4.5 SDK Design and Development

4.5.1 SDK Design Philosophy

This chapter presents the design and implementation of the proposed security orchestration system. Instead of relying on a static, monolithic application, the system adopts a modular, extensible architecture to better respond to modern, fast-evolving ransomware threats. The core contribution is twofold: (1) the development of a Software Development Kit (SDK) that defines the framework's contracts, data models, and developer tools; and (2) a reference implementation composed of a backend Core System and a suite of pluggable modules built using the SDK.

The design philosophy is centered on creating a policy-based decision-making system. The framework eschews a traditional, hard-coded decision matrix in favor of a more flexible and modern orchestration model. The "decision" is not made by a single, intelligent module but is an emergent property of a workflow's execution. This logic is not compiled into the system; it is defined by a human operator in external, auditable workflow files. This "Policy as Code" approach is the cornerstone of the system's adaptability, allowing the security posture to evolve without redeploying the core framework. The SDK is the foundational component that enables this architecture by enforcing standardized contracts, providing a canonical data language, and abstracting away the complexities of the underlying distributed system.

The SDK is the foundation of the system, designed to enforce a standardized contract, provide a canonical data language, and abstract away the underlying complexities of a distributed, message-driven architecture. By providing this mandatory Java library, the SDK ensures that any user-created module can integrate seamlessly into the security framework. The Core System and proof-of-concept modules serve as a comprehensive validation of the SDK's design, proving that developers can rapidly build custom, high-level security logic on top of a stable and resilient orchestration core. The following sections will detail the specific interfaces, classes, and third-party libraries that constitute this design.

4.5.2 The SDK Architectural Design

The SDK is distributed as a self-contained Java library (com.nis1.thesis.sdk.jar). Its architectural design is focused on providing a clean, high-level API that allows developers to create powerful modules without needing expertise in the underlying messaging or network protocols. This is achieved through three key areas: Core Contracts, Standardized Data Models, and Developer Aids.

The contracts are defined as Java interfaces, which is a fundamental design choice. Using interfaces allows the Core System to interact with any module through a consistent, polymorphic API, completely decoupling the core logic from the specific implementation of any given module. At its core is the PluggableModule interface. This contract contains two mandatory lifecycle methods: initialize() and shutdown(). These methods are the entry and exit points that the Core System's Module Lifecycle Manager will call, guaranteeing a uniform mechanism for starting and stopping all modules.

The second critical contract is the CoreSystemApi interface. This is the developer’s secure gateway to the Communication Fabric. It provides two key functions, publishEvent() and subscribeToEvent(), which are the sole methods for sending and receiving standardized events. By exposing the fabric through this high-level interface, the SDK completely abstracts away the underlying protocols like AMQP and the specific client libraries used to communicate with the message broker.

```java
// File: com/nis1/thesis/sdk/PluggableModule.java
public interface PluggableModule {
    String getName();
    void initialize(CoreSystemApi api);
    void shutdown();
}

// File: com/nis1/thesis/sdk/CoreSystemApi.java
public interface CoreSystemApi {
    void publishEvent(Event<?> event);
    void subscribeToEvent(String eventType, Consumer<Event<?>> listener);
}
```

All inter-module communication uses a consistent event format, defined by a set of Plain Old Java Objects (POJOs) within the SDK. The SDK defines a generic Event<T> class that wraps all messages. This class uses Java generics to provide a type-safe envelope containing standard metadata such as a unique eventId, timestamp, and the eventType string, which serves as the AMQP routing key. For the serialization of these objects into a network-transmittable format, JSON was chosen. To perform this serialization, a mature and feature-rich library such as Google's Gson or Jackson would be used internally by the Core System's implementation of the SDK. These libraries are industry standards, chosen for their robust handling of Java generics and their high performance. Specialized payload types like HostAlertData and enum values like MitigationAction ensure strong typing and eliminate guesswork during development.

```java
// File: com/nis1/thesis/sdk/Event.java
public final class Event<T> {
    private final String id = UUID.randomUUID().toString();
    private final Instant timestamp = Instant.now();
    private final String type;
    private final T data;
    // ... constructor and getters ...
}
```

To reduce complexity and boilerplate code, the SDK includes utility classes. A key example is the ModuleHelper class. A developer could instantiate this class with their CoreSystemApi handle, gaining access to simplified methods for common tasks like logging, configuration handling, or publishing standard event types. For instance, a publishIpReputation() method would internally handle the creation of the IpReputationData payload, wrap it in a standard Event object, and call the underlying api.publishEvent()

method. As mentioned, the serialization from Java objects to JSON is handled transparently by the framework, meaning the developer only ever interacts with strongly-typed Java objects.

4.5.3 Reference Implementation

The reference system implements the SDK's abstract concepts into a running backend, demonstrating a complete, end-to-end solution. This implementation relies on a curated set of mature, open-source libraries to perform its functions.

The Communication Fabric is implemented using RabbitMQ as the message broker. The Core System's internal implementation of the CoreSystemApi uses the official RabbitMQ Java Client library (. This library was chosen as it provides direct, low-level access to the AMQP protocol, allowing for fine-grained control over exchanges, queues, and message properties like persistence and acknowledgements.

```
Java
// Conceptual snippet from the internal CoreSystemApi implementation
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;
// ... other imports

public class CoreSystemApiImpl implements CoreSystemApi {
    private Channel amqpChannel;
    private JsonSerializer serializer;

    // ... constructor to establish connection ...

    @Override
    public void publishEvent(Event<?> event) {
        try {
            String jsonMessage = serializer.serialize(event);
                            amqpChannel.basicPublish("security_events_exchange",
event.getType(), null, jsonMessage.getBytes());
        } catch (IOException e) {
            // Handle exceptions...
        }
    }
    // ... implementation for subscribeToEvent ...
}
```

The Module Registry & Lifecycle Manager leverages core Java technologies to achieve its "plug-and-play" functionality. It uses java.io.File to scan a directory for .jar files. For each file, it creates a java.net.URLClassLoader. This classloader makes the code inside the JAR dynamically available to the running application. The manager then uses the Java Reflection API (java.lang.reflect) to inspect the classes within the JAR, searching for one that isAssignableFrom the PluggableModule interface. This is how it validates and instantiates modules without having prior knowledge of their class names.

```java
// Conceptual snippet from the ModuleLifecycleManager
import com.nis1.thesis.sdk.PluggableModule;
import java.io.File;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.jar.JarFile;

public class ModuleLifecycleManager {
    public PluggableModule loadModuleFromJar(File jarFile) throws Exception {
        URL[] urls = { new URL("jar:file:" + jarFile.getAbsolutePath() + "!/")
};
        URLClassLoader classLoader = URLClassLoader.newInstance(urls);

        // Scan the JAR for a class that implements the SDK contract
        try (JarFile jar = new JarFile(jarFile)) {
            // ... logic to iterate through JAR entries ...
            Class<?> loadedClass = classLoader.loadClass(className);
            if (PluggableModule.class.isAssignableFrom(loadedClass)) {
                // If found, create a new instance of it.
                return (PluggableModule)
loadedClass.getDeclaredConstructor().newInstance();
            }
        }
        return null;
    }
}
```

The Workflow Engine requires the ability to parse the .yml workflow files. For this, a library such as SnakeYAML would be integrated. SnakeYAML is a standard, robust library for parsing YAML into Java objects (POJOs), which the engine then uses as its set of instructions.

```java
// Conceptual snippet from the WorkflowEngine
import org.yaml.snakeyaml.Yaml;
import java.io.InputStream;
import java.util.Map;

public class WorkflowEngine {
    public void loadWorkflow(InputStream yamlStream) {
        Yaml yaml = new Yaml();
```

```
        // SnakeYAML parses the .yml file directly into a Java Map or custom
POJO.
        Map<String, Object> workflowDefinition = yaml.load(yamlStream);
        // ... store and process the loaded definition ...
    }
}
```

The Threat Context Store, in its reference implementation, would use a high-performance key-value store like Redis. To communicate with Redis from Java, a client library such as Jedis would be used. Jedis was selected for its high performance and its direct mapping of Redis commands to Java methods (e.g., jedis.hincrBy(), jedis.hset()), which is ideal for the atomic update operations required by the store.

Java

```
// Conceptual snippet from the ThreatContextStoreModule
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;

public class ThreatContextStoreModule implements PluggableModule {
    private JedisPool redisPool;

    @Override
    public void initialize(CoreSystemApi api) {
        this.redisPool = new JedisPool("localhost", 6379);
        api.subscribeToEvent("HOST_ALERT", this::handleHostAlert);
    }

    private void handleHostAlert(Event<?> event) {
        HostAlertData data = (HostAlertData) event.getData();
        try (Jedis jedis = redisPool.getResource()) {
            String redisKey = "host:" + data.getSourceIp();
            // Use atomic Redis commands to update the threat profile.
            jedis.hincrBy(redisKey, "threat_score", data.getSeverity());
        }
    }
}
```

The Suricata module's implementation depends on Suricata's output configuration. If Suricata is configured to output alerts in eve.json format to a log file, this module would use standard Java file I/O libraries (from the java.nio package) to tail the log file in real-time. It would also include a JSON parsing library (like Gson or Jackson) to parse each line of the log. Its code would implement PluggableModule and contain logic to

transform the relevant fields from the Suricata JSON object into a standardized NIDS_ALERT event, which is then published using the SDK.

```java
// Conceptual snippet from the SuricataModule
import com.nis1.thesis.sdk.*;
import com.google.gson.Gson;
import java.nio.file.*;

public class SuricataModule implements PluggableModule {
    private CoreSystemApi api;
    private final Gson gson = new Gson();

    @Override
    public void initialize(CoreSystemApi api) {
        this.api = api;
        // Start a thread to tail the eve.json log file.
        // WatchService watchService =
FileSystems.getDefault().newWatchService();
        // ... logic to read new lines from the file ...
        // String jsonLine = ...;
        // SuricataEveLog eveLog = gson.fromJson(jsonLine,
SuricataEveLog.class);
        // if ("alert".equals(eveLog.eventType)) {
            // ... create NIDS_ALERT payload and publish via api.publishEvent()
...
        // }
    }
}
```

4.5.4 SDK Extensibility and Third-Party Module Development

A primary objective of this research is not merely to create a single, static security application, but to design a foundational Software Development Kit (SDK) that fosters an ecosystem of interoperable security tools. The true measure of the SDK's success is its utility for a third-party developer—an individual or organization with no prior knowledge of the framework's internal implementation—to develop a custom module for an entirely new tool. This section discusses the practical development process for such a third party, detailing the SDK's intended uses, its capabilities, the scope of developer responsibility, and its inherent limitations.

4.5.4.1 Fulfilling the SDK Contract

For a third-party developer, the process begins by including the com.nis1.thesis.sdk.jar as a dependency in their own development environment. The SDK immediately imposes a set of non-negotiable rules through its Core Contracts. The developer's first task is to create a public class that implements the PluggableModule interface. This is the fundamental contract that guarantees their module will be discoverable and manageable by the Core System's Lifecycle Manager. By implementing the initialize() and shutdown() methods, the

developer is forced to consider the module's lifecycle from the outset, providing a standardized entry point for startup logic and a reliable exit point for graceful resource cleanup. This contractual obligation is the cornerstone of the framework's stability, ensuring that even poorly behaved modules can be managed without destabilizing the entire system.

```java
// A developer starting to create a new module for a proprietary tool named
"ThreatIntelX".
package com.thirdparty.modules.threatintelx;

// 1. The developer imports the master contract from the SDK.
import com.nis1.thesis.sdk.PluggableModule;
import com.nis1.thesis.sdk.CoreSystemApi;

// 2. The new class MUST implement the PluggableModule interface.
public class ThreatIntelXConnector implements PluggableModule {

    @Override
    public String getName() {
        return "ThreatIntelX CTI Enrichment Module";
    }

    @Override
    public void initialize(CoreSystemApi api) {
        // 3. The developer is required to provide an entry point.
        // All startup logic, such as connecting to the ThreatIntelX service
        // and subscribing to events, will go here.
        System.out.println(getName() + " is initializing...");
    }

    @Override
    public void shutdown() {
        // 4. The developer is required to provide an exit point.
        // All cleanup logic, such as closing network connections, will go
here.
        System.out.println(getName() + " is shutting down.");
    }
}
```

4.5.4.2 The Developer's Toolkit: Leveraging the SDK's API and Models

Once the contract is implemented, the developer's primary tool for interacting with the framework is the CoreSystemApi object, which is passed to their module's initialize() method. This object is their sole,

secure gateway to the Communication Fabric. To integrate a new tool—for example, a proprietary threat intelligence feed named "ThreatIntelX"—the developer would utilize the API as follows:

First, to contribute data, the developer would use the api.publishEvent() method. Their code would be responsible for connecting to the ThreatIntelX API, retrieving data, and—most importantly—normalizing that data into one of the SDK's Standardized Data Models. For instance, if ThreatIntelX provides IP reputation, the developer would create an instance of the SDK's IpReputationData class, populate it with the results, wrap it in a standard Event object with a unique type (e.g., "IP_REPUTATION_THREATINTELX"), and publish it. This act of normalization is critical; it ensures that the custom data is immediately understandable and usable by any other module in the ecosystem, such as the Workflow Engine, without requiring any changes to those modules.

```java
// Continuing the ThreatIntelXConnector class...
// Imports for SDK data models and other required libraries.
import com.nis1.thesis.sdk.*;
import com.nis1.thesis.sdk.payloads.*; // For IpReputationData,
EnrichmentRequestData
import okhttp3.OkHttpClient; // Example library for making HTTP calls

public class ThreatIntelXConnector implements PluggableModule {
    private CoreSystemApi api;
    private final OkHttpClient httpClient = new OkHttpClient(); //
Tool-specific logic

    @Override
    public String getName() { return "ThreatIntelX CTI Enrichment Module"; }

    @Override
    public void initialize(CoreSystemApi api) {
        this.api = api;
        // The developer uses the SDK to consume data by subscribing to a
generic request topic.
        api.subscribeToEvent("ENRICHMENT_REQUEST_IP",
this::handleEnrichmentRequest);
    }

    // The callback function to handle incoming requests.
    private void handleEnrichmentRequest(Event<?> event) {
        EnrichmentRequestData requestData = (EnrichmentRequestData)
event.getData();
        String ipToCheck = requestData.getIpAddress();

        // 1. Developer implements their own tool-specific logic.
```

```
        boolean isMalicious = queryThreatIntelXApi(ipToCheck);

        // 2. Developer normalizes the result into a standardized SDK data
model.
        IpReputationData resultPayload = new IpReputationData(ipToCheck,
isMalicious, "ThreatIntelX");
        Event<IpReputationData> resultEvent = new
Event<>("IP_REPUTATION_THREATINTELX", resultPayload);

        // 3. Developer uses the SDK to contribute the normalized result back
to the fabric.
        this.api.publishEvent(resultEvent);
    }

    // A placeholder for the developer's custom, tool-specific implementation.
    private boolean queryThreatIntelXApi(String ip) {
        // ... developer writes code using OkHttp to connect to their
proprietary API ...
        return true; // Simulate a malicious result
    }

    @Override
    public void shutdown() { /* ... */ }
}
```

Second, to consume data, the developer would use the api.subscribeToEvent() method. If their new module needed to be triggered by a host alert, they would subscribe to the alerts.host.* topic. This wildcard-based subscription, enabled by the underlying AMQP message broker but abstracted by the SDK, demonstrates a key capability: the new module can listen for and react to events generated by any existing EDR module (Wazuh, CrowdStrike, etc.) without having any direct knowledge of them. The developer simply writes a callback function that accepts a standard Event object, confident that the SDK and the fabric will handle the reliable delivery and deserialization of the message.

The developer is responsible for writing all tool-specific logic. This includes handling network connections to their tool's API, managing authentication (e.g., API keys), parsing the tool's specific data format (e.g., a proprietary XML schema), and managing any necessary state within their own module. The SDK is intentionally designed not to handle these tasks; its purpose is to facilitate the interaction between the module's custom logic and the central framework.

4.5.4.3 Inherent Design Limitations and Scope

While powerful, the SDK's design has intentional limitations that define its scope. The framework is fundamentally asynchronous and event-driven. Therefore, the SDK is not suitable for developing modules that require synchronous, low-latency, in-line packet processing, such as a traditional network firewall or an

intrusion prevention system (IPS). Such functionality belongs in the Data Plane, whereas the SDK is designed exclusively for building out-of-band, Application Plane services.

Furthermore, the reliability of the system is contingent upon the reliability of the underlying message broker. While RabbitMQ provides robust guarantees, a catastrophic failure of the broker would interrupt communication between all modules. The SDK does not, by itself, handle broker-level fault tolerance; this must be addressed at the infrastructure level through clustering and high-availability deployments of RabbitMQ.

Finally, the SDK provides the tools for communication but does not enforce the semantic quality of the data. A poorly written module could publish malformed or low-value data onto the fabric. The framework relies on well-defined workflows and, potentially, dedicated validation modules to filter and prioritize the event stream. The SDK provides the "language," but it is up to the ecosystem of modules to engage in meaningful conversation. These limitations represent the necessary trade-offs for achieving a highly flexible, scalable, and decoupled architectural design.

4.5.5 Operational Principles and Real-World Considerations

Beyond the static definition of its interfaces and data models, the SDK's architectural design directly enables a set of powerful operational principles for any framework built with it. This section discusses these dynamic principles—including policy-driven behavior, flexible trust models, and built-in resilience—not as features of a specific implementation, but as the intended outcomes of the SDK's core design. These principles demonstrate how the SDK empowers developers to build adaptable, fault-tolerant orchestration platforms rather than rigid, monolithic applications.

4.5.5.1 Policy-Driven Decision-Making System

A fundamental principle of the SDK's design is to facilitate a policy-driven decision-making system. The SDK deliberately avoids providing a mechanism for a centralized, hard-coded "Decision Matrix." Instead, its CoreSystemApi is designed to be the bridge between mechanisms (the Pluggable Modules that perform actions) and a separate policy executor (the Workflow Engine). The SDK provides the Event objects and the publish/subscribe methods that allow a workflow to be expressed as a sequence of message exchanges. This design choice is what makes a "Policy as Code" approach possible. By providing a simple, high-level API for event communication, the SDK empowers an architect to externalize the complex decision-making logic into user-definable workflows (.yml files), allowing security administrators to alter the framework's defensive posture without requiring any underlying code to be recompiled. The SDK, therefore, is not just a tool for building modules, but is the key enabler of an agile security posture.

```shell
# A unique, human-readable name for this specific workflow.
name: "Critical Server Incident with Manual Quarantine Approval"

# Version of the workflow.
version: 1.0
```

```yaml
# A detailed description of the workflow's purpose and logic.
description: >
  This workflow implements a highly cautious response policy for assets tagged
  as 'Critical Infrastructure'. It triggers on a high-confidence NIDS alert for
  such an asset. Instead of automatic quarantine, it publishes an approval
request
  to the operator dashboard. The workflow will only proceed with quarantine if
a
  human operator provides explicit approval within 30 minutes.

# --- TRIGGER ---
# This workflow is triggered directly by a high-confidence network alert.
trigger:
  # The event type from a NIDS/DPI module (e.g., Suricata).
  event_type: "NIDS_ALERT"

  # The condition checks two things: the alert severity and a custom tag
  # for the host, which could be provided by an asset management enrichment
module.
  condition: "{{ trigger.data.signatureSeverity == '1' and trigger.data.hostTag
== 'Domain Controller' }}"

# --- STEPS ---
# An ordered list of operations to perform after the workflow is triggered.
steps:
  #
----------------------------------------------------------------------------
---
  # STEP 1: An ACTION to notify and request human intervention.
  # This step runs immediately after the trigger conditions are met.
  - name: "Request Manual Approval from Operator"
    action:
      type: "PUBLISH_EVENT"
      event:
        # This is a special event type that the Web App/Dashboard Module is
designed to handle.
        type: "HUMAN_APPROVAL_REQUIRED"
        data:
          # Provide all the necessary context for the operator to make an
informed decision.
          targetHost: "{{ trigger.data.sourceIp }}"
          suspectedThreat: "{{ trigger.data.signature }}"
          evidenceSource: "Suricata NIDS"
```

```
          recommendedAction: "QUARANTINE"
          # Pass the unique ID of this workflow instance so the approval can be
routed back correctly.
          workflow_instance_id: "{{ workflow.instanceId }}"

  #
----------------------------------------------------------------------------
---
  # STEP 2: A STATEFUL PAUSE to wait for the human's decision.
  # The engine will pause the execution of this workflow instance at this step.
  - name: "Wait for Operator Approval"
    wait_for:
      # The engine now listens for a specific approval event, which would be
published
      # by the Web App/Dashboard Module when an operator clicks the 'Approve'
button.
      event_type: "MANUAL_ACTION_APPROVED"

      # Give the operator a reasonable amount of time to respond.
      timeout: "30 minutes"

      # The conditions ensure that we are acting on the correct approval.
      conditions:
        # Condition 1: Correlate the approval with this specific workflow
instance.
        - "{{ new_event.data.workflow_instance_id == workflow.instanceId }}"

        # Condition 2: Ensure the approved action matches what was requested.
        - "{{ new_event.data.approvedAction == 'QUARANTINE' }}"

  #
----------------------------------------------------------------------------
---
  # STEP 3: The FINAL ACTION, executed only if Step 2 (manual approval)
succeeded.
  - name: "Initiate Approved Host Quarantine"
    action:
      type: "PUBLISH_EVENT"
      event:
        # This is the final mitigation command for the SDN Response Module.
        type: "INITIATE_MITIGATION"
        data:
          # Target the host from the original trigger event.
          targetHost: "{{ trigger.data.sourceIp }}"
```

```
          action: "QUARANTINE"

          # The justification now clearly states that a human approved the
action.
          justification: >
            Automated quarantine of Critical Server was manually approved by
operator
            '{{ new_event.data.approvingOperator }}' in response to NIDS alert
            (Signature: '{{ trigger.data.signature }}').
```

4.5.5.2 Scenarios of Customizability and Precedence Enabled by the SDK

The SDK's decoupled, event-driven design provides developers and operators with unparalleled flexibility to define the precedence and trustworthiness of security tools. The SDK itself is architecturally agnostic about the source of an event; it treats an Event object containing a HostAlertData payload from a trusted commercial EDR and one from a simple open-source tool identically. This deliberate design choice forces the operator to explicitly define trust and precedence within the policy layer (the .yml workflows), rather than having it hard-coded into the framework.

This allows an administrator to implement several distinct and highly customized response scenarios, each reflecting a different security philosophy or risk tolerance, using the same set of underlying SDK primitives.

For instance, in a high-trust EDR scenario, an organization may decide that for certain well-known, high-impact threats, the verdict of their primary EDR is sufficient for immediate action. The SDK enables this by allowing a module developer to publish events with highly specific routing keys. An administrator can then author a workflow that triggers on this specific event type and proceeds directly to mitigation. This policy prioritizes speed of response over correlation, accepting a minimal risk of false positives for maximum containment velocity.

Workflow Example 1: High-Trust EDR Immediate Response

codeYaml

```
Shell
# This workflow trusts a critical-severity alert from a specific EDR (e.g.,
CrowdStrike) implicitly.
name: "High-Trust EDR Immediate Quarantine"
trigger:
   # The event type is very specific, published by a dedicated CrowdStrike
Connector module.
  event_type: "HOST_ALERT_CROWDSTRIKE"
  # The condition ensures this only runs for the most critical alerts.
```

```
  condition: "{{ trigger.data.severity == 'critical' and trigger.data.tactic ==
'Execution' }}"
steps:
  # The workflow has only one step: immediate mitigation.
  - name: "Initiate Immediate Host Quarantine"
    action:
      type: "PUBLISH_EVENT"
      event:
        type: "INITIATE_MITIGATION"
        data:
          targetHost: "{{ trigger.data.sourceIp }}"
          action: "QUARANTINE"
                    justification:  "Automatic  quarantine  due  to  high-trust,
critical-severity CrowdStrike EDR alert."
```

In this scenario, the SDK's role is to provide the reliable, high-speed transport for both the initial alert and the final command. The CoreSystemApi.publishEvent() method is the universal mechanism used by the EDR module and the Workflow Engine to place these messages on the bus, and the standardized HostAlertData and MitigationCommandData payloads ensure the messages are understood at both ends.

Conversely, a more common correlation-driven scenario can be built by leveraging the SDK's asynchronous nature. This policy is designed to reduce false positives by requiring corroborating evidence from multiple, independent security domains. A workflow can use the SDK's subscribeToEvent() method (via the Workflow Engine) to listen for a low-confidence EDR alert. Its primary logic is then a wait_for block that programmatically creates a temporary, secondary subscription for a high-confidence NIDS_ALERT. In this policy, the user is leveraging the SDK's messaging capabilities to define that the NIDS alert is given precedence as the final, required piece of evidence. This demonstrates how a user can compose a more complex trust model where the verdict of a single tool is insufficient.

A third, human-in-the-loop scenario demonstrates the SDK's flexibility in orchestrating complex interactions, including those with human operators. A workflow can be designed for critical infrastructure where the risk of an incorrect automated action is unacceptable. This workflow can perform the entire automated evidence-gathering and correlation process, but its final step is not a mitigation command. Instead, it publishes a custom HUMAN_APPROVAL_REQUIRED event. A developer creating a Dashboard Module would use the SDK to subscribe to this specific event type and present a UI prompt to the operator. The operator's response would cause the Dashboard module to publish a corresponding MANUAL_ACTION_APPROVED event, which a subsequent workflow can consume. This shows how the SDK's simple, universal primitives (publish, subscribe, Event) can be composed by the user to create sophisticated, nuanced, and risk-based automation.

Java

```
// Conceptual snippet from a developer's DashboardModule, built with the SDK.
```

```
import com.nis1.thesis.sdk.*;
import com.nis1.thesis.sdk.payloads.*;

public class DashboardModule implements PluggableModule {
    private CoreSystemApi api;
    private WebSocketServer webSocketServer;

    @Override
    public void initialize(CoreSystemApi api) {
        this.api = api;
         // The developer uses the SDK to subscribe to the custom event defined
in the workflow.
                                  api.subscribeToEvent("HUMAN_APPROVAL_REQUIRED",
this::handleApprovalRequest);
    }

    // Callback to handle the event from the Workflow Engine.
    private void handleApprovalRequest(Event<?> event) {
         // The developer's custom logic to push the request to the browser UI
via WebSockets.
        ApprovalRequestData data = (ApprovalRequestData) event.getData();
        String jsonForUi = convertToJson(data);
        webSocketServer.broadcast(jsonForUi);
    }

     // This method would be called by the WebSocket server when the operator
clicks "Approve" in the UI.
           public   void   onUserApprovalReceived(String   operatorName,   String
workflowInstanceId) {
          // The developer uses the SDK to publish the user's decision back to
the fabric.
         ManualApprovalData approvalData = new ManualApprovalData(operatorName,
"QUARANTINE", workflowInstanceId);
                           Event<ManualApprovalData>   approvalEvent   =   new
Event<>("MANUAL_ACTION_APPROVED", approvalData);
        this.api.publishEvent(approvalEvent);
    }
    // ... other methods ...
}
```

This final example powerfully illustrates the SDK's role as an agnostic facilitator. The SDK itself has no knowledge of "human approval," but its generic Event structure and reliable publish/subscribe mechanism provide the developer with the exact tools needed to build this complex, human-in-the-loop interaction, proving the ultimate flexibility of the design.

4.5.5.3 Designing for Resilience and Fail-Safety

The SDK's design incorporates best practices for building resilient distributed systems, addressing potential failures at both the software and network levels. These mechanisms are not add-ons but are fundamental to the SDK's contracts and API abstractions, ensuring that any framework built upon it has a baseline of robustness.

At the software level, resilience is primarily achieved through the strict contract imposed by the PluggableModule interface. The mandatory initialize() and shutdown() methods provide the necessary hooks for a robust Lifecycle Manager to gracefully manage and, crucially, isolate modules.

The Core System's implementation of this contract is designed to treat each module as a potential point of failure. When loading a new third-party module, it wraps the call to the module's initialize() method in a broad try-catch (Throwable t) block. This design, enabled by the SDK's contract, creates a fault-tolerant loading process. If a module fails to start—due to a bug throwing an Exception, a missing native library causing an Error, or a misconfiguration—the exception is caught and logged, and the faulty module is discarded. This prevents a single misbehaving module from causing a cascading failure that would halt the initialization of the entire framework.

```
Java
// Conceptual snippet from the ModuleLifecycleManager's internal logic.
// This code is NOT part of the SDK, but IMPLEMENTS the SDK's contract.
public class ModuleLifecycleManager {
    private final List<PluggableModule> loadedModules = new ArrayList<>();

    public void loadModulesFromDirectory(File dir) {
        for (File jarFile : findJarFiles(dir)) {
            PluggableModule moduleInstance = null;
            try {
                // Attempt to load and instantiate the module from the JAR.
                moduleInstance = loadModuleFromJar(jarFile);
                if (moduleInstance != null) {
                    // ** THE CRITICAL FAIL-SAFE **
                          // The call to the third-party code is wrapped in a
try-catch block.
                    moduleInstance.initialize(this.coreApi);
                    loadedModules.add(moduleInstance);
                           System.out.println("Successfully loaded module: " +
moduleInstance.getName());
                }
            } catch (Throwable t) {
```

```
                        // Catching 'Throwable' is intentional to handle both
Exceptions and Errors.
                    // If ANY part of the loading or initialization fails for one
module,
                    // the system logs the error and simply moves on to the next
one.
                    System.err.println("Failed to load or initialize module from "
+ jarFile.getName() + ". Error: " + t.getMessage());
                }
            }
        }
    // ... other methods ...
}
```

At the network level, the CoreSystemApi is designed to provide connection resilience by abstracting away the AMQP client. The SDK contract intentionally hides the raw network client from the developer. This allows the internal implementation to manage the AMQPin a robust manner, for instance by initiating an automated, exponential backoff reconnection strategy without propagating raw network exceptions to the modules. If the connection to the RabbitMQ broker is lost, the CoreSystemApi implementation will not immediately fail. Instead, it will attempt to reconnect every 2 seconds, then 4, then 8, and so on, up to a maximum interval. During this outage, calls to publishEvent() can be internally queued in a bounded, in-memory buffer. If the connection is restored, these queued messages are flushed. If the buffer fills or the connection cannot be restored after a configured duration, only then will the API begin to drop messages, logging a critical error. This is a deliberate design choice that allows the module developer to focus on business logic, knowing that the SDK's implementation is responsible for handling transient network failures.

4.5.5.3.1 Handling of Unprecedented Events

A critical aspect of a policy-driven system is defining its behavior in the face of the unknown. The question therefore arises: what happens when an event is published to the Communication Fabric for which no workflow has a matching trigger? For instance, consider a scenario where a newly integrated, third-party database auditing module begins publishing a novel DATABASE_ANOMALY alert, but no administrator has yet authored a .yml file to handle it. The framework is designed to handle this situation gracefully, operating on an implicit "no-match, no-action" principle for its orchestration logic. The event is not lost; it is successfully published to the RabbitMQ exchange as a standardized JSON message, just like any other event.

JSON

```
{
  "id": "f4a1b3c2-db01-44e2-8c1a-9b8d7e6f5c4d",
  "timestamp": "2025-08-22T15:10:05.123Z",
  "type": "alerts.database.anomaly",
  "data": {
```

```
    "sourceDb": "prod-db-01",
    "sourceIp": "10.1.5.25",
    "user": "service-account-backup",
    "anomaly_type": "excessive_row_deletion",
    "details": "User deleted 2,500,000 rows from the 'customers' table.",
    "severity": 9
  }
}
```

However, because the Workflow Engine has not been configured with a workflow that triggers on the "alerts.database.anomaly" routing key, it will not have a corresponding subscription on the message broker. Consequently, the broker finds no binding for this event type from the engine's queue and the message is simply not delivered to it for processing. This behavior is a fundamental and desirable feature of the decoupled, event-driven design. It ensures system stability, as the Workflow Engine is not burdened with unhandled events and continues to operate normally. It also creates an implicit prioritization mechanism, dedicating processing resources only to threats for which an explicit policy exists. Crucially, while the Workflow Engine ignores the event, a safety net is provided by other "promiscuous" modules that subscribe to broader event patterns. The Web App/Dashboard Module, for instance, is designed to provide comprehensive visibility by subscribing to all events using a wildcard topic, a capability enabled directly by the SDK.

```java
// Conceptual snippet from the Dashboard Module's initialize() method.
// This single line of code, using the SDK, is the safety net.
@Override
public void initialize(CoreSystemApi api) {
    // The "#" is the AMQP topic wildcard for "match one or more words".
     // This subscription ensures this module receives a copy of EVERY event on
the fabric.
    api.subscribeToEvent("#", this::forwardEventToDashboard);
    startWebSocketServer();
}
```

This wildcard subscription ensures that even unprecedented events like the DATABASE_ANOMALY alert are immediately captured and made visible. This creates a powerful operational scenario: an unknown threat does not trigger an uncontrolled or default automated response. Instead, it is safely logged for compliance by the Threat Context Store and, as shown, is immediately presented to a human operator on their dashboard. The operator, now aware of this new alert type, is empowered to analyze it manually and, if necessary, author and deploy a new .yml workflow to define how the system should automatically handle this event class in the future. This demonstrates a robust, fail-safe approach to automation, where the system defaults to a safe, observable state in the absence of an explicit policy.

4.5.5.4 Ensuring Network-Level Transparency in an SDN Environment

Given that any framework built with this SDK is intended for an SDN environment, network-level visibility is a primary design concern. The SDK's decision to use JSON as the mandatory serialization format for all Event objects directly addresses this. This deliberate choice over more opaque binary formats ensures that the entire real-time flow of events within the Application Plane is human-readable on the wire. A network administrator using a standard tool like Wireshark can inspect the AMQP traffic flowing to and from the RabbitMQ broker and see the plain-text JSON payloads of every event published via the SDK's publishEvent() method. For example, a Parsed Alert from an EDR module would appear as a legible message on the network:
codeJson

```
None
{
  "id": "e721dc1a-f34a-4c9e-ae82-1827b72e9a1e",
  "timestamp": "2025-07-21T10:35:12.452Z",
  "type": "alerts.host.wazuh",
  "data": {
    "sourceIp": "192.168.1.101",
    "description": "Suspicious file encryption activity",
    "severity": 8
  }
}
```

This transparency, a direct result of the SDK's data model design, is invaluable for debugging module interactions, auditing the behavior of a workflow, and providing a clear, unambiguous record of the system's internal operations directly from the network level.

4.5.6 Proof-of-Concept Module Implementation and Ecosystem Potential

To validate the SDK's design and demonstrate its efficacy in building a cohesive, end-to-end security orchestration system, a reference implementation of the framework was designed around a core set of proof-of-concept modules. Each module is a self-contained application, developed using the SDK, that fulfills a specific architectural role: Collection, Analysis, or Action. This section details the specific open-source technologies chosen for these initial modules, the justification for their selection, and uses practical scenarios, workflow configurations, and code snippets to illustrate how a developer would leverage the SDK to build both these and future modules, thereby demonstrating the SDK's central role in fostering a rich and extensible security ecosystem.

4.5.6.1 Collection Module: Wazuh EDR Connector

For the initial, host-based telemetry source, a Wazuh EDR Connector was designed. The role of this Collection Module is to act as the primary "senses" on the endpoint, providing the initial spark for an investigation. Wazuh was selected as the reference EDR technology for its comprehensive host-level visibility

and its well-documented RESTful API, which provides the exact programmatic interface required for integration. A developer building this module would begin by creating a class that implements the SDK's PluggableModule contract. It would parse the JSON, create an instance of the SDK's standardized HostAlertData class, and populate it with the relevant fields. The final step is to use the CoreSystemApi handle, provided by the SDK during initialization, to publish this normalized data, wrapped in a standard Event object, to the Communication Fabric for consumption by the rest of the system.

```java
// Developer's code within the WazuhConnectorModule, using the SDK
import com.nis1.thesis.sdk.*;
import com.nis1.thesis.sdk.payloads.HostAlertData;
// Other imports for HTTP client, JSON parsing (e.g., Gson), etc.

public class WazuhConnectorModule implements PluggableModule {
    private CoreSystemApi api;

    @Override
    public String getName() { return "Wazuh EDR Connector"; }

    @Override
    public void initialize(CoreSystemApi api) {
        this.api = api;
           // In a real module, a separate thread would continuously poll the
Wazuh API.
        // On receiving a Wazuh JSON alert...
           // WazuhAlert rawAlert = parseWazuhJson(jsonString); // Developer's
custom parsing logic

        // Developer uses the SDK's data models to normalize the alert.
        HostAlertData payload = new HostAlertData(
            "10.0.1.25", // rawAlert.getAgentIp(),
            "5710 - Windows logon success", // rawAlert.getRuleDescription(),
            8 // rawAlert.getRuleLevel()
        );

        // Developer uses the SDK's API to publish the standardized event.
        System.out.println("Publishing normalized Wazuh alert to fabric...");
        this.api.publishEvent(new Event<>("HOST_ALERT_WAZUH", payload));
    }

    @Override
    public void shutdown() { /* Close HTTP client connections */ }
}
```

4.5.6.2 Analysis Module: Suricata NIDS/DPI Module

To provide the high-confidence, network-level evidence required for accurate decision-making, a Suricata NIDS/DPI Module was designed. This module's architectural role is to act as a specialized analysis component that can definitively confirm a threat based on network traffic. Suricata was chosen for its high-performance, multi-threaded architecture and extensive ruleset support. The module's implementation is designed to be a passive listener, tailing Suricata's eve.json log file using the java.nio.WatchService API for efficiency, which avoids costly, continuous file polling. When a new alert is written to the log, the module's code uses a JSON library like Gson to parse the line, normalizes the relevant fields into a standardized NIDS_ALERT Event, and publishes this finding to the fabric using the SDK. This module is the intended consumer of the traffic dynamically redirected by the SDN-enabled SPAN port mirroring.

4.5.6.3 Action Module: OpenDaylight SDN Response Module

To complete the automated response loop, an OpenDaylight SDN Response Module was designed. This Action Module acts as the "hands" of the system, translating logical commands into physical network enforcement. OpenDaylight was selected as the reference SDN Controller for its mature, feature-rich platform and its well-supported RESTCONF Northbound API. A developer building this module would create a class that implements PluggableModule and, in its initialize() method, would use the CoreSystemApi.subscribeToEvent() method to listen for high-level commands from the Workflow Engine, such as INITIATE_MITIGATION. The developer-provided callback function contains the tool-specific logic.

```
Java
// Developer's code within the OpenDaylightResponseModule, using the SDK
import com.nis1.thesis.sdk.*;
import com.nis1.thesis.sdk.enums.MitigationAction;
import com.nis1.thesis.sdk.payloads.MitigationCommandData;
import okhttp3.*; // For making RESTCONF calls

public class OpenDaylightResponseModule implements PluggableModule {
    private final OkHttpClient httpClient = new OkHttpClient();

    @Override
    public String getName() { return "OpenDaylight SDN Response Controller"; }

    @Override
    public void initialize(CoreSystemApi api) {
          // Developer uses the SDK to subscribe to commands from the Workflow
Engine.
                                    api.subscribeToEvent("INITIATE_MITIGATION",
this::handleMitigationCommand);
    }

    // The developer's callback function, triggered by the SDK.
    private void handleMitigationCommand(Event<?> event) {
```

```
                    MitigationCommandData command = (MitigationCommandData)
event.getData();
        if (command.getAction() == MitigationAction.QUARANTINE) {
            // Developer implements their custom, tool-specific logic.
            installQuarantineFlowRule(command.getTargetHost());
        }
    }

     // Developer's private method to handle the specifics of the OpenDaylight
API.
    private void installQuarantineFlowRule(String ipToBlock) {
          // String restconfPayload = buildOdlFlowPayload(ipToBlock); // Custom
logic
        // RequestBody body = RequestBody.create(restconfPayload, ...);
        // Request request = new Request.Builder().url(...).put(body).build();
                System.out.println("[SIMULATED] Sending RESTCONF request to
OpenDaylight to block " + ipToBlock);
    }

    @Override
    public void shutdown() { /* No action needed */ }
}
```

4.5.6.4 Suggested Future Module Development and Ecosystem Potential

The true power of the SDK is demonstrated by the potential for future, third-party module development. A user could orchestrate these new modules with the existing ones to create highly sophisticated, customized defense strategies. Consider a scenario where an organization develops its own proprietary ML Anomaly Detection Module. This module uses the SDK to subscribe to all telemetry, analyzes behavior, and publishes a BEHAVIORAL_ANOMALY event. The security administrator does not want to act on this alert alone, but wants to use it as an early-warning system to trigger a more definitive check. They can author the following .yml workflow to define a new policy that combines their custom ML module with the existing Suricata and OpenDaylight modules, demonstrating the ultimate flexibility of the SDK-enabled ecosystem.

Shell

```
name: "Custom ML Anomaly Correlation with NIDS Confirmation"
version: 1.0
description: >
   This workflow triggers on a high-confidence behavioral anomaly from our
custom ML module.
```

```
   It initiates SPAN port mirroring to gather network evidence. It will only
proceed to
  quarantine if the anomaly is confirmed by a critical Suricata NIDS alert.

trigger:
  # The trigger is the custom event type from the new module.
  event_type: "BEHAVIORAL_ANOMALY"
  condition: "{{ trigger.data.confidenceScore > 0.9 }}"

steps:
  # STEP 1: The workflow uses the existing SDN module's capability.
  - name: "Orchestrate SPAN Port Mirroring for Deep Packet Inspection"
    action:
      type: "PUBLISH_EVENT"
      event:
        type: "INITIATE_TRAFFIC_ANALYSIS"
        data:
          targetHost: "{{ trigger.data.sourceIp }}"

   # STEP 2: The workflow waits for the verdict from a different, existing
module.
  - name: "Wait for Suricata NIDS Confirmation"
    wait_for:
      event_type: "NIDS_ALERT"
      timeout: "5 minutes"
      conditions:
        - "{{ new_event.data.sourceIp == trigger.data.sourceIp }}"
        - "{{ new_event.data.signatureSeverity == '1' }}"

   # STEP 3: The workflow uses the existing Action Module to enforce the final
decision.
  - name: "Initiate Host Quarantine via OpenDaylight"
    action:
      type: "PUBLISH_EVENT"
      event:
        type: "INITIATE_MITIGATION"
        data:
          targetHost: "{{ trigger.data.sourceIp }}"
          action: "QUARANTINE"
            justification: "Quarantine triggered by custom ML anomaly, confirmed
by Suricata NIDS."
```

This scenario is the ultimate validation of the SDK's design. The user was able to create a sophisticated detection and response strategy that gives precedence to the correlation of a brand-new, custom-built tool and a pre-existing tool. They did this without modifying a single line of code in any of the modules, but simply by

defining their desired policy in a declarative workflow. This demonstrates that the SDK successfully enables a truly modular, flexible, and future-proof security orchestration ecosystem. Other high-impact modules that could be readily developed using this same SDK pattern include a Honeypot Interaction Module for zero-day threat detection, a Threat Intelligence (CTI) Enrichment Module to add context from feeds like VirusTotal, and an ITSM Integration Module to automatically create incident tickets in platforms like Jira or ServiceNow.

4.5.7 Architectural Boundaries and Inappropriate Module Types

While the SDK is designed for maximum flexibility within its intended domain, its architectural principles—asynchronous communication, out-of-band operation, and high-level abstraction—create intentional boundaries. These boundaries define the scope of the framework and render certain classes of modules difficult or architecturally inappropriate to develop. Understanding these limitations is key to understanding the SDK's specialized purpose as a security orchestration tool, not a universal networking platform. These inappropriate module types can be categorized by the architectural layer they would need to inhabit or the communication pattern they require.

The most significant category of modules that are unsuited for this SDK are those that must operate in-line within the Data Plane and require synchronous, per-packet decision-making. The quintessential example is a Network Intrusion Prevention System (IPS). Consider a specific scenario where an attacker attempts to use the EternalBlue exploit against a vulnerable server. The IPS's job is to inspect the initial SMB handshake packets, recognize the malicious exploit signature in real-time, and drop those specific packets before they can compromise the server. To achieve this, the IPS must operate at line-rate with microsecond-level latency. The SDK, however, enables an asynchronous, out-of-band workflow that is inherently too slow for this task. The round-trip time for an event—from an SDN switch detecting a novel packet, sending an event to the Communication Fabric, the message being routed, a module consuming and analyzing it, a verdict being published, a command being consumed by the SDN Response Module, an API call being made to the controller, and a final flow rule being installed—would take milliseconds at best. By the time the "drop" command arrived, the thousands of packets constituting the exploit would have already passed and the server would be compromised. Therefore, a developer cannot use the SDK to build an IPS. Instead, the SDK enables a module to brilliantly integrate with an existing, dedicated IPS; the module would subscribe to the IPS's own alert stream and use that high-confidence alert as a trigger for a broader orchestration workflow, such as quarantining other hosts the attacker may have touched.

A second class of modules that would stray from the SDK's nature are those that require implementing low-level Control Plane logic. The SDK is designed to be a client of the SDN Controller's Northbound API, which exposes high-level, abstract functions like "install a flow to block this IP." It is not designed to interact with the Southbound API (e.g., OpenFlow) or to manage the core logic of the controller itself. Imagine a developer wants to create a sophisticated dynamic network load balancer that distributes traffic across a pool of web servers based on real-time server health and application latency. This would require the module to constantly poll servers for health, calculate optimal traffic paths, and frequently manipulate fine-grained flow rules across multiple switches with precise timing. This functionality is a core feature of the Control Plane. A developer attempting to implement this via the Northbound API would find it clumsy and inefficient, as they would be sending high-level, abstract commands to achieve a low-level, high-frequency task, effectively fighting the controller's own internal logic. The SDK is therefore unsuitable

for creating modules that define core network behavior; it is designed for modules that react to security events and command high-level changes to that behavior.

Finally, the SDK's asynchronous, "fire-and-forget" communication model makes it a poor fit for modules that depend heavily on synchronous, blocking request/response interactions. While a request/response pattern can be simulated with a wait_for step in a workflow, it is inefficient and non-deterministic for tasks that require an immediate, guaranteed answer. Consider a developer building a File Access Policy Decision Point (PDP) module for a critical file server. In this scenario, a file system filter driver (acting as part of an Action Module) might intercept a delete system call for a critical file. To be effective, this driver would need to block the operation, publish a synchronous request to the PDP module asking, "Is user 'jsmith' from process 'svchost.exe' allowed to delete 'quarterly_earnings.xlsx'?", and then wait for an immediate "Allow" or "Deny" response before letting the system call proceed. The inherent, variable latency and non-guaranteed response time of an AMQP-based event bus makes it ill-suited for this kind of synchronous authorization check, as it could freeze the user's application or the entire server while waiting for a response. The SDK's design would push the developer towards a different, asynchronous pattern: instead of a synchronous check, the system would allow the operation to happen and rely on a separate EDR alert to be generated after the fact if the action was malicious. This asynchronous model is ideal for detection and response but is not designed to be a synchronous policy enforcement point. These limitations are not flaws, but rather the necessary architectural trade-offs for achieving a highly flexible, scalable, and resilient security orchestration ecosystem.