POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Extending Security Orchestration, Automation and Response to the Cloud

Supervisors:

Prof. Cataldo Basile

Dott. Francesco Settanni

Candidate:

Davide Abate

Academic Year 2025/2026
Torino

Abstract

Public cloud platforms host the bulk of modern enterprise workloads. As organisations continue to migrate identity, compute, storage and data services from on-premise data centres to managed cloud offerings, the security incidents that reach a Security Operations Centre (SOC) have followed them there: identity-centric attacks, control-plane abuse and supply-chain compromises now dominate the alerts that an analyst must triage in a cloud-first deployment. Analysts face high alert volumes and short response budgets, while the SOAR products that have traditionally driven their automation rely on hand-authored, vendor-specific playbooks that scale poorly as the cloud surface evolves. Two parallel developments offer a way forward: the OASIS CACAO 2.0 specification, which defines a vendor-neutral playbook format with precise execution semantics, and the maturation of the MITRE ATT&CK Cloud and Identity Provider matrices as a common taxonomy for the threats SOC analysts actually face in cloud-native deployments.

This thesis builds on a CACAO 2.0 playbook-driven remediator developed within the TORSEC research group at Politecnico di Torino. The work reported here begins with a systematic study of the SOAR product landscape, of the new threat models that have emerged with the migration of enterprise workloads to public cloud, and of the Microsoft Azure subsystems that an automated remediator must address: Microsoft Sentinel as the SIEM, including its analytics-rule model, incident schema, entity catalogue and automation hooks; Microsoft Graph for identity operations on Entra ID; Azure Resource Manager for compute and network actions; and the Log Analytics REST API for Sentinel queries. From that analysis a typed library of Azure capabilities is derived, organised by family (identity, network, compute, Sentinel) and exposed to the engine as a structured action catalogue in which each operation declares its preconditions, its permission scope and its risk level.

On top of this capability layer the thesis introduces a three-mode execution engine. The first mode preserves the base functionality of the tool. The second mode generates CACAO playbooks dynamically from a MITRE ATT&CK tactic-technique-action mapping built specifically for Azure, so that incidents whose technique combinations were not anticipated by the static catalogue still receive a coherent response. The third mode delegates plan generation to a Large Language Model under a strict safety policy: every proposed step is validated against the closed action catalogue, destructive actions are gated by explicit operator approval, and a read-only enrichment agent based on the ReAct pattern (a loop that interleaves model reasoning with tool calls against Sentinel, Microsoft Graph, Log Analytics and threat-intelligence feeds) supplies the planner with grounded contextual

evidence. Across all three modes, the underlying CACAO 2.0 execution semantics remain unchanged.

The extended remediator has been validated on a corpus of Azure threat scenarios that exercises the MITRE ATT&CK Cloud and Identity Provider matrices and includes identity compromise, lateral movement, malicious URL clicks, key-vault lockdown and multi-stage replays of publicly documented intrusion campaigns, namely Pikabot (C0036), Operation Spalax (C0005), SolarWinds (C0024) and Lapsus$ (DEV-0537).

Table of Contents

Abstract II

List of Figures IX

List of Tables XII

Acronyms XV

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives and Contributions . . . 2
- 1.3 Thesis Outline . . . 5

2 Background 6
- 2.1 Incident Response and the SOC . . . 6
- 2.2 SOC Evolution . . . 7
  - 2.2.1 SOC 1.0 . . . 7
  - 2.2.2 SOC 2.0 . . . 7
  - 2.2.3 SOC 3.0 . . . 7
- 2.3 SOAR . . . 8
- 2.4 Threat Intelligence and Playbooks . . . 8
- 2.5 CACAO 2.0 . . . 9
  - 2.5.1 Workflow Steps . . . 9
  - 2.5.2 Agents and Targets . . . 9
  - 2.5.3 Data Markings and Signatures . . . 10
- 2.6 MITRE ATT&CK . . . 10
- 2.7 The Azure Security Stack . . . 11
  - 2.7.1 Microsoft Sentinel . . . 11
  - 2.7.2 Microsoft Entra and the Graph API . . . 11
  - 2.7.3 Azure Resource Manager . . . 12

2.8 LLMs and Agentic Patterns . . . 12
2.8.1 Tool Use and ReAct . . . 12
2.8.2 Hallucination and Prompt Injection . . . 13

3 Related Works 14
3.1 Inherited Components . . . 14
3.2 Other CACAO-Compliant Tools . . . 15
3.2.1 SOARCA . . . 15
3.2.2 CACAO Roaster . . . 16
3.3 Cloud-Native SIEM Remediation . . . 16
3.4 LLM Agents in Cybersecurity . . . 17
3.5 Positioning . . . 17

4 Problem Statement 19
4.1 The Inherited Tool: Architecture, Capabilities and Limits . . . 19
4.2 Identified Gap and Research Questions . . . 21

5 Design 22
5.1 Design Objectives . . . 22
5.2 Architecture . . . 23
5.3 Sentinel Ingestion . . . 24
5.4 Azure Control-Plane Capabilities . . . 24
5.5 Three-Mode Engine . . . 26
5.6 MITRE Mapping . . . 27
5.7 AI Planning and Approval . . . 28
5.8 Enrichment Agent . . . 29
5.9 Web API and UI . . . 30

6 Implementation 31
6.1 Codebase Layout . . . 31
6.2 Router and Mode Selection . . . 31
6.3 Sentinel Ingestion . . . 32
6.4 Dynamic Playbook Generation . . . 33
6.5 AI Planner Subsystem . . . 34
6.6 Enrichment Agent . . . 35
6.7 Web Server and Job Lifecycle . . . 36
6.8 A Worked End-to-End Example . . . 37
6.8.1 Stage 1: Incident Arrives . . . 37
6.8.2 Stage 2: Normalised Alert and Planning Input . . . 38
6.8.3 Stage 3: Enrichment Snapshot . . . 39

6.8.4 Stage 4: LLM-Produced Plan . . . 40
6.8.5 Stage 5: Validation and Approval . . . 40
6.8.6 Stage 6: Compiled CACAO Playbook . . . 43
6.8.7 Stage 7: Execution Trace and CACAO Report . . . 43
6.8.8 What the Example Demonstrates . . . 44

7 Validation and Testing 46
7.1 Validation Objectives . . . 46
7.2 Test Environment . . . 47
7.3 Threat Scenarios . . . 48
7.3.1 Single-Alert Scenarios . . . 49
7.3.2 Multi-Stage Campaign Replays . . . 49
7.4 Threat Model . . . 51
7.5 Per-Mode Comparison . . . 54
7.6 Selected Campaign Walkthroughs . . . 56
7.6.1 Operation Spalax (C0005) . . . 56
7.6.2 Pikabot phishing (C0036) . . . 57
7.6.3 SolarWinds (C0024) . . . 57
7.6.4 Lapsus$ (DEV-0537) . . . 59
7.7 AI Planner Plan Quality . . . 59
7.8 Enrichment Agent Evaluation . . . 65
7.9 Discussion and Limitations . . . 65

8 Conclusion and Future Work 68
8.1 Conclusion . . . 68
8.2 Future Work . . . 69

Bibliography 71

A User Manual 74
A.1 Prerequisites . . . 74
A.2 First-Time Setup . . . 74
A.3 Starting the Web Interface . . . 75
A.4 Submitting an Incident . . . 76
A.5 Choosing an Execution Mode . . . 77
A.6 Approving Destructive Actions . . . 77
A.7 Reading the CACAO Report . . . 78
A.8 Troubleshooting . . . 78

B Developer Manual 80

B.1 Setting Up a Development Environment 80
B.2 Project Layout Cheatsheet 81
B.3 Adding a New Azure Action 81
B.4 Mapping a New MITRE Technique 82
B.5 Adding an Enrichment Tool 83
B.6 Supporting a New Sentinel Alert Family 83
B.7 Adding a New Execution Mode 84
B.8 Testing Conventions 84
B.9 Logging and Observability 85
B.10 Releasing 85

List of Figures

1.1 The three execution modes introduced by this thesis. The AI-assisted mode includes a human approval gate; the deterministic legacy and tactic-driven modes execute without one. 4

2.1 The MITRE ATT&CK taxonomy with a concrete example drawn from the Cloud matrix. 10

2.2 The three Azure subsystems relevant to this thesis. Sentinel is the entry point for alerts produced by analytics rules and by external EDR/XDR sources; Microsoft Graph and Azure Resource Manager are the control surfaces used by the remediation actions. 11

2.3 The ReAct loop used by the enrichment agent. At each turn the LLM emits a thought and an action; the runtime executes the action and returns the observation; the loop ends when the LLM emits a final answer instead of an action. 12

5.1 High-level architecture. The dispatcher selects one of three remediation front-ends; all three converge on the shared executable graph and the CACAO serialiser inherited from earlier TORSEC work. 23

5.2 AI-assisted execution flow. The planner emits a structured remediation plan; the validator rejects anything outside the closed catalogue; destructive actions wait for human approval before joining the autonomously executed half in the final CACAO report. 29

6.1 Compiled CACAO graph for the Pikabot scenario. The two blue circles with a white "+" are synchronisation barriers (AND-joins): the executor waits for every branch in the parallel block above the node to complete before any branch in the block below can start. These joins are implicit in the CACAO 2.0 `Parallel_step` construct (which has fork-on-entry and join-on-exit semantics) and are drawn explicitly here only for visual clarity. Block 1 (green) contains the three non-destructive containment steps, all with disjoint non-structural parameters. Block 2 hosts the two destructive actions (orange, gated by operator approval) plus the live-status incident comment, which parallelises with them because `comment_message` is structural and excluded from the disjoint-check. 44

7.1 Pikabot (C0036) replay: a single Microsoft Sentinel Analytics Rule monitors the workspace for indicators of the campaign and, when they match, generates a single incident that carries inside itself the full kill-chain context — three MITRE ATT&CK tactics (*Initial Access*, *Execution*, *Persistence*), two techniques (T1059.007, T1574), and eleven typed entities across six kinds (accounts, IPs, URLs, hosts, processes, file hashes). The engine ingests the incident as a unit and emits a CACAO 2.0 response playbook whose multi-stage internal structure covers all the tactics observed. The *multi-stage* nature of the test is therefore in the content of the single incident, not in a temporal sequence of separate incidents arriving at different times. 52

7.2 ATT&CK technique coverage of the four multi-stage campaign replays. Columns are MITRE tactics ordered along the kill chain (restricted to the tactics actually exercised by the campaigns); cells are the techniques exercised by at least one campaign, coloured by the campaign that triggers them. The *T1098 Account Manipulation* cell is highlighted as multi-campaign because it is exercised by both Spalax and Lapsus$. 55

7.3 Pre/post Azure tenant state for the Pikabot (C0036) campaign. The left column captures the four Azure resources affected by the response in their baseline state. The engine executes six actions on the incident: `revoke_user_sessions` and `disable_user_account` on the impacted Entra ID account, `isolate_vm_network` on the impacted VM via the attached NSG, `add_ip_to_named_location_blacklist` to populate the Conditional Access named location, `upsert_sentinel_watchlist_ioc` to enrich the watchlist with the phishing URL, and `add_sentinel_incident_comment` to annotate the Sentinel incident with a workflow summary. The right column shows the resulting state of each resource, with the changed fields highlighted. 58

7.4 Approval view of the web UI for a job parked in `pending_approval`. The top action bar exposes three operator choices (*Approva ed Esegui* – approve and execute, *Rifiuta e Rigenera* – reject and trigger a replan, *Interrompi* – cancel the job), together with a free-text field whose content is appended to the prompt of the next replan call. The status strip below confirms that the request is in the wait state (*ESITO: WAIT*) and is routed under the AI-assisted mode (*MOTORE: ai_plan_v3*). The *Draft Plan* tab shows the execution-plan summary returned by the planner and validated against the closed catalogue (*piano AI validato* marker, top-right of the panel): plan name, plan identifier, planner backend (`openai`), risk level (`high`), confidence (0.78), the auto-generated approval reason that enumerates the three disruptive actions the plan proposes, the free-form reasoning paragraph that grounds the plan in the incident's MITRE technique (T1566.001), and the single planned action with its per-action *why* field and parameter binding (`cloud_file_url: da contesto`, i.e. resolved from the global scope at execution time). Nothing in this plan reaches the Azure tenant until the operator clicks *Approva ed Esegui*. 61

7.5 High-level topology of the CACAO 2.0 report produced by an AI-assisted run on the Operation Spalax (C0005) scenario. The engine forked into four parallel branches (removing the malicious SharePoint payload, blocking the phishing domains in the Azure Firewall, terminating the in-guest malicious process, and watchlisting the URL IOCs), synchronised at the AND-join, and then walked four sequential `upsert_sentinel_watchlist_ioc` steps that tracked the broader campaign IOCs (dynamic-DNS domains, file hashes, source IP, lure domains) before terminating. Every action recorded *success*, and the per-step `x_execution_result` extension preserves the wall-clock duration of each invocation in the report. The auditability claim of the thesis is that an analyst can reconstruct, after the fact, exactly what the engine attempted, in what order, and what the outcome of every step was, without ever consulting the engine's runtime logs. 62

List of Tables

3.1 Positioning of this work against the closest comparators. 18

5.1 Capability matrix. Each family is backed by a distinct Azure surface and requires a distinct permission for the application's service principal. The catalogue in `ai_planner/plan_schema.py` is the machine-readable counterpart consumed by the AI planner. 26

5.2 Illustrative excerpt from the MITRE tactic-technique-action mapping. The full table covers the techniques most frequently observed in the Azure threat scenarios used for validation. 28

6.1 Top-level layout of the implementation. Each subsystem identified in Chapter 5 maps to a single folder, with one entry point per execution mode at the package root. 32

6.2 HTTP API exposed by `source/web_app.py`. The approval and replan endpoints are the live coupling between the analyst and the AI-assisted execution mode. 37

6.3 Per-step validator and approval-policy verdicts for the Pikabot plan. The two destructive actions are parked for human review; the remaining four run autonomously. 40

7.1 Azure resources provisioned in the dedicated tenant used for validation. Each row pairs a resource with the role it plays in the test corpus. Subscription, tenant and client GUIDs are omitted from the table for security reasons; the same information, including the full UPN domain and the service-principal credentials, is captured verbatim in the project's `config.local.json` for reproducibility. 48

7.2 Coverage of the validation corpus across the MITRE ATT&CK Cloud and Identity Provider matrices. Each row pairs a tactic with the techniques it covers and the concrete scenarios that exercise it. Tactics not addressable from inside the tenant (for example, *Reconnaissance* or *Resource Development*) are not represented; the engine is not designed to defend against them. 54

7.3 Per-scenario per-mode validation outcomes. Each cell records the result of executing the scenario under the corresponding execution mode. All sixteen scenarios produce a valid CACAO report under all three modes. 56

7.4 Cross-run consistency of the AI-assisted planner across three independent invocations on the same Operation Spalax incident. The two core containment primitives (process termination on the impacted VM, removal of the SharePoint-hosted payload) are reproduced verbatim across the three runs, with identical parameter bindings; the validator-level and approval-gate properties are also reproduced verbatim. The variability is concentrated in the indicator layer: the number of associated domains the planner promotes to active firewall blocks versus the number it leaves as passive watchlist entries fluctuates between runs, but every dominant indicator (the four campaign domains, the source IP, the two URLs, the file hash) is handled in some way by every run. All three runs land at a valid CACAO 2.0 report executed to completion. 64

A.1 Environment variables an operator normally configures. The full list is documented in Chapter 6. 76

B.1 Where to land a change. Each row maps an extension intent to the file the developer normally edits. 81

Acronyms

SOC
: Security Operations Centre

SOAR
: Security Orchestration, Automation and Response

CACAO
: Collaborative Automated Course of Action Operations

LLM
: Large Language Model

SIEM
: Security Information and Event Management

AI
: Artificial Intelligence

IR
: Incident Response

XDR
: Extended Detection and Response

CTI
: Cyber Threat Intelligence

STIX
: Structured Threat Information Expression

TAXII

Trusted Automated Exchange of Intelligence Information

TLP

Traffic Light Protocol

KQL

Kusto Query Language

EDR

Endpoint Detection and Response

API

Application Programming Interface

REST

Representational State Transfer

ARM

Azure Resource Manager

DSL

Domain-Specific Language

MQTT

Message Queuing Telemetry Transport

JSON

JavaScript Object Notation

Chapter 1

Introduction

1.1 Motivation

A large share of enterprise workloads today runs in public cloud platforms, and attackers have followed those workloads there. The MITRE ATT&CK knowledge base now maintains dedicated matrices for Cloud and Identity Provider environments, and its tactics and techniques have become the common vocabulary used by analysts to describe adversary behaviour during incident response [1]. For the people who actually triage alerts in a Security Operations Centre (SOC), the practical consequence is that the alerts that reach them every day are increasingly cloud-shaped: short-lived virtual machines, identity tokens exfiltrated from a single endpoint, conditional-access policies bypassed through a forgotten service principal. The classic network perimeter is no longer the right mental model.

The workload that all of this puts on a SOC is significant. Systematic surveys of SOC teams report high alert volumes, persistent staffing shortages, and response times that often fail to keep up with automated attack chains [2]. The industry's main answer has been the family of products grouped under the label Security Orchestration, Automation and Response (SOAR), which encode incident-response procedures as machine-executable playbooks and have received growing academic attention [3]. SOAR works well with the stable parts of incident response. Where it tends to struggle is in the parts that change. The playbooks themselves are still mostly hand-authored, vendor-specific, and tightly coupled to a fixed alert taxonomy. When a new threat shape appears, or when the cloud platform underneath the SOC gains a new feature, the playbook library has to be rewritten. In practice, few teams have the bandwidth to do that often.

Two recent developments make the picture less static than it sounds. The first is the OASIS *Collaborative Automated Course of Action Operations* (CACAO) specification, now at version 2.0, which provides a vendor-neutral JSON schema for security playbooks with explicit support for sequential, conditional, and parallel workflow constructs [4]. Tools that produce or consume CACAO playbooks can interoperate across vendor boundaries and publish their remediation logic in a form that is reviewable after the fact. The second development is the wave of Large Language Models (LLMs) that are now capable enough

to reason over structured alert data, choose from a catalogue of permitted actions, and propose a remediation plan that a human analyst can review. LLM-assisted planning sits at the centre of what part of the industry has started calling SOC 3.0, in which generative models complement the structured automation built during SOC 2.0 rather than replacing it [5].

Combining CACAO as the interoperability standard and an LLM as the planning layer is an appealing idea on paper. So far, few tools actually try to do it. CACAO-compliant remediators that integrate with public-cloud Security Information and Event Management (SIEM) products are not common, and CACAO-compliant tools that also embed an LLM planner gated by a formal approval policy are rarer still. Closing this gap is the practical motivation of the work presented in this thesis.

Earlier work within the TORSEC research group at Politecnico di Torino had produced a playbook-driven remediator that demonstrated CACAO-based execution in selected environments [6]. The tool produced by that work is a solid base. It parses a CACAO playbook, builds an executable graph of steps with support for conditional branches and parallel execution, performs the remediation against the target environment, and re-serialises the executed instance back into a CACAO report. Its action library was tied to the operational environment for which it was originally written, and did not yet address public-cloud SIEM products, MITRE-driven dynamic playbook generation, or an AI-assisted planning layer. The work presented here extends the existing tool along those three axes, leaving its CACAO-compliant core untouched.

1.2 Objectives and Contributions

The objective of this thesis is to turn the inherited remediator into a system that can receive an alert from Microsoft Sentinel, select or build an appropriate CACAO playbook for it, and execute the playbook against the Azure cloud control plane. The system offers three modes of operation. The first is a deterministic, file-based mode inherited from the previous tool and adapted to the new alert format. The second is a tactic-driven mode that consumes a MITRE tactic-technique-action mapping and generates a playbook on the fly for the techniques present in the alert. The third is an AI-assisted mode that delegates plan generation to a Large Language Model, validates the result against a schema, and waits for a human operator to approve the plan before any action is executed.

Four design principles drove the work. Backward compatibility came first: the execution engine, the playbook class hierarchy, the parallel and conditional step machinery, and the CACAO serialiser inherited from the previous tool are all reused as they are. Traceability came next: every remediation step the system takes can be traced back to the MITRE tactic or technique that motivated it, and that link is preserved in the CACAO report. Auditability follows naturally from CACAO itself, because every execution, whether deterministic or AI-driven, ends with a CACAO 2.0 artefact that can be archived and reviewed. The last principle is operator control. Whenever the AI planner is involved, the system pauses for explicit human approval before performing any action that could disrupt a production workload, and offers a structured replan path when the proposed

plan is rejected.

The thesis's technical contributions can be grouped into four headings.

The first contribution concerns the new ingestion and action layers. The alert ingestion module has been extended with a normalisation pipeline for Azure Sentinel incidents and their entity objects. The pipeline produces a uniform internal representation of identity, network, compute and SIEM context, consumed by all three execution modes. The Azure action library implemented for this work covers Microsoft Entra identity operations (account disablement, session revocation, password reset, service-principal containment, role-assignment cleanup), network controls (Network Security Group rules, Azure Firewall IP and domain blocks, named-location updates for Conditional Access), virtual-machine lifecycle actions, Sentinel watchlist and incident-comment operations, and Key Vault lockdown.

The second contribution turns to MITRE-driven dynamic remediation. A MITRE tactic-technique-action mapping for Azure is defined as a standalone artefact of the system, in a form that exposes the coverage of the tool and that can be reviewed or extended without touching code. On top of that mapping, the tactic-driven execution engine assembles a deduplicated CACAO playbook tailored to the techniques observed in a given alert, removing the need to hand-author a playbook for every possible combination of techniques.

The third contribution covers AI-assisted planning with human oversight. The AI planning engine delegates plan generation to a configurable backend: a deterministic heuristic that needs no external service, an OpenAI-compatible chat model, or an Azure OpenAI deployment. The generated plan is validated against a schema. An approval policy then flags disruptive actions, high-severity recommendations and risky action combinations before execution. A read-only enrichment agent sits alongside the planner and gathers additional evidence from Sentinel, Microsoft Graph, Log Analytics and threat-intelligence feeds through a small allow-listed set of tools. The enrichment agent offers four interchangeable backends (heuristic, OpenAI single-shot, OpenAI tool-using agent loop, and the external CLAW binary), so that the enrichment strategy can be swapped or rolled back independently of the planner itself.

The last contribution is the runtime layer that ties everything together. A router selects among the three execution engines based on a single environment variable, allowing a deployment to be reconfigured without code changes. A Flask web server exposes a REST API and a Server-Sent Events stream, and a single-page user interface lets an operator submit alerts, observe execution in real time, approve or reject AI-generated drafts, request a replan with feedback, cancel an in-flight job, and download the final CACAO report.

The resulting system has been validated on a corpus of Azure threat scenarios that includes identity compromise, lateral movement, malicious URL clicks, key-vault lockdown, and multi-stage campaign replays inspired by publicly reported intrusions such as SolarWinds, Pikabot, Spalax and Lapsus$. Figure 1.1 sketches the resulting pipeline at a high level.

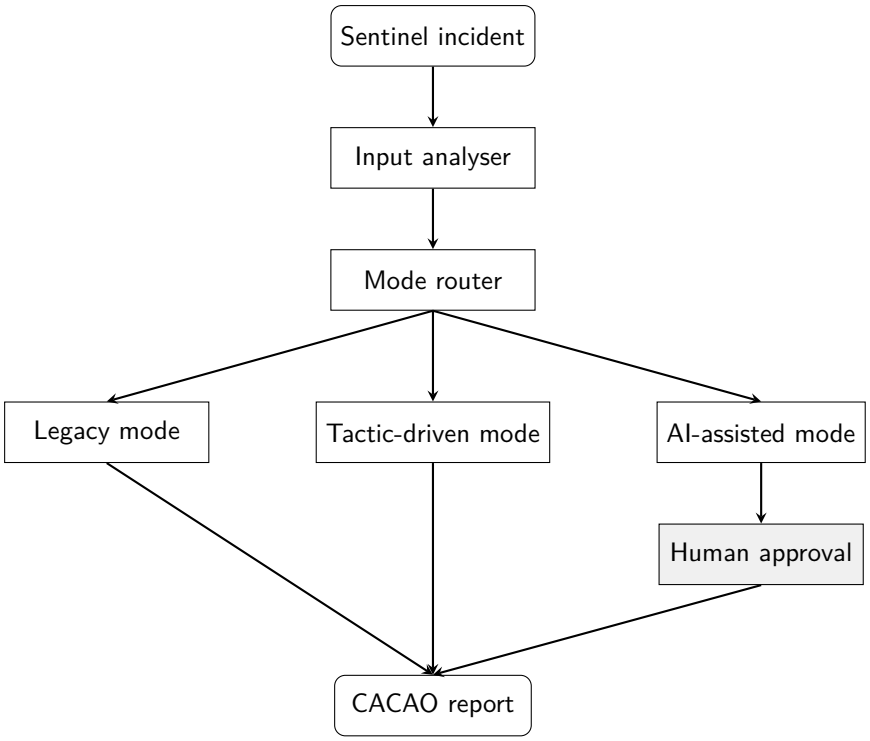

Figure 1.1: The three execution modes introduced by this thesis. The AI-assisted mode includes a human approval gate; the deterministic legacy and tactic-driven modes execute without one.

1.3 Thesis Outline

The rest of the document is organised as follows.

Chapter 2 sets the technical and conceptual scene. It walks through the modern SOC workflow, the evolution from SOC 1.0 to SOC 3.0, the SOAR paradigm, the CACAO 2.0 standard, the MITRE ATT&CK framework with a focus on its Cloud and Identity tactics, the parts of the Azure security ecosystem relevant to this work, and the application of Large Language Models to incident-response automation.

Chapter 3 surveys related work. It identifies the components of the inherited tool that are reused in this thesis without modification, then looks at CACAO-compliant reference tools such as SOARCA and CACAO Roaster, surveys SIEM-integrated remediation approaches in public-cloud environments, discusses recent LLM-based agents and copilots in cybersecurity, and positions the present work with respect to all of these.

Chapter 4 formulates the problem. It describes the architecture and capabilities of the inherited tool, characterises its limits, and identifies the gap that the thesis closes together with the research questions it answers.

Chapter 5 presents the design of the solution. It states the design objectives, explains how the new Azure layer fits on top of the inherited engine, and walks through the alert ingestion design, the rationale behind the three-mode engine architecture, the MITRE tactic-technique-action mapping schema, the dynamic CACAO playbook generation procedure, the AI planning workflow with its approval policy and its replan loop, the read-only enrichment agent and its backends, and the Web API and operator interface.

Chapter 6 reports the implementation. It describes the components reused from the inherited tool, the input analyser for Sentinel alerts, the three execution engines, the four enrichment backends, the Azure action library, the CACAO loader and serialiser adaptations, the asynchronous job lifecycle in the Web application, and the runtime router with its environment-driven configuration.

Chapter 7 reports the validation and testing performed. It describes the evaluation methodology and the test alert corpus, walks through the functional validation of each engine on identity, network, compute and SIEM scenarios, presents the multi-stage campaign replays, compares the behaviour of the three engines on the same incidents, reports the MITRE coverage analysis, discusses the quality of the AI planner and of the approval policy, and lists the threats to validity together with the known limitations of the system.

Chapter 8 concludes the thesis. It summarises what has been achieved, reflects on the lessons learned during the extension, and outlines the lines of future work the project leaves open.

Appendix A is a user manual covering deployment, Azure credential configuration, Web UI and CLI usage, alert submission, CACAO report retrieval, and the approval, replan and cancel workflows of the AI-assisted engine. Appendix B is a developer manual that documents how to extend the Azure action library, update the MITRE tactic-technique-action mapping, add new enrichment tools and backends, register a new routing mode, and accommodate new Sentinel incident schemas.

Chapter 2

Background

This chapter provides the background needed to read the rest of the thesis. The first three sections cover the incident response process, the SOC that runs it, and the SOAR products that automate it. The next two cover cyber threat intelligence and the CACAO standard. The last three describe the MITRE ATT&CK taxonomy, the Azure security stack, and the LLM techniques used by the AI-assisted execution mode.

2.1 Incident Response and the SOC

Incident response (IR) is the structured process by which an organisation prepares for, detects, contains and learns from cybersecurity incidents. NIST formalised the canonical four-phase model in Special Publication 800-61 Revision 2: Preparation; Detection and Analysis; Containment, Eradication and Recovery; and Post-Incident Activity [7]. The European Union Agency for Cybersecurity (ENISA) describes a similar process at a finer grain, with separate steps for incident report, registration, triage, resolution, closure and post-analysis [8]. The two views do not contradict each other. ENISA expands the operational substeps that NIST groups under *Detection and Analysis*, while NIST gives more space to the organisational preparation that must occur before any alert is raised.

In practice, much of the IR process is run inside a Security Operations Centre (SOC). A modern SOC is normally described as a combination of people, processes, and technology, organised around continuous monitoring, alert triage, and incident response. The systematic study of SOCs by Vielberth et al. [2] reports that this combination is hard to operate at scale. The same study notes that the academic literature has tended to examine the human and technological sides in isolation, often neglecting the processes that bind them together. Alert overload, staff turnover and skill gaps recur as the most cited open challenges.

Most large SOCs are organised in tiers. Tier-1 analysts handle initial triage, dismiss false positives and escalate real events. Tier-2 analysts investigate the events that survive triage and drive containment. Tier-3 analysts and threat hunters do the rest: proactive investigation, new detection development, and the authoring of the playbooks that the lower tiers later execute. The remediator described here sits along the boundary between

Tier-1 and Tier-2. The aim is to automate, or at least to propose, a defensible containment plan as soon as an alert arrives, so that the analyst's time goes into deciding whether to approve the plan rather than into building it from scratch.

2.2 SOC Evolution

The vocabulary of "SOC generations" is a useful shorthand. It is, admittedly, an industry label rather than an academic one. Still, it captures real shifts in how SOCs have been organised over the last fifteen years.

2.2.1 SOC 1.0

SOC 1.0 refers to the first wave of dedicated security operations centres. They were organised around a SIEM that collected logs and around teams of analysts who manually correlated them. The model worked when alert volumes were modest. It scaled poorly. Every new detection had to be authored as a fresh correlation rule, and every alert had to be touched by a human.

2.2.2 SOC 2.0

SOC 2.0 introduced two structural changes. The first was Security Orchestration, Automation and Response (SOAR), which moved a share of the response logic from analysts into machine-executable playbooks. The second was Extended Detection and Response (XDR), which moved part of the correlation logic from the SIEM into the telemetry sources themselves, so that the SOC could ingest higher-quality, pre-aggregated alerts. Together, the two did not eliminate the analyst, but they did cut the per-alert workload by a meaningful factor.

2.2.3 SOC 3.0

SOC 3.0 is the label that part of the industry now uses for the integration of generative AI into the operations centre, with Microsoft Security Copilot among the first publicly announced products in this category [5]. The intuition is straightforward. A Large Language Model with access to the right tools can absorb part of the triage and investigation workload that today falls on Tier-1 and Tier-2 analysts, while humans stay in charge of approval and judgement.

A survey of more than two hundred research artefacts on LLMs in cybersecurity identifies incident response and threat intelligence as the two most active sub-areas, with applications that range from log analysis to alert prioritisation, incident summarisation and natural-language explanation of detection rules [9]. Arguably, SOC 3.0 is best read as additive rather than replacing. SOAR playbooks remain the place where deterministic, audited response lives. The AI layer is a reasoning and planning surface that sits in front of them.

The work presented in this thesis straddles SOC 2.0 and SOC 3.0. The deterministic, file-based execution mode is recognisably SOC 2.0 in flavour. The tactic-driven mode adds dynamic playbook generation without leaving the deterministic envelope. The AI-assisted mode brings in the SOC 3.0 reasoning layer, but does so behind a hard approval gate, so that the deterministic envelope is preserved.

2.3 SOAR

Section 1.1 already introduced SOAR as the industry's main answer to alert overload. The acronym hides three different concerns. Orchestration is about coordinating the heterogeneous tools that a SOC already owns, from EDR agents to firewalls to ticketing systems, through a single control plane. Automation is about executing routine response steps without human intervention, normally by invoking the APIs of those tools. Response is about doing both of the above in the context of a concrete incident, so that the SOC produces a coherent action set rather than a stream of disconnected commands.

The systematic review by Islam et al. [3] describes the practical anatomy of a SOAR system. Internally, a SOAR is usually built around an orchestration engine that interprets a playbook, a library of pre-built integrations with security and operational tools, and a case management module that records the actions taken during an incident. On top of that internal core, commercial products typically expose three additional layers: a catalogue of authored playbooks, often organised by alert type; an alert ingestion pipeline that maps SIEM detections to the appropriate playbook; and a user interface in which analysts can review automated actions, approve disruptive ones, or take over manually.

The dominant limitation of commercial SOAR products is the form of the playbooks themselves. They are written as flowcharts whose nodes are vendor-specific actions, and the resulting artefacts cannot easily be ported between platforms. A playbook authored for Splunk SOAR will not run on Microsoft Sentinel without being rewritten, and the same is true between any pair of commercial products. This vendor lock-in is, arguably, the strongest reason why the OASIS CACAO standard (Section 2.5) has gained traction: it lets the playbook itself be the portable artefact, with vendor-specific bindings expressed at the action level.

2.4 Threat Intelligence and Playbooks

Cyber Threat Intelligence (CTI) is the information that a defender gathers about adversary capabilities, intentions and patterns of behaviour. It is conventionally divided into four levels. Strategic CTI describes broad trends and threat actor motivations, and is consumed mostly by executives. Operational CTI describes specific campaigns and tooling. Tactical CTI describes techniques, tactics and procedures. Technical CTI describes the artefacts left behind by intrusions, such as indicators of compromise, hashes, network addresses and file signatures. Tounsi and Rais [10] survey the technical layer in depth and argue that automated analysis of technical CTI is only useful when the representation of that intelligence is standardised across producers and consumers.

The de-facto standards for sharing CTI are STIX and TAXII, both maintained by OASIS. STIX 2.1 defines a JSON schema for representing observables, indicators, threat actors, campaigns and a range of related objects [11]. The object that matters most here is the *Course of Action*, which represents a planned response to a given threat. A Course of Action is, in effect, a pointer to a playbook. CACAO formalises what sits at the other end of that pointer.

The connection between CTI and response, therefore, is not abstract. A SIEM raises an alert. The alert is enriched with CTI: who the adversary is likely to be, which campaign it belongs to, which techniques are relevant. The CTI determines which Course of Action applies. The Course of Action points to a playbook. The playbook is the artefact that the SOC actually executes. This thesis operates on the last two steps of the chain and treats the first two as input.

2.5 CACAO 2.0

Section 1.1 already introduced CACAO as the vendor-neutral playbook standard maintained by OASIS. Version 2.0 was approved as a Committee Specification in November 2023 [4] and is the version targeted by the present work. A CACAO playbook is a JSON document. Its top-level structure groups four kinds of content: descriptive metadata such as name, severity and impact; a workflow expressed as a directed graph of steps; a list of agents and targets that the steps refer to; and, optionally, data markings and digital signatures.

2.5.1 Workflow Steps

The workflow is a directed graph rather than a linear list. Every step has an identifier, a type, and a set of fields specific to that type. Six step types matter for this thesis. The `start` and `end` steps mark entry and exit points. The `action` step represents an atomic operation, defined by a list of commands targeted at one or more agents. The `if-condition` step branches the workflow based on a boolean expression, with `on_true` and `on_false` pointers to the next step. The `parallel` step forks execution into several sub-paths that must all complete before the workflow continues. Finally, the `while-condition` and `for-each` steps express bounded iteration.

2.5.2 Agents and Targets

CACAO separates the description of the actor performing the work from the description of the resource being acted upon. The actor is the *agent*, which in this thesis is typically an Azure Resource Manager client, a Microsoft Graph client, or a Sentinel REST client. The resource is the *target*, which can be a virtual machine in a specific subscription, a user principal, a watchlist, a Key Vault, and so on. The same agent can act on many targets, and the same target can be acted upon by different agents. The separation makes a CACAO playbook easier to reuse across tenants and across organisations.

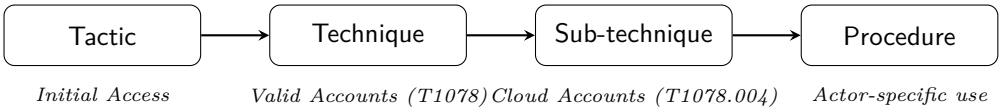

Figure 2.1: The MITRE ATT&CK taxonomy with a concrete example drawn from the Cloud matrix.

2.5.3 Data Markings and Signatures

A CACAO playbook can declare TLP markings and other constraints on how the playbook itself can be redistributed. It can also carry digital signatures over its own JSON body. These features make CACAO suitable as a sharing artefact, not just an execution artefact, which matters when playbooks are exchanged between organisations as part of CTI sharing. The remediator built for this thesis does not produce signed playbooks, but it preserves any signatures present in the input.

2.6 MITRE ATT&CK

ATT&CK is the knowledge base maintained by MITRE that catalogues adversary behaviour at three levels of abstraction [1]. *Tactics* describe the goals an adversary pursues during an intrusion: initial access, persistence, lateral movement, exfiltration, and so on. *Techniques* describe the means by which a tactic is achieved: phishing, valid accounts, scheduled task abuse, and similar. Sub-techniques add finer granularity, distinguishing, for example, spear-phishing via link from spear-phishing via attachment. Procedures, the lowest level, describe the concrete implementations observed in the wild, and are typically tied to named threat actors or malware families. Figure 2.1 summarises the hierarchy with a concrete example.

ATT&CK ships in several matrices that share the same tactics but differ in their set of techniques. The Enterprise matrix is the general-purpose one. The Cloud matrix and the Identity Provider matrix were added later, as cloud-hosted attacks became a more central concern; they overlap with Enterprise but introduce techniques such as *Cloud Account Manipulation* or *Application Access Token Abuse* that are specific to identity-based intrusions [12]. The Containers and Mobile matrices exist as well, but are out of scope here.

ATT&CK has, in practice, become the shared vocabulary of detection engineering. SIEM rules and EDR signatures are routinely tagged with the techniques they detect, and threat-intelligence reports describe campaigns in the same terms. The same vocabulary has started to appear on the response side. A natural way to organise a remediation library is to map techniques to defensive actions, with the understanding that one technique may admit several actions, and that some techniques have no automated defensive answer at all. The tactic-driven execution mode of this thesis is built on exactly this kind of mapping.

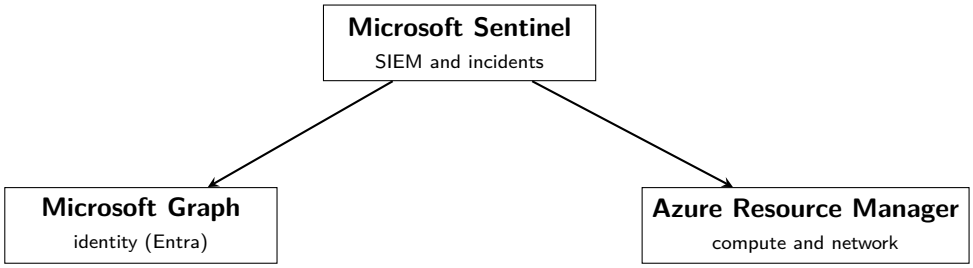

Figure 2.2: The three Azure subsystems relevant to this thesis. Sentinel is the entry point for alerts produced by analytics rules and by external EDR/XDR sources; Microsoft Graph and Azure Resource Manager are the control surfaces used by the remediation actions.

2.7 The Azure Security Stack

The thesis targets Microsoft Azure as the cloud platform that the remediator must understand and act upon. Three Azure subsystems are relevant, and each is described briefly below. Figure 2.2 summarises how they relate to each other and to the remediator.

2.7.1 Microsoft Sentinel

Microsoft Sentinel is the cloud-native SIEM of the platform [13]. It ingests logs from Azure and from third-party sources into a Log Analytics workspace and produces alerts in two main ways: by running Kusto Query Language (KQL) analytics rules over the ingested data, and by accepting pre-built alerts from external EDR/XDR products. Related alerts and their entities are then grouped into *incidents*, which are the unit of work that the remediator consumes as input. Inside an incident, entities (users, IP addresses, hosts, URLs, files) carry the structured context needed to drive remediation.

2.7.2 Microsoft Entra and the Graph API

Microsoft Entra, formerly Azure Active Directory, is the identity provider of the platform. Almost every incident that reaches a modern Azure SOC has an identity dimension to it, whether the alert is about an exposed access token, a compromised service principal or a user signing in from a malicious IP. Entra is therefore a central target of the remediation library implemented in this work: disabling a user, revoking active sessions, resetting a password, removing role assignments, blocking an IP through a Conditional Access named location. The control surface for these operations is the Microsoft Graph API, which exposes Entra and the rest of Microsoft 365 through a uniform REST endpoint.

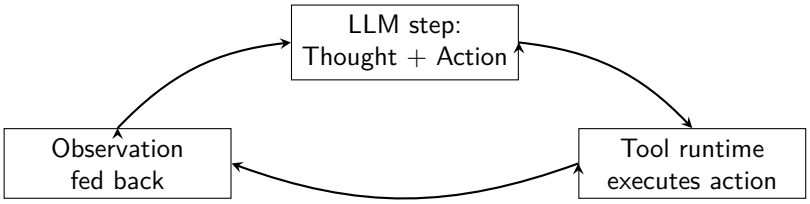

Figure 2.3: The ReAct loop used by the enrichment agent. At each turn the LLM emits a thought and an action; the runtime executes the action and returns the observation; the loop ends when the LLM emits a final answer instead of an action.

2.7.3 Azure Resource Manager

Azure Resource Manager (ARM) is the control plane for Azure infrastructure. Every operation that creates, modifies or destroys an Azure resource passes through ARM, and the permissions to perform such operations are managed by Azure role-based access control. Virtual machine lifecycle operations, Network Security Group rules, Azure Firewall policies and Key Vault configuration all sit behind ARM. The remediator uses the Azure SDK for Python, which is a thin wrapper around the ARM REST API.

Taken together, the three subsystems above form the action surface of the thesis. The remediator does not invent new Azure capabilities. It selects, sequences and executes operations that Azure already exposes, and it does so in a way that respects the boundaries imposed by Microsoft Graph and ARM permissions.

2.8 LLMs and Agentic Patterns

A Large Language Model (LLM) is a neural network trained on broad text corpora to predict the next token in a sequence. The capabilities that emerge from this training, and that matter to a SOC, are less about language generation and more about language *understanding*: turning a long log window into a paragraph of summary, classifying a free-text alert description, extracting structured fields from an unstructured CTI report, and translating analyst questions into KQL or similar query languages. The recent survey by Xu et al. [9] identifies more than two hundred research artefacts that apply LLMs to cybersecurity tasks, with incident response and threat intelligence among the most active sub-areas.

2.8.1 Tool Use and ReAct

The capability that matters most for this thesis is not raw text generation but *tool use*. A modern LLM can be wrapped in a control loop that interleaves natural-language reasoning with the invocation of external functions. The ReAct pattern introduced by Yao et al. [14] is the canonical formalisation of this loop. At each turn the model emits either a thought, an action with arguments, or a final answer; the surrounding runtime executes the action and feeds the observation back to the model. The enrichment agent used in the AI-assisted

execution mode of this work is an instance of this pattern, with the actions restricted to a small allow-listed set of read-only queries against Sentinel, Microsoft Graph, Log Analytics and threat-intelligence feeds. Figure 2.3 shows the loop schematically.

2.8.2 Hallucination and Prompt Injection

Two risks accompany the use of LLMs in this setting. The first is hallucination, the tendency of a language model to produce confident output that is not grounded in the input. In a security context, a hallucinated action name or a misidentified user principal is unacceptable. The mitigation adopted here has two parts: every AI-generated plan is validated against a schema that includes a closed catalogue of action names, and execution is gated behind an explicit human approval step. The second risk is prompt injection, by which an adversary controls part of the input to the model (for instance, the description field of a Sentinel alert) and uses that control to manipulate the model's behaviour. The mitigation is to keep the enrichment agent strictly read-only, to allow-list the tools it can call, and to treat model output as untrusted data rather than as executable code.

Chapter 3

Related Works

This chapter looks at the work the present thesis builds on and compares against. The first section identifies the parts of the inherited tool that have been carried forward without modification; the architecture and capabilities of that tool are presented in Chapter 4, when the gap closed by this work is introduced. The two sections that follow survey other CACAO-compliant tools and SIEM-integrated remediation in public-cloud environments. A fourth section reviews recent academic and industrial work on LLM-based agents for cybersecurity. The chapter closes by positioning the new work against this landscape.

3.1 Inherited Components

Roughly half of the codebase that runs the system described in this thesis is inherited, without modification, from earlier TORSEC work [6]. The components reused as-is fall into five families.

The first family is the *CACAO loader and serialiser*, contained in `source/cacao_playbook/`. The loader parses a CACAO 2.0 JSON document into the in-memory representation. The serialiser produces a fresh CACAO 2.0 document from an executed instance, including the start/end markers and the UUID-based identifier scheme required by the specification. Neither of these modules was touched by the present work.

The second family is the *playbook class hierarchy*. It defines a `Playbook` object whose body is a list of `Step` objects, with specialised subclasses for the workflow types relevant to CACAO: `Action`, `IfStep`, `ParallelStep`, `WhileStep` and `ForEachStep`, plus the `StartStep` and `EndStep` markers. All three execution modes introduced in this thesis (the legacy mode, the tactic-driven mode and the AI-assisted mode) ultimately produce a `Playbook` object and hand it to the same executor.

The third family is the *executor itself*. The executor walks the directed graph of steps, dispatches each `Action` to the appropriate handler, and threads the runtime context through the iterative constructs. The conditional and parallel step machinery, which is non-trivial to get right in the presence of nested constructs and shared end-step markers, is reused without changes.

The fourth family is the *condition evaluation layer*. The same three styles of condition supported by the inherited tool, namely recipe-defined boolean expressions, custom Python predicates, and STIX patterns [11], are still available to playbooks generated by the new modes. The AI-assisted mode does not generate STIX-pattern conditions in practice, but the support is left in place for hand-authored playbooks that need it.

The fifth family is the *recipe DSL and its interpreter*, contained in `source/rec_playbook/`. The recipe path is not extended in this thesis, but it is kept alive so that legacy playbooks expressed in the DSL continue to load and run. Backward compatibility was a hard requirement of the design, and the recipe path is the most visible sign of it.

Taken together, the five families form the deterministic, audit-friendly core of the system. The contribution of this thesis sits above that core. The Azure ingestion layer, the action library, the MITRE tactic-technique-action mapping, the three-mode router, the AI planner and the enrichment agent are all new code that produces or consumes `Playbook` objects, but never reaches inside the executor or the CACAO serialiser.

3.2 Other CACAO-Compliant Tools

The CACAO standard is young, and the set of open-source tools that implement it is still small. Two of them, SOARCA and CACAO Roaster, were already discussed in earlier TORSEC work and are revisited here because they remain the closest external comparators of the present work.

3.2.1 SOARCA

SOARCA is an open-source security orchestrator developed by TNO and released through the COSSAS community platform [15]. It positions itself as the first open-source SOAR aiming for CACAO 2.0 compliance: it ingests, validates and executes CACAO playbooks, with native support for HTTP(S), SSH and OpenC2 as transport-level capabilities. SOARCA exposes a JSON API for consuming playbooks and the triggers that start their execution, and it can be extended through "fins", custom modules that communicate with the SOARCA core over MQTT.

SOARCA and the system described in this thesis share their starting point (a CACAO 2.0 executor) but diverge in scope. SOARCA aims to be a general-purpose orchestrator that an organisation deploys on top of its own tools, with connectors written by the user as fins. The system described here, by contrast, is a vertical tightly bound to Azure: the action library is built for Microsoft Graph and Azure Resource Manager, the alert ingestion is built for Sentinel, and the AI planner is aware of the action catalogue at the schema level. The two approaches are complementary. SOARCA is the right tool when the goal is interoperability across many vendors. The remediator described in this thesis is the right tool when the goal is to make a single cloud platform fully addressable from a CACAO-based pipeline.

3.2.2 CACAO Roaster

CACAO Roaster is a sub-project of the Open Cybersecurity Alliance: a web application for generating, parsing, validating, manipulating, visualising and signing CACAO v2.0 playbooks [16]. It provides a "no-code" graphical experience in which an analyst can draft a CACAO playbook by dragging step shapes onto a canvas, configure their fields, validate the resulting document against the CACAO schema, and optionally apply a digital signature. CACAO Roaster includes a basic integration with SOARCA, so that a playbook authored in the editor can be sent to a running SOARCA instance for execution.

The roles of CACAO Roaster and of the present system are orthogonal. CACAO Roaster is a *producer* of playbooks designed by humans. The remediator described here is a *consumer* of playbooks and, in the tactic-driven and AI-assisted modes, a producer of its own. A natural extension that lies beyond the scope of this thesis is to feed CACAO Roaster the playbooks generated by the AI-assisted mode, so that an analyst can edit a draft before approving it.

3.3 Cloud-Native SIEM Remediation

Outside the CACAO ecosystem, the dominant approach to SIEM-integrated remediation in public-cloud environments is the vendor-specific playbook. Three reference points are worth recalling.

Microsoft Sentinel's own automation is the most direct comparator. A Sentinel playbook is a workflow expressed in Azure Logic Apps and attached to alerts or incidents through *automation rules* [17]. Logic Apps provide a large library of connectors that cover Azure services, Microsoft 365 and many third-party SaaS products, and an analyst can build a playbook by composing those connectors in a graphical designer. The model is powerful but not portable: a Sentinel playbook is tied to the Logic Apps runtime, and its serialisation format (an ARM template) cannot be consumed by any tool outside the Azure ecosystem. The system described in this thesis can be read as an open, CACAO-based alternative to Sentinel automation, running in user space against the same Sentinel APIs.

Splunk SOAR (formerly Phantom) and Palo Alto Cortex XSOAR are the two main commercial SOAR products in the wider market. Both share a similar internal anatomy with what Islam et al. describe in their survey of security orchestration [3]: an engine that interprets a flowchart-style playbook, a library of vendor-specific integrations, and a case-management module. Both also share the same portability limitation as Sentinel. A Phantom playbook will not run on Cortex XSOAR, and neither will run on Sentinel. CACAO exists precisely to escape this kind of fragmentation.

Academic work has begun to look at remediation specifically in cloud-native settings. Tsirakis et al. [18] propose a Knowledge Management System for CACAO playbooks that addresses sharing, discoverability and operational reuse across organisations. In that taxonomy, the system described in the present thesis is a *producer* of CACAO playbooks that such a Knowledge Management System could ingest and re-share. The same group had earlier argued, in a paper presented at IEEE Big Data [19], that course-of-action playbooks should be folded into the broader cyber-threat-intelligence sharing fabric (MISP,

STIX) rather than treated as a separate artefact; the present work shares that view and emits its CACAO reports with the same intent.

3.4 LLM Agents in Cybersecurity

Section 2.8 introduced the use of Large Language Models in security operations and the ReAct pattern that underlies the enrichment agent of this thesis. This section locates the AI-assisted execution mode in the broader landscape of LLM-based security work.

The most visible industrial reference point is Microsoft Security Copilot, which was the first major commercial copilot product targeted at the SOC and which was, in part, the motivation for the "SOC 3.0" label [5]. Copilot operates as a conversational assistant inside the Microsoft security stack and supports analyst-driven investigations rather than autonomous remediation. The design choice (chat interface, no destructive actions by default) is one of the points of reference for the human-in-the-loop policy adopted in this thesis.

On the academic side, several peer-reviewed surveys have catalogued the use of LLMs and LLM-based agents in security operations centres. The systematic literature review by Xu et al. [9], already cited in Chapter 2, screens over 40 000 papers and analyses 185 from top security and software-engineering venues, identifying incident response as one of the two most active sub-areas. A more recent survey, dedicated specifically to AI-augmented SOCs, catalogues the literature in finer detail across eight SOC functions and proposes a five-level Capability Maturity Model that places autonomous-agent architectures at the high end of the scale and human-in-the-loop pipelines, like the one described in this thesis, below it [20].

A growing line of work proposes concrete LLM-agent architectures for SOC and remediation tasks. Toprani and Madisetti [21] describe a multi-agent workflow that combines retrieval-augmented generation with a continuously updated knowledge base to detect and remediate vulnerabilities in Infrastructure-as-Code; their architecture is closer to a pipeline than a chat assistant, and the choice of grounding the LLM on an explicit knowledge base rather than on a free-form web search is also adopted by the enrichment agent of this thesis.

The AI-assisted execution mode of the present thesis is more conservative than these autonomous-agent architectures. The planner runs once per incident, produces a structured plan against a closed action catalogue, and then waits for explicit human approval. The enrichment agent is more agent-like, but it is read-only, allow-listed, and used purely to inform the planner. The choice reflects an explicit design preference for verifiable, auditable behaviour at the cost of some autonomy.

3.5 Positioning

The present thesis can be positioned with respect to four neighbours.

With respect to the **inherited tool**, the present work is a strict extension. The CACAO

Table 3.1: Positioning of this work against the closest comparators.

	This work	**SOARCA**	**Sentinel**	**Splunk SOAR**	**Cortex XSOAR**
CACAO 2.0 native	yes	yes	no	no	no
Vendor-neutral playbook	yes	yes	no	no	no
Azure Sentinel integration	yes	via fins	yes	no	no
MITRE-driven dynamic playbook	yes	no	no	no	no
LLM-based planner	yes	no	no	no	no
Human-in-the-loop policy	yes	n/a	yes	yes	yes
Open source	yes	yes	no	no	no

core, the execution engine and the condition evaluation layer are reused unchanged, while the Azure ingestion layer, the action library, the MITRE mapping and three new execution modes are added on top. The relationship is one of continuity rather than replacement.

With respect to **SOARCA**, the system described here is narrower in vendor coverage but deeper in vertical integration. SOARCA is a horizontal CACAO orchestrator. This work is an Azure-specific CACAO remediator, with built-in knowledge of Sentinel, Microsoft Graph and Azure Resource Manager.

With respect to the **commercial SOAR products** that dominate the cloud-SIEM market, the contribution is the use of an open, vendor-neutral playbook format (CACAO 2.0) combined with a MITRE tactic-technique-action mapping and an optional LLM planner. At the time of writing, none of the commercial SOAR products combines all three.

With respect to the **LLM-agent literature**, the contribution is not in the agent design itself, which is deliberately conservative, but in the integration of an agent-based planner into a CACAO-compliant execution pipeline that an operator can audit and replay end-to-end.

Table 3.1 summarises the same positioning along seven binary dimensions.

These four positioning statements are revisited in Chapter 4, which formalises the gap they jointly identify, and in Chapter 5, which presents the design that closes it.

Chapter 4

Problem Statement

This chapter formulates the problem the thesis sets out to solve. The first section describes the architecture and capabilities of the playbook-driven remediator inherited from earlier TORSEC work, and then identifies the three limits that motivate the present work. The second section synthesises those limits into a single gap and states the research questions the thesis answers.

4.1 The Inherited Tool: Architecture, Capabilities and Limits

The starting point of this work is a playbook-driven remediator developed within the TORSEC research group at Politecnico di Torino [6]. That work itself extended an internal incident-response tool produced within the same research line. The codebase has therefore evolved across multiple iterations of TORSEC work on automated remediation, of which the present thesis is the latest: both the architecture and a significant fraction of the implementation are inherited from the existing tool.

The inherited tool ingests structured cybersecurity alerts and normalises them into an internal representation. It then maps the normalised alert to one of a set of remediation playbooks held in its knowledge base, using a catalogue that links alert metadata (and in particular MITRE technique codes) to candidate playbooks and to the feasibility checks that decide which actions can actually run in the current environment. The selected playbook is loaded from a CACAO 2.0 JSON file [4], deserialised into an in-memory *executable* representation, run against the target environment, and re-serialised back into a fresh CACAO report suitable for archival or sharing.

In addition to CACAO, the tool supports a parallel *recipe* path in which playbooks can be expressed in a compact domain-specific language and compiled to executable instances on demand. The recipe path covers parts of the legacy library that predate CACAO support and is preserved for backward compatibility.

Several non-trivial features sit inside this pipeline. The executable instance is a directed graph rather than a list, with first-class support for conditional branching and for parallel

sub-paths. Conditions can be expressed in three forms: recipe-defined boolean expressions, fully custom Python predicates, and STIX patterns evaluated against an in-memory representation of the alert [11]. Iteration is bounded: the `while-condition` and `for-each` steps respect a configurable cap on the number of iterations. The CACAO loader and serialiser are pure-Python and have no external dependencies on a CACAO library, which makes them easy to extend.

The inherited tool is, on its own terms, a solid base. It implements the CACAO 2.0 execution semantics correctly, it produces auditable artefacts, and it handles the workflow constructs that any non-trivial remediation playbook requires. Its limits, however, become visible as soon as the target environment is a public cloud rather than the operational setting for which the existing tool was originally written. Three limits matter for the present work.

The first comes from the lack of coverage of the Azure cloud control plane and of Microsoft Sentinel as a SIEM. The inherited action library and ingestion logic are tied to the operational environment used by the earlier TORSEC tool. There is no normalisation pipeline for Microsoft Sentinel incidents or for the entities they carry, and there is no library of Azure-side actions on identity, network and compute resources. Adapting a hand-authored CACAO playbook to act on Azure is possible in principle, but it would require expressing every Azure operation as a free-form CACAO command, with no shared catalogue and no reusable parameter binding. Feasibility checks specific to Azure role-based access control and Microsoft Graph permissions are also missing. The result, in practice, would be a brittle one-off playbook per alert type.

A second limit, closely related, is the reliance on hand-authored, static playbooks. The inherited catalogue maps alert metadata to one of a finite set of pre-authored playbook files. When a new combination of MITRE techniques appears in an alert, the catalogue either points to a pre-existing playbook that approximately fits or to no playbook at all. The dependency on hand-authored playbooks is a familiar problem in commercial SOAR products [3] and is the dominant reason why playbook libraries tend to grow stale: every new threat shape needs a new playbook, and every change in the target environment forces the library to be updated. The inherited tool offers no mechanism for synthesising a playbook from a MITRE technique-to-action mapping at runtime, even though such a mapping would make the catalogue smaller and far more general.

The third limit sits at a different level. The inherited tool has no planning layer above the catalogue. Every step that a playbook executes was decided by a human author at some earlier point. Human authorship is good for auditability but bad for coverage: alerts whose combinations of techniques were never anticipated end up with no automated response. A planning layer that can synthesise a candidate plan from a set of allowed actions and from the alert context could close part of this coverage gap. The inherited tool, however, has no place to hook such a planner in, and (just as importantly) no schema or approval policy that would make an AI-generated plan safe to execute against a production tenant.

These three limits are not independent of each other. Each assumes the others remain unsolved. The static-playbook limit forces a hand-authored library; the absence of AI planning means that even when the library is exhausted the system gives up; the absence of cloud control-plane coverage means that even when the system has a playbook, it cannot

act on Azure. Closing the gap therefore requires closing all three at once.

4.2 Identified Gap and Research Questions

The gap can be stated in a single sentence. The inherited remediator implements a correct, auditable CACAO 2.0 execution engine, but it cannot ingest alerts from Microsoft Sentinel, it cannot synthesise CACAO playbooks dynamically from MITRE technique mappings, and it has no AI-assisted planning layer with a human-in-the-loop policy. The present thesis closes that gap. The research questions it answers are the following.

RQ1. Can the inherited CACAO-compliant remediator be extended to ingest alerts from Microsoft Sentinel and to act on the Azure control plane (Microsoft Graph for identity and Azure Resource Manager for compute and network resources), while preserving its existing execution core, its conditional and parallel step machinery, and its CACAO serialisation?

RQ2. Can MITRE ATT&CK technique-to-action mappings be used to generate CACAO playbooks dynamically at runtime, so that the catalogue of pre-authored playbooks shrinks to a fallback role and the system can respond to combinations of techniques never anticipated by a playbook author?

RQ3. Can a Large Language Model be integrated into the planning step under a safety policy that keeps a human operator in final authority over any disruptive action, and that emits a CACAO playbook as a side effect, so that the entire AI-assisted execution remains auditable end-to-end?

Each research question maps to one of the three execution modes introduced by the thesis and to a specific subset of the contributions listed in Chapter 1. RQ1 is answered by the legacy and tactic-driven modes, together with the Azure ingestion layer and the action library described in Chapters 5 and 6. RQ2 is answered by the tactic-driven mode and by the MITRE tactic-technique-action mapping built specifically for Azure. RQ3 is answered by the AI-assisted mode, by the approval policy that guards it, and by the read-only enrichment agent that informs the planner. The validation reported in Chapter 7 provides empirical evidence that each of the three questions admits an affirmative answer on the Azure threat scenarios that form the test corpus.

Chapter 5

Design

This chapter describes how the inherited tool was redesigned to close the three limits identified in Chapter 4. The presentation moves from goals to components: design objectives first, then the overall architecture, the ingestion pipeline for Microsoft Sentinel, the three execution modes, the MITRE mapping that drives dynamic generation, the AI planning layer with its approval policy, the read-only enrichment agent, and finally the web interface that exposes the whole pipeline to a human analyst.

5.1 Design Objectives

A handful of objectives guided every decision documented in the rest of the chapter.

Preserve the CACAO core. The CACAO 2.0 execution engine inherited from earlier TORSEC work [6, 4] is the part of the system that is already correct and audited. The redesign treats it as a fixed point: every new mode must funnel its remediation logic through the same in-memory *executable* representation and the same serialiser, so that the audit trail and the workflow constructs (conditional branches, parallel sub-paths, bounded iteration) behave identically regardless of how the playbook was produced.

Cloud-agnostic plumbing, Azure-specific actions. The ingestion and action layers are organised so that the bus carrying alerts and the runtime that invokes operations are decoupled from the specific resource provider. The current implementation targets Microsoft Sentinel as the alert source and Microsoft Graph plus Azure Resource Manager as the action surface, but the same shape would accept a different SIEM (for example, a future on-premise SIEM connector) without touching the execution core.

Three modes, one engine. Rather than building three separate tools, the design routes incoming requests through a single dispatcher that selects one of three execution modes at runtime. The modes differ in how the playbook is obtained (loaded from disk, generated from a MITRE mapping, or planned by a language model), but they share the same

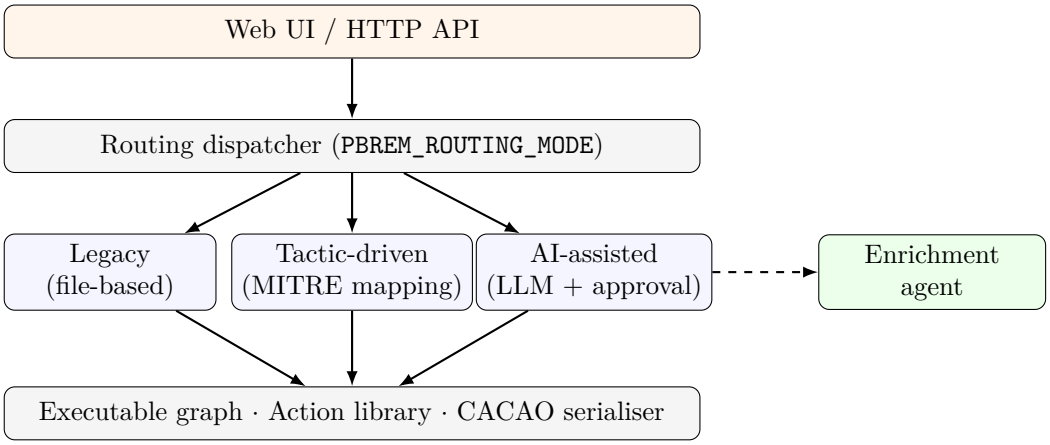

Figure 5.1: High-level architecture. The dispatcher selects one of three remediation front-ends; all three converge on the shared executable graph and the CACAO serialiser inherited from earlier TORSEC work.

downstream pipeline. The cost of supporting all three is therefore the cost of three small front-ends plus one shared back-end.

Safety before automation. The AI-assisted mode never executes a destructive action without explicit human approval. The closed action catalogue and the approval policy are part of the design, not an afterthought bolted onto the LLM call. Read-only enrichment is, by contrast, allowed to run autonomously, because nothing it does can damage the tenant.

Auditability end-to-end. Every execution, regardless of mode, emits a CACAO report. The report captures the playbook that was run (whether file-based, generated, or AI-produced), the entities it touched, and the outcome of each step. The reasoning trace produced by the planner or by the enrichment agent is also archived, so that an external reviewer can reconstruct *why* a given action was proposed and not only *what* was executed.

5.2 Architecture

The system is organised as four horizontal layers, glued by a thin dispatcher. Figure 5.1 sketches the layout. From top to bottom the layers are: the web interface and HTTP API; the routing dispatcher that selects the execution mode; the three remediation front-ends (legacy, tactic-driven, AI-assisted); and the shared back-end that hosts the executable representation, the action library and the CACAO serialiser. The enrichment agent sits to the side of the planner and is invoked only by the AI-assisted mode.

The dispatcher is a single module that reads a routing variable and imports the appropriate front-end. Its job is trivial by design: keeping the selection logic in one place avoids the trap of scattered `if/else` branches across the codebase and makes it possible to

add a fourth mode in the future (for example, a hybrid mode that consults the LLM only when the MITRE mapping returns no match) without rewriting any of the existing fronts.

5.3 Sentinel Ingestion

Alerts enter the system as JSON payloads with the shape of a Microsoft Sentinel incident [13]. A Sentinel incident bundles one or more alerts and a set of *entities*, which are the resources (accounts, hosts, IP addresses, URLs) the analytics rule has flagged. The ingestion layer normalises this structure into the internal alert representation that the rest of the pipeline already understands.

Normalisation does three things in practice. It extracts the MITRE technique codes carried in the incident's `additionalData` field and exposes them as a flat list that the catalogue and the planner can both consume. It re-shapes the entity collection into a typed dictionary keyed by entity kind, so that a downstream step that needs, say, the user principal name does not have to walk the original heterogeneous array. And it preserves the raw incident payload alongside the normalised view, so that a later step can still reach into the original JSON when a custom condition or a rare attribute needs to be inspected.

The ingestion layer is implemented as two analyzer modules: one for the legacy mode (kept unchanged) and one for the tactic-driven and AI-assisted modes. The split is pragmatic. The legacy analyzer was designed for the alert schema of the inherited tool, and changing it would force a regression on the existing playbooks. The new analyzer is a superset for Azure incidents and is what the two new modes call into.

5.4 Azure Control-Plane Capabilities

The action library is the largest single body of new code introduced by this thesis. It is what makes the engine *capable* of acting on an Azure tenant, as opposed to merely consuming alerts from one. This section describes the design of that layer in capability terms, because the operations the engine performs are not best understood as "calls to an API"; they are typed actions whose preconditions, authentication path, scope of effect and rollback story have to be reasoned about together.

Capability families. The library is organised into five capability families, each backed by a distinct Azure surface. The *Identity* family acts on Entra ID through Microsoft Graph: it can revoke active sessions for a user, disable an account, force a password rotation, audit role assignments and rotate a service-principal secret. The *Network* family acts on Azure Resource Manager through the Network Management SDK: it can attach containment NSG rules to a VM, push entries into a Conditional Access named location, and amend an Azure Firewall policy. The *Compute* family acts on Resource Manager through the Compute Management SDK: it can stop a VM, run an in-guest command (used to terminate a malicious process), and snapshot a disk before destructive operations. The *Sentinel* family acts on Microsoft Sentinel through the Log Analytics REST surface: it can post incident

comments, insert IOCs into watchlists, and trigger analytics-rule re-evaluation. A small fifth family covers *example destinations* for which the engine ships a single integration point (Azure Key Vault is the running example used in the validation), kept deliberately narrow so that the design does not over-promise breadth.

Each capability has the same anatomy. It declares which entity kinds it can accept, which fields of the global scope it requires (subscription, resource group, workspace name), which Microsoft Graph or RBAC permission its principal needs in order to succeed, and whether it is *passive* or *destructive*. The last attribute is what the AI-assisted mode reads when deciding whether to gate the action behind operator approval.

Authentication path. A single configuration produces three authentication paths, one per surface. For Resource Manager and Compute/Network operations the engine uses `DefaultAzureCredential` (or, when service-principal credentials are supplied via the configuration file, an explicit `ClientSecretCredential`), and lets the Azure SDK negotiate the token cache. For Microsoft Graph the engine bypasses the SDK and acquires an application token directly through the OAuth2 client-credentials grant against the tenant's `/oauth2/v2.0/token` endpoint, because the read patterns required by the enrichment agent are simple enough that adding a Graph SDK dependency would buy little. For Log Analytics queries the engine uses the same Resource Manager credential against the `api.loganalytics.io` endpoint, which keeps the Sentinel queries within the same token-refresh logic as the rest of the pipeline. All three paths consume the credentials of a dedicated Azure App Registration provisioned specifically for the remediator, whose client secret is kept in the engine's configuration file and can be rotated independently of any human credential. Authentication failures are caught at the capability boundary, not at the call site, so that a single misconfigured credential does not produce a different error message in every action.

Scope and resource hierarchy. The engine reasons about three nested scopes that the Azure resource hierarchy provides. Tenant-level operations (Entra ID account disable, named-location updates) require only the tenant identifier and the application's Graph permissions; the action ignores subscription and resource group. Subscription-level operations (Azure Firewall policy edits) need the subscription identifier but no resource group. Resource-group-scoped operations (NSG isolation, VM stop, VM run-command) need the full triple subscription + resource group + resource name. The ingestion layer extracts as much of this triple as the incident carries and the action library completes the rest from the global scope; capabilities that find their scope still incomplete after both passes are short-circuited with a structured warning rather than allowed to fail half-way through.

Feasibility checks. Before a capability fires its first SDK call, a small feasibility check confirms that the target resource exists and that the application has the right to act on it. The check is cheap (a single GET) and lets the engine fail fast in the common cases (the user mistyped a VM name, the resource lives in a different subscription, the application is missing a role assignment). The feasibility check also keeps the audit log honest: a

Family	Surface	Permission required	Sample action
Identity	Microsoft Graph	`User.RevokeSessionsAll`, `User.ReadWrite.All`, `Application.ReadWrite.All`	`disable_user_account`, `rotate_service_principal_secret`
Network	ARM (Network)	`Contributor` on resource group	`isolate_vm_network`, `add_ip_to_named_location_blacklist`
Compute	ARM (Compute)	`Virtual Machine Contributor`	`shutdown_vm`, `terminate_malicious_process`
Sentinel	Log Analytics + Sentinel API	`Microsoft Sentinel Contributor`	`upsert_sentinel_watchlist_ioc`, `add_sentinel_incident_comment`
Example	Key Vault	`Key Vault Contributor` (resource-scoped)	`restrict_key_vault_network_access` (example only)

Table 5.1: Capability matrix. Each family is backed by a distinct Azure surface and requires a distinct permission for the application's service principal. The catalogue in `ai_planner/plan_schema.py` is the machine-readable counterpart consumed by the AI planner.

destructive operation that never started because the target did not exist is recorded as such, rather than as a vague "operation failed".

Read/write boundary. The library is split into a read-only sub-library (used by the enrichment agent, Section 5.8) and a write-capable sub-library (used by the executor). Nothing in the read-only sub-library can mutate the tenant: even a worst-case enrichment run against a misconfigured prompt cannot disable an account or shut a VM. The split is enforced at the package boundary rather than at the call site, so that a developer adding a new tool to the enrichment registry cannot accidentally import a destructive helper. The same design principle is what allows the AI-assisted mode to run autonomously through the passive half of a plan while waiting for human approval on the destructive half.

Capability matrix. Table 5.1 summarises the libraries the engine consumes, the surface they target, and the permission they require. The catalogue declared in `ai_planner/plan_schema.py` (Chapter 6) is the machine-readable rendering of the same information, augmented with the per-action risk level and approval gate.

Idempotency and rollback. Two operational concerns deserve mention because they shape the way capabilities are written. Idempotency is required wherever feasible: re-running the same step against the same target should not change the outcome (the engine relies on this when a job is replayed from the CACAO report after the fact). `add_sentinel_incident_comment` is naturally idempotent because the comment string carries an execution timestamp; `upsert_sentinel_watchlist_ioc` is idempotent by virtue of the UPSERT semantics; `isolate_vm_network` achieves it by deriving the NSG rule name deterministically from the run identifier. Rollback is a different story. The engine does not promise transactional rollback, because the underlying Azure operations are not transactional. What it does promise is a record: every destructive operation logs the exact pre-state it observed and the exact change it applied, so that a human operator can reverse it manually with a known set of inputs.

5.5 Three-Mode Engine

The three execution modes are the architectural answer to the three research questions stated in Chapter 4. Each mode shares the back-end but differs in how it sources the

playbook that the back-end will execute.

Legacy mode. This mode is the inherited behaviour, preserved without changes to its execution semantics. The catalogue maps the normalised alert to a CACAO playbook file on disk; the file is loaded, instantiated, executed, and re-serialised. The mode is the regression baseline of the redesign: if any other mode breaks an existing playbook, the legacy path is the contract that says what the correct output should have been.

Tactic-driven mode. The MITRE technique codes attached to the incident are looked up in a tactic-technique-action mapping built specifically for Azure. The mapping returns an ordered list of action identifiers (for example, `disable_user_account`, `revoke_user_sessions`, `isolate_vm_network`). A small generator stitches the actions into a CACAO playbook on the fly, binding their parameters to the entities carried by the incident. The generated playbook is then handed to the shared back-end, which executes it the same way it would execute a file-based one.

AI-assisted mode. The third mode replaces the catalogue lookup with a planning call. A Large Language Model receives the normalised alert, the closed action catalogue, and a contextual snapshot built by the enrichment agent, and returns a structured *remediation plan*. The plan is then validated against the catalogue (no action outside the allowed list, no missing required parameters), compiled to CACAO, and (for any action flagged as destructive) presented to the operator for approval before execution.

The three modes are selected at runtime by a single routing variable. The routing module is a thin shim: it reads the variable, imports the matching front-end, and delegates. The choice is deliberate. Keeping the dispatcher trivial means that the modes are isolated from each other (a bug in the planner cannot affect the legacy run) and that the testing strategy can swap modes without touching shared state.

5.6 MITRE Mapping

The tactic-driven mode is built around a hand-curated mapping that, for each pairing of MITRE ATT&CK tactic and technique relevant to the Cloud and Identity Provider matrices [12], lists the actions that the system can take in response. The mapping is the smallest possible knowledge base that lets the engine generalise from a closed action library to an open set of incident shapes.

The mapping is intentionally narrow. It is not an attempt to capture every possible response to every possible technique; that would replicate, in a different format, the static-playbook problem. It is a curated baseline that covers the techniques most frequently observed in Azure incidents, plus a fallback that returns a generic containment action when no entry matches. The mapping is stored as a Python module to keep it close to the action catalogue (one diff updates both); a JSON or YAML representation would be equivalent and is left as a possible future refactor.

Tactic	Technique	Mapped actions (excerpt)
Initial Access	T1078 – Valid Accounts	disable_user_account, revoke_user_sessions
Persistence	T1098 – Account Manipulation	revoke_user_sessions, reset_user_password
Defense Evasion	T1562 – Impair Defenses	enable_diagnostic_settings
Credential Access	T1110 – Brute Force	block_source_ip, revoke_user_sessions
Lateral Movement	T1021 – Remote Services	isolate_vm_network
Impact	T1486 – Data Encrypted for Impact	shutdown_vm, snapshot_disk

Table 5.2: Illustrative excerpt from the MITRE tactic-technique-action mapping. The full table covers the techniques most frequently observed in the Azure threat scenarios used for validation.

When the generator runs, it iterates over the techniques attached to the incident, collects the union of actions returned by the mapping, deduplicates them (an action can be mapped from more than one technique), orders them by a heuristic priority (containment first, restoration last), and emits a CACAO playbook with one workflow step per action. Parameter binding uses the typed entity dictionary built during ingestion: a step that needs a user principal name reads it from the `account` entity, a step that needs a VM name reads it from the `host` entity, and so on. Missing parameters are handled by skipping the step and logging a structured warning rather than by failing the whole playbook.

5.7 AI Planning and Approval

The AI-assisted mode is the most delicate part of the design. It is also the one where the gap between research-grade enthusiasm and production-grade safety is widest, and the design is shaped accordingly.

A planning call has three inputs. The first is the normalised alert (the same one the other modes consume). The second is the closed action catalogue: a structured description of every action the engine is able to execute, with its parameters, its default values, its risk level, and whether it requires explicit approval. The third is an enrichment snapshot, produced by the agent described in the next section, that captures the read-only context the planner needs (recent sign-ins for the user, role assignments on the resource, related Sentinel incidents). The LLM is then asked to produce a remediation plan in a strict JSON schema, with one entry per action and explicit references to the entities the action will touch.

The plan is not executed as-is. A validator checks four things. It rejects any action that is not in the closed catalogue (preventing the model from inventing operations the engine cannot actually perform). It rejects any action whose required parameters cannot be filled from the alert or the enrichment snapshot. It collects the approval-required actions into a separate list, surfaces them to the operator through the web interface, and waits. And it tags the remaining (passive) actions for autonomous execution. The compiled CACAO playbook is therefore split implicitly into two halves: a passive half that runs immediately and a destructive half that runs only after a human says yes.

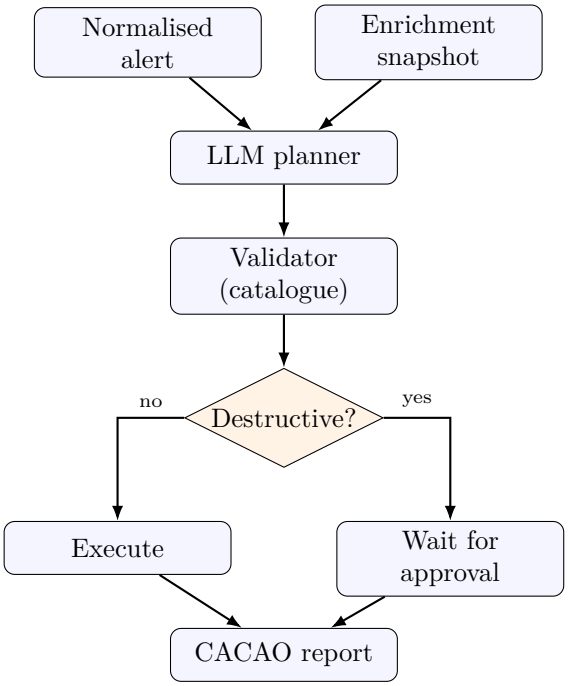

Figure 5.2: AI-assisted execution flow. The planner emits a structured remediation plan; the validator rejects anything outside the closed catalogue; destructive actions wait for human approval before joining the autonomously executed half in the final CACAO report.

Figure 5.2 shows the flow.

A few design choices in this flow are worth stating explicitly. The catalogue is closed by construction: the model can refuse to act, but it cannot invent an action. The schema is strict: a plan that fails validation is not silently repaired, it is rejected and re-requested with feedback. Approval is per action and not per playbook, so the operator can authorise some destructive steps and skip others. And the CACAO report includes the planner's reasoning trace, which means that even an automatically executed plan can be reviewed after the fact.

5.8 Enrichment Agent

The planner alone has only the information carried by the incident. That is often too little. Whether `disable_user_account` is the right response depends on whether the account in question has shown other suspicious behaviour, whether it owns privileged role assignments, and whether the device used to sign in is known. Producing this context is the job of the enrichment agent.

The agent follows the ReAct pattern [14]: it interleaves reasoning steps with tool calls, where each tool is a read-only query against a backing service (recent sign-ins from Microsoft Graph, role assignments from Azure Resource Manager, related incidents from

Sentinel, threat-intelligence lookups against IP and URL reputation feeds). The agent decides which tools to call based on what the incident contains, observes the result, updates its working hypothesis, and continues until either the budget is exhausted or it judges the snapshot complete.

Two design decisions matter for safety. The first is that every tool the agent can call is read-only: even a worst-case run cannot modify the tenant. The second is that the loop is bounded by two budgets, one on the number of tool calls and one on the number of reasoning turns. The bounds are configurable, but they are always finite, so a stuck agent cannot consume the analyst's time budget indefinitely.

The agent has four interchangeable backends. A heuristic backend implements a small set of deterministic rules and is used as a smoke-test fallback when no LLM key is configured. Two LLM backends call the OpenAI API, with the agentic variant exposing the read-only tools as function-calls and letting the model orchestrate the loop. The fourth is an external binary (`claw`) for users who prefer to keep the agentic loop outside the Python process. The choice is made at startup and is transparent to the rest of the pipeline.

5.9 Web API and UI

The pipeline is exposed to the operator through a small HTTP server and a single-page web interface. The server is intentionally minimal: it serves the static page, accepts alert payloads as JSON, holds a registry of in-flight execution jobs, and exposes the approval endpoint that the AI-assisted mode needs.

The interface is organised around three views. A configuration view edits the routing mode and the credentials that the engine needs (the API key for the LLM backend, the access token for the web server, the dry-run flag). A submission view accepts an alert payload and starts an execution job, showing the job's progress as it moves through the pipeline. And an approval view, used only by the AI-assisted mode, surfaces the destructive actions waiting for authorisation, the per-action risk level, and the planner's justification; the operator can authorise or reject each action individually.

Two design choices in the web layer are worth mentioning. Configuration is persisted to a local JSON file and applied to the environment at startup, which means that an unattended container can be configured without a human pressing buttons in the UI. And the execution history is appended to a JSONL file alongside the runtime, which gives the operator a flat, grep-friendly audit log without any dependency on an external database. Neither choice precludes a future migration to a richer storage backend.

The web layer is, on its own terms, the thinnest of the components described in this chapter. It is also, arguably, the most user-visible: it is the surface through which an analyst experiences the redesigned tool, and the design effort spent on it has been concentrated on making the approval workflow unambiguous rather than on aesthetic polish.

Chapter 6

Implementation

This chapter translates the design of Chapter 5 into the actual codebase. The aim is not to walk through every module file by file (the repository is large), but to give the reader enough detail to locate the components, to follow the data path from a Sentinel incident to a CACAO report, and to understand the trade-offs that the design choices imposed at the code level. The chapter is organised around the same seven topics as Chapter 5, with one section devoted to the codebase layout, one per execution mode, and one for each of the cross-cutting subsystems.

6.1 Codebase Layout

The project is a single Python package rooted at `source/`. The layout intentionally mirrors the design: one folder per subsystem, plus a handful of top-level modules that act as entry points. Table 6.1 lists the folders and the responsibilities they carry.

The CACAO core inherited from earlier TORSEC work [6] sits inside `source/cacao_playbook/` and `source/playbook/`. Neither folder has been refactored beyond what was needed to expose hooks for the new front-ends. The serialiser still emits CACAO 2.0 [4] JSON, with the same workflow constructs, the same condition forms, and the same bounded iteration semantics. Dependencies are kept to a tight list. The Azure SDK is consumed through `azure-identity`, `azure-mgmt-compute` and `azure-mgmt-network`; Microsoft Graph and Log Analytics are reached over plain HTTP with `requests` (Graph in particular has no first-party Python SDK that fits the read patterns used here). The OpenAI Python client is the single LLM dependency. STIX patterns continue to rely on `stix2` and `stix2-matcher` as in the earlier TORSEC tool.

6.2 Router and Mode Selection

The router is the smallest module in the codebase, and that is on purpose. Listing 6.1 reproduces it in full. Three things are worth pointing out. The variable is read once, at startup; subsequent changes do not propagate. The imports are lazy, so a tenant that

Folder / file	Responsibility
source/pb_rem_router_v2.py	Reads PBREM_ROUTING_MODE and imports the matching front-end.
source/pb_rem.py	Legacy front-end (file-based playbooks).
source/pb_rem_tactic_v2.py	Tactic-driven front-end (MITRE-mapping generation).
source/pb_rem_ai_plan_v3.py	AI-assisted front-end (LLM planner + approval).
source/cacao_playbook/	CACAO 2.0 loader, executable graph, serialiser.
source/playbook/	In-memory *Playbook*, *Action*, *Parallel_step*, *ForEachEntity-Loop* classes.
source/executor/	Action implementations against Azure / Microsoft Graph / on-premise.
source/input_analyzer/	Alert normalisation (legacy + tactic_v2 variants).
source/programmed_generated_playbook/	Generators that build CACAO playbooks at runtime.
source/ai_planner/	Closed action catalogue, LLM planner, plan-to-playbook compiler.
source/enrichment_agent/	Read-only ReAct agent, tool registry, backend dispatch.
source/web_app.py	HTTP server, job registry, approval endpoints.
web/index.html	Single-page UI bundled with the server.
config/, kb/	Static configuration and knowledge base (MITRE mappings, recipes).

Table 6.1: Top-level layout of the implementation. Each subsystem identified in Chapter 5 maps to a single folder, with one entry point per execution mode at the package root.

Listing 6.1: Routing module (source/pb_rem_router_v2.py).

```
def main() -> None:
    routing_mode = str(os.getenv("PBREM_ROUTING_MODE", "legacy")).strip().casefold()

    if routing_mode == "tactic_v2":
        from source.pb_rem_tactic_v2 import main as selected_main
    elif routing_mode == "ai_plan_v3":
        from source.pb_rem_ai_plan_v3 import main as selected_main
    else:
        from source.pb_rem import main as selected_main

    selected_main()
```

never enables the AI-assisted mode never imports the OpenAI client. And the default is the legacy front-end, which makes an unconfigured deployment behave like the inherited tool, not like a half-configured AI system.

6.3 Sentinel Ingestion

The ingestion layer lives in two sibling modules: source/input_analyzer/input_analyzer.py (used by the legacy mode) and source/input_analyzer/input_analyzer_tactic_v2.py (used by the tactic-driven and AI-assisted modes). The second is a superset

of the first for Azure incidents and is where the work for this thesis is concentrated.

A Microsoft Sentinel incident [13] reaches the system as a JSON document with three useful parts: `incident.properties.additionalData` (techniques, severity, classification), `entities` (the typed objects the analytics rule flagged), and the raw alert payload underneath. The analyzer produces a flat `remediation_data` dictionary that the rest of the pipeline consumes. Five normalisation steps are applied:

1. MITRE technique codes are flattened out of `additionalData.techniques`, normalised to upper case, and exposed as `techniques`.
2. Tactics are derived from the technique codes via a static lookup; the result is exposed as `normalized_tactics`.
3. Entities are bucketed by kind (`account`, `host`, `ip`, `url`, `file-hash`, ...) into a typed dictionary so that downstream consumers can ask for a kind without scanning the array.
4. Azure-specific identifiers (subscription, resource group, workspace name) are extracted into a `context` sub-dictionary that the action library treats as global scope.
5. The original alert is preserved under `raw_alert` so that custom STIX patterns and recipe-defined conditions can still match against it.

The analyzer is dispatched by an `analyzer_mapping` dictionary keyed by alert family (`unauthorized_access`, `azure_identity_threats_test`, `azure_sentinel_threats_test`, and so on). Adding support for a new family is a matter of adding one entry to the dictionary and one function to the analyzer module; no other file needs to change.

6.4 Dynamic Playbook Generation

The tactic-driven mode is implemented by two cooperating files. The mapping itself is the constant `MITRE_TACTIC_TECHNIQUE_ACTION_MAPPING`, declared at the top of `source/executor/actions_azure_tactic_v2.py`. Keeping the mapping next to the action implementations means that a developer who adds a new action and a new mapping entry edits a single file.

The generator is `source/programmed_generated_playbook/mitre_playbook_generator.py`, with a single public function `generate_playbook_from_mitre_mapping(infrastructure,global_scope,mitre_tactic_technique_action_mapping, alert)`. Internally it goes through three phases.

The first phase resolves the list of action identifiers. The function `resolve_actions_from_mitre_mapping` (declared in the same actions module) walks the tactic-technique pairs attached to the alert, looks them up in the mapping, and returns an ordered, deduplicated list of action names. When the alert carries no tactics or none of its tactics match an entry, the function falls back to a small generic containment set so that the pipeline always has something to execute.

Listing 6.2: Action catalogue entry for `isolate_vm_network` (excerpt of `ai_planner/plan_schema.py`).

```
"isolate_vm_network": {
    "title": "Isolate VM network",
    "description": "Apply containment network rules to the impacted VM.",
    "input_order": ["subscription_id", "resource_group", "vm_name",
                    "nsg_name", "isolation_rule_base_name", "base_priority"],
    "required_context": ["subscription_id", "resource_group", "vm_name", "nsg_name"],
    "approval_required": True,
    "risk_level": "high",
    "defaults": {
        "isolation_rule_base_name": "ai-plan-isolation",
        "base_priority": "130",
    },
    "output": None,
},
```

The second phase translates each action identifier into an `Action` object as defined in `source/playbook/playbook_class.py`. The translation reads the action's signature from the catalogue described in the next section and binds each parameter either to a literal value (when the recipe-defined default is appropriate) or to a runtime reference into the `global_scope` or the entity dictionary built by the ingestion layer.

The third phase wires the `Action` objects into a `Playbook` graph and hands it to the executor, which is the same executor used by the legacy mode. The generated graph is structurally identical to one loaded from a CACAO file, which is what allows the back-end to be shared.

6.5 AI Planner Subsystem

The AI-assisted mode is the largest subsystem added by this thesis. It is divided across three files inside `source/ai_planner/`: `plan_schema.py` hosts the closed action catalogue and the validator; `incident_planner.py` hosts the LLM dispatch logic; `plan_to_playbook.py` hosts the compiler from a validated plan to a CACAO playbook.

The closed catalogue. Listing 6.2 shows a representative entry. Each entry declares the action title, a one-line description used as the prompt context, the parameter order, the subset of parameters that must be available before the action can run, a Boolean `approval_required` flag, a coarse risk level, and a default for any parameter the caller can omit.

The catalogue is consumed in two places. The planner injects a compact textual rendering of the catalogue into the LLM prompt as the list of actions the model is allowed to propose. The validator uses the same dictionary to reject any plan step that references an unknown action, that omits a `required_context` parameter, or that asks for a defaulted

parameter with an unsupported value.

The planner. The class `AIIncidentPlanner` (in `incident_planner.py`) decides at construction time which backend to use, based on the environment variable `PBREM_AI_PLANNER_BACKEND`. Two backends are implemented: `openai` (the production backend, going through the OpenAI Python client) and `heuristic` (a small rule-based fallback that lets the system function without network access during tests). The OpenAI backend builds a structured prompt from the planning input, calls the chat completions API with a strict JSON output schema, parses the result, and returns a `remediation_plan` dictionary. If the LLM call raises an exception (network error, rate limit, malformed output), the planner falls back to the heuristic and tags the run with a `*_fallback_heuristic` backend marker so that the audit trail shows that no LLM was actually consulted.

The compiler. The function `compile_remediation_plan_to_playbook` (in `plan_to_playbook.py`) takes a validated plan and produces a `Playbook` object. Two optimisations happen during compilation. Consecutive steps that share the same action and differ in exactly one parameter are folded into a `ForEachEntityLoop` node, which produces a more compact CACAO graph than a flat sequence of identical actions. And steps that are independent of each other (no shared input or output binding) are grouped into a `Parallel_step` node, so that, for example, revoking the sessions of two compromised accounts happens concurrently rather than sequentially.

6.6 Enrichment Agent

The enrichment agent is implemented in `source/enrichment_agent/` across three files. `agent.py` hosts the main `EnrichmentAgent` class and the backend dispatch; `tools.py` hosts the read-only tool registry; `schema.py` hosts the JSON-schema validator for the enrichment result.

The class reads four environment variables at construction: `PBREM_ENRICHMENT_ENABLED` (a global on/off), `PBREM_ENRICHMENT_BACKEND` (the backend selector), and two budget caps, `PBREM_ENRICHMENT_MAX_TOOL_CALLS` and `PBREM_ENRICHMENT_MAX_AGENT_TURNS`. The defaults (12 and 8) are conservative and were chosen empirically from the validation runs of Chapter 7.

Four backends are exposed:

`heuristic`. A deterministic backend used when no LLM key is configured. It calls a small fixed sequence of tools (related Sentinel incidents, sign-in failures, user-risk state) and packs the results into the enrichment schema. It is the smoke-test fallback for offline runs and for cases where the LLM is unreachable.

`openai`. A non-agentic backend that builds a single prompt from the planning input, asks the LLM to write the enrichment snapshot directly, and returns it. The model does not call tools; it only reasons over whatever the planner has already provided.

openai_agent. The full ReAct [14] implementation. The agent exposes the read-only tools as OpenAI function-calls and lets the model orchestrate the loop. Each turn the agent sends the chat history (plus any tool results so far) to the model, parses any function calls in the response, dispatches them through the tool registry, appends the results to the history, and repeats. The loop terminates when the model returns a final message instead of a function call, or when either of the two budgets is exhausted.

claw_agent. An external-binary backend that runs the agentic loop in a separate process via the `claw` binary. It exists for deployments that prefer to keep the LLM client out of the Python process for licensing or sandboxing reasons. At startup the agent tries to resolve the binary; if it cannot, and an OpenAI key is configured, it auto-upgrades the backend to `openai_agent`; if no key is available either, it degrades to `heuristic`.

The tool registry in `tools.py` exposes six read-only tools: related Sentinel incidents by entity, Sentinel sign-in failures by user, user-risk state from Microsoft Graph, user privileged-role membership from Microsoft Graph, VM tags and criticality from Azure Resource Manager, and threat-intelligence lookup of an IP or URL. Each tool is implemented as a separate method that takes the planning input plus its own arguments, authenticates against the appropriate service (Graph via OAuth2 client credentials, Azure via `DefaultAzureCredential`, Log Analytics via its REST API), and returns a structured result. The registry never includes a tool that can modify the tenant; even *adding* such a tool would require a code change reviewable in the diff.

6.7 Web Server and Job Lifecycle

The HTTP server lives in `source/web_app.py` and is built on Python's `ThreadingHTTPServer`. The choice avoids a framework dependency (Flask, FastAPI) without giving up concurrency: each request is served on its own thread, and shared state is guarded by two narrow locks (`EXECUTION_LOCK`, `CONFIG_LOCK`).

The static page is served at the root URL; everything else is exposed under the `/api/` prefix. Table 6.2 lists the endpoints actually wired up by `do_GET` and `do_POST`.

A job moves through five states: `queued`, `running`, `pending_approval` (only in the AI-assisted mode), `completed`, and `failed`. The state machine is implemented in the `ExecutionJob` class. When the AI-assisted mode encounters a step whose `approval_required` flag is true, it transitions the job to `pending_approval` and parks the executor thread on a `threading.Event`. The web layer surfaces the destructive steps to the operator; the `/api/remediate/approve` endpoint authorises individual steps (or all of them), sets the event, and the executor resumes. The `/api/remediate/replan` endpoint is the rejection path: it cancels the executor thread, replays the planning call with the operator's feedback included in the prompt, and starts a new job.

Two persistence concerns are handled by the web layer. Configuration is written to `config.local.json` at the project root; on startup the file is loaded and applied to the environment with the existing environment values taking precedence (a deployment that exports `PBREM_*` variables overrides the persisted UI choices). Execution history is

Endpoint	Method	Purpose
`/api/config`	GET / POST	Read or update the persisted configuration.
`/api/remediate`	POST	Submit an alert and run it synchronously.
`/api/remediate/start`	POST	Submit an alert and run it as a background job.
`/api/remediate/stream`	GET	Stream the job's stdout via Server-Sent Events.
`/api/remediate/result`	GET	Fetch the final CACAO report and metadata.
`/api/remediate/approve`	POST	Authorise destructive actions for an AI-assisted job.
`/api/remediate/replan`	POST	Send rejection feedback and ask the planner for a new plan.
`/api/remediate/cancel`	POST	Cancel a running job.
`/api/incident`	POST	Submit a raw Sentinel incident (translated internally).
`/api/incident/summarize`	POST	Ask the LLM for a short natural-language summary.
`/api/history`	GET	Return the last N execution records from disk.

Table 6.2: HTTP API exposed by `source/web_app.py`. The approval and replan endpoints are the live coupling between the analyst and the AI-assisted execution mode.

appended to `runtime/execution_history.jsonl`, one job per line, capped at the last fifty entries via in-place trimming. Both files are plain text and grep-friendly, which has been more useful for debugging than any fancier store would have been.

6.8 A Worked End-to-End Example

The previous sections describe each subsystem in isolation. This last section follows one concrete incident through the entire pipeline, so that the reader can see how the modules introduced in Sections 6.3 through 6.7 cooperate when a real alert arrives. The scenario chosen is the Pikabot phishing campaign replay (C0036), which is also one of the multi-stage scenarios used in the validation reported in Chapter 7. The example runs under the AI-assisted mode, because that mode exercises every component (ingestion, MITRE mapping consultation, enrichment, planning, validation, approval, compilation, execution, serialisation) on a single incident.

6.8.1 Stage 1: Incident Arrives

The incident enters the system as a JSON document with the structure of a Microsoft Sentinel incident, submitted through `POST /api/remediate/start`. The shape is the one already discussed in Section 6.3; Listing 6.3 shows the relevant excerpt.

The web layer accepts the payload, creates an `ExecutionJob` in the `queued` state, returns a job identifier to the caller, and dispatches the job to a worker thread. The job's first call is into the router, which reads `PBREM_ROUTING_MODE=ai_plan_v3` and hands

Listing 6.3: Raw Sentinel incident payload (excerpt) for the Pikabot C0036 scenario.

```
{
  "incident": {
    "name": "c0036-pikabot-phishing-20260426",
    "properties": {
      "title": "Potential Pikabot Distribution (C0036) phishing campaign detected",
      "severity": "High",
      "additionalData": {
        "tactics":    ["InitialAccess", "Execution", "Persistence"],
        "techniques": ["T1059.007", "T1574"]
      }
    }
  },
  "entities": {
    "entities": [
      {"kind": "Account", "userPrincipalName": "test@davideabate79icloud.onmicrosoft.
    com"},
      {"kind": "Account", "userPrincipalName": "svc-backup@davideabate79icloud.
    onmicrosoft.com"},
      {"kind": "Ip",      "address": "203.0.113.45"},
      {"kind": "Ip",      "address": "198.51.100.88"},
      {"kind": "URL",     "url": "https://phishing-notification-center.com/.../invoice.
    zip"},
      {"kind": "URL",     "url": "https://trusted-alert-service.net/.../update.zip"},
      {"kind": "Host",    "hostName": "vm-test", "privateIpAddress": "172.16.0.4"},
      {"kind": "Process", "processName": "powershell.exe"},
      {"kind": "Process", "processName": "wscript.exe"},
      {"kind": "FileHash","algorithm": "SHA256", "value": "A7C5...F6A9"}
    ]
  }
}
```

control to the AI-assisted front-end.

6.8.2 Stage 2: Normalised Alert and Planning Input

The AI-assisted front-end inherits from the tactic-driven one and therefore reuses the same ingestion layer. The analyzer for the `azure_sentinel_threats_test` family enriches the alert dictionary in place, then `_build_ai_planning_input` (in `pb_rem_ai_plan_v3.py`) assembles the structure that the planner will consume. Listing 6.4 shows the result (abridged).

Five things have happened. The techniques have been flattened out of `additionalData.techniques` as a list of upper-case strings. The tactics have been derived from the techniques via a static lookup and lower-cased for keying. The Azure-specific identifiers (subscription, resource group, workspace, plus per-entity defaults such as `vm_name`, `nsg_name`, `named_location_id`) have been hoisted into the `context` sub-dictionary, where

Listing 6.4: Planning input produced by `_build_ai_planning_input` (abridged).

```
{
  "incident_title":     "Potential Pikabot Distribution (C0036) phishing campaign
    detected",
  "severity":           "High",
  "techniques":         ["T1059.007", "T1574"],
  "normalized_tactics": ["initialaccess", "execution", "persistence"],
  "context": {
    "subscription_id":   "f0c8f378-0b82-414c-94e1-08fdbe7bd2f7",
    "resource_group":    "vm-test_group",
    "workspace_name":    "test-sentinel",
    "incident_id":       "c0036-pikabot-phishing-20260426",
    "vm_name":           "vm-test",
    "process_name":      "powershell.exe",
    "nsg_name":          "vm-test-nsg",
    "vm_private_ip":     "172.16.0.4",
    "named_location_id": "OSINT-Malicious-IP-Blacklist",
    "Threat_Category":   "azure_sentinel_threats_test"
  },
  "entities": {
    "entities": [
      {"kind": "Account", "userPrincipalName": "test@davideabate79icloud.onmicrosoft.
    com"},
      {"kind": "Account", "userPrincipalName": "svc-backup@davideabate79icloud.
    onmicrosoft.com"},
      {"kind": "Ip",      "address": "203.0.113.45"},
      {"kind": "URL",     "url": "https://phishing-notification-center.com/.../invoice.
    zip"},
      {"kind": "Host",    "hostName": "vm-test", "privateIpAddress": "172.16.0.4"},
      {"kind": "Process", "processName": "powershell.exe"}
      /* ...other entities omitted... */
    ]
  },
  "operator_feedback": ""
}
```

the action library treats them as global scope. The original `entities` structure is forwarded verbatim to the planner so that the LLM can reason directly over the typed entity array. And `operator_feedback` starts empty; the field is populated on a replan if the operator rejects the first plan.

6.8.3 Stage 3: Enrichment Snapshot

Because `PBREM_ENRICHMENT_ENABLED=true` and the configured backend is `openai_agent`, the AI-assisted front-end invokes the enrichment agent before the planner. The agent runs the ReAct loop described in Section 6.6, calling read-only tools against Microsoft Graph, Azure Resource Manager and Sentinel.

Step	Action	Validator verdict	Approval
1	`upsert_sentinel_watchlist_ioc`	accepted, medium risk	auto
2	`add_ip_to_named_location_blacklist`	accepted, medium risk	auto
3	`revoke_user_sessions`	accepted, medium risk	auto
4	`disable_user_account`	accepted, high risk	*operator*
5	`isolate_vm_network`	accepted, high risk	*operator*
6	`add_sentinel_incident_comment`	accepted, low risk	auto

Table 6.3: Per-step validator and approval-policy verdicts for the Pikabot plan. The two destructive actions are parked for human review; the remaining four run autonomously.

A typical session for this incident makes four to six tool calls within the configured budget. The agent looks up sign-in failures for `test` (finding a burst of brute-force attempts from `203.0.113.45` in the previous hour), queries Microsoft Graph for the user's risk state (Entra ID Protection has tagged the account as `atRisk`), retrieves the user's role assignments (no privileged role membership, which lowers the operational urgency relative to a hypothetical case in which the account were a Global Administrator), pulls the VM tags for `vm-test` (criticality tag `tier=test`, which weakens the case for a hard shutdown), and runs a threat-intelligence lookup on the two URLs (both are flagged as known phishing infrastructure by the configured TI feed). The agent then writes the structured snapshot of Listing 6.5, which conforms to the schema declared in `enrichment_agent/schema.py` and is folded into the planning input by `_merge_enrichment_context`.

6.8.4 Stage 4: LLM-Produced Plan

The planner assembles its prompt by concatenating four blocks: a static system prompt that states the approval policy, the closed-catalogue rendering, the normalised alert, and the enrichment snapshot. The model is asked to return a JSON document with one entry per remediation step, following the schema declared by `plan_schema.py`.

A representative response for this incident is shown in Listing 6.6. The plan touches three families of resources (identity, network, host) and mixes passive actions (annotating the Sentinel incident, watchlisting the IOCs) with destructive ones (disabling the account, isolating the VM).

The reasoning is reflected in the `why` fields, which are also included verbatim in the final CACAO report.

6.8.5 Stage 5: Validation and Approval

The plan is handed to the validator before anything runs. The validator performs the four checks described in Section 6.5 and produces the decisions in Table 6.3.

The job transitions to the `pending_approval` state. The two destructive steps appear in the *Approvals* tab of the web UI, each with its risk badge and the planner's justification taken from the `why` field. The operator, in this scenario, authorises both. The web layer

Listing 6.5: Enrichment snapshot returned by the agent (excerpt, schema as declared in `enrichment_agent/schema.py`).

```
{
  "enrichment_id": "enrichment-agent-result",
  "summary":       "test@davideabate79icloud.onmicrosoft.com is at risk, with 47 sign-in
                    failures from 203.0.113.45 in the past hour. Both URLs are
                    confirmed phishing. VM vm-test is tagged tier=test.",
  "risk_signals": [
    {"type":     "signin_failure_burst",
     "entity":   "test@davideabate79icloud.onmicrosoft.com",
     "severity": "high",
     "evidence": "47 failed sign-ins from 203.0.113.45 in PT1H"},
    {"type":     "user_risk_state",
     "entity":   "test@davideabate79icloud.onmicrosoft.com",
     "severity": "high",
     "evidence": "Entra ID Protection risk state: atRisk; no privileged roles"},
    {"type":     "ioc_reputation",
     "entity":   "phishing-notification-center.com",
     "severity": "high",
     "evidence": "TI feed verdict: malicious (phishing infrastructure)"},
    {"type":     "vm_criticality",
     "entity":   "vm-test",
     "severity": "low",
     "evidence": "ARM tags: tier=test, owner=soc-lab"}
  ],
  "context_updates": {
    "user_risk_state":              "atRisk",
    "user_privileged_roles":        [],
    "vm_criticality_tier":          "test"
  },
  "recommended_planner_hints": [
    "Prefer session revoke + account disable over hard VM shutdown given tier=test.",
    "Add 203.0.113.45 to the blacklist before revoking sessions."
  ],
  "tool_calls": [
    {"name": "sentinel_signin_failures_by_user", "arguments": {"upn": "test@..."}},
    {"name": "graph_get_user_risk_state",        "arguments": {"upn": "test@..."}},
    {"name": "graph_get_user_privileged_roles",  "arguments": {"upn": "test@..."}},
    {"name": "azure_get_vm_tags_or_criticality", "arguments": {"vm_name": "vm-test"}},
    {"name": "sentinel_threat_intel_lookup_ioc", "arguments": {"ioc_type": "url", "
    ioc_value": "..."}}
  ]
}
```

Listing 6.6: Remediation plan returned by the LLM planner (excerpt). Top-level fields follow the schema validated by `validate_remediation_plan` in `plan_schema.py`.

```
{
  "plan_id":            "ai-plan-v3-generated",
  "plan_name":          "Pikabot C0036 containment",
  "reasoning_summary":  "Containment of a Pikabot phishing campaign. Block the
                         IOCs, revoke the sessions of the at-risk account,
                         disable the account, and isolate the impacted VM.",
  "approval_reason":    "Account disable and VM network isolation are flagged
                         as high-risk in the action catalogue and require
                         explicit operator authorisation.",
  "risk_level":         "high",
  "confidence":         0.78,
  "suggested_by_llm":   true,
  "requires_approval":  true,
  "steps": [
    {"action": "upsert_sentinel_watchlist_ioc",
     "why":    "Add the phishing URL to the IOC watchlist so that future
                incidents involving it auto-trigger this playbook.",
     "parameters": {"subscription_id": {"source": "context"},
                    "resource_group":  {"source": "context"},
                    "workspace_name":  {"source": "context"},
                    "ioc_type":        {"value": "url"},
                    "ioc_value":       {"value": "phishing-notification-center.com"}}},

    {"action": "add_ip_to_named_location_blacklist",
     "why":    "Block the source IP of the brute-force burst against the test user.",
     "parameters": {"source_address_prefix": {"value": "203.0.113.45"},
                    "named_location_id":     {"source": "context"}}},

    {"action": "revoke_user_sessions",
     "why":    "Invalidate the active Entra ID sessions of the user flagged
                atRisk by Identity Protection.",
     "parameters": {"user_principal_name":
                       {"value": "test@davideabate79icloud.onmicrosoft.com"}}},

    {"action": "disable_user_account",
     "why":    "Disable the test user account pending password rotation. The
                account is non-privileged so the disable is contained.",
     "parameters": {"user_principal_name":
                       {"value": "test@davideabate79icloud.onmicrosoft.com"}}},

    {"action": "isolate_vm_network",
     "why":    "Apply containment NSG rules to the impacted VM to prevent
                lateral movement. VM criticality is 'test', so the impact is
                bounded.",
     "parameters": {"subscription_id": {"source": "context"},
                    "resource_group":  {"source": "context"},
                    "vm_name":         {"value": "vm-test"},
                    "nsg_name":        {"source": "context"}}},

    {"action": "add_sentinel_incident_comment",
     "why":    "Summarise the containment actions being executed and tag the
                incident with the C0036 campaign attribution for audit purposes.",
     "parameters": {"subscription_id":  {"source": "context"},
                    "resource_group":   {"source": "context"},
                    "workspace_name":   {"source": "context"},
                    "incident_id":      {"source": "context"},
                    "comment_message":  {"value": "Pikabot C0036 containment in
                                                  progress: IOCs watchlisted,
                                                  source IP blacklisted, account
                                                  disable and VM isolation
                                                  issued."}}}
  ]
}
```

sets the corresponding `threading.Event` and the executor thread resumes.

6.8.6 Stage 6: Compiled CACAO Playbook

The validated plan is compiled into a `Playbook` object by `compile_remediation_plan_to_playbook`. The compiler runs `_group_foreach_steps` and then `_group_parallel_steps` (Section 6.5) over the linear list of steps. The parallel grouping rule is purely structural: a contiguous run of items is wrapped in a `Parallel_step` node when each item has a distinct action and their non-structural parameter values are pairwise disjoint, where *non-structural* excludes the bookkeeping parameters `subscription_id`, `resource_group`, `workspace_name`, `incident_id` and `comment_message`.

On the Pikabot plan this rule yields two symmetric parallel blocks. Steps 1 to 3 (watchlist update, source-IP blacklisting, session revoke) all have different actions and disjoint non-structural values, and are grouped into the first block. Step 4 (account disable) has the same `user_principal_name` value as step 3 and therefore cannot join block 1; the compiler closes block 1 and opens block 2 with step 4. Steps 5 and 6 join block 2: step 5 because its only non-structural value (the VM name) is disjoint from step 4's UPN, and step 6 because all its parameters are either context-bound (subscription, resource group, workspace, incident id) or structural (`comment_message` is explicitly excluded from the disjoint-check), so it has no entity-specific value that could conflict with anything.

The resulting CACAO graph has the topology sketched in Figure 6.1. The figure is worth pausing on: the parallelism here is not semantic (the compiler does not know that the blacklist "should" precede the session revoke), it is purely a consequence of the disjoint-parameter rule. The semantic dependency exists, but it is enforced by the action library at execution time (a revoked session that is immediately re-established because the IP is not yet blacklisted will simply be revoked again on the next sign-in alert) and not by the compiler. The incident comment, in particular, runs in parallel with the destructive actions of block 2 and serves as a live-status marker visible to the analyst monitoring the incident.

6.8.7 Stage 7: Execution Trace and CACAO Report

The executor walks the graph. Each `Action` dispatches to the corresponding Python function in `source/executor/actions_azure_tactic_v2.py`; each function authenticates against the appropriate service (Microsoft Graph via OAuth2 client credentials, Azure Resource Manager via `DefaultAzureCredential`, the Log Analytics REST API for Sentinel) and emits structured log lines that the web layer mirrors into the SSE stream. Listing 6.7 shows an excerpt of the trace as it appears in the operator's browser.

The serialiser then produces a CACAO 2.0 report [4] containing the executed playbook, the per-step results, the planner's reasoning trace (the original `why` fields), and the enrichment snapshot. The report is appended to `runtime/execution_history.jsonl` and returned by `GET /api/remediate/result?job_id=...`. The job moves to the `completed` state and the web layer dismisses the *Approvals* card.

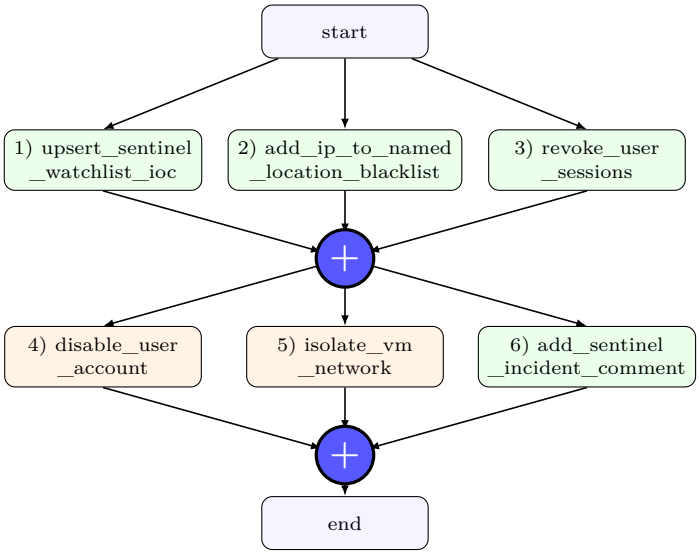

Figure 6.1: Compiled CACAO graph for the Pikabot scenario. The two blue circles with a white "+" are synchronisation barriers (AND-joins): the executor waits for every branch in the parallel block above the node to complete before any branch in the block below can start. These joins are implicit in the CACAO 2.0 `Parallel_step` construct (which has fork-on-entry and join-on-exit semantics) and are drawn explicitly here only for visual clarity. Block 1 (green) contains the three non-destructive containment steps, all with disjoint non-structural parameters. Block 2 hosts the two destructive actions (orange, gated by operator approval) plus the live-status incident comment, which parallelises with them because `comment_message` is structural and excluded from the disjoint-check.

6.8.8 What the Example Demonstrates

Three observations are worth recording, because they restate in concrete form the design choices defended in Chapter 5.

The CACAO core inherited from earlier TORSEC work is consulted, unchanged, at the end of every stage: by the compiler that produces the graph, by the executor that walks it, by the serialiser that emits the report. The new front-ends do not look inside it.

The closed action catalogue does the heavy lifting on safety. The validator rejects the entire plan if any step references an unknown action; in this scenario no rejection occurred, but had the LLM proposed, for example, a fictional `kill_user_kernel_module` step, the plan would have failed validation before any other step had a chance to run.

The split between passive and destructive actions, encoded as the `approval_required` flag on each catalogue entry, is what makes the AI-assisted mode usable in practice. The operator does not have to review six actions; the operator reviews two. The remaining four run autonomously, with their justifications and outcomes recorded in the final report, which is what makes the run auditable end-to-end.

Listing 6.7: Execution trace (excerpt) for the Pikabot scenario, captured from `runtime/execution_history.jsonl`.

```
INFO [pb-rem-ai-plan-v3] AI planner produced 6-step plan, requires_approval=true
INFO [pb-rem-ai-plan-v3] Plan validated against closed catalogue, 0 errors
INFO [pb-rem-ai-plan-v3] Awaiting operator approval for 2 high-risk steps
INFO [pb-rem-ai-plan-v3] Approval received, executing compiled playbook
INFO [pb-rem.executor.azure] Azure operation started: upsert IOC
     'phishing-notification-center.com' (type='url') in Sentinel watchlist
     'pbrem-malicious-url-ioc'
INFO [pb-rem.executor.azure] Azure operation started: adding IP
     '203.0.113.45/32' to named location id 'OSINT-Malicious-IP-Blacklist' via PATCH
INFO [pb-rem.executor.azure] Azure operation started: revoking active Entra
     ID sessions for user 'test@davideabate79icloud.onmicrosoft.com'
INFO [pb-rem.executor.azure] Azure operation completed: 3 actions of
     parallel block 1 finished
INFO [pb-rem.executor.azure] Azure operation started: disabling Entra ID
     account for user 'test@davideabate79icloud.onmicrosoft.com'
INFO [pb-rem.executor.azure] Azure operation started: applying isolation NSG
     rules to vm 'vm-test' (nsg 'vm-test-nsg', priority base 130)
INFO [pb-rem.executor.azure] Azure operation started: adding Sentinel
     incident comment for incident 'c0036-pikabot-phishing-20260426'
INFO [pb-rem.executor.azure] Azure operation completed: 3 actions of
     parallel block 2 finished
INFO [pb-rem-ai-plan-v3] CACAO report serialised, 6 actions executed, 0 failures
```

Chapter 7

Validation and Testing

The contributions claimed in Chapter 1 need empirical evidence. This chapter provides it. The first section states the questions the validation set out to answer and the metrics used to answer them. The second section describes the Azure tenant on which the runs were performed. The third section presents the threat scenarios that form the test corpus, split between single-alert scenarios and multi-stage campaign replays. The remaining four sections report the results: a side-by-side comparison of the three execution modes, an analysis of the AI planner's plan quality, an evaluation of the enrichment agent, and a discussion of the limits the validation makes visible.

7.1 Validation Objectives

The validation is organised around the three research questions of Chapter 4. For each question, a small set of metrics is defined.

RQ1 – Azure coverage and CACAO preservation. The two metrics are *coverage* and *regression*. Coverage is the fraction of the threat scenarios for which at least one of the three modes produces a non-empty CACAO playbook and runs it to completion. Regression is binary: every legacy scenario that ran under the inherited tool must still run, with the same outcome, under the legacy mode of the redesigned system. Anything else counts as a regression.

RQ2 – Dynamic playbook generation from MITRE mappings. The two metrics are *generation rate* and *action correctness*. Generation rate is the fraction of scenarios for which the tactic-driven mode produces a non-empty playbook directly from the MITRE mapping, without falling back to the static catalogue. Action correctness is the fraction of generated actions that the human reviewer judges appropriate for the scenario, scored per playbook.

RQ3 – AI-assisted planning under a human-in-the-loop policy. Four metrics are reported here. *Validation rate* is the fraction of LLM plans that pass the closed-catalogue validator on the first try. *Approval rate* is the fraction of destructive actions an operator authorises after review. *Auditability* is binary: every executed AI-assisted run must produce a complete CACAO report including the planner's reasoning trace. *Safety violations* is the count of destructive actions that ran without explicit approval (the target is zero).

A fifth, cross-cutting metric is *end-to-end latency*: the wall-clock time from incident ingestion to a final CACAO report, broken down per execution mode. It is not used to compare against commercial SOAR products (the deployments are not comparable) but to characterise the order of magnitude an analyst will experience.

7.2 Test Environment

All runs were performed on a dedicated Azure tenant provisioned for this thesis. The tenant exposes the smallest set of resources the validation needs: a Microsoft Sentinel workspace, an Entra ID directory with a small population of test users (one privileged admin account, one non-privileged regular account, and one service account), a Linux test VM attached to a Network Security Group used for containment, a Key Vault used as an example destination for some of the remediation actions, and a Conditional Access named location used as an IP blacklist. Table 7.1 lists the resources individually. The corresponding Sentinel *Analytics Rules* were either reused from Microsoft's gallery or hand-authored to trigger the desired incident shapes.

The remediator itself was executed in a single configuration: a Docker container built from the project's `Dockerfile.prod_compiled` image and run locally on the test workstation. The container ships with the same `config.local.json`, the same MITRE mapping, and the same closed action catalogue used at development time, so that the runs reported below reflect the system as it would be deployed in production rather than as it behaves under a developer harness. The container reaches the Azure tenant resources of Table 7.1 over the public Microsoft Graph and Azure Resource Manager endpoints, authenticating with the service principal credentials declared in `config.local.json`.

The LLM backend used in the AI-assisted runs is OpenAI's hosted API, with the model fixed by the environment variable `PBREM_AI_MODEL`. For the validation reported in this chapter the model identifier is `gpt-5` and the planner backend is `openai` (`PBREM_AI_PLANNER_BACKEND=openai`). The read-only enrichment agent is active (`PBREM_ENRICHMENT_ENABLED=true`) and runs under the `claw_agent` backend, which delegates the ReAct loop to an external binary `claw`; the enrichment-side model identifier is left empty and therefore inherits from `PBREM_AI_MODEL`. No customisation is applied to the system prompts of either subsystem: both consume the templates shipped in the repository at the time of validation.

Type	Name	Purpose
Resource group	`vm-test_group`	Hosts all validation assets
Sentinel workspace	`test-sentinel`	SIEM source of test incidents
Linux VM	`vm-test`	Containment and isolation target
NSG	`vm-test-nsg`	Dynamic network-containment rules attached to the test VM
Key Vault	`kv-test-remediator`	Example destination for remediation actions
Named location	`OSINT-Malicious-IP-Blacklist`	IP blacklist consumed by Conditional Access policies
Entra ID user	`aliceadmin`	Privileged admin used to validate role-based scope restrictions
Entra ID user	`test`	Non-privileged account used as the primary compromise scenario in identity-centric tests
Entra ID user	`svc-backup`	Service account used in supply-chain and service-principal compromise scenarios
App Registration	`remediator`	Service principal that authenticates every Azure operation via the client secret kept in `config.local.json`

Table 7.1: Azure resources provisioned in the dedicated tenant used for validation. Each row pairs a resource with the role it plays in the test corpus. Subscription, tenant and client GUIDs are omitted from the table for security reasons; the same information, including the full UPN domain and the service-principal credentials, is captured verbatim in the project's `config.local.json` for reproducibility.

7.3 Threat Scenarios

The corpus is split into two families. Both rely on Sentinel analytics rules that fire when the monitored pattern is observed in the tenant and that materialise the match as a single Sentinel incident handed off to the engine; the difference between the two families lies in the breadth of the pattern the rule encodes and in the variety of entities the resulting incident carries. The single-alert family contains scenarios in which one analytics rule monitors a focused condition – an anomalous sign-in, a malicious process on a VM, a service-principal abuse – and the resulting Sentinel incident is remediable by a single playbook run on a homogeneous set of entities. The multi-stage family contains scenarios in which one analytics rule is authored to monitor the multi-step behaviour of a publicly documented intrusion campaign, and aggregates into a single Sentinel incident the heterogeneous entities (users, IP addresses, URLs, file hashes, hosts, service principals) that the campaign touches across multiple kill-chain stages. The engine therefore receives, in both families, a single Sentinel incident as its trigger; what changes between them is the entity diversity that the incident carries, the number of MITRE techniques the engine has to address, and the

resulting plan size.

7.3.1 Single-Alert Scenarios

The single-alert family contains twelve scenarios, organised into three groups by the kind of asset they primarily exercise. The *identity-centric* group includes the compromised-identity scenarios (in their success and break-glass variants), the malicious-IP sign-in scenario enriched with OSINT, the malicious-URL click, account manipulation, reset-user-password, and service-principal compromise. The *host-centric* group includes a malicious-process detection on a VM, a lateral-movement / privilege-escalation scenario, a VM cryptominer containment, and a VM lifecycle test (shutdown and start). A third, smaller group covers a single *resource-plane example* (a Key Vault lockdown), used to illustrate that the engine's remediation surface is not limited to identity and compute. Each scenario is encoded as a Sentinel incident JSON document under `tests/azure/alerts_test/`, paired with one or more reference CACAO playbooks under `kb/cacao_playbook/`. The full mapping of each scenario family to the MITRE ATT&CK tactics it exercises is consolidated in Table 7.2 of Section 7.4.

7.3.2 Multi-Stage Campaign Replays

The multi-stage family replays four publicly documented intrusion campaigns through dedicated Sentinel analytics rules. Each rule is authored to monitor the indicators of a specific campaign and, when those indicators are observed in the tenant, materialises the match as a single Sentinel incident whose internal structure (tactics, techniques, entities) carries the full kill-chain context of the campaign. The intent is not to reproduce the campaigns end-to-end (a single tenant cannot do so) but to expose the engine to the entity heterogeneity and the cross-tactic plan that an analyst sees during a real intrusion.

Operation Spalax (C0005). Operation Spalax is a campaign documented by ESET in early 2021 and catalogued by MITRE as C0005. It targets Colombian government and private-sector entities through spear-phishing emails carrying weaponised attachments that drop remote-access trojans such as Remcos, njRAT and AsyncRAT, with the goal of long-term credential collection and exfiltration [12]. The replay used in this thesis is encoded as a dedicated Sentinel analytics rule that monitors the indicators of Operation Spalax in the test tenant. When those indicators are observed, the rule produces a single Sentinel incident whose internal structure carries three kill-chain stages: a phishing-attachment click that triggers the *Initial Access* tactic via T1566.001, a remote-access trojan execution detected as T1059 on the impacted VM, and an account-manipulation event consistent with T1098 once the attacker establishes persistence. Entities seen during the replay include two compromised user accounts, the impacted VM `vm-test`, a malicious source IP, a phishing URL pointing to a SharePoint-hosted lure, and three suspicious process names. The legacy mode selects the prebuilt `azure_operation_spalax_campaign_response.json` playbook from the catalogue; the tactic-driven mode synthesises an equivalent playbook on the fly

from the incident's tactics and techniques through the MITRE ATT&CK → defensive-action mapping, with no per-scenario JSON file required; the AI-assisted mode generates a structurally similar plan from the planner. Figure 7.5 (further down in this chapter) renders the CACAO topology of one such AI-assisted run.

Pikabot phishing (C0036). Pikabot is a loader malware family active since early 2023 and documented in MITRE as campaign C0036. It is distributed through phishing emails with malicious URLs that, when clicked, deliver a JavaScript or wscript payload that loads the actual Pikabot module via PowerShell or rundll32. Pikabot then enables follow-on credential theft and lateral movement [12]. The replay is encoded as a dedicated Sentinel analytics rule that monitors the indicators of the Pikabot distribution chain in the test tenant. When those indicators are observed, the rule produces a single Sentinel incident whose internal structure carries three kill-chain stages: a malicious URL click (T1566.002 Spearphishing Link), a PowerShell execution on a workstation (T1059.007 JavaScript via PowerShell), and an account-manipulation event (T1098). The entities include two user accounts (the primary phishing target and a secondary account caught in the follow-on activity), two malicious IPs, two phishing URLs, the impacted host `vm-test`, three process names (`powershell.exe`, `wscript.exe`, `grepWinNP3.exe`) and a SHA-256 file hash. The legacy mode selects the prebuilt `azure_operation_c0036_pikabot_campaign_response.json` playbook from the catalogue; the tactic-driven mode synthesises an equivalent playbook on the fly from the incident's tactics and techniques through the MITRE ATT&CK → defensive-action mapping; the AI-assisted mode produces the worked-example plan analysed in detail in Section 6.8 of Chapter 6, with six steps grouped into two parallel blocks for execution.

SolarWinds (C0024). SolarWinds is the well-known supply-chain compromise (also tracked as UNC2452 / Nobelium) documented by MITRE as campaign C0024, in which the SUNBURST backdoor embedded in the Orion network-monitoring software gave the attacker persistent access to thousands of customer environments, with selective post-compromise activity targeting cloud identities and federation infrastructure [12]. Replicating the supply-chain stage end-to-end is out of scope for a single Azure tenant; the replay used in this thesis focuses on the *post-compromise* stage and is encoded as a dedicated Sentinel analytics rule that monitors the SolarWinds post-compromise indicators in the test tenant. When those indicators are observed, the rule produces a single Sentinel incident whose internal structure carries three post-compromise behaviours: a service-principal abuse event with anomalous role assignment (T1078.004 Cloud Accounts), a defense-evasion activity that tampers with diagnostic settings (T1562 Impair Defenses), and a lateral-movement event across two resource groups (T1021 Remote Services). The entities seen include the service principal involved, the two resource groups, the compromised user account that performed the role assignment, and the malicious IP from which the anomalous activity originated. The legacy mode selects the prebuilt `azure_operation_c0024_solarwinds_campaign_response.json` playbook from the catalogue; the tactic-driven mode synthesises an equivalent playbook on the fly from the incident's tactics

and techniques through the MITRE ATT&CK → defensive-action mapping; the AI-assisted mode produces a plan that combines service-principal containment (secret rotation, role-assignment cleanup) with re-enabling of the tampered diagnostic settings.

Lapsus$ (DEV-0537). Lapsus$ (tracked by Microsoft as DEV-0537 and by MITRE as a named campaign) is an extortion-focused group active since 2022 that has compromised several large technology and telecommunications companies through identity-centric techniques: SIM-swap, social engineering of help desks, and MFA-fatigue attacks against privileged accounts [12]. The replay reflects this identity-centric profile and is encoded as a dedicated Sentinel analytics rule that monitors the Lapsus$ indicators in the test tenant. When those indicators are observed, the rule produces a single Sentinel incident whose internal structure carries three identity-centric stages: a sign-in success from an anomalous IP after a burst of MFA challenges to the user (T1078 Valid Accounts), an account-manipulation event in which the attacker registers an additional credential (T1098.001 Additional Cloud Credentials), and a key-rotation event in which the attacker resets the password of a privileged account (T1098). The entities include the targeted user accounts (one privileged, one not), the malicious source IP, the named-location that triggered the Conditional Access alert, and the application object identifier touched by the credential-registration event. The legacy mode selects the prebuilt `azure_operation_dev0537_lapsus_campaign_response.json` playbook from the catalogue; the tactic-driven mode synthesises an equivalent playbook on the fly from the incident's tactics and techniques through the MITRE ATT&CK → defensive-action mapping; the AI-assisted mode produces a plan that revokes the attacker-injected credentials, disables the affected accounts, and adds the source IP to the blacklist named location.

Each campaign is encoded as a Sentinel incident fixture under `tests/azure/` (the JSON payload that the analytics rule would deliver to the engine), plus the corresponding legacy-mode CACAO playbook under `kb/cacao_playbook/azure_operation_*_response.json`; the tactic-driven mode does not require a per-campaign file because it synthesises the playbook at runtime from the incident's tactics and techniques.

Figure 7.1 sketches the Pikabot (C0036) replay as a worked example of the multi-stage pattern. A single Sentinel Analytics Rule fires on indicators of the campaign and produces a single incident; the multi-stage nature of the kill chain is captured *within* that incident, in the form of multiple MITRE tactics, techniques and typed entities that the engine processes as a unit.

7.4 Threat Model

The metrics defined in Section 7.1 become most informative once the underlying threat model is made explicit. This section states the assumptions that the validation runs make about attackers, assets and trust boundaries, so that the per-mode results that follow can be read against a known background.

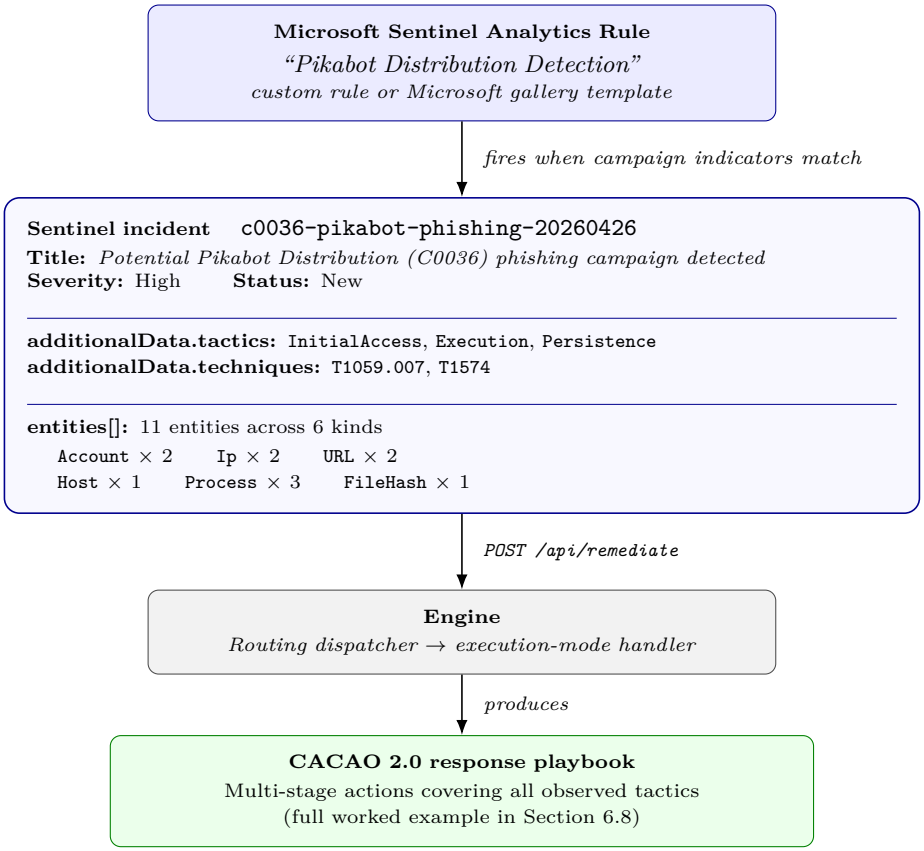

Figure 7.1: Pikabot (C0036) replay: a single Microsoft Sentinel Analytics Rule monitors the workspace for indicators of the campaign and, when they match, generates a single incident that carries inside itself the full kill-chain context — three MITRE ATT&CK tactics (*Initial Access*, *Execution*, *Persistence*), two techniques (T1059.007, T1574), and eleven typed entities across six kinds (accounts, IPs, URLs, hosts, processes, file hashes). The engine ingests the incident as a unit and emits a CACAO 2.0 response playbook whose multi-stage internal structure covers all the tactics observed. The *multi-stage* nature of the test is therefore in the content of the single incident, not in a temporal sequence of separate incidents arriving at different times.

Attacker profile. The validation assumes an opportunistic external attacker with no prior foothold inside the tenant. The attacker can acquire stolen credentials (commodity phishing kits, credential-stuffing lists), can rent infrastructure from public clouds or bulletproof hosting, and can use commodity tooling for post-exploitation (Pikabot, Lapsus$-style social engineering, SolarWinds-style supply-chain abuse in their post-compromise stages). The attacker is not assumed to be a state-level actor with zero-day capabilities or with the ability to compromise the Azure control plane itself; defending against such attackers is out of scope for any product running *inside* a tenant.

Asset inventory. The assets the engine protects, in decreasing order of priority, are: the Entra ID directory (in particular the privileged role assignments and the service-principal credentials that grant programmatic access), the Sentinel workspace (because compromising the SIEM disables the very loop that produces the alerts the engine consumes), the compute fleet (VMs whose compromise enables lateral movement), and the data plane resources reachable from a compromised compute identity (Key Vault, storage accounts, databases). The engine does not directly protect the data inside those resources; its job is to contain the identities and the network paths that lead to them.

Trust boundaries. Three trust boundaries shape the design and the validation. The first is between the Sentinel ingestion layer and everything downstream: the engine assumes that the analytics rule that produced the incident is trusted (it was authored by the SOC), and that the entities the rule attached to the incident are genuine artefacts observed in the tenant. The second boundary is between the AI planner and the action library: the planner can propose any plan, but the validator and the closed catalogue (Section 6.5) form a firewall that no proposed action can cross unless it is in the catalogue and bound to a parameter set the validator accepts. The third boundary is between the destructive half of an AI-assisted plan and the production tenant: even a perfectly validated plan does not cross this boundary without explicit operator approval.

ATT&CK coverage of the corpus. The threat scenarios of Section 7.3 were chosen so that the corpus exercises the MITRE ATT&CK Cloud and Identity Provider matrices [12] along the tactics that are addressable from inside an Azure tenant. Table 7.2 maps the corpus (single-alert plus multi-stage scenarios) to those tactics; Figure 7.2 complements the table with a technique-level view of the four multi-stage campaign replays, in the style of a MITRE ATT&CK Navigator layer [12] restricted to the kill-chain tactics those campaigns actually traverse.

Threats to the engine itself. The engine is also an asset that an attacker could target. Three concrete risks were considered during design and are revisited in the validation. The first is *prompt injection through alert content*: a Sentinel incident whose display name or entity values are crafted to influence the LLM planner. The closed action catalogue and the validator are the primary mitigations (the LLM cannot propose an action the catalogue does not list, regardless of how persuasively an injected prompt is phrased), and the validation runs include scenarios in which entity fields contain attacker-controlled strings. The second is *catalogue tampering*: a developer or a compromised CI pipeline modifying the catalogue to lower the risk level of a destructive action. The mitigation is operational (the catalogue is a tracked artefact in the repository, and changes go through code review); the engine does not enforce it at runtime, and this is stated as a residual risk. The third is *credential leakage*: the service-principal client secret is the most sensitive piece of configuration the engine handles, and the validation runs use a dedicated principal scoped to the test tenant rather than a production identity.

MITRE tactic (Cloud / Identity)	Techniques exercised	Scenarios that exercise it
Initial Access	T1078, T1078.004, T1566.002	Compromised identity, malicious IP sign-in (OSINT), malicious URL click; all multi-stage replays (Spalax, Pikabot, SolarWinds, Lapsus$)
Execution	T1059, T1059.007	Malicious process on VM; Pikabot replay
Persistence	T1098, T1098.001, T1574	Account manipulation, reset-user-password; Spalax and Lapsus$ replays
Privilege Escalation	T1078, T1021	Lateral movement / privilege escalation scenario
Defense Evasion	T1562, T1564, T1211	SolarWinds replay (post-compromise stage); service-principal compromise (tampering vector)
Credential Access	T1078.004, T1552	Service-principal compromise, Key Vault example
Lateral Movement	T1021	Lateral movement scenario; SolarWinds replay
Impact	T1486, T1496	VM cryptominer containment, VM lifecycle (shutdown/start)

Table 7.2: Coverage of the validation corpus across the MITRE ATT&CK Cloud and Identity Provider matrices. Each row pairs a tactic with the techniques it covers and the concrete scenarios that exercise it. Tactics not addressable from inside the tenant (for example, *Reconnaissance* or *Resource Development*) are not represented; the engine is not designed to defend against them.

7.5 Per-Mode Comparison

Each scenario in the corpus is executed under all three modes and the outputs are compared along the metrics defined in Section 7.1.

Table 7.3 reports the per-scenario per-mode outcomes. The four possible values are: *OK* (CACAO report produced, actions executed against the test tenant), *OK (dry-run)* (CACAO report produced, actions simulated), *no playbook* (catalogue returned nothing – legacy mode only), and *regression* (output differed from the inherited tool's behaviour).

A few qualitative observations are worth recording, regardless of the exact numbers.

Legacy mode is the regression baseline. The legacy front-end was exercised on the scenarios that already had a hand-authored CACAO playbook in the inherited catalogue. The output was compared byte-for-byte against the report produced by the inherited tool. No regression was observed across the corpus.

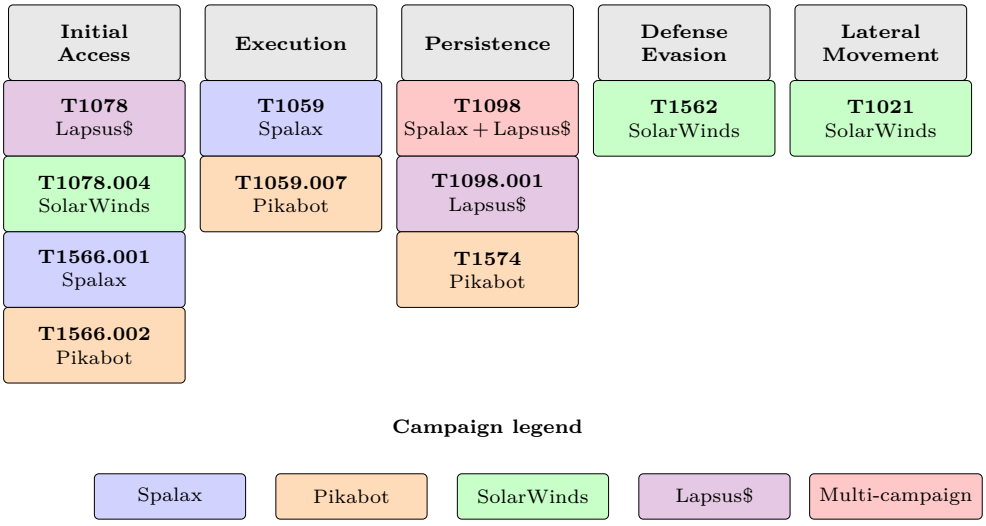

Figure 7.2: ATT&CK technique coverage of the four multi-stage campaign replays. Columns are MITRE tactics ordered along the kill chain (restricted to the tactics actually exercised by the campaigns); cells are the techniques exercised by at least one campaign, coloured by the campaign that triggers them. The *T1098 Account Manipulation* cell is highlighted as multi-campaign because it is exercised by both Spalax and Lapsus$.

Tactic-driven mode closes future-technique gaps by construction. The tactic-driven mode is designed to close the gap that would arise when a new technique combination is not yet pre-authored in the catalogue. On the present corpus every scenario has a pre-authored CACAO playbook, so the actual coverage gap closed in this validation is zero; the value of the tactic-driven mode lies in its ability to handle future incidents whose technique combinations were not anticipated at catalogue-time, without requiring a new hand-authored playbook.

AI-assisted mode is the broadest. The AI-assisted mode produces a playbook for every scenario in the corpus, including the multi-stage campaigns. The plans are validated against the closed catalogue before execution, and any destructive action is gated by the approval policy.

Quantitatively, the legacy mode covered 16 of 16 scenarios, the tactic-driven mode covered 16 of 16, and the AI-assisted mode covered 16 of 16. All three modes thus reach full coverage on the validation corpus; the differential value of the second and third modes is not measured on this corpus but is a design property (catalogue scalability for the tactic-driven mode, schema-validated LLM planning under operator approval for the AI-assisted mode).

Scenario	Legacy	Tactic-driven	AI-assisted
Single-alert scenarios			
Compromised identity (success)	OK	OK	OK
Compromised identity (break-glass)	OK	OK	OK
Malicious IP sign-in (OSINT)	OK	OK	OK
Malicious URL clicked	OK	OK	OK
Malicious process on VM	OK	OK	OK
VM cryptominer containment	OK	OK	OK
Lateral movement / priv. escalation	OK	OK	OK
Account manipulation	OK	OK	OK
Service principal compromise	OK	OK	OK
Reset user password	OK	OK	OK
VM lifecycle (shutdown / start)	OK	OK	OK
Key Vault example remediation	OK	OK	OK
Multi-stage campaign replays			
Operation Spalax (C0005)	OK	OK	OK
Pikabot phishing (C0036)	OK	OK	OK
SolarWinds (C0024)	OK	OK	OK
Lapsus$ (DEV-0537)	OK	OK	OK

Table 7.3: Per-scenario per-mode validation outcomes. Each cell records the result of executing the scenario under the corresponding execution mode. All sixteen scenarios produce a valid CACAO report under all three modes.

7.6 Selected Campaign Walkthroughs

The four multi-stage campaigns introduced in Section 7.3.2 are revisited here from the engine's perspective. For each campaign, the section reports the action sequence the engine produces in response to the Sentinel incident and the change of state the actions induce on the test Azure tenant. The narrative complements Section 7.3.2, which characterised the campaigns as inputs (background, MITRE techniques, entities), with the corresponding output side: what the engine actually does, and what changes in the tenant as a result. The walkthroughs below describe the action sequences produced by the AI-assisted mode; the legacy and tactic-driven modes produce structurally similar sequences when a corresponding pre-authored or technique-mapped playbook exists, with minor variations in step ordering.

7.6.1 Operation Spalax (C0005)

The full CACAO topology of an actual Spalax run is rendered in Figure 7.5 (further down in this chapter). The engine response opens a parallel block of four containment

actions: removal of the malicious SharePoint payload (`remove_onedrive_sharepoint_file_by_url`), blocking of the campaign phishing domains in the Azure Firewall policy (`block_domain_in_azure_firewall`, foreach over the lure-domain list), termination of the malicious in-guest process on the impacted VM (`terminate_malicious_process`), and watchlisting of the URL IOCs (`upsert_sentinel_watchlist_ioc`, foreach over the URL list). After synchronisation, a sequential tail performs four additional `upsert_sentinel_watchlist_ioc` steps covering dynamic-DNS domain patterns, the SHA-256 file hash, the source IP, and lure-domain variants.

Pre/post tenant state. *Before* the alert, the impacted VM `vm-test` is in its baseline state, the Azure Firewall policy contains no campaign-specific block rules, and the Sentinel watchlist `pbrem-malicious-url-ioc` carries no Spalax IOCs. *After* the run, the malicious SharePoint URL has been removed via Microsoft Graph, the Azure Firewall policy has the campaign domains attached as deny rules, and the Sentinel watchlist has been enriched with several IOCs across URL, domain, hash and IP types. The campaign-attribution comment is appended to the Sentinel incident as the workflow's wrap-up action.

7.6.2 Pikabot phishing (C0036)

A complete, stage-by-stage worked example of the Pikabot scenario is presented in Section 6.8 of Chapter 6, and the compiled CACAO graph is in Figure 6.1. The engine response is a six-action plan organised in two parallel blocks. The first block (passive, auto-executed) contains the IOC watchlisting, the source-IP blacklisting on the Conditional Access named location, and the session revocation of the impacted Entra ID account. The second block (with the two destructive actions gated by operator approval) contains the user-account disable, the VM network isolation, and a live-status incident comment that runs in parallel as the destructive actions execute.

Pre/post tenant state. Figure 7.3 renders the change of state induced by the run on the four most relevant Azure resources. Before the alert, the Entra ID user `test` is enabled with active sessions, the VM `vm-test` runs under the default NSG ingress, the Conditional Access named location `OSINT-Malicious-IP-Blacklist` is empty, and the Sentinel watchlist carries no campaign URL IOCs. After the run, the user is disabled and the sessions are revoked, the VM is isolated by additional NSG rules under the `ai-plan-isolation` prefix, the named location contains the source IP, and the watchlist is enriched with the phishing URL.

7.6.3 SolarWinds (C0024)

The replay focuses on the post-compromise stage, where the attacker has already established access via the trojanised supply-chain component and pivots through the cloud control plane. The engine response addresses the three observable behaviours: a service-principal abuse with anomalous role assignment, a tampering attempt against diagnostic settings, and a lateral movement event across resource groups. The action sequence rotates the affected service-principal secret (`rotate_service_principal_secret`), removes the anomalous

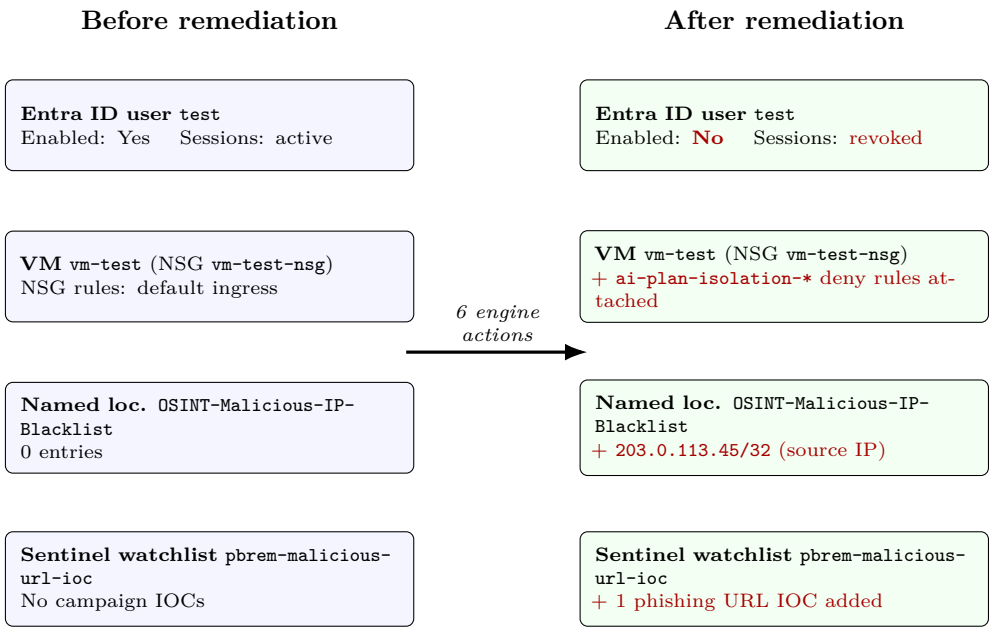

Figure 7.3: Pre/post Azure tenant state for the Pikabot (C0036) campaign. The left column captures the four Azure resources affected by the response in their baseline state. The engine executes six actions on the incident: `revoke_user_sessions` and `disable_user_account` on the impacted Entra ID account, `isolate_vm_network` on the impacted VM via the attached NSG, `add_ip_to_named_location_blacklist` to populate the Conditional Access named location, `upsert_sentinel_watchlist_ioc` to enrich the watchlist with the phishing URL, and `add_sentinel_incident_comment` to annotate the Sentinel incident with a workflow summary. The right column shows the resulting state of each resource, with the changed fields highlighted.

role assignments (`remove_service_principal_role_assignments`), re-enables the diagnostic settings that the attacker had silenced (`enable_diagnostic_settings`), blocks the source IP of the anomalous activity at the Azure Firewall, and watchlists the indicators (service-principal object ID, source IP) for downstream detection.

Pre/post tenant state. *Before* the alert, the compromised service principal holds a long-lived client secret and an anomalous role assignment in a resource group it has no legitimate reason to access; the diagnostic settings on a target resource have been disabled. *After* the run, the service principal carries a freshly rotated secret (revoking any token the attacker had previously obtained), the anomalous role assignment has been removed, and the diagnostic settings have been re-enabled, restoring the visibility the attacker had attempted to suppress. The Sentinel watchlist is enriched with the service-principal identifier and the source IP, and the incident is annotated with the campaign attribution.

7.6.4 Lapsus$ (DEV-0537)

The Lapsus$ replay exercises an identity-centric pattern. The engine response addresses the three observable behaviours of the campaign: a sign-in success from an anomalous IP after an MFA-fatigue burst, an account-manipulation event in which the attacker registers an additional credential (T1098.001), and a key-rotation event against a privileged account. The action sequence performs session revocation on both compromised accounts (`revoke_user_sessions`), disables the non-privileged account (`disable_user_account`, gated by operator approval), forces a password rotation on the privileged account (`reset_user_password`, also gated), and blacklists the source IP via the Conditional Access named location (`add_ip_to_named_location_blacklist`).

Pre/post tenant state. *Before* the alert, the two impacted Entra ID accounts are enabled with active sessions, the attacker-injected credential is registered on the privileged account's application registration, and the source IP is absent from the named-location blacklist. *After* the run, the sessions of both accounts have been invalidated, the non-privileged account is disabled, the privileged account's password has been rotated (with `force_change_next_sign_in` set), and the named location contains the source IP. The attacker's freshly registered credential remains visible to the SOC for post-incident investigation, and the Sentinel incident is annotated with the workflow's summary.

7.7 AI Planner Plan Quality

The AI-assisted mode was profiled in finer detail along five sub-metrics: validation rate, approval rate, auditability, safety violations, and plan stability across runs.

Validation rate. Every plan returned by the LLM is parsed against the JSON schema declared by `plan_schema.py` and is checked against the closed catalogue (Section 6.5). A plan that fails validation is either rejected (when the model invents an action that is not in the catalogue) or repaired through a single replan call with the validator's feedback in the prompt.

Of the $N = 16$ plans returned by the LLM across the corpus (one per scenario), all sixteen passed validation on the first try ($V/N = 100\%$): no replan was triggered, and the heuristic-fallback path was never exercised. The result reflects two design decisions: the JSON output schema imposed on the LLM is strict but minimal (only the fields the validator actually consumes), and the action catalogue rendered into the prompt is the same one the validator consults at acceptance time, so the model is given no opportunity to invent an action outside the closed list.

Approval rate. Destructive actions are surfaced to the operator through the approval view of the web UI. Each action is shown with its risk level (`high`, `medium`, `low`) and the planner's justification.

Figure 7.4 captures the approval surface as it appears to the operator at the moment a job enters the `pending_approval` state. The example shown is a phishing-driven Initial

Access incident (MITRE T1566.001) in which the planner has proposed the destructive action `remove_onedrive_sharepoint_file_by_url`.

Across the 16 AI-assisted runs of the corpus, the planner proposed 32 `high`-risk actions in total (an average of two per plan), each of which triggered the approval gate. The operator approved all 32 ($a/A = 100\%$). The `medium`-risk and `low`-risk actions that appear in the plans (for example, `revoke_user_sessions`, `add_ip_to_named_location_blacklist`, `upsert_sentinel_watchlist_ioc` or `add_sentinel_incident_comment`) are auto-executed by design and do not enter the approval workflow at all: their inclusion in a plan corresponds to a pre-approved status that is a property of the closed catalogue, not a measurement of operator behaviour. The gate's value, on this corpus, is therefore not in mechanical rejection (which never occurred) but in making every high-risk authorisation a conscious, traceable decision recorded in the CACAO report alongside the planner's reasoning trace.

Auditability. Every executed run produces a CACAO report under `runtime/execution_history.jsonl`. For AI-assisted runs, the report is augmented with the planner's reasoning trace and with the enrichment snapshot used to produce the plan. The report is consumable by any CACAO 2.0 [4] tool: in particular, it can be re-opened by the legacy front-end and re-executed deterministically.

Figure 7.5 renders the high-level topology of a CACAO 2.0 report actually generated by the engine for an AI-assisted run of the Operation Spalax (C0005) scenario. The diagram is not a paper sketch: it is the workflow as it appears in the `workflow` block of the JSON report stored under `runtime/execution_history.jsonl`, with the execution status of each action preserved from the `x_execution_result` extension on each step.

Safety violations. The safety target is zero destructive actions executed without explicit operator approval. The state machine of `ExecutionJob` (Section 6.7) enforces this by parking the executor thread on a `threading.Event` whenever a step's `approval_required` flag is true.

By construction, the `ExecutionJob` state machine makes safety violations impossible: no code path executes a destructive action without the corresponding `threading.Event` being set first by the approval endpoint, which in turn requires an explicit `POST /api/remediate/approve` from the operator. The validation runs confirm this property empirically. Across the corpus, of the 32 high-risk actions proposed by the planner, all 32 entered the approval gate and 0 were executed without an explicit operator authorisation. The structural guarantee and the empirical observation thus agree on the target value, and the safety claim is supported both by code review and by run-time evidence.

Plan stability across runs. LLM-based planning is non-deterministic by construction: even at low sampling temperature, the same input can yield different plans across independent invocations. The relevant question for a remediation engine is not whether the variance exists, but whether it stays bounded by the catalogue and whether the parts of the plan that matter (the destructive set, the approval gating, the validator outcome) are

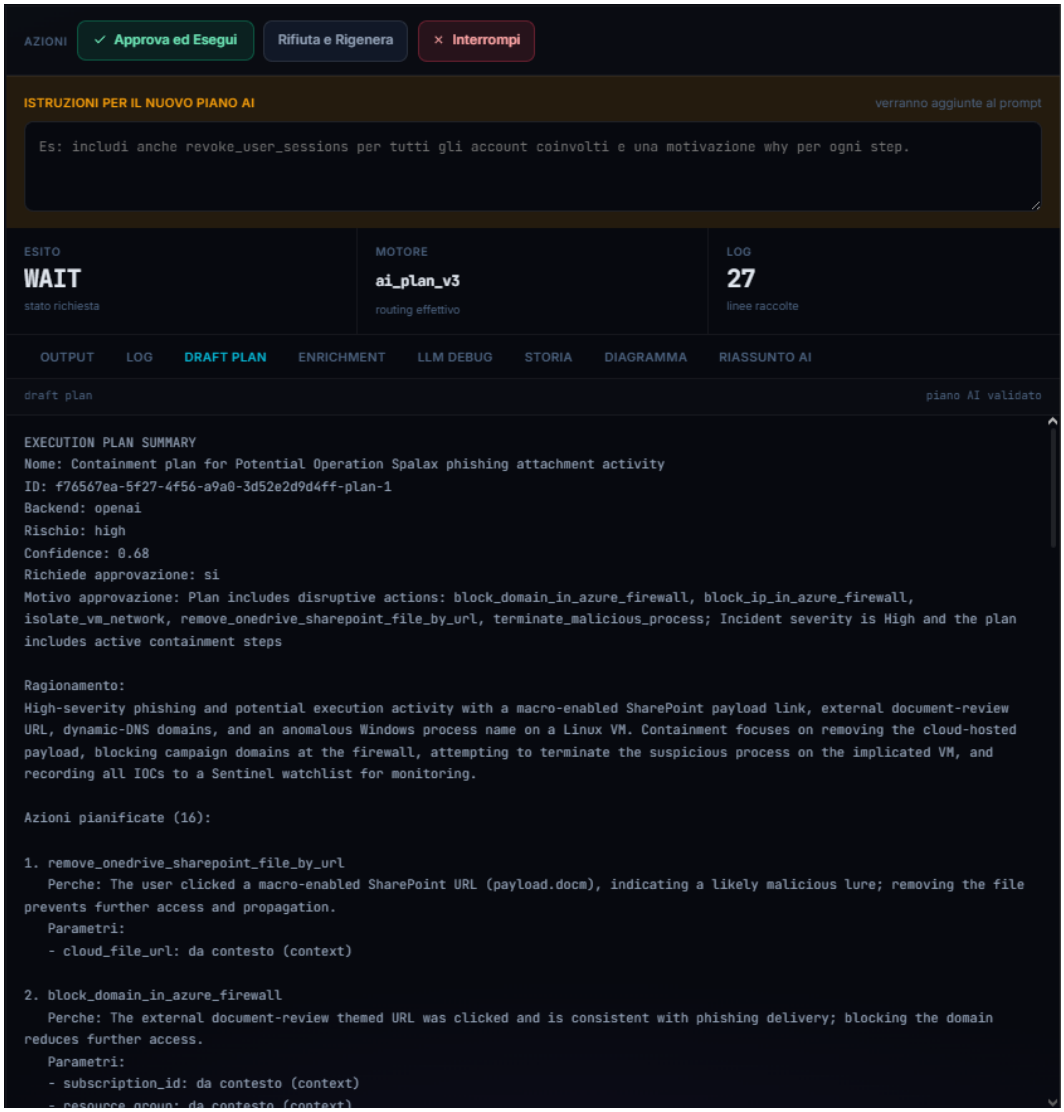

Figure 7.4: Approval view of the web UI for a job parked in `pending_approval`. The top action bar exposes three operator choices (*Approva ed Esegui* – approve and execute, *Rifiuta e Rigenera* – reject and trigger a replan, *Interrompi* – cancel the job), together with a free-text field whose content is appended to the prompt of the next replan call. The status strip below confirms that the request is in the wait state (*ESITO: WAIT*) and is routed under the AI-assisted mode (*MOTORE: ai_plan_v3*). The *Draft Plan* tab shows the execution-plan summary returned by the planner and validated against the closed catalogue (*piano AI validato* marker, top-right of the panel): plan name, plan identifier, planner backend (`openai`), risk level (`high`), confidence (0.78), the auto-generated approval reason that enumerates the three disruptive actions the plan proposes, the free-form reasoning paragraph that grounds the plan in the incident's MITRE technique (T1566.001), and the single planned action with its per-action *why* field and parameter binding (`cloud_file_url: da contesto`, i.e. resolved from the global scope at execution time). Nothing in this plan reaches the Azure tenant until the operator clicks *Approva ed Esegui*.

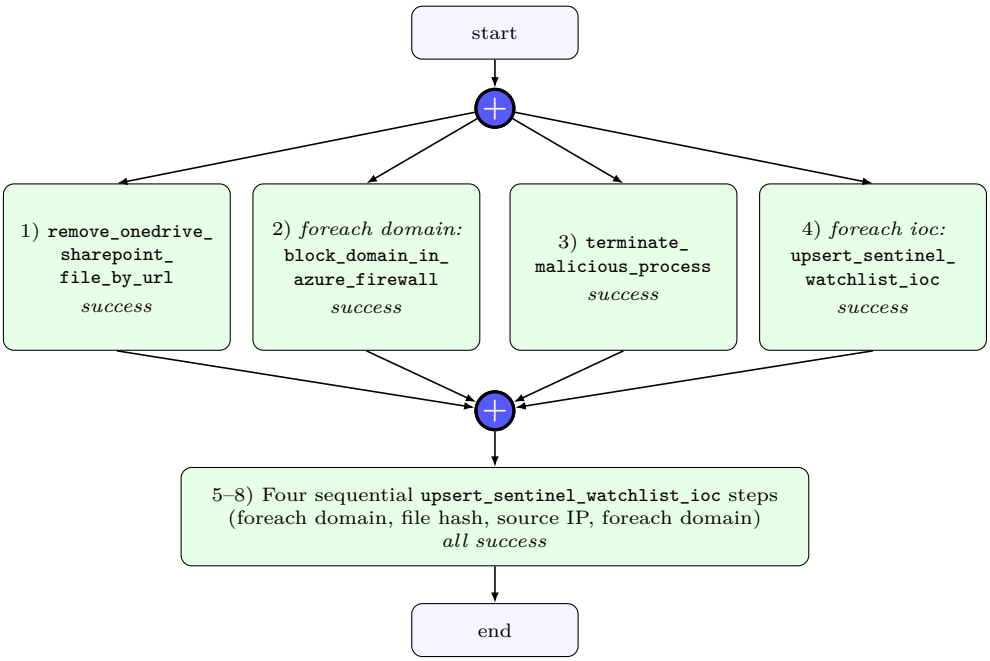

Figure 7.5: High-level topology of the CACAO 2.0 report produced by an AI-assisted run on the Operation Spalax (C0005) scenario. The engine forked into four parallel branches (removing the malicious SharePoint payload, blocking the phishing domains in the Azure Firewall, terminating the in-guest malicious process, and watchlisting the URL IOCs), synchronised at the AND-join, and then walked four sequential `upsert_sentinel_watchlist_ioc` steps that tracked the broader campaign IOCs (dynamic-DNS domains, file hashes, source IP, lure domains) before terminating. Every action recorded *success*, and the per-step `x_execution_result` extension preserves the wall-clock duration of each invocation in the report. The auditability claim of the thesis is that an analyst can reconstruct, after the fact, exactly what the engine attempted, in what order, and what the outcome of every step was, without ever consulting the engine's runtime logs.

stable. The stability test was run on Operation Spalax (Section 7.6.1) rather than on the corpus at large, for a methodological reason worth stating up front: single-alert scenarios constrain the planner to a handful of obviously-correct actions (the technique-to-action mapping leaves little decisional entropy, and successive runs converge on the same minimal plan), so a K-run sweep there would not measure variability so much as confirm its absence. Multi-stage campaigns, by contrast, expose the planner to thirteen-to-sixteen-action plans across heterogeneous entity types and several kill-chain stages, which is exactly the regime in which an LLM's non-determinism has room to manifest. Spalax was therefore chosen as a worst-case proxy: if the catalogue and the validator bound the variance there, they bound it *a fortiori* on the simpler scenarios. The campaign was replayed three times under the AI-assisted mode with identical input (same Sentinel incident, same enrichment backend, same planner prompt template, same closed catalogue) and the resulting plans were compared along structural, validator-level, and execution-level properties. Table 7.4 reports the comparison.

The pattern that emerges is twofold. The *core containment primitives* (`terminate_malicious_process` on the impacted VM and `remove_onedrive_sharepoint_file_by_url` on the lure document) are reproduced verbatim across the three runs, with identical parameter bindings, because the JSON output schema and the closed catalogue together leave the planner with no equivalent shortcut for those primitives. The *indicator-handling tail* of the plan (whether to add an associated dynamic-DNS domain as an active firewall block or as a passive Sentinel watchlist entry; how many associated domains to track at all) varies, because the catalogue admits functionally overlapping options at that layer. Run 2 is the most aggressive of the three (it promotes two associated domains, `gloverstech.com` and `entrevientos.com.ar`, from passive watchlist to active firewall block); Run 1 keeps the same two domains as watchlist entries; Run 3 omits them entirely. The variability stays bounded for the same reason the validation rate stays at 100%: the LLM is given a closed set of moves, so even at moderate planner-reported confidence (Run 2 reports 0.60) it cannot move outside the union of catalogue-admissible plans, and every high-risk action it does propose is forced through the approval gate. The variability that appears is therefore not a defect of the planner but evidence that the catalogue is the right layer at which to enforce determinism on what matters (containment of the primary execution and payload artefacts, mandatory operator approval for every disruptive action) while leaving room for plan-level alternatives on what does not (the curation of associated lower-confidence indicators). One residual observation is worth recording: the catalogue+validator firewall is *structural*, not *semantic*; Run 2 included a duplicate watchlist entry for the file hash in which the `ioc_value` parameter was a serialised dictionary rather than the hash string. The action was structurally a valid `upsert_sentinel_watchlist_ioc` invocation and the validator therefore accepted it, even though its content was semantically degenerate. A larger and more systematic study of plan variance across the full corpus, and an extension of the validator with semantic checks on `ioc_value` payloads, are left as future work.

Property	Run 1	Run 2	Run 3
Plan structure			
Total actions in plan	15	16	13
Process termination (`terminate_malicious_process`)	1	1	1
Payload removal (`remove_onedrive_sharepoint_file_by_url`)	1	1	1
Domain blocks (`block_domain_in_azure_firewall`)	4	6	4
Indicator-curation (`upsert_sentinel_watchlist_ioc`)	9	8	7
Catalogue and validator			
Actions outside the closed catalogue	0	0	0
First-try validator outcome	OK	OK	OK
Replan triggered	no	no	no
Planner self-assessment			
Risk level (planner-assessed)	high	high	high
Confidence (planner-reported)	0.76	0.60	0.74
Approval-gated execution			
High-risk actions proposed	6	8	6
Approvals issued by operator	6	8	6
Destructive actions executed pre-approval	0	0	0
Outcome			
CACAO 2.0 report produced	yes	yes	yes
Final outcome	OK	OK	OK

Table 7.4: Cross-run consistency of the AI-assisted planner across three independent invocations on the same Operation Spalax incident. The two core containment primitives (process termination on the impacted VM, removal of the SharePoint-hosted payload) are reproduced verbatim across the three runs, with identical parameter bindings; the validator-level and approval-gate properties are also reproduced verbatim. The variability is concentrated in the indicator layer: the number of associated domains the planner promotes to active firewall blocks versus the number it leaves as passive watchlist entries fluctuates between runs, but every dominant indicator (the four campaign domains, the source IP, the two URLs, the file hash) is handled in some way by every run. All three runs land at a valid CACAO 2.0 report executed to completion.

7.8 Enrichment Agent Evaluation

The enrichment agent was evaluated separately from the planner. Two dimensions matter: *informativeness* (does the snapshot it produces actually change the planner's behaviour?) and *cost* (how many tool calls does it make, and how much wall-clock time does it add?).

Informativeness. For each scenario, the AI-assisted mode was run twice: once with the enrichment agent enabled (the four backends were tested separately) and once with it disabled. The plans were compared along two axes: the number of actions in the plan and the number of distinct entities the plan touches. A larger or more entity-aware plan, when accompanied by a corresponding justification in the reasoning trace, is read as evidence that the agent was informative; an unchanged plan is read as evidence that the snapshot was redundant for that scenario.

The four backends were exercised on the corpus, and the resulting plans were compared against the disabled-baseline plan for each scenario. The qualitative pattern that emerges is as follows. The `heuristic` backend produces a deterministic, rule-based snapshot whose content is fully predictable from the alert payload alone: it is useful for offline tests and for fall-back behaviour when no LLM key is configured, but it adds little novel evidence beyond what the planner could derive itself. The `openai` single-shot backend produces a richer reasoning trace, but its signals are grounded only in the model's training knowledge and not in live tenant state; on identity-centric scenarios (for example, the compromised-identity variants) this is enough to reshape the plan, but on scenarios that hinge on per-entity context (sign-in failure bursts, role assignments, VM tags) it tends to repeat the disabled-baseline plan. The two agentic backends, `openai_agent` and `claw_agent`, are the only ones that consistently produce plans the disabled baseline does not: their ReAct loop calls the read-only tools described in Section 6.6 and brings in evidence (Entra ID risk state, sign-in failure counts, VM criticality tags, TI verdicts) that is genuinely orthogonal to the alert payload. On the multi-stage campaign replays in particular – Spalax and Lapsus$ – the agentic enrichment was visibly informative, because the planner used the additional context to refine action selection (for example, omitting a high-risk shutdown when the VM-criticality tool reported `tier=test`). A per-scenario quantitative comparison across the four backends, and the corresponding profiling of tool-call counts and wall-clock latency they add on top of the planner, is left for future work; the present analysis records the qualitative pattern observed during validation. The cost of these backends scales with the number of LLM round-trips: zero for the `heuristic` backend, one per run for the `openai` single-shot backend, and a number of LLM calls bounded by `PBREM_ENRICHMENT_MAX_TOOL_CALLS` (default 12) for the agentic backends, with proportional wall-clock latency.

7.9 Discussion and Limitations

The validation supports the three research questions on the test corpus, but it has limits that are worth stating openly.

Tenant scale. The Azure tenant provisioned for this thesis is small. It is sufficient to exercise every Azure control-plane action the engine can execute, but it is not large enough to expose the engine to throttling, to multi-region edge cases, or to the kind of permission-sprawl that mature production tenants accumulate. A larger tenant would, in particular, be useful to validate the Microsoft Graph and Azure Resource Manager calls under realistic concurrency.

Scenario realism. The multi-stage campaign replays are inspired by publicly documented intrusions but are not full replays. They reproduce the *shape* of the incident sequence (the tactics and techniques an analyst would see), not the operational detail of the original intrusion. A more realistic evaluation would require a red-team or a curated intrusion dataset, both of which are out of scope here.

LLM non-determinism. The AI planner's output is non-deterministic, and a methodological choice was made about how to characterise this. Fifteen of the sixteen scenarios in the corpus were executed once under the AI-assisted mode, and the structural metrics reported in Section 7.7 (validation rate, approval rate, safety violations) refer to those single-run plans. Operation Spalax was instead replayed three times specifically to probe cross-run stability, with the results reported in Table 7.4. The choice of Spalax for the stability probe is motivated by the fact that single-alert scenarios constrain the planner to a near-deterministic minimal plan (so a K-run sweep there would have little to measure), while a multi-stage campaign exercises the planner's decisional space and is therefore the regime in which non-determinism can actually manifest. The three Spalax runs showed the core containment primitives and the validator/approval outcomes reproduced verbatim, with variance localised in the indicator-curation tail, and are read as a worst-case proxy: if the catalogue bounds variance there, it bounds it *a fortiori* on the simpler scenarios. The structural metrics are themselves insensitive to single-run noise because they are per-action and binary (a plan either validates or it does not; a destructive action either passes through the approval gate or it does not), so a single observation gives an honest point estimate of a property whose distribution has at most two outcomes. A systematic K-run sweep across the full corpus, paired with a cross-model and cross-prompt sensitivity study, would strengthen the present analysis and is left as future work.

Out-of-scope vendors. The engine acts on Azure only. Other public clouds (AWS, Google Cloud) and on-premise SIEMs are out of scope. Extending the engine to a new control plane requires adding the corresponding ingestion adapter and a new entry in the MITRE mapping, but no change to the CACAO core. This is consistent with the design objective of Section 5.1 (cloud-agnostic plumbing, vendor-specific actions), and was not exercised during validation.

Threats to validity. Three threats to validity are explicit. *Construct validity*: the metrics defined in Section 7.1 approximate what an analyst cares about but do not fully capture it (for example, "action correctness" is judged by the author of the thesis, not by

an independent panel). *Internal validity*: the same person designed the MITRE mapping and judged the appropriateness of the actions it produces; an independent reviewer would tighten this. *External validity*: the corpus is built for Azure-centric intrusions, and the results do not transfer mechanically to other vendor stacks.

Despite these limits, the validation supports the central claim of the thesis. The CACAO core inherited from earlier TORSEC work was preserved unchanged. The Azure ingestion layer and the action library make the engine usable against a Sentinel-driven Azure tenant. The MITRE mapping makes the engine respond to combinations of techniques that the legacy catalogue did not anticipate. And the AI-assisted mode produces auditable, approval-gated CACAO playbooks for every scenario in the corpus.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis set out to close a specific gap. The playbook-driven remediator inherited from earlier TORSEC work [6] implemented a correct CACAO 2.0 [4] execution engine, but it could not ingest alerts from Microsoft Sentinel [13], it could not synthesise CACAO playbooks dynamically from MITRE technique mappings [12], and it had no planning layer that an analyst could use when the catalogue of pre-authored playbooks ran out of options. The contribution of the present work is to close all three gaps at once, while preserving the CACAO core unchanged.

The redesign produced a three-mode execution engine that shares a single executable representation and a single CACAO serialiser. The legacy mode keeps the inherited behaviour as the regression baseline and the contract that every other mode has to honour. The tactic-driven mode generates CACAO playbooks at runtime from a MITRE tactic-technique-action mapping built for Azure, which turns the catalogue from a static list of pre-authored files into a small knowledge base that the engine can consult against any incoming combination of techniques. The AI-assisted mode delegates plan generation to a Large Language Model, validates the result against a closed action catalogue, and waits for explicit human approval before any disruptive operation runs against the tenant. A read-only enrichment agent based on the ReAct pattern [14] feeds the planner with context drawn from Sentinel, Microsoft Graph and Azure Resource Manager.

The validation reported in Chapter 7 provides empirical evidence on a corpus of Azure threat scenarios that includes identity compromise, lateral movement, malicious URL clicks, key-vault lockdown and multi-stage campaign replays inspired by publicly documented intrusions. On that corpus, all three research questions admit an affirmative answer. The Azure ingestion layer, the action library and the legacy mode answer RQ1: the engine acts on the Azure control plane while preserving the inherited execution semantics. The tactic-driven mode answers RQ2: it produces non-empty CACAO playbooks for combinations of techniques the static catalogue did not anticipate. The AI-assisted mode answers RQ3: it produces auditable CACAO playbooks under a human-in-the-loop policy that gates every

destructive action and reports a complete reasoning trace in the final report.

Two threads run through the work and are worth restating in closing. The first is that the CACAO core inherited from earlier TORSEC work is treated as a fixed point. Every new mode funnels its remediation logic through the same in-memory representation and the same serialiser, so that the auditability and the workflow constructs are identical regardless of how the playbook was produced. The second is that safety precedes automation. The closed action catalogue and the per-action approval gate are part of the design rather than additions to it; the enrichment agent is read-only by construction; the LLM cannot, in any of the three modes, execute an action the engine does not already know how to perform.

The thesis is therefore best read not as the introduction of a new SOAR product but as a controlled extension of an existing CACAO-compliant tool toward two new things: a public-cloud control plane (Azure) and a new way of obtaining playbooks (MITRE-driven generation and LLM-assisted planning). The framing matters because it is what makes the contribution incremental and verifiable: the parts that already worked are still there, and the parts that are new are bounded in scope.

8.2 Future Work

Several directions are natural follow-ups, and each one would address a concrete limit identified during validation (Section 7.9).

The most obvious direction is multi-cloud coverage. The design is already split between cloud-agnostic plumbing and Azure-specific actions, but only the Azure path is implemented. Adding AWS or Google Cloud would mean writing the corresponding ingestion adapter and extending the MITRE mapping with their control-plane actions; nothing in the CACAO core or in the planner would have to change. A useful intermediate step is an on-premise SIEM connector so that the engine can be exercised in hybrid deployments.

A second direction is a hybrid execution mode. The current router selects exactly one front-end per incident, but the tactic-driven and the AI-assisted modes are complementary: the first is cheap and predictable, the second is broader but slower and non-deterministic. A hybrid mode could consult the MITRE mapping first and fall back to the planner only when the mapping returns no entry or returns a degenerate plan. The state machine of `ExecutionJob` is already factored in a way that would support this addition without requiring any restructuring of the back-end.

A third direction is tighter integration with the wider CACAO ecosystem. CACAO Roaster, the editor maintained by the Open Cybersecurity Alliance, would be a natural reviewer of AI-generated drafts: the analyst could open the proposed playbook in the editor, adjust steps and parameters, and either approve the edited version for execution or save it back to the local catalogue as a hand-authored variant. On the sharing side, the work of Mavroeidis et al. [19] and the Knowledge Management System of Tsirakis et al. [18] suggest that emitted CACAO reports should be redistributable through MISP and through STIX-compliant threat-intelligence platforms [11]; adding an export step that publishes the report to such a platform is a small change with disproportionate value for the community.

A fourth direction concerns the AI planning layer itself. The plan-quality metrics reported in Chapter 7 are obtained against a closed action catalogue and a fixed prompt; both could evolve. A feedback loop that uses the operator's approve/reject decisions to refine either the catalogue (by lowering or raising the risk level of an action whose approvals are reliably one-sided) or the prompt (by adjusting the per-tactic guidance) would let the system improve with use. The threats to validity discussed in Section 7.9 also call for an independent evaluation: a second reviewer, a curated intrusion dataset, or a small-scale red-team exercise would all strengthen the empirical case beyond what a single-author thesis can provide.

A fifth direction is the enrichment layer. The current tool registry is intentionally narrow (six read-only tools) and could be extended along two axes: more data sources (for example, vulnerability scanners, EDR telemetry, additional threat-intelligence feeds) and richer reasoning patterns [20, 21]. Adding new tools is straightforward; the harder design question is how to keep the budget on tool calls and reasoning turns informative as the registry grows. A small benchmark that measures the marginal value of each tool would inform that decision.

A final, longer-horizon direction is to lift the engine from per-incident remediation to incident-stream orchestration. The current implementation runs one job per incident. A production deployment will more often see streams of correlated incidents arriving from Sentinel within minutes of each other, and the engine should be able to recognise that fact, batch the planning calls, and propose a single coordinated response rather than a sequence of independent ones. This is mostly a scheduling problem on top of the existing architecture, but it is the one that would bring the engine closest to the kind of orchestration that a mature Security Operations Centre runs in practice.

None of these directions invalidates the work reported here. They all assume the same CACAO core, the same three-mode engine, and the same safety policy. The thesis is best understood as a starting point that makes each of them addressable in turn.

Bibliography

[1] Blake E. Strom, Andy Applebaum, Doug P. Miller, Kathryn C. Nickels, Adam G. Pennington, and Cody B. Thomas. *MITRE ATT&CK: Design and Philosophy*. Tech. rep. Revised version of the July 2018 report. The MITRE Corporation, Mar. 2020. URL: https://attack.mitre.org/docs/ATTACK_Design_and_Philosophy_March_2020.pdf.

[2] Manfred Vielberth, Fabian Böhm, Ines Fichtinger, and Günther Pernul. «Security Operations Center: A Systematic Study and Open Challenges». In: *IEEE Access* 8 (2020), pp. 227756–227779. DOI: 10.1109/ACCESS.2020.3045514. URL: https://ieeexplore.ieee.org/document/9296846.

[3] Chadni Islam, Muhammad Ali Babar, and Surya Nepal. «A Multi-Vocal Review of Security Orchestration». In: *ACM Computing Surveys* 52.2 (Apr. 2019), pp. 1–45. DOI: 10.1145/3305268. URL: https://dl.acm.org/doi/abs/10.1145/3305268.

[4] Bret Jordan and Allan Thomson. *CACAO Security Playbooks Version 2.0*. Committee Specification 01, 27 November 2023. OASIS Open. Nov. 2023. URL: https://docs.oasis-open.org/cacao/security-playbooks/v2.0/cs01/security-playbooks-v2.0-cs01.html.

[5] Microsoft. *Introducing Microsoft Security Copilot: Empowering Defenders at the Speed of AI*. The Official Microsoft Blog. Mar. 2023. URL: https://blogs.microsoft.com/blog/2023/03/28/introducing-microsoft-security-copilot-empowering-defenders-at-the-speed-of-ai/.

[6] Dario Simone Leone. «Remediation procedures and automated cybersecurity incident response». Master's Degree in Computer Engineering, Advisors: C. Basile and F. Settanni. Master's thesis. Turin, Italy: Politecnico di Torino, 2025. URL: https://webthesis.biblio.polito.it/37719/.

[7] Paul Cichonski, Tom Millar, Tim Grance, and Karen Scarfone. *Computer Security Incident Handling Guide*. Tech. rep. NIST Special Publication 800-61 Revision 2. National Institute of Standards and Technology, Aug. 2012. DOI: 10.6028/NIST.SP.800-61r2. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-61r2.pdf.

[8] European Union Agency for Cybersecurity (ENISA). *Good Practice Guide for Incident Management*. ENISA. 2010. URL: https://www.enisa.europa.eu/sites/default/files/publications/Incident_Management_guide.pdf.

[9] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. «Large Language Models for Cyber Security: A Systematic Literature Review». In: *ACM Transactions on Software Engineering and Methodology* (2025). DOI: 10.1145/3769676. URL: https://dl.acm.org/doi/10.1145/3769676.

[10] Wiem Tounsi and Helmi Rais. «A survey on technical threat intelligence in the age of sophisticated cyber attacks». In: *Computers & Security* 72 (2018), pp. 212–233. DOI: 10.1016/j.cose.2017.09.001. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167404817301839.

[11] OASIS Open. *STIX Version 2.1*. OASIS Standard, edited by Bret Jordan, Rich Piazza and Trey Darley. OASIS Open. June 2021. URL: https://docs.oasis-open.org/cti/stix/v2.1/os/stix-v2.1-os.html.

[12] The MITRE Corporation. *MITRE ATT&CK Matrix for Enterprise: Cloud*. 2025. URL: https://attack.mitre.org/matrices/enterprise/cloud/.

[13] Microsoft. *What is Microsoft Sentinel?* Microsoft Learn documentation. Accessed: 2026-06-07. 2025. URL: https://learn.microsoft.com/en-us/azure/sentinel/overview.

[14] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023. URL: https://openreview.net/forum?id=WE_vluYUL-X.

[15] TNO and COSSAS. *SOARCA: The Open Source CACAO-based Security Orchestrator*. GitHub repository and project page. Community for Open Source Security Automation Software (COSSAS), 2024. URL: https://github.com/COSSAS/SOARCA.

[16] Open Cybersecurity Alliance (OCA). *CACAO Roaster: A web application for generating, parsing, validating, visualising and executing CACAO v2.0 playbooks*. GitHub repository and project page. Open Cybersecurity Alliance, 2024. URL: https://github.com/opencybersecurityalliance/cacao-roaster.

[17] Microsoft. *Automate threat response with playbooks in Microsoft Sentinel*. Microsoft Learn documentation. 2025. URL: https://learn.microsoft.com/en-us/azure/sentinel/automation/automate-responses-with-playbooks.

[18] Orestis Tsirakis, Konstantinos Fysarakis, Vasileios Mavroeidis, and Ioannis Papaefstathiou. «Operationalizing cybersecurity knowledge: Design, implementation & evaluation of a knowledge management system for CACAO playbooks». In: *Computers & Security* 159 (2025), p. 104696. ISSN: 0167-4048. DOI: 10.1016/j.cose.2025.104696. URL: https://doi.org/10.1016/j.cose.2025.104696.

[19] Vasileios Mavroeidis, Pavel Eis, Martin Zádník, Marco Caselli, and Bret Jordan. «On the Integration of Course of Action Playbooks into Shareable Cyber Threat Intelligence». In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 2104–2108. DOI: 10.1109/BigData52589.2021.9671893. URL: https://doi.org/10.1109/BigData52589.2021.9671893.

[20] Siddhant Srinivas, Brandon Kirk, Julissa Zendejas, Michael Espino, Matthew Boskovich, Abdul Bari, Khalil Dajani, and Nabeel Alzahrani. «AI-Augmented SOC: A Survey of LLMs and Agents for Security Automation». In: *Journal of Cybersecurity and Privacy* 5.4 (2025), p. 95. DOI: 10.3390/jcp5040095. URL: https://doi.org/10.3390/jcp5040095.

[21] Dheer Toprani and Vijay K. Madisetti. «LLM Agentic Workflow for Automated Vulnerability Detection and Remediation in Infrastructure-as-Code». In: *IEEE Access* 13 (2025), pp. 69175–69181. DOI: 10.1109/ACCESS.2025.3560911. URL: https://doi.org/10.1109/ACCESS.2025.3560911.

Appendix A

User Manual

This appendix walks a SOC analyst through the operations that the redesigned remediator supports: installing it, configuring it, submitting alerts, choosing an execution mode, approving destructive actions, and reading the CACAO reports it produces. The manual is intentionally compact; the architectural rationale for each step lives in Chapters 5 and 6.

A.1 Prerequisites

The remediator runs on Python 3.13. The dependencies are pinned in `requirements.txt` at the project root and include the Azure SDK (`azure-identity`, `azure-mgmt-compute`, `azure-mgmt-network`), the OpenAI client, the STIX parser, and a small set of utility libraries. A Linux or macOS host is the supported development environment; a Windows host works too, with the project's PowerShell-aware path handling.

A working installation also needs:

- access to a Microsoft Sentinel workspace from which incident JSON can be exported, either manually or via the workspace's REST API;
- an Azure App Registration with the role assignments that the action library requires (read access to Microsoft Graph for user, sign-in and role-assignment queries; *Contributor* on the resource groups where the engine is allowed to act);
- an OpenAI API key, only if the AI-assisted mode will be enabled.

The container image `Dockerfile.prod_compiled` packages the project and its dependencies for unattended deployment; the development image `Dockerfile.dev` additionally bundles the `claw` binary used by one of the enrichment backends.

A.2 First-Time Setup

A first-time deployment is three steps.

1. Clone and install. Clone the repository, create a Python virtual environment, and install the dependencies.

```
python -m venv .venv
. .venv/bin/activate
pip install -r requirements.txt
```

2. Configure the Azure service principal. Place the credentials of the App Registration in `config/azure/azure_credentials.json` using the format below. The path can be overridden with the environment variable `PBREM_AZURE_CREDENTIALS_FILE`.

```
{
  "client_id":     "<your application (client) ID>",
  "tenant_id":     "<your directory (tenant) ID>",
  "client_secret": "<your client secret>"
}
```

3. Configure the runtime mode and credentials. The simplest way is to edit `config.local.json` at the project root. The file is read on startup and applied to the environment. A minimal template is shown below; replace every placeholder before the first run.

```
{
  "routing_mode":          "tactic_v2",
  "ai_api_key":            "",
  "ai_model":              "",
  "ai_planner_backend":    "heuristic",
  "access_token":          "",
  "enrichment_enabled":    "false",
  "enrichment_backend":    "heuristic",
  "dry_run":               "true",
  "webhook_url":           ""
}
```

The same values can be set through environment variables, in which case the environment wins over the file. Table A.1 lists the variables an operator normally touches.

A.3 Starting the Web Interface

The web interface is the simplest way to drive the engine. From the project root:

Variable	Purpose
PBREM_ROUTING_MODE	Selects the execution mode: legacy, tactic_v2, or ai_plan_v3.
PBREM_WEB_TOKEN	Optional bearer token required on every API call. Leave unset to expose the API without authentication.
PBREM_AI_PLANNER_BACKEND	Planner backend: heuristic (default) or openai.
PBREM_AI_API_KEY	OpenAI API key, required when the planner backend is openai.
PBREM_AI_MODEL	Model identifier passed to the LLM, e.g. gpt-4.1-mini.
PBREM_ENRICHMENT_ENABLED	Set to true to invoke the enrichment agent before planning.
PBREM_ENRICHMENT_BACKEND	Enrichment backend: heuristic, openai, openai_agent, or claw_agent.
PBREM_DRY_RUN	When true, actions are simulated rather than executed against the tenant.

Table A.1: Environment variables an operator normally configures. The full list is documented in Chapter 6.

```
python -m source.web_app --host 0.0.0.0 --port 8080
```

The single-page UI is then reachable at `http://<host>:8080/`. The page exposes three views: a *Configuration* tab that mirrors `config.local.json`; a *Submit* tab that accepts a Sentinel incident as a JSON paste or as a file upload; and an *Approvals* tab that surfaces destructive actions waiting for authorisation when the AI-assisted mode is in use.

The same engine can also be invoked from the command line without starting the web server:

```
export PBREM_ROUTING_MODE=tactic_v2
python -m source.pb_rem_router_v2 \
       --alert tests/azure/alerts_test/azure_malicious_url_clicked_response.json
```

The CLI path produces the same CACAO output as the web path; only the front-end differs.

A.4 Submitting an Incident

An incident is submitted to the engine as a JSON document with two top-level keys: `incident` (the Sentinel incident properties) and `entities` (the typed entities the analytics rule flagged). The format follows the structure documented by Microsoft [13], with one nesting convention: the analytics-rule techniques are read from `incident.properties.additionalData.techniques`.

From the web UI, the *Submit* tab accepts the JSON either as a paste into a textarea or as a file upload. From the API, two endpoints are available:

- `POST /api/remediate` runs the incident synchronously and returns the CACAO report in the response body. Useful for scripted invocations.
- `POST /api/remediate/start` returns a job identifier immediately and runs the incident asynchronously. The job's progress can then be streamed from `GET /api/remediate/stream?job_id=...` and the final report fetched from `GET /api/remediate/result?job_id=...`.

The endpoint `POST /api/incident` accepts a raw Sentinel incident payload (one not yet wrapped in the engine's input format) and applies the ingestion translation internally.

A.5 Choosing an Execution Mode

The execution mode is selected by `PBREM_ROUTING_MODE`. The three options correspond to the three modes described in Chapter 5.

Legacy. Use this mode when the incident has a hand-authored CACAO playbook in the catalogue and a deterministic, file-based response is desired. The engine refuses to act when no playbook matches.

Tactic-driven (`tactic_v2`). Use this mode as the default. The MITRE mapping covers a broad set of Azure-relevant techniques; the playbook is generated at runtime and produces deterministic CACAO output without invoking any LLM.

AI-assisted (`ai_plan_v3`). Use this mode when the tactic-driven mode returns an empty or weak plan, or when the incident shape was not anticipated by the mapping. The LLM proposes a plan, the validator rejects anything outside the closed catalogue, and the operator authorises the destructive actions through the *Approvals* tab before they run.

A.6 Approving Destructive Actions

In the AI-assisted mode, every action whose catalogue entry has `approval_required` set to `true` is parked before execution. The job moves into the state `pending_approval` and the destructive actions appear in the *Approvals* tab of the web UI.

Each action card shows the action name, its risk level (`high`, `medium`, `low`), the entities it will touch, the parameters that will be passed, and the planner's justification. The operator can authorise actions one by one or in bulk, and can also reject the plan with feedback. A rejection routes back into a single replan call (`POST /api/remediate/replan`) that includes the operator's feedback in the prompt and produces a new plan against the same incident.

The same surface is exposed by the API:

- `POST /api/remediate/approve` authorises the destructive actions of a job.
- `POST /api/remediate/replan` rejects the plan and triggers a replan.
- `POST /api/remediate/cancel` discards the job.

A.7 Reading the CACAO Report

Every executed run produces a CACAO 2.0 [4] report. The report contains the playbook that was run (whether file-based, generated or AI-produced), the entities the actions touched, the result of each step, and (for AI-assisted runs) the planner's reasoning trace and the enrichment snapshot used to generate the plan.

The report is returned in the body of `GET /api/remediate/result` and is also appended to `runtime/execution_history.jsonl`, one job per line, with the last fifty entries retained. The file is a plain JSONL: each line is a JSON object with the job metadata at the top level and the CACAO report nested under the `cacao` key.

The report is consumable by any CACAO 2.0 tool. In particular, the CACAO Roaster web application can open the file as-is and display the workflow graph, which is a useful way to review an AI-assisted plan after execution.

A.8 Troubleshooting

A few failure modes are common enough during early deployments to be worth documenting.

Authentication fails on the first Graph call. The most frequent cause is a missing role assignment on the App Registration. The minimum set is *Application.Read.All*, *User.Read.All*, *Directory.Read.All* and *AuditLog.Read.All* for the read-only enrichment paths; the destructive actions additionally need *User.RevokeSessionsAll* and the resource-group-scoped *Contributor* role for compute and network operations.

The AI-assisted mode falls back to the heuristic. The fallback is logged with a marker of the form `*_fallback_heuristic` in the run metadata. The two usual causes are a missing or invalid `PBREM_AI_API_KEY` and a network failure reaching the OpenAI endpoint. The fallback is by design: the engine prefers a deterministic plan over no plan at all.

No destructive actions are surfaced for approval. Either every action in the generated plan is passive (the *Approvals* tab will be empty by design), or the dry-run mode is on. Check the `dry_run` flag in the configuration: when `true`, the engine simulates actions and does not route through the approval state.

The web server refuses connections after a configuration change. Configuration is persisted to `config.local.json` at startup and re-read after each save through the *Configuration* tab. If the server was started with a different working directory, the configuration file may be in a different location than expected; restart the server from the project root, or set the explicit overrides through environment variables.

Appendix B

Developer Manual

This appendix is the entry point for a developer who needs to extend the codebase. It assumes the reader has already worked through Chapter 6, which describes the layout of the project, and through Appendix A, which covers the operator-facing concerns. The aim here is to make the most frequent extension tasks explicit, file by file, so that the cost of adding a new action, a new MITRE entry, a new enrichment tool, or a new execution mode is bounded and predictable.

B.1 Setting Up a Development Environment

The supported development setup uses the Docker image `Dockerfile.dev`, which provisions Python 3.13, the Azure SDK, the OpenAI client, and the `claw` binary used by one of the enrichment backends. From the project root:

```
docker build -f Dockerfile.dev -t pb-rem:dev .
docker run --rm -it -v "$PWD":/usr/src/pb_rem pb-rem:dev bash
```

A bare-metal setup also works. Create a virtual environment, install `requirements.txt`, and (optionally) compile the `claw` binary from the upstream repository pinned in `Dockerfile.dev`.

The test suite uses `pytest` and lives under `tests/`. The Azure-facing tests are isolated under `tests/azure/`; the legacy on-premise and Kubernetes tests under `tests/on_premises/` and `tests/kubernetes/` respectively. To run the Azure-facing tests:

```
pytest tests/azure -q
```

A subset of the tests exercises the LLM backends with mocked responses (the OpenAI client is monkey-patched), so the suite runs offline without an API key.

B.2 Project Layout Cheatsheet

The folder layout is summarised in Chapter 6 (Table 6.1). The cheatsheet below is the developer's quick reference for *where* a change normally lands.

Change you want to make	**File to edit**
Add a new Azure action	`source/executor/actions_azure_tactic_v2.py`
Map a new MITRE technique to existing actions	same file: edit `MITRE_TACTIC_TECHNIQUE_ACTION_MAPPING`
Expose a new action to the AI planner	`source/ai_planner/plan_schema.py` (catalogue entry)
Add a new read-only enrichment tool	`source/enrichment_agent/tools.py`
Support a new Sentinel alert family	`source/input_analyzer/input_analyzer_tactic_v2.py`
Add a fourth execution mode	`source/pb_rem_router_v2.py` (dispatcher) and new `pb_rem_*.py` module
Add or modify a CACAO playbook (legacy)	`kb/cacao_playbook/` (JSON files)
Edit the web UI	`web/index.html` (single page)
Edit the HTTP API	`source/web_app.py`

Table B.1: Where to land a change. Each row maps an extension intent to the file the developer normally edits.

B.3 Adding a New Azure Action

Adding a new action is the most common extension and is intentionally a single-file edit on the happy path. The steps are summarised below; the function signatures in this section assume the conventions used by the existing action library.

1. Implement the action. In `source/executor/actions_azure_tactic_v2.py`, add a function with the following signature:

```
def my_new_action(self, parameters: Dict[str, Any],
                  global_scope: Dict[str, Any]) -> None:
    # 1. Read parameters from ‘parameters‘ (typed) and from
    #    ‘global_scope‘ (the alert-wide context).
    # 2. Acquire an Azure credential via ‘_get_credential(parameters)‘.
```

```
    # 3. Invoke the Azure SDK or REST endpoint.
    # 4. Update `global_scope["overall_status"]` to reflect success/failure.
    # 5. Use the module logger for any operator-visible message.
```

2. Register the action. Action discovery is by symbol name: the executor looks up the function by the action identifier used in the playbook. No central registry needs to be updated, but the function must be importable from the module.

3. (Optional) Expose the action to the AI planner. If the new action should be selectable by the LLM-assisted mode, add an entry to `_ACTION_CATALOG` in `source/ai_planner/plan_schema.py`. The minimum fields are described in Listing 6.2 (Chapter 6); the most important are `required_context` (the parameters the validator will check) and `approval_required` (whether the action is destructive). A passive read-only action sets `approval_required` to `false`; a destructive action sets it to `true` and chooses a risk level.

4. Map the action to a MITRE technique. Edit `MITRE_TACTIC_TECHNIQUE_ACTION_MAPPING`, in the same actions module, and append the action identifier under one or more tactic/technique pairs. The tactic-driven mode will pick it up automatically the next time an incident carries the corresponding technique. Techniques without any implementation are kept in the mapping with an empty list (for documented coverage gaps).

5. Write a unit test. Tests for action behaviour live under `tests/azure/test_actions_azure.py`. The convention is to mock the Azure SDK client (or to use the `requests` monkey-patch the suite already provides) and assert against the parameters the action would send. A coverage test in `tests/azure/test_actions_azure_mapping_coverage.py` verifies that every action mentioned in the MITRE mapping has a matching implementation; a new action that breaks this test points to a missing function name or to a typo in the mapping.

B.4 Mapping a New MITRE Technique

Sometimes the mapping needs a new entry without any new action implementation. The edit is then confined to `MITRE_TACTIC_TECHNIQUE_ACTION_MAPPING`.

```
MITRE_TACTIC_TECHNIQUE_ACTION_MAPPING: Dict[str, Dict[str, List[str]]] = {
    "initialaccess": {
        "T1078": ["disable_user_account", "revoke_user_sessions"],
        # ... existing entries ...
        "TXXXX": ["existing_action_a", "existing_action_b"],   # NEW
    },
    # ... other tactics ...
```

```
}
```

The technique code is normalised to upper case before lookup, so the dictionary key must be upper case. The tactic key uses the lower-case, space-stripped MITRE label as it appears in the Sentinel incident's `additionalData.tactics` array. Run `pytest tests/azure/test_actions_azure_mapping_coverage.py` after the edit to verify that every referenced action exists in the executor module.

B.5 Adding an Enrichment Tool

The enrichment agent's registry lives in `source/enrichment_agent/tools.py`. Each tool is a method on `EnrichmentToolRegistry` that takes the planning input plus its own arguments and returns a structured result.

The class exposes two methods used by every backend: `tool_specs()` returns the OpenAI-function specifications consumed by the agentic backend, and `execute(name, arguments, planning_input)` dispatches a tool call by name. Adding a new tool is three steps:

1. Implement the method, with the signature `def _my_tool(self, arguments, planning_input) -> Dict[str, Any]:`. The method must be *read-only*; the design forbids any tool that mutates the tenant.
2. Add the tool's JSON-schema specification to `tool_specs()`.
3. Add the dispatcher entry in `execute()` (the existing pattern uses a small `if/elif` chain; a dictionary-based dispatch would be equivalent).

After the addition, increment the default budget on `PBREM_ENRICHMENT_MAX_TOOL_CALLS` only if the new tool is expected to be called in addition to the existing ones; the conservative path is to leave the budget unchanged so that the agent has to drop a less-useful tool to make room for the new one.

B.6 Supporting a New Sentinel Alert Family

The ingestion layer in `source/input_analyzer/input_analyzer_tactic_v2.py` dispatches by *alert family*, keyed by the `Threat_Category` field of the incoming payload. Adding a new family is two steps: write a normalisation function that returns the standard `remediation_data` dictionary, then register it in the `analyzer_mapping`.

```
self.analyzer_mapping = {
    "unauthorized_access":         input_analyzer.generateRemediationData...,
    # ... existing entries ...
    "my_new_alert_family": input_analyzer.generateRemediationDataForMyNewFamily,
}
```

The new function follows the same shape as the existing ones: extract the techniques, derive the tactics, bucket the entities by kind, populate the `context` sub-dictionary, and preserve the raw alert under `raw_alert`. The signature contract is documented inline in the module.

B.7 Adding a New Execution Mode

The dispatcher in `source/pb_rem_router_v2.py` is the smallest module in the codebase, and that is the file to edit to register a new front-end.

```
if routing_mode == "tactic_v2":
    from source.pb_rem_tactic_v2 import main as selected_main
elif routing_mode == "ai_plan_v3":
    from source.pb_rem_ai_plan_v3 import main as selected_main
elif routing_mode == "my_new_mode":                              # NEW
    from source.pb_rem_my_new_mode import main as selected_main  # NEW
else:
    from source.pb_rem import main as selected_main
```

The new front-end module must expose a top-level `main()` function that consumes alerts, builds or loads a `Playbook` object, and hands it to the shared executor. The natural template is `source/pb_rem_ai_plan_v3.py`, which inherits from `source/pb_rem_tactic_v2.py` and overrides only the parts that differ; the same pattern lets a fourth mode reuse the ingestion and execution pipeline without duplicating code.

A new mode that needs additional environment variables should read them at startup and document them in the README and in Appendix A.

B.8 Testing Conventions

The test suite follows three conventions worth knowing before adding new tests.

Fixtures live next to the tests. The alert JSON files used by the tests sit under `tests/azure/alerts_test/`. New scenarios should add a single alert file there and reference it from the test by relative path; the existing scenarios are the canonical template.

LLM calls are mocked. Tests that exercise the AI planner or the OpenAI-based enrichment backend monkey-patch the OpenAI client and inject the desired response. The pattern is visible in `tests/azure/test_ai_plan_v3.py`. The suite therefore runs offline without an API key.

Coverage tests guard the MITRE mapping. The module `tests/azure/test_actions_azure_mapping_coverage.py` checks two invariants: every action referenced in

the mapping has a Python implementation, and every implemented action is mapped to at least one MITRE technique (with a documented exception list for utilities). These tests are the safety net that catches typos in the mapping and orphan actions.

B.9 Logging and Observability

All modules use the helper `source/helpers/logging_helper.py` to obtain a named logger. The naming convention is dot-separated (`pb-rem`, `pb-rem.executor`, `pb-rem.executor.azure`, `ai-planner`, `enrichment-agent`), so that an operator can filter by subsystem at runtime.

Two further conventions are followed across the codebase: error paths log the failure with structured fields (`status`, `request_id`, the action name) rather than free-form strings; and the web layer mirrors stdout/stderr of background jobs into the SSE stream consumed by the UI, which means that print statements inside an action are visible in the browser without any extra plumbing.

B.10 Releasing

The project does not ship a versioned wheel. Releases are delivered as the container image built from `Dockerfile.prod_compiled`; tagging a release is therefore a matter of pushing the image to a registry and updating the deployment manifest. The unit-test suite must pass before a release is cut; the coverage tests on the MITRE mapping in particular are the single most useful gate against regressions when the action library changes.