds as follows:

1. The ScanOrchestrator initiates a scan (manually or via the SchedulingService).
2. The CloudConnector retrieves storage configuration data.
3. The RuleEngine evaluates configurations and produces misconfiguration findings.
4. The MLModel analyzes synthetic access logs and produces anomaly findings.
5. Findings are aggregated and passed to the ReportingService for presentation and export.

At no stage are cloud credentials or raw configuration data persisted beyond the active scan.

5.4 Architectural Rationale

The selected architecture directly supports the system's requirements and design goals:

- **Modularity:** Clear separation of concerns enables independent development and testing.
- **Extensibility:** New rules, providers, or analysis components can be added with minimal impact.
- **Explainability:** Rule-based analysis produces transparent and interpretable findings.
- **Security:** Sensitive data handling is constrained through architectural boundaries.

This high-level design provides a stable foundation for the detailed design described in the following section.

5.5 Threat Model and Assumptions

The system is designed to detect *configuration-driven security risks* in cloud storage resources rather than active exploitation attempts. The primary threat considered is an external, unauthenticated attacker who can discover and access publicly exposed or weakly protected storage resources due to misconfiguration.

The system operates under the following assumptions:

- Read-only access to storage configuration metadata.
- No inspection of runtime traffic, request payloads, or stored data contents.
- No storage or handling of cloud credentials, secrets, or sensitive data.

Threats such as runtime exploitation, insider abuse, credential compromise, or privilege escalation beyond configuration analysis are explicitly out of scope.

Given this threat model, a rule-based detection approach provides deterministic and explainable results that align with security auditing, compliance validation, and preventive risk assessment use cases.

6. Low-Level Design

This section presents the low-level design (LLD) of the **Rule-Based Misconfiguration Detection module**.
This module was selected for detailed design because it implements core business logic, is non-trivial (multiple independent rules, deterministic aggregation, conservative handling of missing data), and is highly testable in isolation.
The design follows a layered architecture principle: the module belongs to the **Logic/Analysis tier** and is intentionally isolated from cloud API integration, persistence, scheduling, reporting/export, and UI concerns.

6.1 Purpose and Scope

The purpose of the module is to evaluate **normalized, provider-agnostic** cloud storage configurations against a predefined set of best-practice rules, and to produce structured, explainable findings including severity (Low/Medium/High) and remediation guidance.
In scope

- Apply predefined rules to each storage resource configuration.
- Produce findings with severity and short remediation suggestions.
- Handle missing/invalid configuration data gracefully.
- Ensure deterministic evaluation behavior suitable for validation on synthetic inputs.

Out of scope

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

6.2 Internal Components

The module is composed of the following internal elements:

- **RuleEngine -** Coordinates rule execution for a resource and aggregates findings.
- **Rule (interface/abstract) -** Encapsulates detection logic for exactly one misconfiguration condition; returns at most one finding.
- **ResourceConfig (input contract) -** Provider-agnostic normalized representation of storage configuration used by rules.
- **Finding (output contract) -** Output object describing a detected misconfiguration, including severity and remediation.
- **Severity**
  Enumeration of LOW, MEDIUM, HIGH.

6.3 Data Structures and Contracts

6.3.1 Resource Configuration Input Contract

The module operates on a normalized configuration object produced by higher layers (after provider-specific retrieval and normalization). It must not contain secrets; all processing is in memory only.

Fields (conceptual contract):

- resource_id (str): unique identifier of the storage resource (with account context).
- provider (str): cloud provider identifier (AWS/Azure/GCP).
- resource_type (str): storage type (bucket/container).
- public_access (bool | None): whether public access is enabled; None represents unknown/unavailable.
- encryption_enabled (bool | None): whether encryption at rest is enabled.
- logging_enabled (bool | None): whether access logging is enabled.
- policy_summary (dict): simplified access policy attributes required for permission analysis.
- metadata (dict): non-sensitive contextual information (e.g., region, account alias).

Invariant: missing/unknown values must be explicitly represented (e.g., None) to enable conservative evaluation behavior.

6.3.2 Finding Output Contract

A finding represents a detected misconfiguration produced by a rule evaluation. Findings are structured to support investigation, reporting, and remediation workflows.

Fields (conceptual contract):

- finding_type: fixed value "misconfiguration".
- rule_id: identifier of the rule that detected the issue.
- title: short summary.
- description: explanation of the issue.
- severity: LOW / MEDIUM / HIGH.
- resource_id: affected resource identifier.
- evidence: non-sensitive facts supporting the finding.
- remediation: short recommendation.
- timestamp_utc (optional): may be assigned by higher-level components.

6.4 Interfaces

6.4.1 Rule Interface

Each rule evaluates one specific condition and returns at most one finding. Rules are stateless to ensure deterministic behavior and facilitate unit testing.

Methods:

- getRuleId() -> str
- getDescription() -> str
- getDefaultSeverity() -> Severity
- evaluate(resource) -> Finding | None

Constraints:

- Must handle missing/incomplete data gracefully.
- Must not propagate unhandled exceptions to callers.

6.4.2 RuleEngine Interface

The engine coordinates execution of registered rules and aggregates findings.

Methods:

- registerRule(rule) -> None
- evaluateResource(resource) -> list[Finding]
- evaluateAll(resources) -> list[Finding]

Constraints:

- Rule execution order must be deterministic.
- Failure in one rule must not interrupt execution of other rules.

6.5 Control Flow and Error Handling

6.5.1 Single Resource Evaluation (High-Level Flow)

For each resource:

1. The engine iterates over all registered rules in a deterministic order.
2. Each rule evaluates the resource configuration independently.
3. If a rule detects a violation, it returns a finding; otherwise returns none.
4. The engine aggregates all returned findings and returns them to the orchestration layer.

6.5.2 Error Handling Policy

The module applies a defensive, deterministic policy to ensure predictable behavior under imperfect inputs:

- **Missing/invalid fields:** if required configuration data is missing or invalid, the rule returns no finding.
- **Rule evaluation failure:** exceptions raised by a rule are caught at the engine level; evaluation continues with other rules.
- **Empty rule set:** evaluation returns an empty finding list.
- **Empty resource set:** evaluation returns an empty finding list.
- **Incomplete configurations:** conservative behavior is preferred to reduce false positives.

6.6 Extensibility

The design supports extension by adding new rules without modifying existing rules or the engine:

1. Implement the Rule interface and assign a unique rule_id.
2. Define default severity and remediation.
3. Register the rule with the engine.

This preserves modularity and maintains stable interfaces over time.

6.7 Testability

The module is explicitly designed for unit-level validation using synthetic resource configurations with controlled misconfiguration scenarios. Testing focuses on:

- Detection of injected misconfigurations.
- Correct severity classification.
- Absence of findings for correctly configured resources (false-positive control).
- Stable behavior under incomplete/invalid inputs.
- Isolation of failures so one rule does not affect overall evaluation.

The minimum test set includes public access detection, encryption detection, excessive permissions detection, logging detection, and missing-field behavior.

7. Additional Information

This section highlights technical characteristics of the project that are specific to its nature and are not fully captured by the architectural or low-level design descriptions. The focus is on implementation principles that influence correctness, validation, and extensibility.

7.1 Rule Evaluation Pattern

The detection logic follows a **one-rule–one-condition–one-finding** pattern. Each rule evaluates exactly one misconfiguration condition and may produce at most one finding. Rules are evaluated independently and in a deterministic order, and all findings are aggregated without short-circuiting execution. This ensures clear responsibility boundaries and fault isolation.

7.2 Deterministic and Conservative Evaluation

Deterministic execution is enforced: given the same input configurations, the system always produces the same findings in the same order. This property enables reproducible validation and stable regression testing.
To reduce false positives, the system applies a **conservative evaluation strategy**. When required configuration attributes are missing or unknown, rules do not generate findings. Findings are therefore based only on explicit evidence.

7.3 Normalized Data Contracts

All rule evaluations operate on normalized, provider-agnostic configuration objects and produce structured findings. This abstraction decouples detection logic from cloud-provider-specific representations and simplifies rule implementation and extension.

7.4 Validation-Oriented and Extensible Design

Rule logic is stateless and side-effect free, enabling precise unit-level validation using synthetic inputs. New detection logic can be added by introducing additional rules without modifying existing components, supporting incremental evolution of the system.

The technical properties described above reflect deliberate design choices that prioritize explainability, determinism, and testability, and complement the architectural and low-level design presented in earlier sections.

7.5 Scalability and Computational Complexity

The rule-based misconfiguration detection process evaluates each storage resource independently against a fixed set of predefined rules. The overall time complexity of the evaluation process is therefore $O(R \times N)$, where R is the number of registered rules and N is the number of evaluated storage resources.
Memory consumption is linear with respect to the number of input resources and the number of generated findings, as all processing is performed in memory without persistent state.
Due to the stateless and independent nature of rule evaluation, the system is inherently parallelizable across resources or rules. This enables straightforward horizontal scaling in future implementations without requiring changes to the detection logic.

8. Development Environment and Technical Choices

This section describes the development environment used in the project and justifies key technical choices, including programming language selection, execution environment, and data handling decisions.

8.1 Programming Language Selection

The system was implemented in **Python**. Python was selected due to its suitability for rapid prototyping, readability, and strong support for modular software design. These properties align well with the project's goals of clarity, explainability, and ease of validation.
Python also provides a mature ecosystem for testing and scripting, enabling concise implementation of rule-based logic and efficient development of unit tests. The language's expressive syntax allows security rules to be implemented in a clear and maintainable manner, reducing the likelihood of implementation errors.

8.2 Development and Execution Environment

Development was performed in a **local development environment**, using a standard desktop operating system. The system is executed as a command-line application and does not require a graphical user interface or specialized runtime environment.
This choice supports:

- Ease of development and debugging
- Reproducibility of results
- Platform independence

The system can be executed on common operating systems without environment-specific dependencies.

8.3 Data Handling: Local and In-Memory Processing

The project operates exclusively on **local, synthetic configuration data** and **in-memory processing**. Cloud storage services are not accessed directly during execution.
This design choice was intentional and motivated by:

- Security considerations (no handling of real credentials or sensitive configurations)
- Reproducibility and controlled experimentation
- Independence from external cloud environments

Raw configuration data and credentials are never persisted. Only derived, non-sensitive findings are produced as output during execution

.

8.4 Testing and Tooling

Testing was performed using the **pytest** framework, which supports concise and expressive unit tests. Tests focus on validating rule behavior, aggregation logic, and error handling under controlled synthetic inputs.
Using a lightweight testing framework enables fast iteration and repeatable execution, supporting the project's emphasis on determinism and validation.

8.5 External Dependencies and Simplicity

The implementation minimizes external dependencies and avoids reliance on cloud SDKs, databases, or external services. This decision reduces system complexity and ensures that the behavior of the system is fully determined by the provided input data.
By keeping the environment simple and self-contained, the project remains easy to understand, extend, and evaluate.
The development environment and technical choices made in this project prioritize simplicity, reproducibility, and security. The use of Python, local execution, and in-memory data handling supports rapid development while maintaining clarity and validation-friendly behavior, in line with the project's overall design goals.

9. Full Validation Report

This section presents the validation of the implemented system. Validation focuses on verifying that the implemented functionality satisfies the defined functional and non-functional requirements and behaves correctly under controlled and reproducible conditions.
Validation was performed using **automated unit tests** and **demonstration execution** on synthetic input data. No real cloud environments, credentials, or external services were used.

9.1 Validation Strategy

The validation strategy is based on **deterministic, unit-level testing** of the core rule-based misconfiguration detection module, combined with an end-to-end demonstration run.
Synthetic cloud storage configurations were constructed to represent both compliant and non-compliant scenarios. This approach enables precise control over test conditions and ensures reproducibility of results.
The validation strategy was designed to:

- verify correctness of individual misconfiguration detection rules,
- ensure deterministic and repeatable behavior,
- validate aggregation and fault isolation mechanisms,
- avoid dependency on external cloud providers.

9.2 Test Environment

Validation was conducted in a local development environment using:

- Python as the execution platform,
- the pytest framework for automated testing,
- in-memory synthetic configuration objects.

No cloud SDKs, credentials, or persistent storage were involved in test execution. All validation activities are fully reproducible on any standard Python environment.

9.3 Functional Requirements Validation

The following functional behaviors were validated through automated tests and demo execution:

- **Misconfiguration detection -** Rules correctly identify publicly accessible resources, missing or incorrect encryption, disabled access logging, and overly permissive access policies when these conditions are present in the input configuration.
- **Multiple findings per resource -** When multiple misconfigurations exist for a single resource, the system produces a separate finding for each violated rule.
- **False-positive control -** Correctly configured resources do not produce findings.
- **Rule independence -** Failure or absence of one rule does not affect the execution of other rules.
- **Graceful handling of incomplete data -** Missing or unknown configuration attributes do not cause runtime errors and do not result in speculative findings.

All validated functional behaviors conform to the requirements defined in the SRS.

9.4 Non-Functional Requirements Validation

Validation also addressed key non-functional requirements:

- **Determinism -** Repeated execution with identical inputs produces identical findings in the same order.
- **Robustness -** Exceptions or failures in individual rule evaluations are isolated and do not interrupt overall execution.
- **Maintainability and testability -** Rules are stateless and independently testable, supporting modular validation.
- **Security -** No sensitive data, credentials, or real cloud configurations are processed or persisted during validation.

These properties were validated through repeated test execution and inspection of system behavior.

9.5 Automated Test Results

Automated validation was performed using pytest. The test suite includes:

- rule-level tests validating individual misconfiguration detection logic,
- engine-level tests validating aggregation, determinism, and fault isolation.

All automated tests completed successfully, with **nine test cases executed and zero failures observed**.

9.6 Demonstration Execution

In addition to automated testing, a demonstration script was executed to simulate a full scan over synthetic storage configurations. The demo produced structured findings including:

- affected resource identifiers,
- rule identifiers,
- severity levels,
- descriptions and remediation guidance.

The demonstration confirms correct end-to-end behavior of the system and illustrates the format and content of generated findings.

9.7 Validation Summary

The validation results confirm that the implemented system:

- satisfies the defined functional requirements,
- meets key non-functional requirements,
- behaves deterministically and robustly,
- produces explainable and reproducible results.

No deviations from the specified behavior were observed during validation.

Validation Aspect	Validation Method	Result
Rule correctness	Unit tests	Pass
Fault isolation	Engine robustness tests	Pass
Deterministic behavior	Repeated execution	Pass
Non-functional requirements	Design review and tests	Pass
End-to-end execution	Demo run	Pass

10. Summary

This project presented the design and implementation of a rule-based misconfiguration detection system for cloud storage resources. The system focuses on identifying common security misconfigurations using predefined best-practice rules applied to normalized, provider-agnostic configurations.

A layered architecture was adopted to separate cloud integration, orchestration, analysis, and reporting concerns. The core rule-based detection module was designed with determinism, explainability, and testability as primary goals. These properties were validated through automated unit testing and demonstration execution on synthetic inputs.

The implemented system satisfies the defined functional and non-functional requirements, produces structured and interpretable findings, and behaves predictably under both valid and imperfect input conditions. The project demonstrates a complete software engineering process, from requirements analysis and architectural design to implementation and validation.

The current implementation focuses on static analysis of synthetic, normalized cloud storage configurations and does not integrate with real cloud provider APIs. Runtime behavior, access patterns, and data-level threats are not analyzed.

The implemented rule set targets common and well-understood storage misconfigurations and does not aim to provide exhaustive coverage of all possible security risks.

Future work may include integration with real cloud provider APIs, expansion of the rule set, incorporation of machine learning–based anomaly detection, enhanced persistence and reporting mechanisms, and the development of a user-facing interface. Due to the system's modular and extensible design, these enhancements can be incorporated without significant architectural changes.

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.

X.6.2 Evaluation of Multiple Resources

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
```

X.6.3 Typical Rule Evaluation Pattern

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