POLITÉCNICO DE LEIRIA
ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Integrating Disparate Security Tools into a Single Splunk Interface

Multi-Domain Infrastructure Integration, Baseline Anomaly Detection, and Compliance Automation

Alexandre Jerónimo

School of Management and Technology
Department of Computer Engineering
Master in Cybersecurity & Digital Forensics

Leiria, April 2026

POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Integrating Disparate Security Tools into a Single Splunk Interface

Multi-Domain Infrastructure Integration, Baseline Anomaly Detection, and Compliance Automation

Alexandre Jerónimo
Student No. 2230467

Supervisor: Paulo Jorge Cordeiro
Adjunct Professor, Polytechnic of Leiria

School of Management and Technology
Department of Computer Engineering
Master in Cybersecurity & Digital Forensics

Project

Leiria, April 2026

Integrating Disparate Security Tools
into a Single Splunk Interface

Preparation of this work was facilitated by the use of the *IPLeiria-Thesis* template.

Acknowledgements

I wish to thank my supervisor, Professor Paulo Jorge Cordeiro, for his guidance, availability, and trust in letting me shape this project.

I owe particular gratitude to João Carvalho, my team lead and mentor, who taught me Splunk and the surrounding infrastructure from the ground up. Without his encouragement, this project would not have existed at all. His patience and seemingly inexhaustible supply of ideas kept it moving forward, often faster than I could keep up. That he integrated the system into his own daily workflow and tested it extensively gave me both motivation and confidence in the work.

I also thank fellow team members Tomás Ferreira and Fernando Simões, for adopting the system into their routines, providing feedback that improved it throughout, and for their advice on the project report.

To Elisa Matias, my manager at the host organisation, I am grateful for the support she always offered and for connecting me with colleagues across the organisation who could help. More broadly, the cybersecurity operations team provided a welcoming environment throughout.

I also thank Helder Cruz and João Oliveira at Capgemini Engineering, whose support made this placement possible and who first opened the door to the professional environment in which this work was carried out.

Finally, I thank my family — my parents, my sister, and my grandparents — for their understanding during the more demanding periods, their motivation, and their unconditional support.

Abstract

Security analysts at the host organisation rely on separate platforms for infrastructure management, log source monitoring, vulnerability scanning, email monitoring, and identity verification. Switching between these tools delays investigations and leaves processes such as monthly reporting dependent on manual procedures.

USO API is a custom Splunk search command that lets analysts query six operational data domains directly from Splunk's search interface. Built on a modular architecture, the command is organised into six modules covering managed hosts, vulnerabilities, Splunk health, database connectivity, email processing, and identity records. A centralised tabbed dashboard consolidates every module into a single view. Three companion subsystems extend the solution: one automates compliance reporting with automatic retries on failure; another supports low-noise infrastructure diagnostics through historical baseline comparison; a third adds a privacy-monitoring extension that scans indexed data for personal information.

The project shows that a single Splunk search command can give a small security team direct access to five external platforms and one internal data consolidation layer without deploying additional middleware or monitoring infrastructure. The project yields three core transferable engineering contributions, validated within the Splunk environment: a SIEM-centric integration pattern for unified infrastructure querying, a multi-level priority architecture for efficient dashboard loading with staggered searches, and a historical-baseline diagnostics approach that maintained low-noise monitoring whilst reducing reliance on manual calibration. A secondary engineering contribution extends the same platform with a high-throughput pattern-scanning approach for Personally Identifiable Information (PII) detection within a Security Information and Event Management (SIEM) pipeline.

Keywords: SIEM, Splunk, Custom Search Command, Infrastructure Integration, Anomaly Detection, Compliance Automation, PII Detection.

Resumo

Os analistas de segurança da entidade de acolhimento recorrem a plataformas distintas para gestão de infraestrutura, monitorização de *logs*, análise de vulnerabilidades, monitorização de correio eletrónico e verificação de identidades. A alternância entre estas ferramentas atrasa as investigações e torna processos como relatórios mensais dependentes de procedimentos manuais.

O USO API é um comando de pesquisa Splunk personalizado que permite consultar seis domínios operacionais de dados diretamente a partir da interface de pesquisa do Splunk. Assente numa arquitetura modular, o comando está organizado em seis módulos que abrangem máquinas geridas, vulnerabilidades, instâncias Splunk, conectividade a bases de dados, processamento de correio eletrónico e registos de identidade. Um painel centralizado com separadores reúne todos os módulos numa vista única. Três subsistemas complementares estendem a solução: o primeiro automatiza relatórios de conformidade com repetição automática em caso de falha; o segundo suporta diagnósticos de infraestrutura com baixo ruído através de comparação com histórico; o terceiro acrescenta uma extensão de monitorização de privacidade que deteta dados pessoais em conteúdos indexados.

O projeto mostra que um único comando de pesquisa Splunk permite a uma equipa de segurança de pequena dimensão aceder diretamente a cinco plataformas externas e uma camada de consolidação de dados internos sem recorrer a *software* intermediário ou a infraestruturas de monitorização adicionais. Do trabalho resultam três contribuições centrais de engenharia transferíveis, validadas no ambiente Splunk: um padrão de integração centrado no SIEM para consulta unificada de infraestrutura, uma arquitetura de prioridades multinível para carregamento eficiente de painéis com pesquisas faseadas, e uma abordagem de diagnóstico baseada em histórico que manteve monitorização com baixo ruído e reduziu a dependência de calibração manual. Uma contribuição de engenharia secundária estende a mesma plataforma com uma abordagem de análise intensiva de padrões para deteção de Personally Identifiable Information (PII) num *pipeline* Security Information and Event Management (SIEM).

Palavras-Chave: SIEM, Splunk, Comando de Pesquisa Personalizado, Integração de Infraestrutura, Deteção de Anomalias, Automação de Conformidade, Deteção de Dados Pessoais.

Contents

List of Figures

List of Tables

Acronyms

AD Active Directory. (*p. 36, 70*)
AI Artificial Intelligence. (*p. 29, 64, 95*)
API Application Programming Interface. (*p. xi, 7, 8, 12, 17–19, 21, 23, 25, 27–29, 31, 40, 43–46, 49, 51, 55, 57, 58, 61, 67, 69, 70, 83, 87–89, 91, 95, 97, 104, 117, 120, 124, 125, 128, 129, 148, 149*)
AQL Ariel Query Language. (*p. 30*)
AWS Amazon Web Services. (*p. 18*)

CIM Common Information Model. (*p. 14*)
CLI Command-Line Interface. (*p. 57, 96, 117, 123*)
CMDB Configuration Management Database. (*p. 17*)
CRUD Create, Read, Update, Delete. (*p. 15*)
CSS Cascading Style Sheets. (*p. 79, 120*)
CSV Comma-Separated Values. (*p. 15, 51, 52, 64, 79, 80, 89, 103, 120, 167*)
CVE Common Vulnerabilities and Exposures. (*p. 20, 69, 75*)
CVSS Common Vulnerability Scoring System. (*p. 20*)

DNS Domain Name System. (*p. 75*)
DOM Document Object Model. (*p. 79, 119, 120*)
DSL Domain-Specific Language. (*p. 11, 30*)
DSR Design Science Research. (*p. 4, 13, 26, 93, 111, 116*)

EDR Endpoint Detection and Response. (*p. 16, 22*)
ELK Elasticsearch, Logstash, Kibana. (*p. 12*)
EPS Events Per Second. (*p. 125*)
ES Enterprise Security. (*p. 10, 12, 14*)

GDPR General Data Protection Regulation. (*p. 10, 36*)

HTML HyperText Markup Language. (*p. 10, 84, 120*)
HTTP HyperText Transfer Protocol. (*p. 27, 61, 70, 84, 90, 96, 97*)
HTTPS HyperText Transfer Protocol Secure. (*p. 46, 56, 61*)

IaC Infrastructure as Code. (*p. 18*)
IDS Intrusion Detection System. (*p. 12*)
IEC International Electrotechnical Commission. (*p. 34, 58*)
IEEE Institute of Electrical and Electronics Engineers. (*p. 34*)
IP Internet Protocol. (*p. 38, 42, 49, 51, 68, 70–72, 87*)
IPC Inter-Process Communication. (*p. 54, 91*)
IPS Intrusion Prevention System. (*p. 12*)
ISO International Organisation for Standardization. (*p. 34, 58*)
IT Information Technology. (*p. 20, 86*)

JSON JavaScript Object Notation. (*p. 10, 23, 54, 67, 69, 80, 82, 85, 87, 91, 96, 117, 118, 128*)
JWT JSON Web Token. (*p. 28, 49, 55, 87, 126*)

KQL Kibana Query Language. (*p. 11, 30*)
KV Key-Value. (*p. xii, 15, 35–39, 46, 53, 71, 82, 83, 89, 96, 97, 102, 105, 127, 162, 169*)

LDAP Lightweight Directory Access Protocol. (*p. 71, 87*)
LogQL Log Query Language. (*p. 30*)

MD5 Message-Digest Algorithm 5. (*p. 80, 83*)
MTTR Mean Time to Respond. (*p. 16*)

NIST National Institute of Standards and Technology. (*p. xiii, 8, 9, 20, 37, 40, 60, 61, 129*)

OAuth Open Authorisation. (*p. 44, 49, 70, 88, 148*)
OWASP Open Web Application Security Project. (*p. 42*)

PCI DSS Payment Card Industry Data Security Standard. (*p. 10*)
PII Personally Identifiable Information. (*p. ii, iii, xi, 2, 3, 5–7, 36–38, 43, 46, 50, 54–56, 77, 78, 85, 86, 90, 95–100, 107, 108, 110, 117, 118, 121, 122, 124–128, 161, 168*)
PromQL Prometheus Query Language. (*p. 30*)

RAM Random Access Memory. (*p. 107*)
RBAC Role-Based Access Control. (*p. 43, 61*)
REST Representational State Transfer. (*p. 12, 15, 16, 21, 31, 45, 46, 56, 57, 61, 67, 68, 76, 79, 83, 84, 89–91, 97, 102, 117, 123, 129, 141, 146*)
RHEL Red Hat Enterprise Linux. (*p. 21*)
RQ Research Question. (*p. 25, 94, 107, 114*)

SANS SysAdmin, Audit, Network, and Security. (*p. 22, 24, 26*)
SDK Software Development Kit. (*p. 16, 27, 45, 48, 54, 56, 59, 64, 67, 68, 80, 82, 87, 88, 96, 101, 117, 118, 123*)
SEM Security Event Management. (*p. 8*)
SFTP SSH File Transfer Protocol. (*p. 97*)
SHC Search Head Cluster. (*p. 13–15, 60*)
SIEM Security Information and Event Management. (*p. ii, iii, 2, 3, 6–13, 16–32, 58, 109, 111, 116, 121, 123–125, 128, 129*)
SIM Security Information Management. (*p. 8*)
SMTP Simple Mail Transfer Protocol. (*p. 52, 53, 84*)
SOAR Security Orchestration, Automation, and Response. (*p. 2, 10–12, 16, 17, 19, 29, 30, 101*)
SOC Security Operations Centre. (*p. 8, 9, 13, 21–26, 31, 129*)
SPL Search Processing Language. (*p. 6, 11, 12, 15, 16, 19, 23, 26–28, 30–32, 35, 41, 42, 45, 46, 48, 49, 51, 59, 60, 64, 70–72, 79, 92, 101, 109, 110, 113, 114, 123, 163*)
SSH Secure Shell. (*p. xi, 3, 18, 19, 34, 38, 40, 44–46, 52, 56–61, 63, 66, 67, 80, 88, 89, 96, 97, 103, 109, 112, 113, 118, 123, 129, 146, 148, 151, 171*)
SSPL Server Side Public License. (*p. 12*)
SUS System Usability Scale. (*p. 5, 94, 99, 101, 110, 111, 113, 114, 116, 122, 125, 126, 129*)

TLS Transport Layer Security. (*p. 98, 99, 122*)
TOFU Trust On First Use. (*p. xi, 58, 59, 112, 171*)
TSIDX Time-Series Index. (*p. 76*)
TTL Time To Live. (*p. 128*)

UEBA User and Entity Behaviour Analytics. (*p. 9*)
UI User Interface. (*p. 103, 113*)
URL Uniform Resource Locator. (*p. 70, 97*)
USO Unified Security Operations. (*p. ii, iii, xi, 4–6, 17, 19, 20, 30, 31, 39, 42, 44, 45, 47, 48, 52, 53, 57, 91, 92, 95, 97, 103, 112, 113, 121, 142*)

VMDR Vulnerability Management, Detection, and Response. (*p. 20*)

WinRM Windows Remote Management. (*p. 18*)

XDR Extended Detection and Response. (*p. 22*)
XML eXtensible Markup Language. (*p. 10, 45, 111*)

YAML YAML Ain't Markup Language. (*p. 18, 19*)

1 Introduction

1.1 Context and Motivation

Security operations within large telecommunications providers involve managing a substantial and heterogeneous infrastructure. At the host organisation, the cybersecurity operations team is responsible for maintaining the security posture of servers, network devices, and applications that support both internal operations and customer-facing services. This responsibility spans vulnerability management, log collection, compliance monitoring, and incident response, in an environment where the threat landscape requires continuous adaptation and visibility across all infrastructure layers (ENISA, 2024).

The team's primary operational platform is Splunk Enterprise, deployed as a search head cluster to handle the volume and variety of security data. The production estate comprises approximately 20 Splunk instances, including log collectors, indexers, and a search-head cluster, supported by dedicated deployment-server and cluster-manager roles[1]. Splunk is the central repository for security events, infrastructure logs, and operational metrics. However, not all relevant information resides within Splunk. Infrastructure details are managed through platforms like Red Hat Satellite, automation is orchestrated via Ansible Automation Platform, and vulnerability data originates from dedicated scanning tools. Each of these systems has its own interface and query mechanisms.

This fragmentation creates friction in daily operations. An analyst investigating an alert may need to check the host's patch status in Satellite, verify its configuration through Ansible, and correlate this with vulnerability scan results, all whilst the primary investigation occurs in Splunk. Context switching between these tools slows response times and increases the cognitive load on analysts (Tilbury et al., 2024).

The motivation for this project arose from an analysis of operational workflows within the cybersecurity operations team aimed at reducing this operational friction.

[1] Specific node counts for each role are anonymised in this work to maintain infrastructure confidentiality.

The project began as a dashboard to bring infrastructure visibility into Splunk. As the project developed, the scope broadened to include automation of manual reporting workflows that consumed approximately one hour each month. Subsequent operational analysis led to a third focus area: infrastructure health diagnostics, which automates the detection of anomalies in log sources, search health, and data pipeline consistency. Throughout the project, the primary thesis thread remained unified observability within Splunk; the compliance report pipeline and later PII capability are presented as secondary operational extensions built alongside that core contribution.

1.2 Problem Statement

The core problem this project addresses is the operational inefficiency caused by tool fragmentation in security operations. Analysts must interact with multiple platforms to gather the information needed for routine tasks, and certain workflows require manual intervention that could be automated.

Specifically, the project targets three related challenges. First, infrastructure visibility is scattered across systems that do not integrate natively with the SIEM system. Querying host status, disk usage, or pending updates requires leaving Splunk and accessing separate administrative interfaces. The same fragmentation affects database connectivity monitoring and email ingestion pipelines, whose health is observable only through their respective management consoles.

Second, monthly compliance reporting involves a repetitive and error-prone manual process. Data must be extracted from Splunk, reformatted to match partner specifications, and uploaded to external servers. This process takes place under deadline pressure, and failures due to network issues or data problems require immediate attention to avoid missing delivery windows.

Third, infrastructure health monitoring relies on manual, ad-hoc checks. Determining whether log sources are reporting correctly, whether scheduled searches complete within acceptable timeframes, or whether data stores approach capacity limits requires individual verification across multiple interfaces. Without automated baselines, degradation in these areas can go unnoticed until it affects downstream operations.

The project scope is deliberately bounded to querying and visibility rather than response automation. It does not implement real-time streaming ingestion, automated remediation, or machine learning-based detection; these capabilities fall within the domain of Security Orchestration, Automation, and Response (SOAR) platforms, which are complementary to rather than replaced by this work. The system also targets a single team's workflows within the host organisation's cybersecurity unit and does not address multi-tenant deployment scenarios.

1.3 Research Questions and Objectives

The challenges identified in Section 1.2 give rise to three research questions. Each targets a measurable aspect of the proposed solution and defines criteria against which Chapter 6 evaluates the outcome.

RQ1: *To what extent does a unified SIEM query interface for heterogeneous infrastructure systems reduce context-switching in security operations workflows?*
Success criterion: analysts complete representative tasks (host investigation, vulnerability triage, pipeline health checks) without leaving the SIEM, reducing the operational context switches required—from an SSH terminal on the Ansible host, several Splunk dashboards, the Satellite web interface, and selected supporting portals—to one unified Splunk workflow.

RQ2: *Can a multi-level priority architecture keep time to first usable active-tab dashboard content below 20 seconds during routine production use?*
Success criterion: observed time to first usable active-tab dashboard content at or below 20 seconds during routine production use after priority-based search scheduling is applied.

RQ3: *Can historical-baseline diagnostics support low-noise infrastructure health monitoring whilst reducing manual threshold calibration?*
Success criterion: historical baselines accommodate natural variance in production diagnostics telemetry without generating spurious alerts over a sustained monitoring period. Comparative performance against static threshold alerting is treated as a qualitative discussion only, since no controlled experiment was conducted within the project timeline.

To investigate them, the project pursues three complementary goals: unified observability, fault-tolerant automation, and proactive infrastructure health monitoring. These goals are not weighted equally within the project. The primary contribution is the unified observability framework inside Splunk, evaluated through RQ1 and RQ2 and extended by the diagnostics work in RQ3. The compliance report pipeline and the PII capability are secondary operational extensions that broaden the platform's practical value without displacing that central observability focus. The first goal centres on a dashboard that consolidates data from the organisation's SIEM platform, infrastructure management systems such as Ansible Automation Platform, and vulnerability scanning tools. This unified view presents infrastructure health, security events, and compliance status within a single application.

The second goal addresses the manual compliance reporting described in Section 1.2. A dedicated compliance report pipeline automates extraction, transformation, and secure delivery, with fault tolerance mechanisms to handle transient failures without human intervention.

The third goal targets proactive infrastructure health monitoring. A diagnostics subsystem automates checks on log sources, scheduled searches, and data stores

through alert actions that compare current metrics against historical baselines.

A modular integration framework, referred to as USO API[2], supports the first two goals by giving analysts a single interface for querying external systems from within Splunk searches, hiding authentication and protocol complexity from end users. The diagnostics subsystem extends this observability focus as an independent Technology Add-on, whilst the compliance report pipeline follows the same deployment pattern as an adjacent operational extension.

1.4 Methodology

This project adopts Design Science Research (DSR) as its methodological framework (Hevner et al., 2004). DSR suits information systems projects whose primary output is a purposeful artefact evaluated against defined objectives (Peffers et al., 2007; Wieringa, 2014). The six-phase process model proposed by Peffers et al. (2007) structures the project; Table 6.1 in Chapter 6 maps each phase to its corresponding chapter.

The project followed an iterative, requirements-driven development approach (Larman et al., 2003), proceeding incrementally in response to concrete operational needs rather than from a complete upfront specification. Initial requirements gathering, conducted through ongoing discussions with the cybersecurity operations team, focused on identifying workflow pain points through direct operational analysis. These discussions revealed the key sources of friction: the need to switch between Splunk and other administrative tools, and the time spent on manual reporting tasks. From this analysis, a clear priority was established: consolidate infrastructure visibility into a Splunk dashboard.

Development began with the integration framework. Before building dashboard visualisations, there needed to be a mechanism for Splunk queries to retrieve data from external systems. This led to the design of USO API, a custom search command that abstracts connections to Ansible, Red Hat Satellite, and other sources. The framework was built incrementally, adding support for each data source as dashboard requirements demanded it.

Once the framework was in place, attention turned to the dashboard. Panels were added iteratively, starting with basic host status information and expanding to include vulnerability summaries, work order tracking, and automation monitoring. User feedback from team members guided decisions about which metrics to display and how to organise the interface.

Beyond the dashboard, two subsystems addressed distinct operational needs. The compliance report pipeline, driven by an immediate need to automate monthly compliance reports, gained its fault-tolerance features (retry logic and data validation) after

[2] "USO API" is an anonymised label used throughout this project report. The term "API" designates the component's architectural role: an operational abstraction layer exposing a uniform programmatic interface to heterogeneous backends, invoked through Splunk's custom search command mechanism rather than through a network service endpoint.

early versions encountered the kinds of temporary failures that occur in production environments. The diagnostics subsystem arose from the recognition that infrastructure health checks were performed manually and inconsistently, and its modular architecture allows individual diagnostic checks to be added incrementally as monitoring needs are identified.

Throughout development, testing occurred in a dedicated Splunk development instance before changes were deployed to production. Validation initially adopted a production-centric strategy, with reliability verified through user acceptance testing and continuous monitoring of real-world usage patterns. As the codebase matured, an automated test suite spanning all three subsystems was added across the core USO API, diagnostics subsystem, and PII detection tool, covering action parsing, module dispatch, credential management, notification delivery, and PII detector validation. These tests must pass before changes are submitted for production deployment. The testing approach is discussed in detail in Chapter 6.

1.5 Project Structure

The remainder of this project is organised as follows:

Chapter 2: Background and Related Work surveys the theoretical foundations and related tools, identifying gaps in current solutions that motivate the contributions presented in subsequent chapters.

Chapter 3: Requirements Analysis defines the functional, non-functional, and security requirements derived from operational needs.

Chapter 4: System Architecture presents the high-level design of the solution, including the integration framework, the compliance report pipeline (automated monthly delivery to external partners), the diagnostics subsystem, and the PII detection subsystem (automated scanning for exposed personal data).

Chapter 5: Implementation describes the technical realisation of the system, covering the development environment, the construction of USO API, dashboard logic, diagnostics implementation, and PII detection.

Chapter 6: Evaluation and Discussion evaluates the solution against its requirements and research questions, presenting performance analysis, operational impact metrics, and lessons learned.

Chapter 7: Conclusions and Future Work summarises contributions and outlines directions for future work.

The project report concludes with five appendices and one annex. **Appendix A** provides extended implementation details for the dashboard, row expansion architecture, and credential management tools. **Appendix B** reproduces high-resolution anonymised screenshots of the production dashboard. **Appendix C** presents annotated query examples illustrating the system's integration patterns. **Appendix D** documents the operational security controls and anonymisation methodology. **Appendix E** presents the synthetic evaluation corpus for the PII detection subsystem. Finally, **Annex L** reproduces the SUS questionnaire used during the usability assessment.

1.6 Expected Contributions

This project contributes to security operations practice and to the study of SIEM-infrastructure integration. The contributions fall into four categories.

Primary Contribution: SIEM-Centric Integration Pattern. The project's principal contribution is a registry-dispatch architecture that extends the SIEM's native query language to heterogeneous infrastructure systems without external middleware. The pattern unifies six backend access mechanisms behind a single `| usoapi module=X action=Y` invocation syntax, abstracts authentication and protocol complexity from analysts, and produces output that composes with standard Search Processing Language (SPL) operators. This integration pattern is described in Chapter 4 and validated in Chapter 6.

Technical Contributions. Six operational artefacts implement and extend the integration pattern within the host organisation's production environment:

- The integration framework (USO API), a custom Splunk search command exposing six integration modules that query infrastructure, vulnerability, compliance, and identity data from external platforms.
- A unified dashboard that consolidates infrastructure health, vulnerability status, and automation state into a multi-tab interface with priority-based search scheduling.
- A compliance report pipeline, treated as a secondary operational extension, that automates the monthly reporting process described in Section 1.2, running end-to-end with retry mechanisms for transient failures.
- A diagnostics subsystem that performs automated health checks on log sources, scheduled searches, and data stores against historical baselines.
- A privacy-monitoring extension for PII detection that scans indexed data for exposed personal information through regex-based detectors with algorithmic validation, using a compiled detection engine for high-throughput scanning.
- A token-based authentication scheme that centralises credential management through Splunk's encrypted credential store and search head cluster replication.

Empirical Contributions. The framework was validated through sustained production deployment (five months, approximately 29,000 command invocations across the observation period) and an exploratory usability pilot study ($N = 3$, census sample from the complete eligible expert population). Operational metrics include workflow comparison data, compliance pipeline delivery rates, diagnostics pass rates, and dashboard load-time observations. These findings are presented in Chapter 6.

Transferable Contributions. Three core engineering contributions are transferable beyond the specific deployment context: the registry-based integration pattern itself (applicable to any SIEM with an extensibility API), the multi-level priority architecture for rapid first-view dashboard usability under routine production conditions, and the historical-baseline diagnostics approach for infrastructure health monitoring. A fourth, secondary contribution is a high-throughput pattern-scanning approach for PII detection within a SIEM pipeline. Each is bounded by explicit scope limitations discussed in Section 7.2.5.

2

Background and Related Work

This chapter surveys the theoretical foundations, peer-reviewed academic research, and existing industrial solutions relevant to unified security operations dashboard design. It covers SIEM architecture and Security Operations Centre (SOC) operations, the Splunk platform, security orchestration and automation, vulnerability management, dashboard design, and API integration patterns, then synthesises these academic and industrial perspectives in the gap analysis that motivates the project.

2.1 SIEM Fundamentals and Modern Security Operations

2.1.1 SIEM Architecture and Evolution

Security Information and Event Management (SIEM) systems are central infrastructure for modern cybersecurity operations (González-Granadillo et al., 2021). Organisations use them to detect, analyse, and respond to security threats through centralised log aggregation and real-time monitoring.

A SIEM aggregates event data from heterogeneous sources (applications, networks, endpoints, and cloud platforms) (Davies et al., 2025) and applies correlation rules, analytics, and threat intelligence to identify security incidents. The architecture comprises three main components.

Data collection agents gather logs from diverse sources, and normalisation processes convert these disparate formats into a common schema for correlation across different data types. Once normalised, correlation engines apply rules and statistical models to identify threat patterns, triggering alerts that notify security personnel of incidents requiring investigation.

SIEM technology emerged from the convergence of Security Information Management (SIM) and Security Event Management (SEM), fusing long-term compliance storage with real-time alerting to address both immediate threat detection and historical analysis (González-Granadillo et al., 2021). This convergence relies on structured log management, a discipline formalised by the NIST in Special Publication 800-92, which

provides guidelines for establishing log management infrastructures, handling diverse log formats, and ensuring data integrity (Scarfone et al., 2006). Modern frameworks such as the NIST Cybersecurity Framework (CSF) 2.0 (Pascoe et al., 2024) and its guidelines for continuous monitoring (Dempsey et al., 2011) formalise these capabilities within the Detect and Respond functions, emphasising the need for automated visibility across heterogeneous environments.

Modern SIEM platforms have moved well beyond log collection. They now function as advanced analytical systems. Machine learning algorithms detect anomalous behaviour patterns automatically without requiring manually defined rules, whilst User and Entity Behaviour Analytics (UEBA) capabilities help identify insider threats and compromised user accounts by establishing behavioural baselines and flagging statistically significant deviations. Threat intelligence feeds add global context to locally observed events, helping analysts distinguish genuine threats from false positives based on external threat actor information. Many platforms now also support hybrid and cloud-native deployments (Davies et al., 2025).

2.1.2 SIEM in SOC Operations

Within contemporary SOC, SIEM systems form the operational core, offering unified visibility across an organisation's security posture. This reliance aligns with the Detect and Respond functions of the NIST incident handling framework (Nelson et al., 2025), which require comprehensive observability to identify the scope and impact of potential compromises. SOC analysts rely on these platforms to triage alerts, investigate incidents, and coordinate responses (Crowley, 2024), treating them as a unified view through which teams monitor multiple security controls.

Vielberth et al. (2020) survey SOC architectures across the technology, process, and people dimensions, identifying tool integration as a persistent challenge when technology stacks comprise numerous products with limited interoperability. Kokulu et al. (2019) corroborate this finding through qualitative fieldwork across several SOCs, observing a fundamental mismatch between deployed tools and actual analyst workflows: contextual information required for alert triage frequently resides outside the analyst's primary interface, forcing time-consuming context switches between applications. Sundaramurthy et al. (2016) apply activity theory to SOC operations, revealing how analysts construct informal tools and workarounds to bridge gaps in their official toolsets. Recent 2024 studies further quantify these challenges through the lens of human-automation teaming; these research findings demonstrate how tool fragmentation and the "swivel chair" effect contribute to cognitive overload and "alert tyranny" (Tilbury et al., 2024). These studies collectively characterise a working environment in which tool fragmentation impedes the speed and accuracy of security investigations.

Modern deployments support several operational workflows, beginning with threat detection through correlation rules and machine learning algorithms that identify potential incidents (Splunk Inc., 2024b). Investigation tools offer analysts search capabilities and visualisations to examine alerts in depth, whilst response orchestration

integrates with SOAR platforms to automate containment and remediation.

Regulatory frameworks have accelerated SIEM adoption across industries. General Data Protection Regulation (GDPR), the NIS2 Directive, Payment Card Industry Data Security Standard (PCI DSS), and sector-specific regulations mandate continuous monitoring, audit logging, and incident detection capabilities (European Parliament and Council of the European Union, 2016; European Parliament and Council of the European Union, 2022; PCI Security Standards Council, 2025), which means many implementations serve the dual purposes of security operations and regulatory compliance.

Despite these capabilities, SIEM deployments face persistent operational challenges that limit effectiveness (Davies et al., 2025; Agyepong et al., 2020). Alert fatigue remains among the most widely reported challenges, as security analysts are frequently overwhelmed by high volumes of false positives that obscure genuine threats and lead to important alerts being missed or dismissed (Jalalvand et al., 2024). Rule development and tuning demands deep expertise in both security domain knowledge and platform-specific query languages, requiring continuous refinement as threat landscapes and infrastructure evolve. Integration with legacy systems that lack modern logging capabilities or use proprietary log formats often presents technical hurdles requiring custom parsers. These operational limitations have driven development of next-generation SIEM architectures focused on automation through SOAR integration and contextual analytics that reduce false positive rates.

2.1.3 Commercial SIEM Solutions

The three dominant commercial platforms each take distinct architectural approaches that influence their extensibility for infrastructure monitoring use cases.

Splunk Enterprise Security (ES) extends the core Splunk platform with security-specific dashboards, correlation searches, and notable event management (Splunk Inc., 2026h). The Common Information Model (CIM) handles data normalisation across log sources, requiring ingested data to conform to predefined field mappings for dashboards to function correctly (Splunk Inc., 2024b).

Beyond data normalisation constraints, dashboard development in Splunk also presents architectural choices. Classic Simple XML dashboards use an eXtensible Markup Language (XML) schema with panel definitions, form inputs, and token-based interactivity. Dashboard Studio, introduced in Splunk 8.2, uses a JavaScript Object Notation (JSON) structure offering more layout flexibility but deprecating HyperText Markup Language (HTML) dashboards. For security operations centres requiring custom visualisations beyond Enterprise Security (ES)'s built-in offerings, the transition to Dashboard Studio adds migration overhead whilst removing capabilities some workflows depend upon.

The platform's extensibility through custom search commands supports integration with external systems, though ES itself offers no native connectivity to infrastruc-

ture management tools like Ansible or Red Hat Satellite. Security analysts must either build custom integrations or rely on separate monitoring stacks for infrastructure health visibility.

IBM QRadar positions itself as an AI-augmented security platform with user behaviour analytics, network detection and response, and SOAR integration (IBM, 2026). Machine learning models score threats based on magnitude, credibility, and relevance factors. The platform's consumption-based licensing and complex deployment requirements present barriers for smaller teams, as each data source and analytics module adds to both cost and administrative overhead.

Operational complexity extends to automation capabilities; QRadar provides no direct integration with Ansible for infrastructure automation. Red Hat's certified Ansible content collection for QRadar permits playbooks to configure log sources and manage offences, but this represents SIEM-to-automation connectivity rather than the reverse: QRadar dashboards cannot directly query infrastructure state through Ansible.

Elastic Security builds on the Elastic Stack, featuring a resource-based cost model that contrasts with the volume-based licensing of traditional commercial platforms (Elastic N.V., 2025b). Elasticsearch manages distributed search and data storage, Kibana delivers the interface for analysis and visualisation, and the detection engine evaluates data against rules and machine learning models to generate alerts. Under this model, costs scale with consumed resources rather than with data-volume-based licence tiers.

Kibana Query Language (KQL) and Elasticsearch Query Domain-Specific Language (DSL) serve different use cases. KQL offers a simpler syntax for ad-hoc searches whilst Query DSL supports programmatic precision. Security analysts familiar with Splunk's SPL face a learning curve, and the dual-language model complicates knowledge transfer between teams.

Integration with SOAR platforms requires external orchestration tools. Elastic includes detection rules and alerting, but automated response playbooks depend on third-party solutions or custom scripting. Shuffle, an open-source SOAR platform, integrates with Elastic deployments to close this gap, though the combination requires separate deployment and maintenance.

Comparative Analysis Table 2.1 summarises the capabilities relevant to infrastructure monitoring integration. None of the commercial platforms provide native connectivity to Ansible or Red Hat Satellite, requiring custom development for organisations seeking unified visibility across security and infrastructure domains.

The absence of infrastructure monitoring integration in commercial SIEMs reflects their design focus: security event correlation rather than operational visibility. Organisations running Red Hat infrastructure cannot query patch status, errata (advisory notices describing available patches, categorised as security, bugfix, or enhancement) counts, or subscription compliance through their SIEM dashboards without building custom data pipelines.

Table 2.1: *Commercial SIEM Infrastructure Integration Capabilities*

Capability	Splunk ES	QRadar	Elastic
Custom Search Commands	Yes	Limited	Via plugins
Ansible Infrastructure Querying[a]	No	No	No
Satellite Integration	No	No	No
Open Source	No	No	Core only[b]
Dashboard Customisation	Simple XML/Studio	Built-in	Kibana

[a] All three have Ansible collections for SIEM management (e.g. `splunk.es`, `ibm.qradar`), but none supports querying infrastructure hosts through the SIEM search interface.
[b] Elastic changed from Apache 2.0 to Server Side Public License (SSPL)/Elastic License 2.0 in 2021 (Elastic N.V., 2021).

2.1.4 Open-Source Alternatives

The SIEM market includes both commercial and open-source solutions, each with distinct advantages and constraints. Self-managed implementations of the Elasticsearch, Logstash, Kibana (ELK) Stack and Graylog offer cost-effective alternatives with high customisability, benefiting from active community development and open-core flexibility (Elastic N.V., 2025a). These solutions require substantial internal expertise for deployment and maintenance, however, and often lack the full support structures that commercial vendors supply.

Commercial platforms including Splunk Enterprise Security, Microsoft Sentinel, and Exabeam deliver enterprise-grade features alongside professional support (Davies et al., 2025). Splunk has established market leadership through its SPL and extensive app ecosystem, though commercial platforms generally command significant licensing costs offset by reduced implementation complexity.

Platform selection depends on architectural fit rather than just cost. Organisations with mature engineering teams may prefer open-source flexibility for custom pipelines, whilst those prioritising rapid capability adoption often select commercial platforms to minimise maintenance overhead. Hybrid approaches are common, using open-source tools for initial log aggregation before forwarding to commercial SIEMs for advanced analytics.

Practical open-source deployments combine multiple tools into layered stacks: a perimeter firewall with Intrusion Detection System (IDS)/Intrusion Prevention System (IPS) capabilities, lightweight shippers forwarding logs to Elasticsearch, Kibana dashboards rendering operational views, and a SOAR layer automating response through Representational State Transfer (REST) API calls. Such architectures demonstrate that cost-effective security automation is achievable without commercial licensing (Elastic N.V., 2025a). These implementations require expertise across numerous platforms, however, and version compatibility between components introduces upgrade complexity. Organisations with limited security engineering resources may find the total effort exceeds that of managed commercial alternatives despite the licence savings.

The common thread across both commercial and open-source SIEM solutions is their exclusively inward focus: they excel at correlating data they already ingest but provide no native mechanism for querying external infrastructure state. Commercial platforms invest in pre-built detection content and compliance frameworks, whilst open-source alternatives offer customisation at the cost of operational overhead. Neither category extends the analyst's query language to encompass infrastructure management tools such as Ansible or Red Hat Satellite. This gap forces security operations teams to maintain parallel monitoring stacks and manually correlate findings across disconnected interfaces, a workflow pattern that empirical research consistently identifies as a primary source of analyst inefficiency (Kokulu et al., 2019; Agyepong et al., 2020). Metrics-oriented research confirms this gap: Forsberg et al. (2023) propose a DSR-based performance metric framework for SOC technology and note that tool integration remains an under-measured dimension of SOC effectiveness.

2.2 Splunk Platform Architecture

Since the project targets Splunk Enterprise as its runtime environment, this section introduces the platform's architecture, data model, and extensibility points. These concepts underpin the system design presented in Chapter 4 and the implementation decisions discussed in Chapter 5.

2.2.1 Distributed Architecture and Data Flow

Splunk Enterprise follows a distributed architecture in which three principal roles cooperate to ingest, store, and query machine data (Splunk Inc., 2026h).

Search heads execute user queries, serve the web interface, and run dashboards and saved searches. In production environments, multiple search heads form a Search Head Cluster (SHC) that shares configuration and distributes workload across members. On the storage side, **indexers** receive incoming data, parse it, and write it to disk; when a search head issues a query, each indexer scans its local data stores and returns partial results. Deploying additional indexers increases both storage capacity and query throughput.

Forwarders are lightweight agents installed on data sources. Universal Forwarders collect and forward raw data without parsing, whilst Heavy Forwarders can parse and filter data before transmission. In many environments, intermediate collectors aggregate streams from multiple forwarders before routing them to indexers. Data therefore flows in one direction through the pipeline: sources send events to forwarders, forwarders relay them to indexers, and search heads query the indexed data on demand. Query execution reverses this path: the search head distributes a query to all relevant indexers, each indexer scans its local buckets, and the search head merges their partial results into the final output.

Several supporting roles manage cluster coordination. The *deployer* pushes configuration bundles to SHC members; the *deployment server* distributes application bundles and configuration updates to its registered forwarders (*deployment clients*), which periodically poll the server for updated content; and the *cluster manager* (formerly master node) coordinates indexer clustering for data replication and search factor compliance.

The query distribution described above operates within a single Splunk deployment and is termed *distributed search* (Splunk Inc., 2026b). Splunk also supports *federated search*, which extends this model across independent deployments: a search head queries saved searches or datasets on a remote Splunk instance through the `| from federated:` command (Splunk Inc., 2026f). Whilst distributed search scales query throughput within one deployment, federated search links separate deployments without replicating data between them. Chapter 5 uses this capability to query the deployment server from the search head cluster for forwarder fleet monitoring.

2.2.2 Data Model: Indexes, Sourcetypes, and Sources

Splunk organises ingested data around three primary metadata fields (Splunk Inc., 2026h).

An **index** is a logical data repository backed by a set of directories on disk. Data within each index is partitioned by time into buckets, which allows the platform to restrict searches to relevant time ranges without scanning the entire data store. Splunk ships with multiple default indexes: `_internal` for platform logs, `_audit` for access records, and `main` as the default destination. Administrators create custom indexes to isolate data by type, retention policy, or access control requirements.

To classify the format of ingested data, Splunk assigns each event a **sourcetype** (for example, `syslog`, `access_combined`, or `WinEventLog`) that determines how fields are extracted from raw text. Sourcetype definitions reside in `props.conf`. A **source** identifies the origin of data, such as a file path, network port, or script output.

Together, index, sourcetype, and source form the metadata triple attached to every event (Splunk Inc., 2026h). At a higher level, the Common Information Model (CIM) normalises field names across sourcetypes to support cross-source correlation (Splunk Inc., 2024a). Splunk Enterprise Security requires CIM-compliant data for its standard correlation searches (Splunk Inc., 2024b), though this project queries data through its own field mappings rather than ES-specific CIM models.

2.2.3 Application and Configuration Model

Splunk organises functionality into **apps**: self-contained packages that bundle dashboards, search commands, saved searches, lookups, and static assets under a single directory in `$SPLUNK_HOME/etc/apps/` (Splunk Inc., 2026c). Each app contains a `default/` directory with shipped configuration and an optional `local/` directory for administrator customisation.

Technology Add-ons (TAs) are lightweight apps that deliver data inputs, field extractions, and data normalisation without user-facing dashboards. TAs typically supply `inputs.conf`, `props.conf`, and `transforms.conf`.

Splunk merges configuration files across directories using a deterministic precedence order: system defaults, then app `default/`, then app `local/`, then user-level overrides. This layering allows apps to ship baseline settings that administrators can customise without modifying original files. Several configuration files are particularly relevant to the project described in later chapters:

- `commands.conf` registers custom search commands;
- `savedsearches.conf` defines saved and scheduled searches, including alert triggers;
- `alert_actions.conf` declares custom alert action parameters;
- `app.conf` stores app metadata such as label, version, and visibility;
- `collections.conf` defines KV Store collections;
- `transforms.conf` specifies lookup definitions and field transforms.

Chapters 4 and 5 show how the project uses these files to register custom commands, schedule automated pipelines, and define credential and collection schemas.

2.2.4 Data Storage Beyond Indexes

Besides time-series indexes, Splunk offers two additional storage mechanisms suited to structured reference data (Splunk Inc., 2026h).

The **KV Store** is a MongoDB-backed key–value store built into the platform. Collections are defined in `collections.conf` and accessed through the REST API or SPL commands (`| inputlookup` and `| outputlookup`). Unlike indexes, KV Store collections support full Create, Read, Update, Delete (CRUD) operations, which makes them appropriate for stateful data such as configuration tables, enrichment caches, or diagnostic history records. In SHC deployments, KV Store collections replicate across members automatically.

Lookups are static enrichment files, most commonly Comma-Separated Values (CSV), that map field values to additional context. They are defined in `transforms.conf` and invoked with the `| lookup` command in SPL. Lookup files reside in the app's `lookups/` directory and can be managed through Splunk Web or the REST API.

Both KV Store collections and CSV lookups appear in the project's identity resolution, health monitoring, and diagnostic subsystems, as described in Chapters 4 and 5.

2.2.5 Search Execution and Extensibility

When a user submits an SPL query, the search head parses it and distributes work to the relevant indexers. Streaming operations such as `eval`, `where`, and `regex` execute on indexers in parallel, whilst transforming operations (`stats`, `chart`, `timechart`) run

on the search head after partial results have been collected. This distribution model is central to Splunk's ability to query large data volumes within acceptable response times.

Saved searches run on a configurable schedule. When configured as alerts, they trigger actions (email notifications, webhook calls, or custom alert actions) whenever results match specified conditions. Alert actions receive a payload containing the triggering search results and associated metadata, which allows downstream processors to act on the data without re-executing the original query.

Custom search commands, registered in `commands.conf`, extend SPL with operations implemented in Python or other languages; Section 2.7.1 examines the Software Development Kit (SDK) framework and command types in detail. Splunk also exposes a management-plane REST API under `/services/`, through which scripts and custom commands can query server health, index configurations, credential storage, and search job status programmatically. Section 2.7.3 discusses how integration credentials are stored and retrieved through this interface.

2.3 Security Orchestration and Automation

2.3.1 SOAR Concepts and SIEM-SOAR Integration

SOAR has reshaped security operations, letting organisations integrate disparate security tools, automate repetitive tasks, and accelerate incident response workflows (Islam et al., 2019). SOAR platforms address a persistent operational limitation: the manual, time-intensive processes traditionally required to investigate alerts and execute remediation actions. By codifying security procedures into automated playbooks, these systems improve both the speed and consistency of incident response whilst freeing analysts to focus on complex investigative work.

SOAR emerged from the convergence of incident response management and standalone automation, reflecting a shift from isolated task execution towards integrated workflows spanning the full incident lifecycle (Islam et al., 2019). This transition addressed the limitations of disparate security toolsets, which often necessitated manual coordination and hampered rapid response capabilities (Vielberth et al., 2020).

Contemporary platforms rest on three core capabilities. Orchestration integrates security technologies (SIEM systems, Endpoint Detection and Response (EDR) platforms, firewalls, threat intelligence services, and vulnerability scanners) into unified workflows. Through programmatic playbooks, automation eliminates manual tasks by executing predefined action sequences in response to defined triggers or conditions. Response capabilities offer full incident management frameworks covering case management, task assignment, collaboration, and documentation.

These capabilities translate into measurable operational gains. Mean Time to Respond (MTTR) decreases as automation accelerates both investigation and remediation activities, whilst analyst productivity increases through the elimination of repetitive

manual tasks. Standardised playbooks also increase response consistency by codifying practitioner expertise into documented, executable workflows.

SIEM and SOAR are complementary: SIEM excels at data aggregation, correlation, and alert generation whilst SOAR provides the orchestration and automation capabilities needed to act efficiently on those alerts. The most prevalent pattern involves automated alert enrichment and triage workflows. When SIEM generates an alert, SOAR automatically initiates investigative playbooks that gather contextual information from multiple sources: querying threat intelligence platforms, examining endpoint detection tools, and correlating findings with vulnerability scan data. The resulting enriched alert includes threat reputation scores, endpoint telemetry, and vulnerability exposure data that the raw SIEM alert lacks.

Bi-directional integration patterns maintain synchronised state throughout the incident lifecycle, ensuring that response actions taken by SOAR are logged back to the SIEM platform. This creates detailed audit trails and ensures that correlation engines factor remediation activities into ongoing analysis.

Vendors are blurring the traditional boundaries between SIEM and SOAR through unified platform architectures. Splunk SOAR (formerly Phantom) integrates natively with Splunk Enterprise Security (Splunk Inc., 2024b), whilst Microsoft Sentinel includes built-in automation and orchestration capabilities as native platform features. Industry analysis identifies a bifurcation: standalone SOAR platforms offering vendor-agnostic integration across heterogeneous environments versus ecosystem-first solutions that prioritise deep integration within a single vendor's product suite (Islam et al., 2019).

Beyond security tool orchestration, infrastructure automation forms an important dimension of SOAR response capabilities. Many incidents require actions beyond dedicated security tools, such as system patching and network device reconfiguration. Configuration management tools such as Ansible, originally developed for DevOps workflows, now serve security response contexts (Red Hat, Inc., 2025c). Integration patterns between SOAR and infrastructure automation frameworks vary in sophistication, from basic playbook invocation via API to advanced deployments that query Configuration Management Databases (CMDBs) to determine remediation targets based on real-time infrastructure state.

Despite these integration possibilities, SOAR platforms introduce additional infrastructure complexity and often lack native integration with bespoke monitoring systems. The USO API approach implements targeted automation within the SIEM layer itself, using custom search commands to query infrastructure state directly, without routing requests through separate orchestration platforms.

2.3.2 MITRE ATT&CK Framework

The MITRE ATT&CK (Adversarial Tactics, Techniques, and Common Knowledge) framework is a globally accessible knowledge base of adversary behaviour derived from real-world observations (The MITRE Corporation, 2026). Organised as a matrix of tactics (the adversary's objectives) and techniques (the methods used to achieve those objectives), ATT&CK offers a structured vocabulary for describing how threat actors operate across the stages of an intrusion (Strom et al., 2018). The framework's relevance to SIEM integration extends beyond log-based detection: when a correlation rule flags a technique such as T1021 (Remote Services), the investigation benefits from knowing whether the target host has recently been patched, whether its configuration aligns with hardening baselines, and whether related infrastructure changes are scheduled. This contextual enrichment illustrates the need for cross-domain visibility, where understanding a host's patch status, automation queue position, and configuration state requires data from systems outside the SIEM's native data stores.

2.3.3 Infrastructure Automation

Infrastructure as Code (IaC) replaces manual configuration processes with machine-readable definitions that specify desired infrastructure states (Morris, 2021). Provisioning tools such as Terraform and Amazon Web Services (AWS) CloudFormation focus on resource creation, defining cloud infrastructure declaratively. Configuration management tools, including Ansible, Puppet, and Chef, address post-provisioning system state: installing software, enforcing security baselines, and maintaining configuration consistency across server estates (Kumara et al., 2021). The declarative versus imperative distinction is relevant: declarative tools specify desired end states whilst the platform determines required actions, whereas imperative approaches like Ansible playbooks define explicit step-by-step execution sequences. For security remediation workflows, the imperative model offers explicit control over execution order and clear visibility into each action taken.

Ansible is an agentless automation platform distinguished by operational simplicity (Red Hat, Inc., 2025b). It requires no agents on managed nodes. Unlike competing tools, Ansible communicates through standard protocols including SSH for Unix-like systems and Windows Remote Management (WinRM) for Windows environments, eliminating deployment overhead whilst reducing the attack surface.

The architecture comprises multiple interconnected components. Playbooks, written in accessible YAML Ain't Markup Language (YAML) syntax, define ordered task sequences that describe automation workflows in human-readable format. Tasks invoke modules, which are discrete functionality units that abstract platform-specific implementation details, supporting cross-platform automation through consistent interfaces. The inventory system defines managed hosts and organises them into logical groups, supporting both static file-based definitions and dynamic scripts that query external sources such as cloud provider APIs or Configuration Management Databases.

Variables and templates, powered by the Jinja2 templating engine, handle parameterisation and reusability across different deployment contexts.

Ansible's module library includes thousands of modules organised into functional collections. Core modules handle system management tasks including command execution, file manipulation, package installation, and service control.

Ansible's agentless design and human-readable YAML syntax lower the adoption barrier relative to agent-based alternatives such as Puppet and Chef, which demand dedicated infrastructure for certificate management and agent deployment. This simplicity, however, still imposes scaling limits: Ansible uses five concurrent forks by default unless configured otherwise (Red Hat, Inc., 2025a), meaning response time increases with host count. For security operations where query latency directly affects investigation speed, complementary patterns such as increasing fork concurrency or result caching become necessary as managed estates grow.

2.3.4 Ansible in Security Operations

Ansible's capabilities extend naturally into security operations, with security teams deploying the platform for varied use cases spanning preventive, detective, and responsive security controls (Red Hat, Inc., 2025c; Billoir et al., 2024). The platform's idempotent nature and detailed logging make it particularly suitable for security contexts where auditability and predictability are essential operational requirements.

Compliance automation is another common use case, with organisations encoding hardening standards into playbooks that audit configurations and enforce consistent system states (Center for Internet Security, 2022). Ansible Automation Platform and its open-source counterpart AWX add role-based access control, credential management, and RESTful APIs for enterprise integration.

The USO API implementation uses Ansible's agentless architecture to query managed hosts directly via SSH, retrieving host status, uptime data, and system state. Exposing this information within SPL lets analysts correlate infrastructure state with security events and determine, for instance, whether a host flagged in an alert has pending restarts or recent configuration changes.

Collectively, these automation and orchestration capabilities reveal a consistent architectural pattern: security operations benefit most when detection, investigation, and response actions share a common execution context. SOAR platforms advance this objective by integrating security tools into automated workflows, and ATT&CK establishes a shared vocabulary for mapping detection coverage against adversary behaviour. Infrastructure automation extends this logic to remediation and state queries. The remaining difficulty is bridging these domains at the analyst's query interface. Whilst SOAR platforms orchestrate actions across tools, they do not expose infrastructure state within the SIEM's search language, leaving analysts unable to correlate host health or automation status with security events during ad-hoc investigations.

2.4 Vulnerability Management and Patch Workflows

2.4.1 Lifecycle and Scanning Tools

Vulnerability management is a continuous process through which organisations identify, assess, prioritise, and remediate security vulnerabilities across their Information Technology (IT) infrastructure. NIST frames patch management as "preventive maintenance for technology", an ongoing effort that reduces risk before exploitation occurs (Souppaya et al., 2022). The lifecycle spans discovery, assessment, prioritisation, remediation, and verification, with effective programmes balancing broad coverage against risk-based resource allocation.

Vulnerability scanning tools automate the discovery and assessment of security vulnerabilities across IT assets. Commercial solutions such as Tenable Nessus and Qualys Vulnerability Management, Detection, and Response (VMDR) offer broad vulnerability detection with enterprise management features (Tenable, Inc., 2025; Qualys, Inc., 2025). Both platforms are deployed within the host organisation's environment, where their findings are indexed in Splunk and surfaced through the dashboard's vulnerability views.

Vulnerability identification relies on curated databases maintained by scanning vendors and security research organisations. The Common Vulnerabilities and Exposures (CVE) system assigns standardised identifiers that allow consistent vulnerability tracking across tools and organisations (The MITRE Corporation, 2025). The National Vulnerability Database aggregates these entries and enriches them with severity scores, reference links, and affected product metadata (National Institute of Standards and Technology, 2025), whilst the Common Vulnerability Scoring System (CVSS) quantifies severity through factors such as exploitability, technical impact, and scope (FIRST.org, 2025).

Scanning platforms increasingly integrate with SIEM systems for event correlation, though this connectivity does not extend to querying infrastructure state from within the SIEM's search language.

Prioritisation solely by CVSS scores often proves ineffective because it fails to account for factors such as asset criticality, existing compensating controls, and active exploitation intelligence (Allodi et al., 2014). Modern risk-based approaches, such as the Exploit Prediction Scoring System (EPSS) (FIRST.org, 2023) and Stakeholder-Specific Vulnerability Categorization (SSVC) (CERT Coordination Center, 2024), incorporate these factors to generate better remediation guidance. CISA's Vulnrichment project (Cybersecurity and Infrastructure Security Agency, 2024) demonstrates the implementation of SSVC at scale by enriching CVE records with decision-based metadata to assist organisations in triage.

Vulnerability data integration forms a core component of the USO API dashboard. The team can view remediation status alongside infrastructure health within a unified interface, whilst the dashboard correlates vulnerability counts with Ansible host groups to identify which infrastructure segments require attention.

2.4.2 Red Hat Satellite

Red Hat Satellite centralises content management, patching, and provisioning for Red Hat Enterprise Linux (RHEL) infrastructure (Red Hat, Inc., 2024a). Content Views stage package versions through lifecycle environments to permit controlled promotion from development through production. The REST API (version 2) exposes host inventory, errata applicability, subscription status, and hostgroup membership for programmatic access (Red Hat, Inc., 2024b).

The native Satellite web interface serves administrators performing patch management and content synchronisation. Security teams seeking visibility into vulnerability exposure across hosts encounter limitations: the interface emphasises management actions over monitoring dashboards. Errata reports list applicable patches per host but do not integrate with security event correlation or threat detection workflows.

Authenticated API access supports automation without embedding credentials directly in dashboard definitions (Red Hat, Inc., 2024a; Red Hat, Inc., 2024b). No native Satellite connector was identified in the surveyed commercial SIEM product documentation, so organisations must build custom integrations to surface this data in security dashboards.

2.4.3 Nagios, Icinga, and Monitoring Tools

Nagios established the open-source infrastructure monitoring pattern: check plugins execute against hosts, returning status codes that the core engine aggregates into service and host states (Nagios Enterprises, LLC, 2024). The architecture scales through distributed monitoring and passive check acceptance, supporting deployments across thousands of hosts.

Nagios Core offers host and service monitoring through check plugins, but its web interface and file-based configuration model lack modern API capabilities for integration with external platforms. Icinga originated as a Nagios fork, introducing improved APIs and a contemporary web interface (Icinga GmbH, 2024b; Icinga GmbH, 2024a); it likewise lacks native SIEM integration. In the surveyed documentation, both tools focus on host and service availability monitoring rather than the security event correlation, log aggregation, and vulnerability context that a SOC requires.

The tools surveyed in this section address vulnerability lifecycle management from different angles: scanning tools detect vulnerabilities, management platforms like Satellite track patch deployment, and monitoring tools such as Nagios and Icinga offer host-level health visibility. None integrates directly with a SIEM's query interface. Security teams seeking to correlate vulnerability exposure with detected threats or infrastructure health must extract data from each tool separately and perform manual correlation, a limitation that extends to the dashboard domain examined in the following section.

2.5 Dashboard Design for Security Operations

2.5.1 Challenges and Patterns

Security operations dashboards present unique design challenges that distinguish them from general business intelligence interfaces. The SysAdmin, Audit, Network, and Security (SANS) 2024 SOC Survey identifies lack of enterprise-wide visibility as a recurring barrier to SOC effectiveness, with respondents ranking it alongside competing priorities and staffing shortages (Crowley, 2024). This visibility gap persists. It manifests most clearly in organisations where security monitoring, infrastructure management, and vulnerability remediation operate through separate toolsets that do not share context.

SOC analysts face cognitive demands that directly impact dashboard requirements. The same survey reports that 2–10 analysts constitute the most common SOC team size, yet these teams must monitor alerts from EDR/Extended Detection and Response (XDR) systems (the most common incident trigger), correlate events with SIEM data, and execute remediation across distributed infrastructure. Dashboard design must accommodate this workload concentration by minimising context-switching between tools, a challenge compounded when infrastructure state resides in systems external to the SIEM platform.

Alert fatigue represents another persistent challenge. Respondents ranked "too many alerts that we can't look into" and "lack of context related to what we are seeing" among their reported challenges (Crowley, 2024), a finding corroborated by systematic reviews of SOC operations (Jalalvand et al., 2024). This lack of integration often forces analysts to develop informal tools and scripts to bridge functionality gaps between vendor products—a phenomenon sometimes described as the "swivel-chair" effect (Sundaramurthy et al., 2016; Kokulu et al., 2019). These studies suggest that effective security dashboards must deliver contextual enrichment beyond raw alert counts to inform rapid triage decisions. For infrastructure-focused security teams, this context includes asset health, pending maintenance tasks, and automation queue status, information often absent from traditional SOC dashboards.

The information visualisation literature presents theoretical grounding for dashboard design principles. Few (2013) establishes foundational patterns for operational dashboards, emphasising at-a-glance monitoring through reduced visual clutter and data-pixel ratio optimisation. Building on this, Shneiderman (1996) proposes a visual information-seeking mantra: "overview first, zoom and filter, then details on demand." This taxonomy maps directly to the hierarchical dashboard architecture observed in security operations: summary panels present aggregate indicators, interactive filters narrow the scope to particular host groups or time ranges, and drill-down navigation exposes individual records. Applied to infrastructure security dashboards, the overview corresponds to fleet-wide health indicators, filtering corresponds to host-group or severity selection, and details-on-demand corresponds to individual host inspection through row expansion or drill-down searches.

Infrastructure monitoring dashboards differ from alert-centric SOC interfaces in their emphasis on system state rather than discrete events. Whilst SOC dashboards prioritise incident queues and threat indicators, infrastructure dashboards track host availability, service health, configuration compliance, and remediation progress. Drawing from foundational cognitive science, Few (2013) argues that effective dashboards must align with the operator's mental model of the underlying system to reduce the cognitive effort required for situational awareness—a principle corroborated by recent neuroergonomic research into security operations. Security teams responsible for infrastructure protection require hybrid approaches that synthesise both perspectives.

Splunk's dashboard framework supports such integration through Dashboard Studio and the classic Simple XML format (Splunk Inc., 2026h). Dashboard Studio employs a JSON-based definition language with absolute positioning for precise layout control suited to operational displays. The framework supports data binding to SPL queries, allowing dashboards to combine native Splunk data with results from custom search commands that query external systems.

Splunk Enterprise Security includes pre-built dashboards addressing incident review, asset investigation, and threat intelligence workflows (Splunk Inc., 2024b). These dashboards demonstrate design patterns applicable to custom implementations: hierarchical navigation from summary to detail views, consistent colour semantics (red for critical, amber for warning, green for normal), and integration of diverse data sources through correlated searches. Dashboard performance considerations become significant at scale. Splunk's search head architecture distributes query execution across indexers, but dashboard load times depend on query complexity and result set sizes. For dashboards incorporating custom commands that query external APIs, additional latency from network requests must be managed through timeout handling and caching strategies where data freshness requirements permit.

2.5.2 Grafana, Prometheus, and Kibana

Open-source visualisation platforms offer flexibility that commercial SIEM dashboards constrain. The trade-off involves operational complexity and the absence of integrated security correlation capabilities.

Prometheus collects time-series metrics through a pull model by scraping endpoints that expose instrumented data (Prometheus Authors, 2024). Grafana subsequently renders these metrics through configurable dashboards that support templating, alerting, and multiple data source backends (Grafana Labs, 2024a). The combination is widely used in Kubernetes and cloud-native observability stacks (Cloud Native Computing Foundation, 2020).

For security use cases, the architecture presents limitations. Prometheus stores metrics, not logs or events (Prometheus Authors, 2024); security monitoring requires complementary log aggregation through Loki or a similar system (Grafana Labs, 2024b). Grafana features its own alerting engine (Grafana Labs, 2024a), but alert handling lacks

the behavioural analysis and threat intelligence integration that commercial SIEMs offer. Organisations adopting Grafana for security visibility build bespoke solutions combining several tools instead of deploying an integrated platform.

Kibana's capabilities for security monitoring overlap with Elastic Security, discussed in Section 2.1.3. The distinction matters for organisations using Elasticsearch without the security-focused additions: raw Kibana delivers visualisation but not detection rules, threat intelligence integration, or case management.

Dashboard creation in Kibana supports both visual builders and saved-object import and export, offering flexibility beyond Splunk's SimpleXML (Splunk Inc., 2026h; Elastic N.V., 2025c) but requiring familiarity with Elasticsearch's data model and query syntax.

2.5.3 Role-Based Dashboard Design

Different security roles require fundamentally different information presentations. The SANS survey distinguishes between security administrators, SOC analysts, and security managers as primary SOC roles (Crowley, 2024). Each persona approaches dashboards with distinct objectives and time constraints.

SOC analysts require detailed alert queues with contextual enrichment that allows rapid triage. For infrastructure-focused analysts, this includes system health indicators alongside security alerts—determining whether an affected host is already scheduled for maintenance affects incident prioritisation. Information density must balance completeness against cognitive load; progressive disclosure patterns present summary indicators with drill-down access to supporting detail.

Security engineers focus on trending metrics for system health and configuration compliance, whilst leadership requires aggregated programme-effectiveness summaries for governance decisions.

Multi-tenancy capabilities let a single dashboard platform serve this range of requirements through personalised views. Splunk's role-based access control determines dashboard visibility, whilst configurable default views can present role-appropriate initial displays. This approach maintains a single source of truth whilst adapting presentation to audience requirements. Khan et al. (2023) corroborate the importance of these role-specific patterns, identifying eight groups of design patterns from a systematic review of 144 dashboards across domains. Their analysis confirms that dashboards serving operational decision-making benefit from consistent spatial layouts and progressive disclosure, whilst dashboards targeting strategic audiences require aggregated trend visualisations. For security operations dashboards that span multiple data sources, these patterns must accommodate varying data freshness and latency characteristics alongside role-specific information requirements.

Research and practice converge on several principles for security dashboard design: hierarchical information architecture from overview to details on demand, role-appropriate presentation, and reduced context-switching through consolidated views.

The practical challenge lies in implementing these principles when the data originates from heterogeneous sources outside the SIEM. Commercial dashboard features focus on data that the SIEM has already ingested; Grafana and Kibana depend on their own backend data stores. None provides a mechanism for querying live infrastructure state alongside security events within a single dashboard, a gap that extends to the API design considerations examined in the next section.

2.6 Linking Research Questions to Literature

The preceding sections surveyed SIEM architectures (Section 2.1), security orchestration and infrastructure automation (Section 2.3), vulnerability management (Section 2.4), and dashboard design for security operations (Section 2.5). Across these domains, three recurring gaps emerged that no commercial product or research prototype fully addresses. This section makes the connection explicit by mapping each research question from Section 1.3 to the literature that motivates it and to the evaluation method employed in Chapter 6.

Table 2.2: *Research question grounding in surveyed literature*

RQ	Literature Gap	Key Sources	Evaluation Method
RQ1	Tool fragmentation forces analysts to context-switch across disconnected interfaces; no SIEM offers native infrastructure-state querying	Kokulu et al. (2019), Vielberth et al. (2020), Sundaramurthy et al. (2016), Tilbury et al. (2024), Agyepong et al. (2020)	Workflow comparison, context-switching reduction, exploratory usability pilot
RQ2	Security dashboards aggregating external sources face latency from API calls; no framework addresses SIEM dashboard load under multi-source conditions	Few (2013), Shneiderman (1996), Khan et al. (2023), Forsberg et al. (2023)	Browser-observed load times, priority queue telemetry
RQ3	Static alerting thresholds generate false positives during routine variance; adaptive baselines lack empirical SOC validation	Jalalvand et al. (2024), Crowley (2024), Forsberg et al. (2023)	Production telemetry, four-week deviation monitoring

RQ1: Unified query interface and context-switching. Kokulu et al. (2019) conduct qualitative fieldwork across several SOCs and identify a fundamental mismatch between the tools organisations deploy and the workflows analysts actually follow: contextual information needed for alert triage frequently resides outside the analyst's primary interface. Vielberth et al. (2020) corroborate this finding through a systematic survey of SOC architectures, cataloguing tool integration as a persistent challenge across the technology, process, and people dimensions. Sundaramurthy et al. (2016) apply

activity theory to the same operational context and show that analysts construct informal workarounds to bridge gaps that their official toolsets leave open. More recent work by Tilbury et al. (2024) quantifies the consequences through the lens of human-automation teaming, demonstrating how the "swivel chair" effect across fragmented tools contributes to cognitive overload and alert fatigue. Agyepong et al. (2020) reinforce these observations through a systematic review of SOC analyst challenges and performance metrics. Collectively, these studies establish that tool fragmentation degrades investigative speed and accuracy, yet none proposes extending the SIEM's own query language to encompass infrastructure management systems. RQ1 addresses this gap directly by asking whether a unified SPL interface reduces context-switching in practice.

RQ2: First usable dashboard content under priority loading. Few (2013) establishes foundational principles for operational dashboards, emphasising at-a-glance monitoring through reduced visual clutter and data-pixel ratio optimisation. Shneiderman (1996) formalises the "overview first, zoom and filter, then details on demand" mantra that maps to the hierarchical dashboard architecture common in security operations. Khan et al. (2023) extend this foundation through a systematic review of 144 dashboards, identifying design patterns for progressive disclosure and consistent spatial layouts that improve operational decision-making. Forsberg et al. (2023) apply DSR to SOC performance measurement and observe that tool integration remains an undermeasured dimension of SOC effectiveness. These works provide design guidance for dashboards that display pre-ingested data, but none addresses how a SIEM dashboard should sequence mixed-latency searches so that analysts receive useful content quickly during routine use. RQ2 targets this gap by evaluating whether priority-based search scheduling keeps time to first usable active-tab content within an operationally acceptable bound.

RQ3: Historical-baseline anomaly detection. Jalalvand et al. (2024) survey alert prioritisation methods across the SOC domain and identify false positive management as a core challenge: the volume of alerts generated by static detection rules overwhelms analyst capacity and erodes trust in monitoring systems. The SANS 2024 SOC Survey confirms this operationally, with respondents ranking "too many alerts" and "lack of context" among their most significant challenges (Crowley, 2024). Forsberg et al. (2023) note that technical performance metrics for SOCs remain underdeveloped, particularly for detection quality and the trade-off between sensitivity and false positive rates. These findings motivate an alternative to static thresholds for infrastructure health monitoring: historical baselines that accommodate natural data-volume variance without generating spurious alerts. RQ3 evaluates this approach through sustained production monitoring.

2.7 API Design and Custom Search Commands

Enterprise application integration (EAI) theory presents a broader architectural lens for understanding SIEM extensibility patterns. Hohpe et al. (2003) catalogue sixty-five messaging and integration patterns, several of which map to custom search command designs. A registry-based dispatch architecture, where incoming requests specify a target module and action that a central dispatcher routes to the appropriate handler, follows the Message Router pattern: the router examines message content to determine the destination without the sender needing to know implementation details. The related Content-Based Router pattern applies when dispatch decisions depend on request parameters such as target system type or authentication requirements. Framing custom search commands within this pattern vocabulary positions SIEM integration architectures alongside established enterprise middleware designs, clarifying the architectural decisions involved in bridging heterogeneous systems through a unified query interface.

2.7.1 Custom Search Commands in Splunk

Splunk's extensibility architecture lets organisations augment the SPL with custom commands that integrate external data sources, encode domain-specific logic, or abstract complex query patterns into reusable interfaces (Splunk Inc., 2026e). These extensions operate transparently within SPL pipelines, which lets analysts invoke custom functionality using syntax indistinguishable from built-in commands. For security operations, custom commands unlock integration scenarios that native Splunk capabilities cannot address: querying infrastructure management APIs or triggering automated remediation.

The Splunk SDK for Python supplies the development framework for custom command implementation (Splunk Inc., 2024c). Commands inherit from base classes that handle protocol negotiation, input parsing, and output serialisation, so that developers can focus on business logic. The SDK categorises commands by their data flow characteristics: generating commands create events from external sources without requiring pipeline input, streaming commands process events individually as they flow through the pipeline, reporting commands aggregate events to produce statistical summaries, and events commands receive the full set of pipeline events for collective processing. Command type determines execution semantics. Streaming commands execute in parallel across indexers, whilst generating commands run on a single node.

2.7.2 External API Integration Patterns

Custom commands commonly implement patterns that bridge Splunk's query environment with external systems. API integration commands issue HyperText Transfer Protocol (HTTP) requests to external services during search execution, allowing SPL queries to incorporate data from threat intelligence platforms or infrastructure moni-

toring APIs. The command receives target specifications through SPL arguments, constructs appropriate API requests, parses responses, and emits results as Splunk events.

Database integration commands extend this pattern to relational data sources. Using database connector libraries, commands can query external databases and return results as Splunk events. This capability supports correlation between SIEM event data and business context stored in data warehouses—enrichment that would otherwise require data replication or manual lookup procedures.

Infrastructure automation integration represents a third pattern relevant to security operations. Commands can invoke Ansible playbooks or query Ansible Automation Platform APIs. Through this integration, SPL queries correlate security events with infrastructure state: identifying which hosts require restarts, checking automation queue depths, or verifying that remediation playbooks have executed successfully.

Performance considerations constrain external API integration patterns. Network latency from API calls accumulates across large result sets, potentially causing search timeouts. Effective implementations employ connection pooling, batch requests where APIs support them, and timeout handling that degrades gracefully rather than failing entirely. Caching strategies reduce redundant API calls when data freshness requirements permit, though cache invalidation complexity must be weighed against latency benefits.

2.7.3 Credential Management for Integrations

External API integration demands secure credential storage. Custom commands must access these credentials at runtime. Splunk's storage/passwords API grants this capability, encrypting credentials at rest and managing access through Splunk's role-based access control system (Splunk Inc., 2026g). Custom commands retrieve credentials programmatically during execution, avoiding embedded secrets in source code or configuration files that create exposure risks through version control or backup systems.

The storage/passwords API organises credentials through a namespace hierarchy. Each credential belongs to an application context and realm, grouped logically by target system or credential type. For multi-system integrations, a label-based organisation pattern proves effective: credentials include descriptive labels that support programmatic selection of appropriate tokens for intended operations.

Token-based authentication patterns dominate modern API integrations. JSON Web Tokens (JWTs) encode authorisation claims in self-contained, cryptographically signed payloads that achieve stateless authentication (Jones et al., 2015). For machine-to-machine scenarios where custom commands authenticate to external services, long-lived tokens simplify operations but require secure storage. The storage/passwords API addresses this requirement by centralising token management, generating audit trails of credential access, and supporting credential rotation.

Service accounts created specifically for automation and custom command execution inherit Splunk roles that grant necessary capabilities whilst adhering to least-privilege principles. Token-based authentication using these accounts permits custom commands to access external APIs with appropriate, audited credentials.

2.8 Academic Research on SIEM Integration

2.8.1 SIEM-SOAR Integration Studies

The integration of SIEM with external data sources has received sustained attention in the systems security literature. González-Granadillo et al. (2021) survey the evolution of SIEM architectures and identify persistent challenges in data normalisation across heterogeneous log formats, scalability limitations as log volumes grow, and cross-service visibility gaps in multi-tenant deployments. A more recent systematic review by López Velásquez et al. (2023) traces a parallel evolution, with growing emphasis on reducing analyst workload and on compliance-driven deployment patterns. Cloud-native environments intensify these challenges: security data spans multiple providers and account boundaries, requiring API-based ingestion, agentless data capture, and real-time event streaming to maintain centralised visibility.

Existing integration patterns share a common directionality. SOAR orchestration platforms address operational challenges by automating response workflows, but the data flow is predominantly inbound: cloud logs and threat intelligence feeds flow *into* the SIEM, and automated actions flow *out* to SOAR playbooks. The reverse pattern—exposing infrastructure management state (host health, patch compliance, automation queue depth) within the SIEM's own query language so that analysts can correlate it with security events during ad-hoc investigations—remains unaddressed in the surveyed literature. This directional gap motivates the integration approach explored in this project.

2.8.2 AI-Driven Security Automation

Jalalvand et al. (2024) survey the intersection of machine learning and SIEM-based alert prioritisation, documenting how human-Artificial Intelligence (AI) teaming can reduce false positive rates and combat alert fatigue through automated triage and contextual enrichment (Ban et al., 2021). Their analysis highlights that effective automation requires feedback loops between analyst expertise and algorithmic detection, rather than purely autonomous classification. Machine learning integration for anomaly identification in user behaviour and network traffic features prominently in 2024-era security architectures, yet the emphasis remains on detection and prioritisation of security events rather than on querying infrastructure state.

Existing Ansible integrations with SIEM platforms follow a SIEM-to-automation direction: the security platform triggering playbook execution (Red Hat, Inc., 2025c; Billoir et al., 2024). The reverse direction, where Ansible execution results appear in SIEM dashboards for operational visibility, remains absent from the studies surveyed.

2.9 Gap Analysis and Project Positioning

The preceding sections reveal a consistent industrial-academic mismatch: tool fragmentation is well described in the literature, whilst commercial and open-source platforms still optimise for their own data models and operational ecosystems. SIEM platforms aggregate security data but require operator intervention to correlate infrastructure health with security events. SOAR systems automate response workflows but depend on pre-integrated connectors that rarely extend to bespoke infrastructure management tools. Ansible enforces configuration and drives remediation, but does so outside the SIEM's analytical scope; vulnerability management programmes produce prioritised patch lists yet have no direct channel to the infrastructure teams who execute them. No existing tool spans these domains without bespoke integration work.

Table 2.3 synthesises these findings, positioning the USO API framework against existing alternatives.

Table 2.3: *Infrastructure-SIEM Integration Gap Analysis*

Requirement	Commercial SIEM	Grafana	Nagios	USO API
Unified Query Language	Partial[a]	No[b]	No	Yes
Ansible Execution Results	No	No	No	Yes
Satellite Patch Status	No	No	No	Yes
SPL Pipeline Integration	Splunk only	No	No	Yes
Single Credential Store	No	No	No	Yes
No Additional Licensing	No	Yes	Yes	Yes

[a] Within each platform's own query language only
[b] Grafana delegates queries to backend-specific languages (Prometheus Query Language (PromQL), Log Query Language (LogQL))

Unified Query Interface Commercial SIEMs expose query languages for their own data stores. Splunk's SPL queries indexed events; QRadar's Ariel Query Language (AQL) searches its database; Elastic accepts KQL and Query DSL. None extends to query external systems through the same interface. Security analysts seeking infrastructure state must either consult separate tools or build custom data ingestion pipelines.

The USO API addresses this through a custom Splunk search command (Section 2.7.1) that accepts parameters specifying the target system and action. The same SPL query syntax that analysts use for security investigations extends to infrastructure queries, eliminating context switching and supporting correlation between security events and infrastructure state.

Ansible Integration Direction Existing Ansible-SIEM integrations flow from SIEM to Ansible: security events trigger playbook execution for automated response. The reverse direction, where Ansible execution results appear in SIEM dashboards, lacks equivalent tooling. Organisations seeking visibility into Ansible-managed infrastruc-

ture health through their security platform find no commercial or open-source solution addressing this requirement.

The USO API's Ansible module executes ad-hoc commands against infrastructure targets and returns results as SPL-compatible events. Dashboard panels can display disk usage, reboot requirements, or custom command output alongside security metrics, delivering operational context for security decisions.

Red Hat Satellite Visibility Satellite's REST API exposes patch status, errata applicability, and subscription compliance, but the data remains isolated from security monitoring workflows. No SIEM platform indexed on this research provides native Satellite integration. Organisations running Red Hat infrastructure cannot assess vulnerability exposure through their security dashboards without custom development.

The Satellite module in the USO API queries errata, host information, and host-group membership through authenticated API calls. Results surface in Splunk dashboards where analysts can correlate patch compliance with security event patterns.

Credential Centralisation Multi-system monitoring requires credentials stored across separate configuration files, each with its own management lifecycle. The USO API consolidates authentication through Splunk's `storage/passwords` API, inheriting its encryption, access control, and cluster-replication capabilities. Three credential families are managed this way: per-host bearer tokens for remote Splunk authentication, a username-and-password pair for Red Hat Satellite, and a client-credential configuration for Microsoft Graph that is used at runtime to obtain short-lived access tokens.

2.9.1 Operational Challenges at the Host Organisation

Two operational challenges faced by the host organisation's cybersecurity operations team directly inform the project objectives.

Challenge 1: Infrastructure visibility within SIEM query workflows. The team's existing Splunk deployment affords limited native support for querying external infrastructure state during security investigations. When an analyst investigates an alert, they cannot readily determine whether the affected host requires a restart, whether its critical processes are responding, or whether queued remediation tasks are pending. This context resides in separate systems (Ansible Automation Platform, infrastructure monitoring tools, custom databases), requiring analysts to context-switch between interfaces and manually correlate information from disparate sources.

Challenge 2: Dashboards for infrastructure-focused security operations. Standard SOC dashboards focus on threat detection and alert triage. The cybersecurity operations team requires different visibility: host inventory status, automation queue depths, vulnerability remediation progress, and system restart requirements. These

infrastructure-centric dashboards must integrate with existing Splunk workflows without requiring separate tooling, and must present data from multiple external systems in a unified view.

Contribution Positioning This work contributes a framework that extends Splunk's query model to heterogeneous infrastructure systems without requiring additional commercial licensing. The approach differs from commercial SIEM add-ons that address security use cases exclusively, and from open-source monitoring stacks that lack security event correlation. By implementing the integration as a custom search command, the framework preserves compatibility with existing SPL skills, dashboard patterns, and operational workflows. Chapter 3 formalises the requirements derived from these gaps and operational challenges.

3 Requirements Analysis

This chapter defines the functional, non-functional, security, and privacy requirements for the unified security operations dashboard. The requirements address the operational inefficiencies identified in Chapter 1: fragmented infrastructure visibility, manual compliance reporting, and reactive health monitoring. Requirements follow IEEE 29148 conventions (IEEE, 2018), using *shall* for mandatory provisions and *should* for recommended provisions.

3.1 Stakeholder Analysis

The primary stakeholders are the Splunk infrastructure team within the host organisation's cybersecurity department. This team administers the department's Splunk platform and the operational workflows for infrastructure monitoring, vulnerability management, and compliance reporting.

Requirements emerged through ongoing conversations with the team, following the iterative and face-to-face practices characteristic of agile requirements engineering (Cao et al., 2008). Initial discussions confirmed the workflow friction described in Section 1.1. As development progressed, additional requirements surfaced through direct observation of analyst workflows and feedback on early prototypes.

The team requires visibility into infrastructure health and automation status to support both rapid status verification and detailed technical investigations. This shaped the dashboard design: summary panels offer at-a-glance confirmation of fleet-wide health, whilst detailed tables support the diagnostic work that follows when an anomaly surfaces.

3.2 Functional Requirements

The functional requirements are organised around the core capabilities that the system must deliver, following the functional and non-functional classification described in International Organisation for Standardization (ISO)/International Electrotechnical Commission (IEC)/Institute of Electrical and Electronics Engineers (IEEE) 29148 (IEEE, 2018). Each requirement corresponds to a particular operational need identified during the requirements gathering process.

3.2.1 FR1: Infrastructure Querying

The system shall query host status and health information from within Splunk searches.

Analysts need to determine whether hosts are reachable without leaving Splunk. A connectivity test action verifies that the Ansible control node can reach target hosts via SSH. Basic host information including hostname resolution and system uptime shall be retrievable to support incident investigation and change verification. Disk usage monitoring is equally important because storage exhaustion on Splunk infrastructure causes indexing failures; the system shall report partition usage with adjustable threshold alerts to identify hosts approaching capacity limits early.

Host restart status is required to identify systems pending reboot after kernel or critical package updates. This information, requiring direct system access via SSH, shall be queryable through the Splunk interface. File operations support troubleshooting and configuration verification: the system shall allow searching for configuration files across hosts and retrieving file contents for comparison, particularly when analysts need to verify configuration consistency across Splunk infrastructure components.

3.2.2 FR2: Splunk Integration

The system shall monitor Splunk infrastructure beyond what analysts can observe through the standard web interface.

Instance health monitoring shall report the status of Splunk services across the infrastructure. The system shall retrieve overall health status and, optionally, detailed feature-level health information for troubleshooting. This check requires appropriate user permissions, specifically the `list_health` capability on target instances.

Internal queue status monitoring is necessary for identifying ingestion bottlenecks. Queue fill percentages for indexer pipelines shall be reported so that analysts can detect data flow problems before they cause visible symptoms. This requires the `list_introspection` capability.

Beyond real-time monitoring, search job history supports both troubleshooting and capacity planning. Recent job records and, importantly, failed job logs that include error details shall be retrievable. These logs are otherwise only accessible through the file system of the search head where the job executed.

Data management operations require visibility into KV store collections. The system shall list collections with their configurations and field definitions, so that analysts can review the state of application data stores.

To support rapid identification of recurring problems, internal log analysis consolidates ERROR and WARN messages from Splunk's internal logs, ranked by frequency. This avoids requiring analysts to construct complex SPL queries against the `_internal` index.

3.2.3 FR3: Database Connectivity Monitoring

The system shall expose the status of database extraction processes. The team operates extraction scripts that pull data from external databases on scheduled intervals. Failures in these processes are not immediately visible through Splunk's native monitoring.

The system shall list configured database extractions with their connection parameters. The system shall also support testing connectivity to individual database targets, reporting whether the extraction script can establish a connection. The check builds on existing extraction infrastructure instead of implementing direct database connectivity.

3.2.4 FR4: Vulnerability Management

The system shall integrate with Red Hat Satellite to report vulnerability and patch status. The team manages approximately 50 hosts through Satellite, and analysts need to correlate security events with host patch status during incident investigation.

Host inventory queries shall return registered hosts with their operating system version, lifecycle status, and global health status as reported by Satellite. The global status indicates whether hosts have pending actions or configuration issues.

The system shall list available errata across the managed estate, filterable by type (security, bugfix, enhancement) and severity. For incident investigation, analysts need host-specific errata queries that identify exactly which patches a particular host requires.

Subscription and hostgroup information supports inventory management, though these represent lower-priority requirements compared to errata visibility.

3.2.5 FR5: Email Pipeline Monitoring

The system shall monitor email ingestion pipelines that feed Splunk indexes. The team manages automated extraction of emails from shared mailboxes, with ingestion rates and pipeline health representing key operational metrics.

Mailbox health queries shall return folder statistics including message counts and size totals. This information supports capacity planning and anomaly detection when ingestion rates deviate from expected patterns. The system shall retrieve mailbox data through the Microsoft Graph API instead of legacy Exchange protocols.

Pipeline status monitoring shall track extraction job execution, reporting success rates, failure details, and processing timestamps. Ingestion status queries shall verify that emails are reaching their target Splunk indexes at expected rates, with user-defined thresholds for alerting on stale data.

3.2.6 FR6: Identity and Access Visibility

The system shall query identity and access information from within Splunk searches. Security investigations frequently require establishing the organisational context of accounts and hosts involved in an event: who owns a given server, which groups a suspect account belongs to, what systems a user can access, or whether an account remains active. These queries currently involve switching between Active Directory (AD) administration tools, the access governance portal, and HR systems, each with its own interface and query semantics.

User profile lookups shall return consolidated identity records drawn from Active Directory, the access governance platform, and HR records. Group membership queries shall list the groups a user belongs to and, conversely, the users within a given group. Access rights lookups shall return the catalogue of systems and permissions associated with a user through the access governance platform, for access reviews and audit responses.

Splunk role mapping shall derive the effective Splunk roles for a given user by matching their Active Directory group memberships against the role-mapping configuration. This supports permission troubleshooting and access reviews without requiring Splunk administrator access. Host identity enrichment shall return operational context for infrastructure hosts, including team ownership, environment classification, security zone, and support contact, sourced from the internal infrastructure inventory.

Proactive identity monitoring shall detect accounts that have not authenticated within operator-specified time windows and identify gaps in identity coverage where hosts or users lack expected records in identity stores. All identity queries operate against local Splunk lookups and KV stores, requiring no external credentials beyond the standard command execution context.

3.2.7 FR7: PII Exposure Detection

The system shall scan indexed data for PII within Splunk search pipelines. Log sources across the enterprise infrastructure may inadvertently contain personal data such as tax identifiers, phone numbers, email addresses, and financial account numbers. Detecting this exposure is an operational requirement driven by regulatory obligations under the GDPR and the Network and Information Security Directive (NIS2) (European Parliament and Council of the European Union, 2016; European Parliament and Council of the European Union, 2022). These mandates are further formalised by international standards such as ISO/IEC 27701:2019, which extends established information

security management controls to include specific privacy information management requirements (International Organization for Standardization, 2019). Compliance is also supported by modern operational frameworks such as the NIST Cybersecurity Framework (CSF) 2.0 (Pascoe et al., 2024), which defines continuous monitoring and data protection as core functional requirements for critical infrastructure operators. Beyond regulatory compliance, PII detection prevents accidental disclosure during investigation workflows.

The detection capability shall cover at least fourteen PII types spanning both Portuguese-specific identifiers (such as fiscal numbers and IBANs) and universal patterns (such as email addresses and credit card numbers). Each detector applies validation logic appropriate to its type, including algorithmic verification and contextual keyword matching to suppress false positives in machine-generated log data. Section 4.7.2 presents the full detector catalogue with its validation strategy.

Three operating modes shall address distinct analyst workflows. A counts mode augments each event with per-type detection counts for dashboard aggregation. A detail mode returns matched text and detector metadata for forensic investigation. A redaction mode replaces detected PII inline, which supports safe data sharing and export.

Confidence scoring classifies each detection as low, medium, or high based on validator outcomes and contextual evidence. Detectors prone to false positives require supporting context keywords within a proximity window before reporting a match, and analysts can filter results by minimum confidence level.

Detection results shall be presented through a dedicated dashboard tab for operational monitoring. A scheduled search scans selected indexes at regular intervals, indexing summaries for trend analysis and alerting.

3.2.8 FR8: Unified Dashboard

The system shall present a unified dashboard that consolidates infrastructure status without requiring analysts to execute ad-hoc queries for routine monitoring.

The dashboard shall be organised into sixteen tabs addressing distinct operational concerns:

1. **Summary**: At-a-glance metrics showing overall infrastructure health, including host counts, Splunk component status, database connection status, and recent activity indicators.
2. **Infrastructure**: Host monitoring combining Satellite inventory data with real-time status from Ansible queries. The tab shall display host health, errata status, disk usage, and pending reboot requirements.
3. **Sources and Data**: Data ingestion monitoring showing KV store collections, lookup file freshness, and database extraction configurations. The tab shall support data quality assurance and troubleshooting of missing data.

4. **Firewall**: Firewall log monitoring and analysis, displaying connection events, policy violations, and traffic patterns relevant to the managed infrastructure.
5. **Data Health**: Data pipeline integrity monitoring, displaying diagnostic results from automated health checks across log sources, KV stores, and scheduled searches.
6. **Vulnerabilities**: Patch status visibility sourced from Red Hat Satellite and other vulnerability management tools, displaying security errata counts, affected hosts, and remediation priorities.
7. **Suspicious Activity**: Correlation of host security logs to detect anomalies including failed SSH authentication patterns, brute-force attempts, identification of external IP addresses, and unauthorised `sudo` activity.
8. **Work Orders**: Integration with the organisation's ticketing system to display active work orders affecting infrastructure.
9. **Licenses**: Splunk licence consumption metrics with trend visualisation, to identify capacity issues before violations occur.
10. **Forwarders**: Universal forwarder monitoring showing deployment status, version distribution, and connectivity health.
11. **Automation**: Compliance report pipeline monitoring showing execution status, validation results, retry history, and failure details.
12. **Performance**: Splunk search performance metrics including search head cluster activity, job completion rates, and resource utilisation.
13. **Logs Status**: Data freshness monitoring across all ingested log sources, with colour-coded indicators for sources that have not received recent events.
14. **Extractor**: Database extraction pipeline status showing scheduled jobs, execution history, and connectivity test results.
15. **Email**: Email pipeline monitoring displaying mailbox health, ingestion rates, and extraction job status for all configured email sources.
16. **PII Exposure**: PII detection results across indexed data, showing detection counts by type, confidence distribution, and affected indexes. The tab shall consume output from scheduled scans and support investigation of data protection exposure.

Interactive elements shall support drill-down investigation. Row expansion capabilities shall reveal detailed information for selected items without navigating away from the dashboard. Conditional formatting shall highlight items requiring attention through colour coding for status indicators and threshold-based alerts.

The dashboard shall support periodic refresh at intervals appropriate to each component's update frequency, instead of continuous real-time polling. Infrastructure status changes infrequently enough that refresh intervals of several minutes are sufficient, balancing data freshness against resource consumption.

The Automation tab warrants clarification regarding scope. The compliance report pipeline operates as an independent subsystem with its own scheduled execution, error handling, and notification mechanisms. The dashboard observes this pipeline by

querying its logging output rather than controlling its execution. This loose coupling means the pipeline functions independently of the USO API; the dashboard delivers visibility into its status without creating operational dependencies.

Although the compliance report pipeline represents a significant operational deliverable, it is treated as a secondary extension of the main observability platform and is specified here under FR8 because this chapter defines requirements for the integrated operational interface rather than specifying every autonomous subsystem in isolation. The pipeline's requirement-facing role in this scope is observability: analysts must monitor execution status, validation outcomes, and failures through the Automation tab. Its internal processing stages (extraction, transformation, upload, and verification) are engineered as a standalone subsystem and are detailed in Chapter 4 and Chapter 5.

3.2.9 FR9: Infrastructure Health Diagnostics

The system shall automate infrastructure health diagnostics that validate the integrity of data pipelines, search operations, and application state. Manual verification of these components across approximately 50 managed hosts does not scale; the system shall therefore perform scheduled validation runs that surface anomalies as indexed events and operator notifications.

The diagnostics capability shall cover three categories of infrastructure health checks:

- **Search job validation**: Failed saved searches, excessive runtimes, and abnormal result counts shall be detected and flagged.
- **File integrity monitoring**: The system shall verify essential configuration and data files against expected checksums, sizes, and age thresholds.
- **KV Store validation**: Collection row counts shall be checked against expected thresholds to detect data loss or synchronisation failures.

The system shall maintain historical baselines for comparison, so that anomaly detection accounts for normal operational variance rather than relying solely on static thresholds. Diagnostic results shall be indexed to a dedicated summary index (`uso_diagnostics`), separating diagnostic telemetry from operational data.

The system shall support configurable notification channels, allowing operators to select which channels receive alerts and to distinguish between success and failure conditions. Reliability mechanisms shall follow the same patterns established in NFR3, including adjustable retry logic with exponential backoff.

The architectural design of the diagnostics subsystem, including its module dispatcher, historical comparison engine, and notification backend model, is described in Section 4.6.

3.3 Non-Functional Requirements

Non-functional requirements define the quality attributes that the system must exhibit. These requirements emerged from operational realities and organisational standards rather than explicit stakeholder requests.

3.3.1 NFR1: Performance

Command execution shall complete within a timeframe acceptable for interactive use. The acceptance threshold is 20 seconds for typical queries against a single host or small host group. In practice, most such queries are expected to return faster, but 20 seconds defines the upper bound used for evaluation. This constraint reflects both user expectations and practical limitations of the underlying infrastructure.

Network latency dominates execution time for most operations. Queries that invoke SSH connections to the Ansible control node, which then connects to target hosts, accumulate latency at each hop. The system shall not introduce significant additional overhead beyond these inherent network delays.

Dashboard loading shall prioritise perceived responsiveness. Searches shall execute in parallel where dependencies permit, and panels shall render progressively as results become available. Initial page load shall present meaningful content within 20 seconds under normal conditions.

3.3.2 NFR2: Security

The system shall not store credentials in plain text. All authentication materials, including API tokens and SSH key references, shall be stored using Splunk's encrypted credential storage mechanism. This requirement aligns with organisational security policies and the credential management controls prescribed by NIST SP 800-53 (Joint Task Force Interagency Working Group, 2020). Security mechanisms should further align with industry best practices for API security, such as those defined in the OWASP API Security Top 10, particularly regarding broken object-level authorisation and insecure consumption of APIs (OWASP Foundation, 2023). Audit logging shall record all command executions to maintain accountability and support forensic analysis. Section 4.8 presents the security architecture and its alignment with international standards.

Token-based authentication shall be preferred over service accounts where technically feasible. Bearer tokens confer several advantages: they inherit user role capabilities automatically, support straightforward revocation, and eliminate the need to maintain dedicated service accounts across distributed infrastructure.

The system shall operate within the user's existing permission boundaries. Operations shall fail cleanly when users lack required capabilities rather than attempting privilege escalation. Error messages shall indicate the exact missing capability for troubleshooting.

3.3.3 NFR3: Reliability

The system shall handle errors gracefully without crashing or producing cryptic failures. When external systems are unreachable or return errors, the command shall report the failure clearly and exit normally, allowing subsequent SPL pipeline operations to process the error indication. Specifically, graceful handling requires that the command exits with a zero return code and emits at least one result row containing an `error` field with a human-readable description identifying the affected target and the failure category.

The system shall achieve a search completion rate of at least 99% under normal operating conditions, measured as the ratio of completed search invocations to total search invocations recorded in Splunk audit logs over a 30-day observation window.

For automated pipelines such as the report processing workflow, reliability requirements are more stringent. The pipeline shall implement retry logic for transient failures, with exponential backoff (Nygard, 2018) to avoid overwhelming recovering systems. Three attempts (two retries) with exponential backoff delays of two and four seconds (computed as 2^n for each retry) represent the baseline configuration.

Data validation shall prevent silent corruption. The compliance report pipeline shall verify row counts before proceeding with uploads, blocking transmissions that fall below expected thresholds. The default minimum threshold shall be 100 rows, with report-specific validated overrides permitted for legitimate low-volume reports. This preserves a safeguard against incomplete exports without blocking low-volume reporting workflows.

3.3.4 NFR4: Scalability

The system shall support batch operations across multiple hosts. Rather than requiring analysts to execute separate queries for each host, the command shall accept host groups and iterate across members automatically. Results shall aggregate into a single output set.

The current scope covers approximately 50 hosts and 20 Splunk instances. The architecture shall support at least double the current managed inventory (approximately 100 hosts and 40 Splunk instances) without requiring architectural redesign. Growth beyond this range is not anticipated, and the system is not intended for deployment at scales significantly beyond the current order of magnitude.

3.3.5 NFR5: Maintainability

The system shall follow a modular architecture that isolates functionality by integration target. Adding support for a new external system shall not require modifications to existing modules. Each module shall be independently testable (importable in isolation with mocked external dependencies) and independently deployable as a separate Technology Add-on, installable without requiring other application packages to be present.

A registry pattern shall support dynamic discovery of available actions. New actions can be added by creating appropriately structured modules without modifying central dispatch logic. This pattern reduces the risk of introducing regressions when extending functionality.

Code organisation shall follow established patterns within the Splunk application development ecosystem. The Technology Add-on naming convention and directory structure supports deployment through standard Splunk mechanisms.

Adding a new action to an existing module shall require no more than one source file (the action implementation) and one registry entry, without changes to the module executor, command parser, or other existing actions.

3.3.6 NFR6: Usability

The command shall present a consistent interface regardless of the target system. All modules shall follow the invocation syntax:

```
| usoapi module=<module> action=<action> target=<target>
```

with no module-specific deviations from this pattern. This uniformity reduces the learning curve for analysts, who need only learn one command syntax to query any integrated system.

Help documentation shall be accessible from within the command itself. The command shall support three help levels:

- General: `help=true`
- Module-specific: `help=true module=<module>`
- Action-specific: `help=true module=<module> action=<action>`

Each level shall return structured output listing the available items, required parameters, and usage examples.

Target resolution shall support meaningful aliases alongside literal addresses. Analysts shall be able to specify host groups like `collectors` or `indexers` instead of remembering IP addresses or fully qualified hostnames.

An analyst familiar with SPL syntax shall be able to construct a valid USO API query for any supported action within two minutes, given access to the built-in help system.

3.4 Supplementary Security and Privacy Requirements

To formalise the system's operational boundaries, the following supplementary requirements govern data protection and credential lifecycle management, aligned with the controls defined in the Open Web Application Security Project (OWASP) Application Security Verification Standard (OWASP Foundation, 2025):

- **SPR1 (Data Minimisation):** The system shall process PII detections in memory and avoid intentional persistence of raw matches to local application storage. Success is evidenced by the detector's stateless processing model, in-stream annotation and redaction behaviour, and the absence of implementation paths that write raw matches to the application workspace.
- **SPR2 (Credential Rotation):** The system shall support revoking and replacing stored tokens through Splunk's encrypted credential store, without requiring code changes or plaintext credential handling.
- **SPR3 (Access Segregation):** The system shall enforce Role-Based Access Control (RBAC) and capability checks for external integrations. Users or service identities lacking the required capability shall be denied execution or data access.

Table 3.1 summarises all functional, non-functional, and security requirements identified in this chapter.

Table 3.1: *Summary of Functional and Non-Functional Requirements*

ID	Category	Description
FR1	Infrastructure	Query host health, disk usage, and status via Ansible
FR2	Splunk	Monitor Splunk instances, queues, jobs, and logs
FR3	Database	Test database connectivity and list extraction configs
FR4	Vulnerability	Query Red Hat Satellite for hosts and errata
FR5	Email	Monitor email pipelines, mailbox health, and ingestion status
FR6	Identity	Query user, group, host, and access identity from local lookups
FR7	PII Detection	Scan indexed data for personally identifiable information
FR8	Dashboard	Unified 16-tab interface consolidating all data sources
FR9	Diagnostics	Automated infrastructure health diagnostics via alert actions
NFR1	Performance	`| usoapi` command response within 20 seconds
NFR2	Security	Encrypted credential storage, token-based auth
NFR3	Reliability	Graceful errors, retry logic, data validation
NFR4	Scalability	Batch operations across host groups
NFR5	Maintainability	Modular architecture, registry pattern
NFR6	Usability	Consistent syntax, built-in help, target aliases
SPR1	Minimisation	In-memory PII scanning without local persistence
SPR2	Rotation	Event-driven token revocation
SPR3	Segregation	RBAC enforcement for all integrations

3.5 Constraints and Assumptions

Technical Constraints The system operates within Splunk's Python execution environment, which imposes restrictions on available libraries and execution context. Custom search commands execute in a sandboxed environment with limited network access and restricted file system operations. The implementation must work within these boundaries.

Network connectivity is required to the Ansible control node, Red Hat Satellite API endpoint, and target Splunk instances. Firewall rules permitting this connectivity are assumed to be in place or obtainable through standard change management processes.

The Splunk deployment uses a search head cluster configuration. Custom commands execute on a single cluster member rather than distributed across all members. This affects how commands interact with local state and credentials.

Organisational Constraints Credential provisioning differs by system. Access credentials for Red Hat Satellite and Open Authorisation (OAuth) 2.0 client credentials for the Microsoft Graph API require requests to the respective platform administrators. By contrast, the SSH key used for the Ansible control node connection is managed within the Splunk infrastructure team rather than requested from an external platform administrator.

The code will not be released publicly. This relaxes certain requirements around documentation completeness and generalisation that would apply to open-source projects.

Assumptions Authentication to remote Splunk instances uses bearer tokens from a dedicated service account with the necessary capabilities (`list_health`, `list_introspection`, and related monitoring capabilities). A separate token is stored per remote host in Splunk's `storage/passwords` endpoint and synchronised automatically across search head cluster members. Regular users do not require elevated capabilities; they need only the ability to execute the custom search command.

Target hosts are registered in Ansible's inventory and accessible via SSH from the control node. The team's existing Ansible infrastructure supplies this foundation.

Red Hat Satellite contains current inventory and patch status information. The accuracy of vulnerability visibility depends on the timeliness of Satellite's data, which is managed outside this project's scope.

With these requirements, constraints, and assumptions established, Chapter 4 presents the system architecture that translates them into concrete design decisions across USO API, the dashboard, and the credential management subsystems. The verification approach, including functional completeness assessment and non-functional property evaluation through production telemetry, is documented in Chapter 6.

4

System Architecture

4.1 Architecture Overview

The architecture of USO API follows a layered structure that separates concerns across four distinct tiers: presentation, application, integration, and infrastructure (Buschmann et al., 1996).

At the presentation layer, Splunk Web serves the unified dashboard through XML form definitions. Analysts interact with tabbed views that display host health, Splunk instance status, vulnerability data, and automation pipeline states. Each panel issues SPL queries that invoke the custom command, receiving results as standard Splunk table data suitable for rendering in charts, tables, or status indicators.

The application layer consists of a single custom search command, `usoapi`, implemented as a `GeneratingCommand` using the Splunk SDK for Python. This command accepts parameters specifying the target module, action, and hosts, then dispatches execution to the appropriate integration module.

Six integration modules connect the command layer to operational data sources. The Ansible module executes ad-hoc commands against managed hosts via SSH, collecting system metrics such as uptime, disk usage, and service health. For platform observability, the Splunk module queries remote instances through their REST APIs, retrieving server information, queue status, index metadata, and job history. Vulnerability and compliance data flow through the Satellite module, which interfaces with the Red Hat Satellite API to obtain host inventory, errata status, and subscription data.

The remaining three modules address data-oriented concerns. Database connectivity monitoring spans Oracle, Microsoft SQL Server, and Sybase instances through the Extractor module, whilst the Mail module presents a unified view of email data flows, authenticating against Microsoft Graph API to check mailbox health. It combines these checks with Splunk index ingestion statistics and Extractor log analysis to present an end-to-end view of the email processing pipeline. Unlike the preceding five modules, the Identity module operates entirely within the local Splunk environment, querying

existing lookups and KV Store collections to resolve user profiles, group memberships, and host ownership from Active Directory, access management, and human resources data.

The infrastructure layer covers the target systems themselves: a fleet of Linux hosts managed through Ansible, multiple Splunk instances from development through production, and the Red Hat Satellite server that tracks host compliance. Communication protocols vary by target: SSH for Ansible operations, HTTPS for Splunk REST and Satellite APIs, and native database protocols for Extractor tests.

Not all dashboard data flows through the custom command. Where data is already indexed in Splunk or queryable through native platform features such as `tstats`, `mstats`, `rest`, or federated search, native SPL is preferred, as it avoids the overhead of the command framework and executes with lower latency. The custom command is reserved for integrations with external systems that Splunk cannot query natively, or for complex data-fusion tasks, such as those performed by the Identity module, that would be syntactically unwieldy in pure SPL. The Forwarders tab illustrates this principle: it retrieves deployment server metadata through Splunk's federated search feature, which allows a search head cluster to query saved searches on remote Splunk instances. A federated provider connects the cluster to the deployment server using a dedicated service account, and the dashboard pulls forwarder status via the `| from federated:` command. This avoids embedding deployment server queries in the custom command framework whilst still presenting the data within the unified dashboard.

Three separate subsystems extend the platform beyond the synchronous query interface. The diagnostics subsystem, deployed as `uso_TA_diagnostics`, remains tightly aligned with the main observability contribution by monitoring infrastructure health through search-job, file, and KV Store validation before indexing diagnostic events and dispatching notifications. Alongside it, two secondary extensions broaden the platform's operational reach. The compliance report pipeline, deployed as `uso_TA_compliance`, executes scheduled searches that extract monthly compliance data, transform results into partner-specified formats, and upload artefacts to external servers. Deployed as `uso_TA_piifinder`, the PII detection subsystem scans event data for personally identifiable information through a hybrid Rust and Python streaming command, complemented by a scheduled search that periodically indexes exposure metrics.

All three subsystems operate independently of the synchronous query infrastructure, communicating state through shared summary indexes that the main dashboard queries for visualisation. This *summary-index integration model* recurs throughout the architecture: each subsystem writes structured events to a dedicated index without shared code or coupling to the dashboard, and failures in any subsystem do not affect the presentation layer. Figure 4.1 illustrates the overall system architecture and the relationships between these layers.

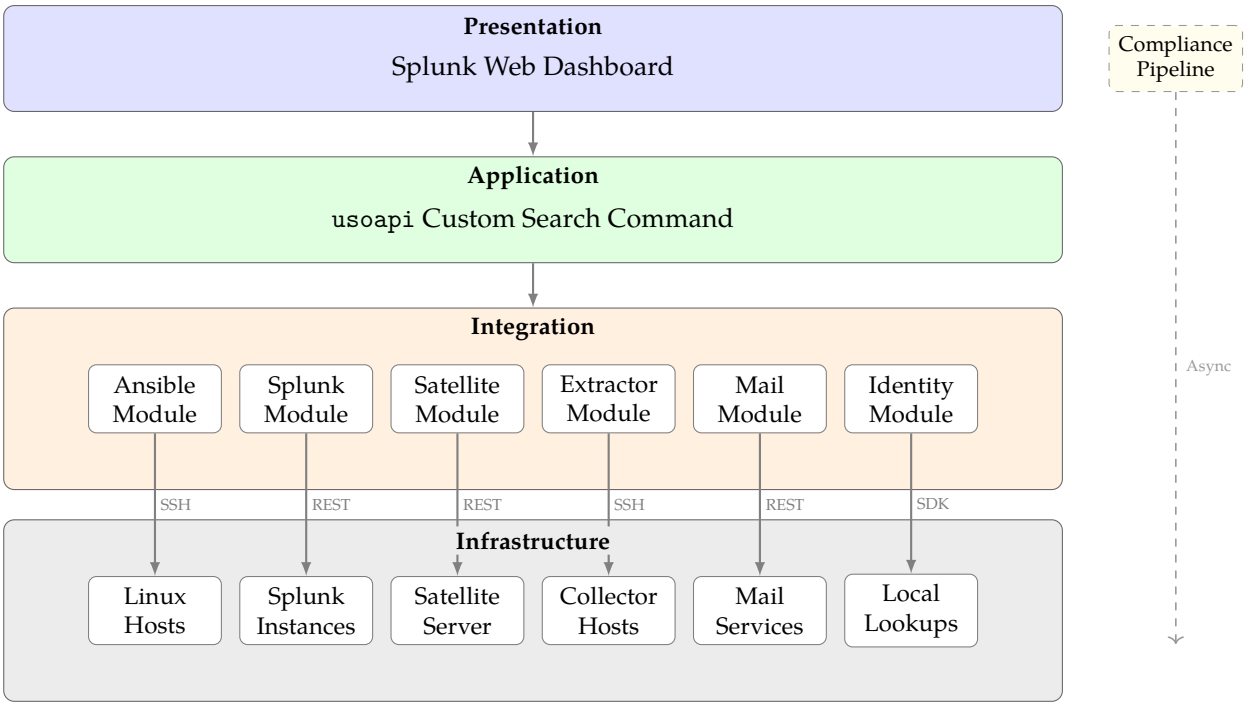


Figure 4.1: *USO API system architecture overview. Six integration modules connect the unified dashboard to heterogeneous infrastructure backends through protocol-specific adapters.*

4.2 Component Relationships

The command architecture implements a registry-based dispatch mechanism, drawing on the Factory Method and Strategy patterns (Gamma et al., 1994), where the main command class delegates execution to specialised module handlers. When an analyst invokes a query such as `| usoapi module=ansible action=health target=indexers`, the command parses the parameters, loads the requested module dynamically, and passes control along with a context object containing all relevant options. This dispatch logic follows the Content-Based Router pattern (Hohpe et al., 2003) introduced in Section 2.7: the `module` parameter determines which handler receives the request, without requiring callers to know the underlying transport or authentication details.

Each module maintains its own action registry mapping action names to handler classes. This registry pattern supports extension without modifying core command code: adding a new action requires creating a handler class that implements the standard interface and registering it within the appropriate module's action map. The dispatch mechanism discovers available actions at runtime, supporting both built-in documentation generation and error messages that enumerate valid options. Figure 4.2 depicts this dispatch process from command invocation through module selection to action execution.

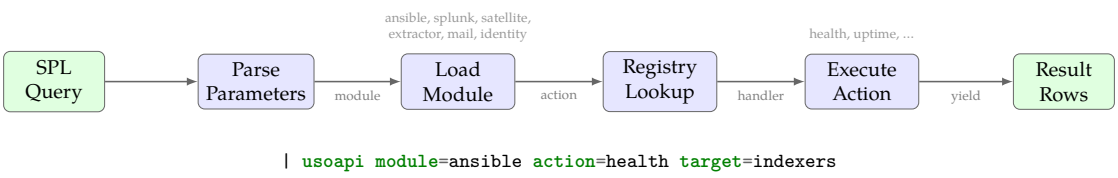


Figure 4.2: *Command Dispatch Flow*

4.2.1 Technology Stack

Python 3.13 is the primary implementation language. The Splunk SDK for Python exposes the `GeneratingCommand` base class and handles protocol-level communication with the search pipeline. Dashboard visualisations use Splunk's Simple XML framework extended with custom JavaScript for interactive features such as row expansion, which reveals additional detail when an analyst clicks a table row. Simple XML was chosen over the newer Dashboard Studio framework for its JavaScript extensibility, as discussed in Section 4.11.1.

4.3 Core Components Design

The core components form the runtime backbone of the USO API command, handling dispatch, module isolation, and credential access.

4.3.1 Command Framework

The `usoapi` command extends Splunk's `GeneratingCommand` base class, positioning it as a data source at the start of search expressions instead of a transformation that processes pipeline input. The command produces result rows by querying external systems, yielding data that downstream SPL commands can filter, aggregate, or visualise.

Each integration module implements a domain-specific exception hierarchy that lets callers differentiate between configuration errors, authentication failures, permission denials, and transient service issues. This classification permits differentiated error handling: dashboard panels can retry on rate-limit responses whilst immediately surfacing permission errors to administrators.

All actions within a module inherit from an abstract base class that defines the standard interface, validates parameters, formats results, and resolves targets. This pattern maintains consistent behaviour across actions whilst reducing code duplication.

4.3.2 Integration Modules

The six integration modules described in Section 4.1 share a common architectural foundation. Each module extends a common action base class that standardises parameter validation, result formatting, and error propagation. Actions raise typed exceptions (`ActionError`, `ModuleError`, `AuthenticationError`, `USOConnectionError`) that the command framework catches and translates into structured error events, isolating callers from transport-level failures. Five modules authenticate against remote systems using credentials retrieved from the credential management system described in Section 4.3.3; the Identity module is the exception, operating entirely within the local Splunk environment through the SDK's `self.service` connection and requiring no external credentials.

4.3.3 Credential Management System

The Splunk module's authentication requirements drove development of a dedicated credential management system. Remote Splunk instances require authentication for API access, and traditional approaches using service account passwords introduce operational overhead: password rotation policies force regular updates across all configured hosts, and password storage in configuration files raises security concerns during code reviews and deployments. The implemented solution uses bearer tokens generated through Splunk's authentication token API, which encode user identity and role capabilities in JWT format. The system accommodates permanent tokens with no expiration, eliminating rotation overhead whilst maintaining the ability to revoke individual tokens when required.

Machine-to-machine authentication uses dedicated service accounts with defined capability requirements. A representative naming pattern such as `svc.secmon-m2m` reflects organisational conventions: a service-account prefix, a team-scope identifier, and a suffix denoting the machine-to-machine purpose. The service account requires particular monitoring capabilities: `list_health` for health endpoint access, `list_introspection` for queue monitoring, `search` for executing SPL queries, and `rest_properties_get` for metadata retrieval. Section 5.8.1 documents the full capability set. Role membership inheritance means that adding a capability to the role automatically extends to all tokens derived from accounts holding that role.

Target resolution handles multiple methods for translating analyst-friendly identifiers into actionable host references, including direct IP addresses, mnemonic labels, wildcard patterns, and predefined group shortcuts. A separate credential type accommodates OAuth 2.0 client credentials (Hardt, 2012) for Microsoft Graph API access, stored in a distinct namespace within the credential store. Chapter 5 details the credential storage mechanisms, token lifecycle management, and administrative tooling that support this architecture.

4.4 Dashboard Architecture

The dashboard architecture translates the module data flows into an interactive analyst interface built on Splunk's Simple XML framework.

4.4.1 Tab Structure and Layout Patterns

The unified dashboard uses Splunk's Simple XML framework with form inputs that drive panel queries through token substitution. Form inputs at the top establish shared filtering context: a host selector populates from credential store queries, a time range picker constrains searches, and a scope filter switches between all hosts and production systems. These inputs bind to tokens that panels reference in their SPL queries, so that filtering remains consistent across all visible data.

The sixteen tabs catalogued in Section 3.2.8 are organised into four thematic groups: Overview (Summary), Platform (Infrastructure, Performance, Forwarders, Logs Status, Sources/Data), Security (Suspicious Activity, Vulnerabilities, Firewall, Data Health, PII Exposure), and Services (Work Orders, Licenses, Automation, Extractor, Email). The resulting tab structure is shown in Figure 4.3.

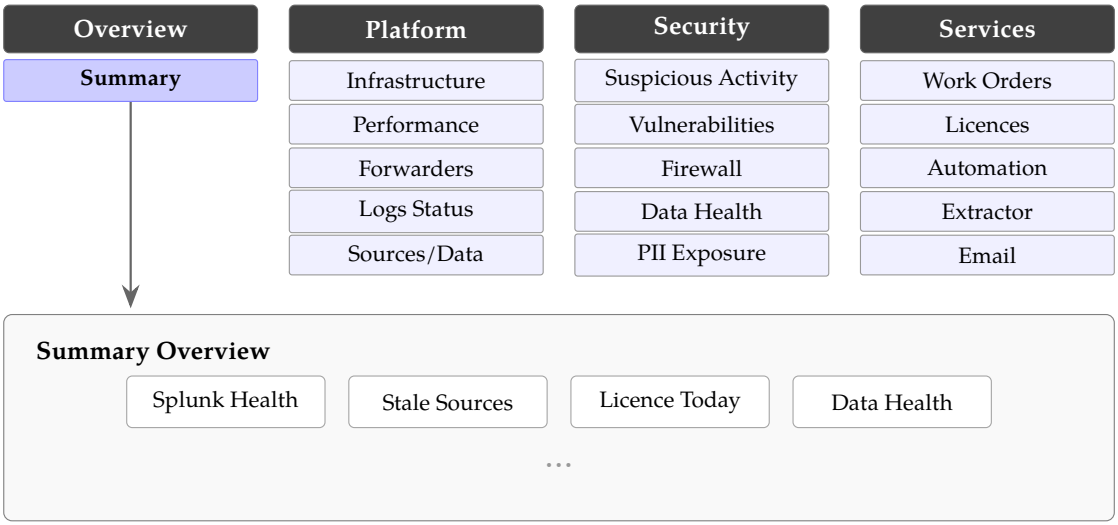


Figure 4.3: *Dashboard tab structure with sixteen tabs organised into four thematic groups. The Summary tab (highlighted) shows the first row of KPI panels; the full layout is visible in Figure 5.2.*

Each tab applies a consistent layout pattern: summary statistics appear as single-value visualisations across the top, followed by detailed tables or charts in the main content area. The architectural patterns governing shared tab structure are presented below; individual tab content is catalogued in Section 3.2.8.

4.4.2 Visualisation Components

Three visualisation types serve different data characteristics. Single-value panels highlight core metrics such as total hosts, active alerts, or licence utilisation percentages. Tables present detailed records with inline status indicators using icon formatting rules. Charts plot trends over time, particularly useful for licence usage patterns and queue depth monitoring.

Row expansion extends table functionality by surfacing additional detail when an analyst clicks a table row, executing a secondary search scoped to that row's context and injecting results into an expandable region below the clicked row. Section 5.4.2 describes the implementation.

4.4.3 Data Refresh Strategy

The refresh strategy balances data freshness against search head load by combining token-gated execution with mixed refresh cadences. Many panels do not execute immediately at dashboard load; instead, searches are released only when the relevant tab or prerequisite token becomes active, and refresh intervals vary by workload rather

than following a single five-minute cycle. By reading from lookup or summary-index materialisations, these panels reduce interactive search cost compared to recomputing the same results from raw events.

Panels requiring relatively current values, such as Splunk health indicators, current event-rate counters, and running-job views, issue live SPL queries on their refresh cycle. Where the data model permits it, these queries use commands such as `tstats`, Splunk's accelerated statistics command, for efficient access to recent data. Heavier derived views rely on the materialised snapshot pattern described below, which moves more expensive computations away from the user-facing refresh cycle.

A complementary pattern, referred to as *materialised snapshots*[1], uses scheduled saved searches to compute derived data and write the results to storage targets (primarily CSV lookup tables and, where appropriate, summary indexes) at fixed intervals. Dashboard panels then query these datasets rather than executing live searches, trading bounded staleness[2] for predictable render times. This two-tier arrangement also allows the dashboard to present broader security context without reaching external APIs or cache-refresh limits; for example, the index-status monitor consumes materialised CSV lookups (Appendix C.5), whilst selected source-IP reputation views consult an AbuseIPDB[3] (AbuseIPDB, 2025) cache (Appendix C.7). Two further implementations of this pattern appear in Section 5.4.4 and Section 5.7.4, which use lookup and summary-index storage respectively.

4.4.4 Tab Priority Controller

A sixteen-tab dashboard with dozens of panels presents a search scheduling challenge: executing all searches simultaneously on page load overwhelms the search cluster and delays rendering of the tab the analyst is viewing. The tab priority controller addresses this through token-based dependency gating. A JavaScript component monitors tab selection events and manipulates dashboard tokens that panel searches depend upon, preventing execution until the analyst selects a given tab. Once visited, tokens persist so that returning to a tab does not re-execute its searches. This mechanism prioritises the active tab whilst deferring background tabs for progressive loading.

4.5 Compliance Report Pipeline

The compliance report pipeline is a complementary operational extension that automates monthly delivery of compliance reports to external partners, replacing a manual process that previously required operator intervention at each stage.

[1] The term *materialised snapshot* as used in this work denotes a scheduled saved search that pre-computes a derived dataset and writes the result to a CSV lookup or summary index for later dashboard consumption.

[2] Bounded staleness describes a state where data lags behind reality by a fixed, known interval. In this architecture, that interval corresponds to the duration between scheduled background materialisation runs.

[3] AbuseIPDB is a collaborative IP address reputation database used to identify and report malicious activity such as hacking attempts, spam, and DDoS attacks.

4.5.1 Architecture Overview

The pipeline executes as a standalone Splunk application, `uso_TA_compliance`, with its own scheduled searches and alert actions. Operating independently of the USO API, it follows the summary-index integration model defined in Section 4.1. The alert action pattern decouples pipeline logic from search scheduling: the saved search defines what data to extract, whilst the processor script handles how the data is transformed and delivered.

4.5.2 Pipeline Components

The processing pipeline begins with a scheduled search that executes against source indexes on the first day of each month. Upon completion, Splunk's alerting subsystem invokes a custom alert action that manages four sequential runtime stages: validated CSV export, transformation into the required output format, upload to the external partner's server using rsync[4] over SSH, and post-upload verification. Logging to the summary index and operator notifications span all four stages rather than forming a separate processing step. Figure 4.4 illustrates this flow from scheduled-search trigger through verification.

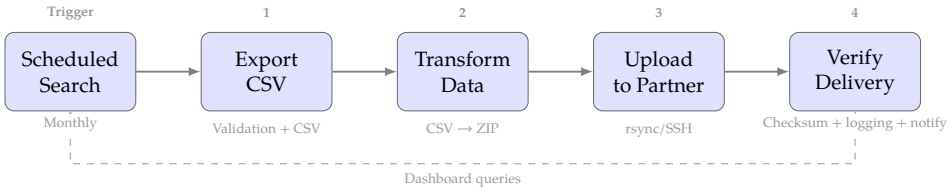


Figure 4.4: *Compliance Report Processing Pipeline*

4.5.3 Integration Points

Each pipeline stage emits a log entry to the `uso_compliance` summary index, recording pipeline telemetry. The dashboard's Automation tab queries this index to display recent processing runs, highlight failures, and provide drill-down access to error details. Notification hooks extend this logging model with active alerting: the processor dispatches webhook messages to Mattermost, a self-hosted messaging platform (Mattermost, Inc., 2026), and sends email notifications over Simple Mail Transfer Protocol (SMTP) to designated recipients at critical stages, informing operators of validation outcomes, upload success, or retry exhaustion. This three-channel approach (summary-index logging, Mattermost webhooks, and email) suits a pipeline that runs once monthly, where failures must be caught promptly rather than discovered during the next scheduled execution.

[4] `rsync` is a Unix utility for efficient incremental file transfer over SSH.

4.6 Diagnostics Subsystem

The diagnostics subsystem is packaged as a further Technology Add-on, `uso_TA_diagnostics`, and follows the same deployment pattern as the compliance report pipeline. Its purpose differs, however: where the pipeline delivers data to external partners, the diagnostics subsystem turns Splunk's alerting infrastructure inward, monitoring the health of search jobs, filesystem artefacts, and KV Store collections that the broader platform depends upon.

4.6.1 Module Architecture

The subsystem employs a dispatcher with registry-driven module discovery. When a saved search triggers the `uso_diagnostics` alert action, the dispatcher iterates over registered modules, executes those enabled in the alert configuration, and aggregates results. Three modules handle diagnostic checks. The job diagnostics module captures search metadata and flags anomalies such as zero-result searches or unusually long durations. The file diagnostics module validates filesystem artefacts against configurable size and row count thresholds. The KV Store diagnostics module verifies collection existence, accessibility, and row counts to detect both data loss and unexpected growth.

4.6.2 Historical Comparison Engine

Both the file and KV Store modules support an optional historical comparison mode. When enabled, the engine queries previous diagnostic results to compute statistical baselines, then calculates the percentage deviation of the current measurement from the historical average. This approach distinguishes genuine problems from normal variation without requiring administrators to hard-code absolute thresholds for every monitored resource.

4.6.3 Notification Architecture

Notifications follow a pluggable backend model. A `BaseNotifier` abstract class defines the notification interface, with concrete implementations for three channels: Mattermost webhooks, Splunk bulletin board messages, and SMTP email delivery. A fan-out handler dispatches notifications to all registered backends, applying level-based gating: error notifications are sent by default, whilst success notifications require explicit opt-in, preventing alert fatigue during normal operation.

4.6.4 Integration Model

The diagnostics subsystem follows the summary-index integration model (Section 4.1), writing events to the `uso_diagnostics` index. Because it shares no code with the USO API, the subsystem can monitor artefacts across any Splunk application.

4.7 PII Detection Subsystem

The Personally Identifiable Information (PII) detection subsystem is a secondary privacy-monitoring tool addressing a complementary security concern: identifying personally identifiable information exposed within Splunk event data. Deployed as `uso_TA_piifinder`, this subsystem introduces a streaming command, `piifinder`, that scans event fields for patterns matching fourteen categories of personal data, with particular attention to Portuguese regulatory identifiers. Unlike the compliance and diagnostics subsystems, which operate exclusively through scheduled searches and alert actions, `piifinder` also functions as an interactive command that analysts invoke directly within search expressions.

4.7.1 Architecture Overview

The subsystem adopts a hybrid architecture combining Python and Rust. The Python layer implements a `StreamingCommand` via the Splunk SDK, handling integration with the search pipeline: parameter parsing, event routing, and result formatting. The compute-intensive pattern matching delegates to a compiled Rust binary that communicates with the Python wrapper via newline-delimited JSON Inter-Process Communication (IPC). Each event passes to the Rust process as a JSON object; the binary applies all configured detectors and returns match summaries.

This separation reflects a deliberate trade-off: Python remains responsible for Splunk command registration and pipeline communication, whilst a native compiled core handles the performance-critical scanning workload (Lavrijsen et al., 2016). The Splunk SDK mandates Python for command registration and pipeline communication, yet Python's regex engine lacks the throughput needed for scanning large event volumes. The choice of Rust over alternatives such as C or C++ is motivated by its combination of systems-level performance with compile-time memory safety guarantees that do not rely on garbage collection (Jung et al., 2021), which suits production data-processing components where predictable throughput and memory safety both matter. By isolating detection logic in a compiled binary, the architecture achieves a fourfold improvement in processing speed at scale, as evaluated in Chapter 6, whilst retaining full compatibility with Splunk's command infrastructure.

Graceful degradation preserves availability when the Rust binary is absent or fails to start. The Python wrapper contains an equivalent detection implementation and falls back to it transparently, accepting reduced throughput instead of failing the search. The Python fallback ensures that deployment to environments where the compiled binary cannot run (new operating system versions, restrictive filesystem policies) does not prevent PII scanning entirely.

4.7.2 Detection Pipeline

The detection pipeline processes each event through fourteen specialised detectors organised into two groups. Six target Portuguese-specific identifiers: fiscal numbers (NIF) checked for structural consistency, international bank account numbers (IBAN) verified by the modulo-97 check digit algorithm (Banco de Portugal, 2024), social security numbers (NISS), citizen card numbers, vehicle registration plates, and postal addresses matching Portuguese street-type patterns. Eight handle universal patterns: email addresses, credit card numbers validated by the Luhn algorithm[5], credentials such as passwords and API keys, structured personal names, IPv4 addresses, dates of birth, JWT and bearer tokens, and telephone numbers in Portuguese and international formats.

Six of these detectors are context-gated: they fire only when relevant keywords appear within a configurable character window around the candidate match. This mechanism suppresses false positives for patterns that frequently occur in non-PII contexts, such as nine-digit numeric sequences resembling fiscal numbers but originating from log identifiers or trace IDs.

Confidence scoring assigns each detection a three-tier rating (low, medium, or high). Context keywords within the surrounding text boost confidence by one tier, whilst negative keywords suppress it by one tier or discard the match entirely. The same pattern may therefore produce different confidence levels depending on textual context, giving analysts a filtering parameter to control detection sensitivity.

Two algorithmic choices underpin the Rust backend's throughput. A `RegexSet` prefilter compiles all fourteen detector patterns into a single finite automaton; for each input text, only detectors whose patterns matched in the set proceed to individual evaluation. Keyword matching for context and negative terms uses Aho-Corasick automata (Aho et al., 1975), reducing matching cost from linear in the number of keywords to linear in the length of the text.

4.7.3 Integration Points

The subsystem integrates with the broader platform through three channels. As a streaming command, `piifinder` can be appended to any search expression, allowing analysts to scan arbitrary data on demand. Three operating modes serve different use cases: counts mode adds aggregate PII statistics to each event for dashboard consumption, detail mode exposes individual match locations for investigation, and redact mode replaces detected PII inline for data sanitisation.

A saved search executes every thirty minutes, scanning application indexes and writing exposure metrics to the `uso_diagnostics` summary index. This scheduled component tracks PII exposure trends historically without requiring analyst intervention.

[5] The Luhn algorithm is a checksum formula used to validate identification numbers such as credit card numbers (Luhn, 1960).

The dashboard's PII Exposure tab queries this summary index to present detection statistics, confidence distributions, and per-type breakdowns, as described in Section 4.4.1, following the same summary-index integration model as the compliance and diagnostics subsystems (Section 4.1).

4.8 Security Architecture

4.8.1 Authentication and Authorisation

Authentication flows depend on the integration module invoked. Local Splunk operations authenticate through the `self.service` property, which the SDK populates automatically from the search context session. Because the command inherits the invoking user's session and permissions, no additional credential handling is required for local queries.

Remote Splunk queries authenticate using bearer tokens retrieved from the credential store. The module resolves the target host, looks up its associated token, and includes the token in the `Authorization` header. The remote instance validates the token and applies the associated role capabilities to the request.

Ansible operations use SSH key-based authentication: the Splunk service account on the search head holds an SSH private key whose matching public key is installed in the Ansible control node's authorised keys file, allowing ad-hoc commands to execute without password prompts. The application assumes this key pair has been provisioned and is maintained through the team's operational SSH procedures.

4.8.2 Secure Communication

Remote Splunk REST calls, Satellite requests, and Microsoft Graph requests target HTTPS endpoints. In the current implementation, however, peer certificate verification is disabled for the Splunk and Satellite code paths, so those integrations rely on encrypted transport but do not yet enforce full end-to-end certificate validation on every remote connection. This choice was accepted for the current environment because those endpoints reside on organisation-managed private networks and are accessed through constrained service identities for monitoring operations only. Encrypted transport, least-privilege roles, and audit logging reduce the residual risk; full certificate-chain validation, whilst outside this project's scope, remains a hardening requirement rather than a completed control.

Report processing file transfers use rsync over SSH, encrypting both the control channel and data stream. Unlike the internal Ansible connections, the compliance pipeline enforces a strict host key verification policy (`StrictHostKeyChecking=yes`), requiring target host keys to be pre-provisioned in the search head's `known_hosts` file to prevent man-in-the-middle attacks during external transfers. The stricter setting is deliberate: the upload destination crosses a stronger trust boundary than the internal Ansible estate, so the extra operational burden of key pre-distribution is justified there.

4.8.3 Credential Storage Security

Splunk's `storage/passwords` API encrypts secrets, stores them in `passwords.conf`, and replicates updates automatically across search head cluster members (Splunk Inc., 2026g). In this deployment, the application's entries are created under the `search` app namespace and protected through role capabilities and knowledge-object permissions (Splunk Inc., 2026d). Three credential categories organise the stored secrets: one for per-host Splunk bearer tokens, one for Red Hat Satellite username-and-password credentials, and one for the Microsoft Graph client-credential configuration used to obtain short-lived access tokens at runtime.

4.8.4 Audit and Logging

Command executions generate log entries that flow to Splunk's internal logs through a designated log file processed by the `_internal` index, with each entry recording the timestamp, invoking user, target hosts, action requested, and execution outcome. Security teams can query these logs to audit USO API usage patterns.

Token Manager Command-Line Interface (CLI) operations log separately, capturing administrative actions such as credential creation, deletion, and test invocations. These logs support forensic investigation if credential misuse is suspected, establishing baselines for normal administrative activity.

4.8.5 Principle of Least Privilege

Service accounts follow a least-privilege model, possessing only the capabilities required for monitoring (Section 4.3.3). Tokens inherit these constrained permissions, limiting blast radius if a token is compromised.

Dashboard access respects Splunk's role-based access model. Because roles determine which indexes, dashboards, apps, and capabilities a user can access, the dashboard presents only data that the current user is already permitted to search or view (Splunk Inc., 2026a).

4.8.6 Security Considerations

This section identifies the principal trust boundaries and threat vectors relevant to the system architecture, complementing the mechanism-level descriptions in the preceding subsections.

Trust Boundaries The architecture spans three trust zones with distinct security postures. The *trusted zone* comprises the Splunk search head cluster, where authenticated users execute queries and the application's remote-API credentials reside in encrypted storage. The *semi-trusted zone* covers the organisation-managed infrastructure: Ansible targets, remote Splunk instances, and the Red Hat Satellite server, all accessible over SSH or REST API channels that the organisation controls but that reside outside the

SIEM perimeter. The *untrusted zone* covers the Microsoft Graph API, where trust is established only through provider-issued credentials and the organisation has no control over the service endpoint.

Key Threats and Mitigations A systematic threat assessment using the STRIDE methodology (Shostack, 2014), conducted in accordance with the risk management principles of ISO/IEC 27005:2022 (ISO/IEC, 2022), formalises the evaluation of the system's trust boundaries (Van Landuyt et al., 2021) and its automated infrastructure (Tran et al., 2024). The architecture shown in Figure 4.5 illustrates the boundaries separating the core SIEM components from external endpoints.

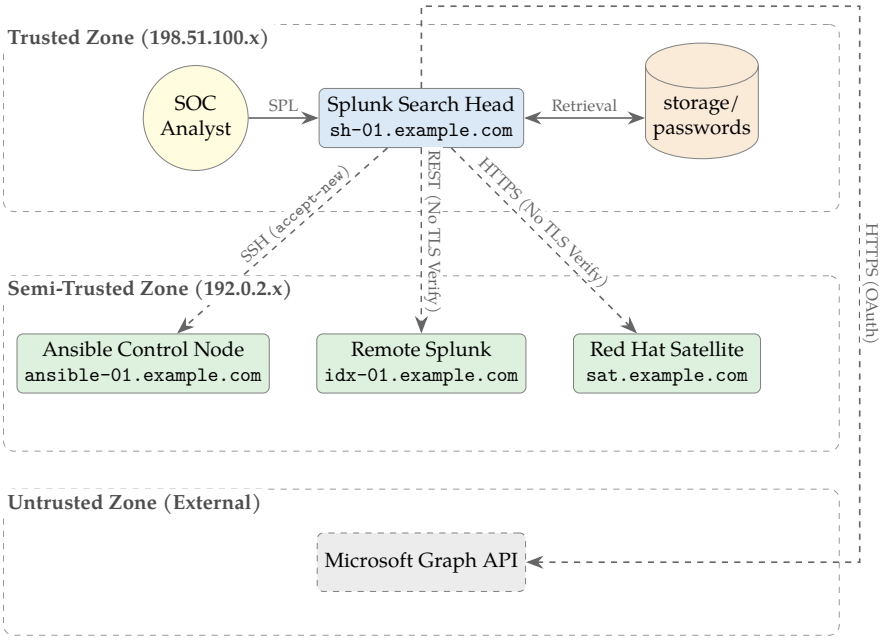


Figure 4.5: *Trust boundaries isolating the Splunk core platform from external components. The diagram highlights the SSH TOFU enforcement and external API connections.*

Table 4.1 details the application of the STRIDE model across these boundaries, evaluating the specific risks and their corresponding technical mitigations.

These mitigations include deliberately bounded trade-offs. Internal Ansible traffic uses TOFU because the monitored host set is relatively small, remains on restricted private subnets, and rejects subsequent host-key changes, whereas pre-distributing keys to the full estate would have added recurring operational overhead. Explicit query rate limiting was likewise deferred because the user population is small and authenticated, whilst token caching and search scheduling already reduce avoidable external load.

Command Abuse Analysis and Defences

The introduction of a custom search command that exposes commands executed as Ansible ad-hoc operations through a fixed registry introduces a risk of command injection and unauthorised enumeration. To mitigate this, the architecture enforces strict execution allowlists at the API layer. Analysts cannot specify arbitrary Ansible modules or targets; they are restricted to the registered action set—comprising actions such as `test`, `health`, `services`, and `disk`—executed as Ansible ad-hoc commands against

Table 4.1: *STRIDE Threat Model and Risk Assessment*

STRIDE Category	Threat & Mitigation	Asset	Prob.	Impact
Spoofing	**Threat:** SSH man-in-the-middle interception between search heads and target hosts. **Mitigation:** Enforced TOFU (`accept-new`) for internal Ansible connections; strict verification (`yes`) for external compliance transfers.	SSH Channels	Low	High
Tampering	**Threat:** Parameter injection through user-supplied SPL arguments. **Mitigation:** `splunklib` SDK validates and constrains query inputs before evaluation.	Search Head	Low	High
Repudiation	**Threat:** Actions executed via API cannot be traced to origin. **Mitigation:** All custom search executions are logged persistently to the `_internal` index.	Audit Logs	Low	Medium
Information Disclosure	**Threat:** Credential exposure during rest or transit. **Mitigation:** Credentials stored in Splunk's encrypted `storage/passwords`.	Credentials	Med	High
Denial of Service	**Threat:** API rate limit exhaustion against Microsoft Graph. **Mitigation:** In-memory token caching avoids repeated authentication requests; query access is limited to a small authenticated analyst group.	Ext. API	Low	Medium
Elevation of Privilege	**Threat:** Unauthorised access to system endpoints. **Mitigation:** Service account uses least-privilege capabilities.	Endpoints	Low	High

registered inventory groups. The system also masks sensitive operational errors from the end-user. If an underlying infrastructure component returns a detailed error trace containing internal network paths or stack traces, the executor intercepts and sanitises this output, returning only generic failure codes to the Splunk interface to prevent information disclosure.

4.9 Deployment Architecture

Development occurs against a standalone Splunk instance isolated from production infrastructure. This environment mirrors production configuration, with the same Splunk version, comparable index structures, and equivalent access patterns, so that changes can be tested locally before submitting to version control.

Production deployment targets the search head cluster through a dedicated deployer host, which distributes application bundles to all cluster members through the standard SHC bundle push mechanism.

The wider core Splunk estate monitored by the platform comprises multiple collectors, several indexers, and the search head cluster, with separate deployment-server and cluster-manager roles supporting configuration distribution.

Beyond the Splunk platform itself, the system depends on the Ansible control node's inventory reflecting current host configurations, network paths to Red Hat Satellite and remote Splunk instances, and SSH key distribution for report upload servers managed separately from the main application deployment.

4.10 Alignment with International Security Standards

The architecture and its operational workflows have been designed to align with industry best practices, specifically the NIST Cybersecurity Framework (CSF) 2.0 (Pascoe et al., 2024) and its underlying control families defined in NIST Special Publication 800-53 Revision 5 (Joint Task Force Interagency Working Group, 2020). Table 4.2 maps the core system components to these formal security and privacy controls.

4.11 Alternative Approaches Considered

4.11.1 Design Alternatives

Five architectural alternatives were evaluated, each rejected for introducing complexity disproportionate to the team's operational constraints.

SOAR Platform vs Custom Integration The project's scope aligns superficially with Security Orchestration, Automation, and Response (SOAR) platforms surveyed in Chapter 2. Deploying Splunk SOAR (or an equivalent) would provide a mature playbook framework and pre-built connectors for some target systems. However, SOAR platforms solve a different problem: they orchestrate incident response workflows (containment, remediation, case management) rather than providing ad-hoc query-time infrastructure visibility within SPL searches. The security operations team's core need was to embed external data retrieval into Splunk's native search language so that host investigation, vulnerability triage, and pipeline health checks execute as standard SPL queries, composable with other commands, embeddable in saved searches, and

Table 4.2: *High-Level Alignment with NIST SP 800-53 Rev. 5 Control Families*

Control Family	Implementation Strategy	System Component
AC (Access Control)	Capability-based authorization enforces least-privilege access (`sec-mon-access`).	Splunk RBAC
AU (Audit & Accountability)	Centralised, immutable logging of all custom command executions to the `_internal` index.	Audit Logging
IA (Identification & Authentication)	Token-based authentication and encrypted credential management via `storage/passwords`.	Credential Store
SC (System & Communications)	Internal transport encryption over SSH and HTTPS protocols.	API Integrations
SI (System & Information Integrity)	Automated vulnerability enrichment and continuous monitoring of infrastructure health.	Threat Intelligence

displayable in dashboard panels without middleware. A SOAR deployment would also introduce additional infrastructure, separate RBAC, and licensing costs disproportionate to the team's use case. The custom search command addresses the query-time integration gap directly, without the operational overhead of a separate orchestration layer.

External API Service vs Embedded Command One option was to implement the integration layer as an external REST API service that Splunk would call via HTTP. This approach would decouple the integration logic from Splunk, potentially easing maintenance and making it available to other consumers. It would, however, introduce additional infrastructure requirements (a dedicated server, deployment pipeline, monitoring) and add network latency to every query. The embedded custom command approach keeps all logic within the Splunk application package, which simplifies deployment and reduces operational complexity.

Direct SSH vs Ansible Orchestration Infrastructure queries could connect directly to target hosts via SSH rather than routing through Ansible. Direct connections would reduce latency by eliminating the intermediate hop but would require managing SSH keys and known hosts entries on every search head cluster member. Routing through Ansible centralises SSH management on a single control node and reuses existing inventory and authentication infrastructure. The latency overhead proved acceptable for the monitoring use case.

Polling vs Event-Driven Dashboard Updates The dashboard uses polling-based refresh as opposed to event-driven updates. A WebSocket-based approach could offer real-time updates when monitored data changes. Yet the underlying data sources (infrastructure status, vulnerability scans) change infrequently enough that periodic polling at multi-minute intervals yields sufficient freshness. Implementing real-time updates would add complexity without proportionate benefit.

Simple XML vs Dashboard Studio A related decision was the use of Simple XML over Splunk's newer Dashboard Studio framework for the unified dashboard. The operations team had existing familiarity with Simple XML, and the JavaScript extensibility it offers was essential for implementing custom interactive panels such as the row expansion architecture described in Appendix A.2.1. Dashboard Studio does not support this level of JavaScript customisation, which would have prevented several core dashboard features.

4.11.2 Decision Rationale

The consistent thread through these decisions was prioritising operational simplicity. Each alternative that was rejected introduced additional infrastructure, dependencies, or complexity that the use case did not justify. The selected approaches minimise operational overhead whilst meeting functional requirements, aligning with the constraint that the solution must be maintainable by a small team without dedicated development resources.

The architectural decisions and component designs presented in this chapter form the blueprint that Chapter 5 translates into working code.

5 Implementation

5.1 Development Environment

5.1.1 Version Control Strategy

Development follows a feature branch workflow hosted on GitLab. The main development branch receives changes through merge requests, with each feature occupying its own branch until ready for integration. This separation permits parallel development of dashboard enhancements and new module actions without blocking ongoing work.

The repository structure separates application code from environment-specific configuration. The `default/` directory contains production-ready settings, whilst `local/` directories hold developer-specific overrides excluded from version control through `.gitignore` entries. This arrangement prevents accidental deployment of development configurations and keeps production settings under explicit version control.

5.1.2 Development Tools

Development occurred with the Visual Studio Code editor via SSH on the development server, with direct access to the Splunk instance and immediate deployment of changes during development cycles. This arrangement permitted rapid iteration on both command implementation and dashboard layouts.

The Python toolchain uses `uv` (Astral Software Inc., 2026) for deterministic dependency resolution and project management, which guarantees identical dependency trees across development and production. Code quality is enforced through `ruff` (Astral Software Inc., 2026) for linting and formatting, whilst `ty` (Astral Software Inc., 2026) performs type checking across all four Python codebases; all three tools share a single configuration file, `pyproject.toml`, and are developed by the same vendor. This automated verification supports maintainable, production-ready code practices (Martin, 2008).

An AI coding assistant supported unit test development, most notably by scaffolding test files. All generated code was reviewed, corrected, and adapted to match project conventions before being included in the test suite.

The main command file `bin/usoapi.py` contains the core command class and base module definitions. Integration modules reside in `bin/usoapi_modules/`, each with its own submodule directory containing action implementations, registry definitions, and shared utilities.

5.2 Core API Implementation

The architectural components from Chapter 4 materialise as Python modules sharing a common base class, registry pattern, and credential access layer. Appendix A contains supplementary implementation details, whilst Appendix C documents representative SPL invocation patterns.

5.2.1 Base Command Class

The `USOAPICommand` class extends Splunk's `GeneratingCommand` (Splunk Inc., 2024c; Splunk Inc., 2026e), registered in `commands.conf` and decorated with configuration specifying reporting type and local execution. Versioning and visibility metadata for the broader package reside in `app.conf`. The class defines options for module selection, action specification, and target identification, along with module-specific parameters such as thresholds, filters, and output preferences. A legacy `object` parameter remains available for backwards compatibility, but `module` is the canonical parameter name.

The command communicates with the search pipeline through the SDK's protocol version 2, reading search parameters from standard input and writing results as CSV.

The `generate()` method is the execution entry point. It first checks whether help should be displayed, either due to missing parameters or explicit help requests. For actual queries, it builds a context dictionary containing all relevant options, loads the requested module dynamically, validates the action against the module's registry, and iterates the module's results to yield enhanced result dictionaries. Appendix C.1 shows a minimal dashboard search invoking this command.

5.2.2 Module Registry Pattern

Module discovery occurs during command initialisation. The `_discover_modules()` method scans the `usoapi_modules/` directory for Python files, building a registry that maps module names to their file paths and class names, following the design patterns described in Section 4.2. This approach permits extension without modifying core command code, as described in Section 4.2: adding a new module requires creating the appropriately named file with the expected class structure. The `USOAPICommand` class caches module instances in a `_module_cache` dictionary after first instantiation, avoiding repeated module imports during the same command execution.

Each module maintains its own action registry (Gamma et al., 1994) through a standardised interface. The `registry.py` file in each submodule directory defines an `ActionFactory` that maps action names to handler classes, and an `ActionRegistry` that allows introspection for documentation generation and error messages. The `ActionRegistry` caches resolved action classes in an internal `_cache` dictionary after the first dynamic import via `importlib.import_module()`. Subsequent lookups return the cached class directly, bypassing repeated module resolution. Figure 5.1 shows this registry structure across the codebase.

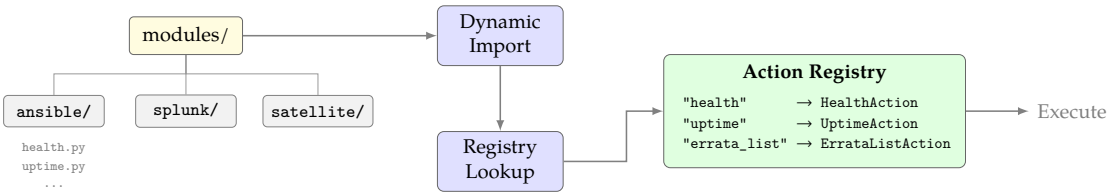


Figure 5.1: *Module Registry Pattern*

Listing 1: *Module discovery scanning the `usoapi_modules/` directory*

```
def _discover_modules(self) -> None:
    module_dir = Path(MODULE_PATH)
    if not module_dir.exists():
        return

    for module_file in module_dir.glob("*.py"):
        if module_file.name.startswith("__"):
            continue

        object_type = module_file.stem
        class_name = f"{object_type.title()}Module"

        self._available_modules[object_type] = {
            "module_name": object_type,
            "class_name": class_name,
            "file_path": str(module_file),
        }
```

5.2.3 Action Base Class

Actions inherit from a common base class with consistent interfaces for execution, result formatting, and credential access. The base class constructor receives a reference to the parent module, which gives actions access to logging facilities, the Splunk service object, and module-level utilities.

The `execute()` method receives the command context and returns an iterable of result dictionaries. Actions may yield results incrementally for streaming output or collect all results before returning. Error conditions produce error result dictionaries instead of exceptions. No stack traces reach the search bar. Instead, the command presents diagnostic information within the normal search results interface.

Listing 2: *Abstract base class defining the action interface*

```python
class CommonActionBase(ABC):

    def __init__(self, module_instance: Any) -> None:
        self.module = module_instance

    @property
    @abstractmethod
    def action_name(self) -> str:
        """Return the action name (e.g., 'test', 'hostname')"""

    @property
    def required_params(self) -> List[str]:
        return []

    @property
    def optional_params(self) -> List[str]:
        return ["debug"]
```

5.2.4 Self-Documenting Help System

The command framework includes a built-in help system that generates documentation dynamically from action metadata. The help system employs a two-tier architecture that keeps documentation synchronised with code: a centralised `USOAPIDocumentation` class handles overview and module-level help, whilst each action's `get_help()` method supplies action-specific documentation.

When analysts invoke the command without parameters or with a `help=true` option, the `USOAPIDocumentation` class iterates through all registered modules and their actions. For each action, it retrieves the action name, required parameters, optional parameters with their defaults, and a description of the action's purpose. This metadata assembles into formatted output that displays directly in search results, so analysts discover capabilities without leaving the Splunk interface.

The help output structure follows a consistent format across all modules. Each action listing includes a usage example showing the minimum viable invocation, a description paragraph explaining what the action does, and tables enumerating parameters with their types and purposes.

When analysts specify an invalid action name, the error message includes the list of valid alternatives, guiding them toward correct usage.

5.2.5 Ansible Executor

The `AnsibleExecutor` class handles execution of Ansible ad-hoc commands through SSH. The executor establishes an SSH connection to the Ansible control node and runs commands remotely, keeping all Ansible configuration and inventory on a dedicated host. The search head triggers operations through this relay rather than invoking Ansible locally.

Command construction follows a specific pattern. The executor builds an SSH command specifying the private key path, strict host key checking set to `accept-new` (Trust On First Use), and password authentication disabled. The remote command changes to the Ansible configuration directory and executes the requested Ansible command with JSON output formatting. The executor sets `ANSIBLE_FORKS=30` to control fan-out concurrency, alongside host-level (10-second) and subprocess-level (120-second) timeout constants that bound execution duration.

Return code handling accommodates Ansible's semantics. Exit code 0 indicates success across all hosts. Exit code 2 signals that some hosts experienced failures whilst others succeeded. Exit code 4 indicates unreachable hosts alongside successful results. The executor parses output for all these cases, extracting per-host results from the JSON structure. Special handling applies to the `needs_restarting` and `health` actions, which treat exit code 1 as a valid non-error result. For `needs_restarting`, this code indicates that a reboot is required.

Host targeting defaults to native multi-target execution. When an action spans several host groups, the executor passes a comma-separated pattern (e.g., `collectors,search-heads,indexers`) directly to Ansible, which resolves and contacts all matching hosts within a single SSH invocation to the control node. This approach replaced an earlier design that split groups into sequential per-target calls, each requiring a separate SSH round-trip. Observed timings from the migration showed the Host Status query dropping from roughly 30 seconds to around 15 seconds, a reduction of 50%. Other multi-host actions showed similar improvements: `needs_restarting` fell from approximately 15 seconds to around 5 seconds, with reductions ranging from 40–70% depending on the number of target groups. The sequential mode remains available through a `native_multi_target=False` parameter for actions that require per-host result isolation.

5.2.6 Splunk SDK Wrapper

The Splunk module wraps the Splunk SDK for Python and gives consistent access to remote Splunk instance APIs. Authentication uses bearer tokens retrieved from the credential store based on target host resolution. The executor constructs a service connection using the SDK's `connect()` function with the `splunkToken` parameter for token-based authentication. The executor caches hostname lookups, resolved from the `service.info.serverName` attribute, in an in-memory dictionary to avoid repeated REST calls when multiple actions query the same host within a single command execution.

Actions query designated REST endpoints depending on their purpose. The health action queries `/services/server/health/splunkd` to retrieve the health tree structure. The queue status action queries `/services/server/introspection/queues` for internal queue metrics. Each action class encapsulates the endpoint knowledge and response parsing appropriate to its data source.

5.2.7 Target Resolution Implementation

Target resolution transforms user-provided identifiers into addressable hosts with associated credentials. The implementation handles several resolution methods that the system attempts in sequence until one succeeds.

Direct IP addressing bypasses resolution entirely: inputs matching IP address patterns proceed directly to credential lookup. Label-based resolution searches the credential store for entries matching the provided label string. Wildcard patterns expand against all stored labels, returning multiple hosts for patterns such as `host*col*`. Group shortcuts resolve to predefined host lists, with names such as `collectors` or `all` expanding to their constituent members.

5.3 Individual Action Implementations

Each integration module contains domain-specific actions that implement the query patterns described in Section 4.3.

5.3.1 Infrastructure Health Monitoring

The Ansible module implements actions for system-level diagnostics. The health action collects memory usage, load averages, and process counts from remote hosts, aggregating several shell commands into a single health assessment. Each action constructs an Ansible command that pipes system statistics through text processing, parsing the structured output into fields suitable for Splunk visualisation. A related action, `needs_restarting`, identifies hosts requiring reboots after package updates by invoking the operating system's `needs-restarting` utility. This utility signals a required reboot through exit code 1, which under normal shell and Ansible conventions would indicate failure; the executor override described in Section 5.2.5 accommodates this for both `needs_restarting` and `health`, the latter embedding the same utility at the end of its compound command.

Disk monitoring warrants special treatment. The disk action collects filesystem utilisation with support for threshold-based filtering: analysts may specify a percentage threshold to surface only partitions exceeding a certain fullness level. Partition selection uses a comma-separated parameter that narrows output to particular mount points such as root filesystems or data volumes. Results include usage percentages, free space, and size class indicators that drive dashboard colour coding.

5.3.2 Splunk Monitoring Actions

Splunk monitoring actions query remote instance REST APIs through the SDK wrapper. The health action retrieves the health tree from the `splunkd/health` endpoint, parsing the hierarchical structure into flat results that indicate feature-level health status. The detailed mode expands this tree, yielding a row per health indicator with disabled state, status level, and any attached messages.

Queue monitoring addresses ingestion performance. The queue status action queries introspection endpoints to retrieve internal queue depths and fill percentages. When indexer pipelines fall behind, these queues grow; persistent high fill percentages indicate capacity problems that may lead to event loss. Results surface queue names, current sizes, maximum sizes, and percentage calculations that dashboard visualisations use for alerting thresholds.

Index metadata retrieval reveals data governance details. The index info action enumerates indexes with size statistics, event counts, and configuration details. Wildcard filtering allows analysts to focus on specific index patterns, such as internal indexes or application-specific indexes. The action calculates fill percentages relative to maximum size limits and surfaces bucket count metrics that indicate index health.

5.3.3 Satellite Integration

The Satellite module queries the Red Hat Satellite API for vulnerability and compliance data. Host enumeration retrieves managed systems with their patch status, operating system versions, and hostgroup memberships. The host listing action accepts search patterns for filtering by hostname, reducing result sets when investigating individual systems.

Errata reporting adds vulnerability context. The `errata_list` action retrieves available patches across the infrastructure, categorised by type (security, bugfix, enhancement) and severity level. Results include CVE identifiers where applicable, affected package counts, and reboot requirements. The `errata_by_host` action narrows this view to patches applicable to individual systems, populating the dashboard's vulnerability indicators.

5.3.4 Database Monitoring

The Extractor module tests connectivity to database sources used by the data collection infrastructure. Enterprise data extraction depends on scheduled connections to operational databases; silent failures interrupt data flows without generating alerts until derived metrics become unavailable.

The `db_test` action attempts connections to configured database endpoints, reporting success, failure, or timeout for each. Connection parameters reside in a separate configuration system tied to the extraction infrastructure. Connectivity tests cover Oracle, Microsoft SQL Server, and Sybase database types through appropriate Python database libraries.

To reduce load on the extraction infrastructure, the module employs a file-based caching layer, `ResultCache`, which stores query results as JSON files within the `local/cache/` directory with a configurable time-to-live of 1800 seconds (30 minutes). On each read, stale entries are purged automatically. A `force=true` parameter bypasses cached results when analysts require fresh data. Routine queries therefore read from the cache, and on-demand investigations always retrieve current state.

The `db_test` action discovers all configured extraction stanzas[1], distributes them round-robin across multiple collector IP addresses, and tests connectivity concurrently using a `ThreadPoolExecutor` with one worker per collector. Results are collected via `as_completed` and merged into the output stream, which reduces total execution time from the sum of individual timeouts to the duration of the slowest single collector.

5.3.5 Graph Client Implementation

The `GraphClient` class encapsulates all interaction with Microsoft's Graph API (Microsoft Corporation, 2026), handling authentication, request construction, response parsing, and error classification. Authentication implements the OAuth 2.0 client credentials grant (Hardt, 2012): the client constructs a POST request to Azure AD's token endpoint and caches the returned access token until it approaches expiry, refreshing lazily to avoid blocking application startup. An optional proxy URL routes all traffic through the corporate proxy for environments lacking direct internet access.

Error handling distinguishes between recoverable and terminal failures. Rate limit responses (HTTP 429) raise `GraphRateLimitError` with the `Retry-After` header value for caller-managed backoff. Authentication errors (HTTP 401) and permission errors (HTTP 403) raise distinct exceptions that signal credential or configuration issues. Pagination handling follows `@odata.nextLink` references until the complete result set has been accumulated.

5.3.6 Email Pipeline Monitoring

The Mail module coordinates several data sources to present unified email pipeline status. Instead of implementing direct integrations, the module delegates to existing modules and aggregates their results. This composition pattern reduces code duplication whilst offering a single point of access for email-related queries.

The `mailbox_health` action uses the Graph client library to verify mailbox accessibility. The action retrieves Graph credentials, instantiates the `GraphClient`, and enumerates mail folders for each configured mailbox. Results confirm that authentication succeeds and the mailbox remains accessible. Through this pattern, the Mail module can present Graph functionality without duplicating authentication or API interaction logic.

The `ingestion_status` action queries Splunk indexes for email event volumes. The implementation constructs an SPL search that counts events in configured email indexes over the past twenty-four hours and identifies the timestamp of the most recent event. These values feed into threshold checks: event counts below expected baselines trigger warning or critical status, as do excessive gaps since the last event.

[1] A stanza is a named configuration block in Splunk's `.conf` files, delimited by square brackets.

Threshold configuration uses module-level constants that permit adjustment without code modification. The implementation defines `INGESTION_WARNING_THRESHOLD` at 0.5 (50 per cent of expected volume) and `INGESTION_CRITICAL_THRESHOLD` at 0.1 (10 per cent). A single staleness constant, `INGESTION_STALE_HOURS`, set to six hours, flags sources whose most recent event exceeds the configured window. Operations teams can tune these values based on observed email volumes and acceptable detection latency.

The `pipeline_status` action examines Extractor execution logs stored in a summary index. The implementation queries for recent extraction script runs, checking exit codes and completion timestamps. Failed extractions surface immediately, whilst missing scheduled runs trigger alerts based on expected execution frequency. This check catches both explicit failures and silent scheduling issues where extraction scripts fail to run entirely.

5.3.7 Identity and Access Queries

As described in Section 4.3.2, the Identity module operates locally within the Splunk environment, querying lookups and KV Store collections through the inherited `self.service` connection without external credentials. These collections are defined in `collections.conf`, whilst lookup files are mapped to field names in `transforms.conf`.

All eleven identity actions share a `LookupExecutor` class that wraps Splunk's `service.jobs.oneshot()` method into a reusable query interface. The executor accepts a lookup name, optional `where` clauses, field selections, and a configurable `max_results` limit. It validates lookup names against a strict regex pattern before constructing SPL, which prevents injection through malformed input. An `enrich_lookup()` method extends this base capability with multi-lookup joins, building a single SPL pipeline that queries a base lookup and enriches rows with fields from additional lookups.

Six data sources feed the identity actions:

Active Directory User accounts and group memberships from the directory accounts and directory groups lookups, synchronised from the organisation's Lightweight Directory Access Protocol (LDAP) directory.

Access Governance Platform Access management records from the access records, access catalogue, and user records lookups.

HR System Employee records, department assignments, and manager chains from the HR data lookup.

Infrastructure Inventory Host records from the internal infrastructure inventory, with ownership, environment classification, and support contacts.

Splunk Accounts A Splunk role-mapping lookup that links Active Directory groups to Splunk roles.

Host Resolver A host resolution lookup mapping IP addresses to hostnames, used by the `host_identity` action.

Group queries use substring matching through SPL's `like()` function in place of exact equality, so analysts can search for partial group names without knowing the full distinguished name. Manager resolution follows a two-step lookup: the action first retrieves the manager's common name from the user's Active Directory record, then searches the accounts lookup by email alias or CN to return the manager's full profile. This approach handles the common case where the `manager` field stores a distinguished name rather than a username.

The module exposes eleven actions. `user_profile` and `identity_resolve` return consolidated views by fusing records across Active Directory, the access governance platform, the HR system, and Splunk accounts. `user_groups` and `group_members` query group membership in both directions. `user_access` retrieves access governance entitlements. `group_profile` returns metadata about a group itself. `host_identity` enriches an IP address or hostname with ownership and environment data from the internal infrastructure inventory. `managed_users` lists direct reports for a manager. `splunk_roles` derives a user's Splunk role assignments by matching their Active Directory groups against the role-mapping table. `inactive_accounts` identifies accounts where the last logon exceeds a configurable threshold. `identity_gaps` cross-references Active Directory accounts against HR records to detect orphaned identities or missing access reviews.

5.4 Dashboard Implementation

The dashboard combines a single Simple XML definition with JavaScript extensions that support its interactive features.

5.4.1 Dashboard XML Structure

The unified dashboard uses Splunk's Simple XML framework to define panel layouts and query bindings. Simple XML constrains each dashboard to a single monolithic XML file; accordingly, all sixteen tabs and their associated panels reside in one document. The performance optimisations described in Section 5.4.3 keep load times acceptable despite this scale. The dashboard consists of form inputs that establish global filtering context, followed by tabbed sections that organise content by monitoring domain. Each panel specifies its data source as either an inline search or a reference to a saved search, with results rendered through configured visualisation types.

Token variables link form inputs to panel queries. When an analyst selects a host from the dropdown, the selection value populates a token that panel searches reference in their SPL. This binding permits dynamic filtering without requiring each panel to implement its own input controls. Token inheritance cascades through the dashboard hierarchy, which maintains consistent filtering across all visible panels.

Simple XML does not include a native tab component. The original dashboard used a dropdown `<input>` element where each selection triggered hundreds of XML `<change>` conditions, making modifications error-prone and navigation cluttered as the view count grew.

A custom two-tier tab bar, implemented in JavaScript (`tabs.js`) and CSS (`tabs.css`), replaced this dropdown. The component organises sixteen pages into four thematic groups (Overview, Platform, Security, and Services) using a primary bar for group selection and a secondary bar for pages within each group. The JavaScript manages all `tab_*` and `show_*` dashboard tokens directly, eliminating the verbose XML condition blocks. Because Splunk re-renders HTML panel contents on search completion (wiping inline DOM attributes), the component injects its styles through dynamic `<style>` elements that persist across these re-renders.

This three-level navigation hierarchy (group overview, individual tab, and row expansion for detail) realises the visual information-seeking mantra of "overview first, zoom and filter, then details on demand" discussed in Section 2.5 (Shneiderman, 1996).

Table 5.1 lists all sixteen tabs with their group membership, primary data sources, and analytical purpose.

Appendix A.1 documents each tab's panel layout and data sources, including the Vulnerabilities tab's Satellite errata integration, chart configuration conventions, and the filtering controls available to analysts. Figure 5.2 presents the anonymised overview tab as rendered in the production deployment.

Figure 5.2: *Dashboard overview tab with high-level operational metrics*

5.4.2 Interactive Elements

Three interaction patterns were implemented to support exploratory analysis within the dashboard: drilldown navigation from summary KPIs to detail tabs, row expansion that surfaces secondary data inline beneath table rows, and cell tooltips that reveal truncated content on hover.

The Summary tab is the dashboard's entry point, presenting aggregated KPIs across all monitored subsystems. Clicking any summary panel triggers a drilldown that routes the analyst directly to the relevant detail view. Each panel configures a

Table 5.1: *Dashboard tab inventory by group, data source, and purpose*

Tab	Group	Data Source	Purpose
Summary	Overview	Native SPL + custom JS drilldowns	Cross-dashboard KPI landing view
Infrastructure	Platform	`| usoapi`	Host inventory and platform health
Performance	Platform	Native SPL / `mstats`	Host resource trend monitoring
Forwarders	Platform	Federated search	Forwarder fleet coverage and inventory
Logs Status	Platform	Saved search + custom JS	Log-source freshness overview
Sources/Data	Platform	Native SPL + snapshots + `| usoapi`	Index freshness and data-asset oversight
Suspicious Activity	Security	Native SPL + reputation lookup	SSH brute-force and external IP anomaly triage
Vulnerabilities	Security	Lookup-backed vulnerability dataset	Managed vulnerability backlog review
Firewall	Security	Native SPL	Blocked-traffic and connection analysis
Data Health	Security	Diagnostics summary index	Validation status for data assets
PII Exposure	Security	Diagnostics summary index	PII hit trends by index/type
Work Orders	Services	Lookup snapshot (`inputlookup`)	Ticket backlog and ageing view
Licenses	Services	Native SPL on licence index	Licence consumption and spike analysis
Automation	Services	Summary index + custom JS	Compliance pipeline run monitoring
Extractor	Services	Hybrid `| usoapi` + native SPL	Extractor reliability and stanza health
Email	Services	Hybrid `| usoapi` + native SPL	Email ingestion and mailbox status

Simple XML drilldown that sets a `summary_nav_target` token with the identifier of the destination. A navigation coordinator module listens for changes to this token and resolves the target through a mapping table that associates identifiers with particular tab and panel combinations. For targets that specify only a tab, the coordinator activates that tab and scrolls to the top of the page. For targets that reference a given panel within a tab, the coordinator activates the tab and delegates to a utility module that locates and highlights the target element.

The panel location logic accounts for lazy-loaded content by polling the DOM at 200-millisecond intervals for up to fifteen attempts after an initial 500-millisecond de-

lay, which ensures that panels rendered by deferred searches are found reliably. Once located, the target element is brought into view using `scrollIntoView` with smooth scrolling centred vertically in the viewport. A two-pulse CSS animation based on `panel-glow` keyframes then draws attention to the target panel through a transient glow effect. For users who have activated reduced-motion preferences, the animation is replaced with a static accent-coloured border. This drilldown pattern extends the information-seeking principle discussed in Section 2.5, giving analysts a single-click path from aggregate indicators to the corresponding detail panels across the dashboard's sixteen tabs.

Whilst drilldowns navigate between views, row expansion surfaces detail within the current view. When an analyst clicks a table row, JavaScript code intercepts the click event, extracts identifying information from the row's data, and executes a secondary search to retrieve additional detail. The script then injects the results into an expandable region below the clicked row, displaying the detail alongside the summary view.

This pattern applies across several tabs, each surfacing different detail when expanded:

Infrastructure Host metadata, disk usage, Splunk health features, applicable errata, critical services, and essential ports.

Vulnerabilities Vulnerability details, ownership, CVE references, errata summaries, and remediation guidance.

Automation Retry performance, data quality metrics, validation errors, and file paths.

Forwarders Application inventories, server class memberships, and Domain Name System (DNS) resolution results.

Extractor Stanza configuration details and recent error history.

Each expansion uses the same underlying JavaScript framework with tab-specific configuration. Figure B.1 presents the Infrastructure tab's host status view, and Figure B.2 illustrates the expanded detail panel for a selected host; both are reproduced at landscape scale in Appendix B.

The full implementation details of the row expansion architecture, including the caching strategy, cache invalidation mechanism, and chevron injection, are documented in Appendix A.2. Appendix C.3 presents the Infrastructure tab's host-status search, which fuses data from three modules and a forwarder lookup through successive joins.

The third interaction pattern addresses cell overflow in table rendering. The `cell_tooltip.js` module truncates overlong cells at a configurable character threshold, appending an ellipsis and a corner indicator. Hovering over a truncated cell displays the complete value in a positioned overlay using a `<pre>` element to preserve monospace formatting; clicking the indicator pins the overlay for text selection. A companion configuration file, `cell_tooltip_tables.js`, declares per-column truncation limits for thirteen tables across the dashboard.

5.4.3 Dashboard Performance Optimisation

Executing all searches simultaneously strains the search cluster and produces an unpredictable loading experience. The implementation addresses this through three complementary strategies: priority-based search sequencing, tab-aware search gating, and staggered autorefresh intervals.

Search Priority Queue

The dashboard implements a seven-stage priority queue using Splunk's token dependency system. Hidden base searches execute in sequence, each setting a token upon completion that gates the next tier. Panels declare dependencies on these tokens through the `depends` attribute, preventing execution until prerequisite stages complete.

The priority stages order searches into three broad categories by typical execution time. Fast searches (lookups and metadata queries completing in under one second) execute first, followed by moderate searches (REST API calls and licence queries requiring five to sixty seconds), and finally slow searches (accelerated `tstats` and vulnerability scans that may require several minutes). Each stage's sentinel search releases a token upon completion, gating the next tier.

Beyond priority ordering, the dashboard employs base search reuse to eliminate duplicate query execution. Named base searches each execute once and supply their results to several downstream panels through Splunk's `base=` attribute. Query construction favours pre-indexed data structures (`tstats`, `mstats`) over raw event scans, querying pre-built Time-Series Index (TSIDX) summaries and metric store structures instead.

Panels on the Licenses tab illustrate the dependency pattern. Instead of loading immediately, licence charts display a "Waiting for other searches to complete" message until the `license_queries_done` token appears. This feedback informs analysts that loading is progressing rather than stalled.

Figure B.6 in Appendix B presents the Licenses tab, including its licence-usage gauges, trend panels, and per-index and per-sourcetype breakdowns. The per-index and per-sourcetype panels use a third-party treemap visualisation add-on available on Splunkbase.[2]

[2] Treemap Viz: `https://splunkbase.splunk.com/app/7890`

Listing 3: *Priority queue token chain gating search execution*

```
<!-- Priority 1: FAST - Lookup-based searches (~0.5s) -->
<search id="priority_fast_lookup" depends="$workorders_ready$">
  <query>| inputlookup vulnerability_lookup | stats count</query>
  <done><set token="fast_lookups_done">1</set></done>
</search>

<!-- Priority 2: MODERATE - REST API calls (~5-10s) -->
<search id="priority_moderate_stats" depends="$fast_lookups_done$">
  <query>| rest /services/server/info | head 1</query>
  <done><set token="moderate_stats_done">1</set></done>
</search>

<!-- Priorities 3-5 follow the same chaining pattern -->

<!-- Priority 6: SLOWEST - Vulnerability timechart (~260s) -->
<search id="priority_vuln_scan_timechart" depends="$tstats_queries_done$">
  <query>| tstats count WHERE (...) earliest=-1d@d | head 1</query>
  <done><set token="vuln_timechart_done">1</set></done>
</search>
```

Appendix C.4 presents the full token dependency chain, including the `<fail>` handler that prevents stalls when a gated search fails.

Staggered Autorefresh

Continuous monitoring requires periodic data refresh, but simultaneous refresh of all panels creates the same load spikes that priority queuing addresses at page load. The implementation therefore uses no single global freshness interval. Instead, refresh cadences are assigned per data source according to volatility and acquisition cost. The `refreshType=delay` configuration ensures countdown begins after search completion rather than from a fixed timestamp, avoiding pile-ups caused by searches finishing at the same time.

The Logs Status materialiser runs every five minutes and writes a `materialized_at` timestamp into `log_sources_status.csv`; the dashboard then refreshes the corresponding base search every 300 seconds and recomputes age from the stored timestamp instead of rerunning the underlying `dbinspect` and `tstats` queries. The Sources/-Data tab consumes a 30-minute index-status snapshot alongside a daily profile job that recalculates day-of-week baselines from the previous four weeks, separating current freshness from historical expectation. The PII exposure summaries are rebuilt every thirty minutes, which is sufficient for operational monitoring without imposing continuous regex scans on production indexes.

Each panel exposes its data age directly to analysts. The Sources/Data table shows `Last Event`, `Age`, and `Health` fields, classifying freshness as healthy below one hour, delayed below six hours, and missing thereafter unless a source is baseline-suppressed

as low volume. The PII tab includes a `Last Scan` panel that displays minutes since the most recent scheduled scan, whilst the Logs Status grid shows the age of the latest event and exposes the last-event timestamp in its tooltip. Analysts therefore see both the current status and the recency of the evidence behind it.

Tab Priority Controller

The token-gating mechanism described in Section 4.4.4 is implemented in `tab_priority_controller.js`. Each tab's panels depend on a token following the `<tabname>_visited` naming convention; the controller sets only the active tab's token on page load, blocking all other tabs from dispatching searches.

An earlier iteration unloaded tabs on navigation, forcing analysts to wait for fresh searches on every switch. The current implementation uses progressive loading instead: once the initial tab completes, background tabs activate at five-second intervals in the order defined by the priority queue from Section 5.4.3. User navigation, however, takes absolute priority. When an analyst clicks a tab that has not yet loaded, the controller intercepts the event, cancels any pending background activation timers, and immediately triggers the `activate_tab` sequence for the selected view. This "queue-jumping" behaviour ensures that the dashboard remains responsive to active investigation needs whilst still warming the cache for secondary views during idle periods. Once all tabs have loaded, subsequent tab switches complete without delay since results remain cached in memory.

Listing 4: *Progressive tab-activation and search-prioritisation algorithm*

```
def activate_tab(target_tab):
    # Pre-populate filters to avoid unresolved token errors
    populate_defaults(target_tab)
    set_token(f"{target_tab}_visited", True)

    # Dispatch searches in priority order
    for priority in [FAST, MODERATE, SLOW]:
        searches = get_searches(target_tab, priority)
        # ... logic to dispatch searches based on token dependencies ...

def on_tab_click(selected_tab):
    # User interaction takes priority over background queue
    cancel_background_timers()
    activate_tab(selected_tab)
    # Resume background queue after a short delay
    resume_background_queue(delay=5000)

def dashboard_init(initial_tab):
    activate_tab(initial_tab)
    # Background activation of other tabs to warm cache
    for tab in other_tabs:
        wait(5000) # Staggered 5s background delay
        if not is_visited(tab):
            activate_tab(tab)
```

5.4.4 Logs Status Grid

The Logs Status tab employs a custom JavaScript visualisation, `logs_status_grid.js`, to display log source health as a grid of status boxes. Traditional table layouts scale poorly when monitoring dozens of log sources; a grid presentation delivers immediate status overview without scrolling.

The `logs_status_grid.js` module renders status boxes directly into the dashboard Document Object Model (DOM), bypassing the base view class pattern. Cascading Style Sheets (CSS) Grid layout positions these boxes in a responsive grid that adjusts column count based on container width. Each status box displays the log source name, a colour-coded status indicator, and optional metrics such as event counts or most recent event time.

A scheduled saved search, `Logs Status Materialiser`, runs every five minutes and writes freshness metrics to the `log_sources_status.csv` lookup. It computes these metrics from `dbinspect` and `tstats` queries, stamps each row with `materialized_at=now()`, and hands the dashboard a pre-computed dataset instead of forcing render-time recalculation. The dashboard panels then join this lookup with threshold definitions, recompute `age_seconds` from `materialized_at`, and render both an `age_display` string and the underlying last-event timestamp. Sources exceeding threshold counts render as green; those below threshold but non-zero render as amber; sources with zero events or missing entirely render as red. This CSV-driven design also simplifies maintenance: adding a new log source requires only appending a row to the lookup file with the source's index, category, and threshold values, rather than modifying long SPL queries.

The grid monitors 41 log sources across 14 categories spanning different indexes, sourcetype filters, and wildcard patterns. Sequential `map` subsearches, consolidated `tstats` queries, and REST/`dbinspect` hybrid strategies each introduced either excessive query time or edge cases with metadata accuracy. Moving all computation into the scheduled saved search resolved these trade-offs and reduced perceived dashboard load time to sub-second levels.

Sort ordering prioritises critical and warning states, placing sources requiring attention at the top left of the grid. Figure B.5 in Appendix B presents the complete Logs Status tab, including the status grid and ingestion trend panels.

Several additional tabs use native SPL rather than `| usoapi`. The Forwarders tab uses Splunk's federated search feature to retrieve deployment server metadata via `| from federated:forwarders_ds01`, extracting application inventories and computing polling recency from `lastPhoneHomeTime`. Version visualisations are derived at search time from the reported `splunkVersion` field, with one chart grouping full version strings and another extracting major versions for operating-system comparison. This avoids hardcoded full-version lists and reduces the maintenance burden as new major releases appear. Performance and Firewall tabs query indexed metrics and native firewall logs directly. The `| usoapi` command is reserved for tabs bridging external

systems not natively queryable from Splunk, or for the multi-source record fusion required by the Identity module. Appendix A.1 documents the panel layouts and query strategies for each tab, whilst Appendix B includes screenshots of the Logs Status, Performance, and Forwarders tabs (Figures B.5, B.3, and B.4).

Listing 5: *Materialised saved search computing log source freshness*

```
[Logs Status Materialiser]
search = | inputlookup log_sources_lookup
  | rename source_id as title
  | eval has_st_filter=if(isnotnull(sourcetype), 1, 0)
  | join type=left title
  [| dbinspect index=infra_* OR index=security_*
       | stats max(endEpoch) as latest by index
       | rename index as title]
  | append [| inputlookup log_sources_lookup
       | where isnotnull(sourcetype) AND source_type="index"
       | map maxsearches=20 search="| tstats latest(_time)
         as st_latest WHERE (index=$title$
         sourcetype=$sourcetype$) by index"]
  | stats first(label) as index_label ... by title
  | eval materialized_at=now()
  | outputlookup log_sources_status.csv
```

5.5 Compliance Report Pipeline

The compliance report pipeline uses a Python alert action and a pair of shell scripts to implement four processing stages triggered by the scheduled search.

5.5.1 Processor Implementation

The central component is a Python script, `compliance_processor.py`, registered as a custom alert action in `alert_actions.conf`. When a scheduled search completes, Splunk invokes this script with a JSON payload containing the search results file path, session credentials, and user-defined parameters such as the report name and destination environment.

After initialising a Splunk SDK connection for summary-index logging, the processor reads search results from the temporary file that Splunk provides, counts rows, and writes validated data to a dated CSV file within a local staging directory. Subsequent stages delegate to external scripts: `compliance_transform.sh` converts comma-separated values to pipe-delimited format using the `csvformat` utility and compresses the output as a ZIP archive, whilst `compliance_upload_prd.sh` transfers the archive via rsync over SSH with explicit host key verification and ownership flags. Each file also receives an Message-Digest Algorithm 5 (MD5) checksum, recorded alongside the outcome in the summary index to maintain an audit trail linking pipeline runs to delivered artefacts.

5.5.2 Robustness Mechanisms

Fault tolerance relies on a decorator-based retry mechanism (Gamma et al., 1994) and threshold-driven validation, shown in Listings 6 and 7 respectively.

The retry decorator wraps network-dependent functions (upload, post-upload verification) and re-invokes them on failure up to a configurable maximum, with the delay computed as $base^{attempt}$ (Nygard, 2018). Module-level constants `MAX_RETRY_ATTEMPTS` and `RETRY_BACKOFF_BASE` govern the bounds, defaulting to three attempts with a base of two seconds.

To handle varying data volumes, the system uses a per-report threshold dictionary, `REPORT_ROW_THRESHOLDS`, for row-count validation, which falls back to a global minimum defined by `MIN_ROW_COUNT_THRESHOLD`. This dictionary-based approach accommodates reports whose volumes differ by an order of magnitude without requiring separate logic.

Listing 6: *Retry decorator with exponential backoff for network operations*

```python
def retry_with_backoff(
    max_attempts: int = MAX_RETRY_ATTEMPTS,
    backoff_base: int = RETRY_BACKOFF_BASE,
) -> Callable[..., Any]:
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @wraps(func)
        def wrapper(*args: Any, **kwargs: Any) -> Any:
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts:
                        raise
                    backoff_time = backoff_base ** attempt
                    logger.warning(
                        f"{func.__name__} failed "
                        f"(attempt {attempt}/{max_attempts}): {e}")
                    time.sleep(backoff_time)
        return wrapper
    return decorator
```

Listing 7: *Row-count validation blocking upload of incomplete reports*

```python
min_rows = REPORT_ROW_THRESHOLDS.get(
    report_name, MIN_ROW_COUNT_THRESHOLD)
if validation_results["total_rows"] < min_rows:
    is_valid = False
    errors.append(
        f"Row count {validation_results['total_rows']} "
        f"below minimum threshold {min_rows}")
```

To verify these mechanisms without affecting production partners, the pipeline includes a dedicated test report configuration. This test report exercises the full pipeline path against a staging SFTP environment isolated from production, using a system index that produces approximately one hundred rows. A reduced validation threshold confirms that row-count checks, retry logic, and notification channels function correctly. A second test configuration intentionally produces empty result sets, validating that the pipeline detects and marks zero-row reports through its marker mechanism rather than attempting delivery. Both configurations are disabled by default and dispatched manually when changes to the pipeline require validation.

5.5.3 Dashboard Integration

The Automation tab described in Section 4.5.3 displays a table of recent pipeline executions, colour-coded by outcome, with expandable rows that reveal failure details (Figure B.7 in Appendix B).

The row expansion behaviour is implemented in JavaScript, `compliance_row_expansion.js`, using Splunk's Web Framework. When an analyst expands a row, the script executes a secondary search against the summary index to retrieve the full log entry for that execution, including error messages and validation metrics. This information appears inline within the table, so analysts can diagnose issues without navigating away from the dashboard or accessing server logs directly.

5.6 Diagnostics Implementation

Automated health checking runs through a separate Splunk application, `uso_TA_diagnostics`, which monitors scheduled search outputs, file system artefacts, and KV Store collections. Splunk invokes the entry point script, `uso_diagnostics.py`, as a custom alert action with the `--execute` flag, passing a JSON payload through standard input that contains the alert's session key, search metadata, and user-configured parameters (Figure B.9). Once the script initialises a Splunk SDK connection from the session key, it delegates execution to the module dispatcher.

5.6.1 Module Dispatcher

Implemented in `dispatcher.py`, the module dispatcher uses a `MODULES` registry dictionary that maps module names to their handler classes. Module discovery is static: each handler class is imported and registered at the module level, which keeps the mapping explicit and avoids dynamic import complexity. When invoked, the dispatcher iterates through the modules that the alert configuration has enabled, wrapping each execution in an optional retry loop with configurable exponential backoff. If a module raises an exception on its first attempt, the dispatcher waits 2^n seconds before retrying, up to a maximum number of attempts defined in the alert parameters.

Each module receives the alert payload and configuration parameters, executes its checks, and returns normalised result dictionaries containing a status code, a human-readable summary, and metric-specific fields. After all modules complete, the dispatcher logs every result as an event in a configurable summary index, then assembles a consolidated notification summary from the individual module outcomes, grouping findings by severity level before forwarding them to the notification subsystem.

Listing 8: *Diagnostics module registry with per-module configuration*

```
@dataclass
class ModuleConfig:
    name: str           # Module identifier (job, file, kvstore)
    module: Any         # Module with run_diagnostics() method
    enable_key: str     # Config key to check if enabled
    default_enabled: bool

MODULES = [
    ModuleConfig("job", job_diagnostics,
                 "enable_job_diagnostics", True),
    ModuleConfig("file", file_diagnostics,
                 "enable_file_diagnostics", False),
    ModuleConfig("kvstore", kvstore_diagnostics,
                 "enable_kvstore_diagnostics", False),
]
```

5.6.2 Diagnostic Modules

Three diagnostic modules address distinct monitoring concerns. Of these, job diagnostics operates closest to Splunk's internal APIs: it captures the search identifier from the alert payload and enriches this data by querying the REST endpoint `search/v2/jobs/{sid}`. Configurable checks include search string capture, zero-result warnings when a scheduled search produces no events, and maximum-duration warnings when execution time exceeds a threshold.

File validation addresses a different concern: confirming that generated artefacts on disk meet configurable expectations. Input accepts three formats (a single file path, a semicolon-delimited list, or glob patterns for dynamic resolution) and for each resolved path, the module checks existence, readability, size, row count, MD5 checksum, and file age. Two batch modes govern aggregation across multiple files. Under `all_required`, a single absent or invalid file fails the entire check; under `any_required`, the check succeeds when at least one file passes validation. Summary statistics then report total files checked, pass and fail counts, and aggregate size.

Unlike the job module, which queries transient search artefacts, KV Store diagnostics validates persistent collection health through Splunk's REST interface. After confirming that a collection exists and is accessible, the module counts rows using paginated reads against the `/storage/collections/data` endpoint. Configurable minimum and maximum thresholds flag collections that have grown unexpectedly or shrunk below operational minimums. Both the file and KV Store modules support optional historical anomaly comparison, described below.

5.6.3 Historical Comparison Engine

Beyond static threshold checks, the subsystem detects gradual drift through historical comparison. Implemented in `historical.py`, this engine queries the diagnostics summary index for previous execution results and requires at least three historical records before computing baselines. From those records, it derives average and median values for each tracked metric, then applies configurable per-metric deviation thresholds to determine whether a current value exceeds acceptable deviation.

Listing 9: *Historical baseline anomaly detection algorithm*

```
def check_anomaly(current_val, history, threshold_pct):
    if len(history) < 3:
        return PASS # Insufficient data for baseline

    baseline_mean = sum(history) / len(history)
    deviation = abs(current_val - baseline_mean)
    max_allowed = (threshold_pct / 100) * baseline_mean

    if deviation > max_allowed:
        return ANOMALY(
            observed=current_val,
            expected=baseline_mean,
            diff_pct=(deviation/baseline_mean)*100
        )
    return PASS
```

5.6.4 Notification System

The notification subsystem implements the pluggable architecture specified in Section 4.6.3 and extends it with a third channel and transport-level resilience. Beyond the `WebhookNotifier` and `SplunkBulletinNotifier` backends defined in the architecture, an `EmailNotifier` delivers HTML-formatted diagnostic summaries over SMTP via Python's `smtplib` with configurable server, port, and recipient list; the `email_enabled` parameter gates dispatch and restricts it to error, warning, and success conditions.

Each transport backend applies retry logic through the shared retry library (Section 5.9.4). Webhook delivery retries on HTTP 403, 429, and 5xx responses with exponential backoff, bulletin board writes recover from transient REST errors, and email delivery handles connection failures. All notification parameters (module toggles, per-metric thresholds, historical comparison settings, retry counts, and notification preferences) reside in `alert_actions.conf` alongside the output index and sourcetype configuration. Figure B.9 shows the resulting configuration interface as rendered by Splunk.

5.7 PII Detection Implementation

A separate Splunk application, `uso_TA_piifinder`, implements the secondary privacy-monitoring extension. The architecture reflects privacy-by-design considerations (Spiekermann et al., 2009; Gürses et al., 2011) and can be read against privacy-engineering frameworks such as LINDDUN (Deng et al., 2011; Wuyts, 2015), though no formal LINDDUN threat modelling process was conducted. In particular, the stateless nature of the detector enforces data minimisation—it scans, annotates, and forwards events without retaining any PII in persistent memory or local storage.

5.7.1 Streaming Command Architecture

The `PIIFinderCommand` class extends Splunk's `StreamingCommand` with four configurable options: `fields` (which event fields to scan, defaulting to `_raw`), `types` (which PII categories to detect), `mode` (output format corresponding to the three modes defined in Section 4.7.3), and `min_confidence` (filtering threshold). Splunk passes events one at a time through the `stream()` method, which annotates each event with detection results and yields it back into the pipeline.

Each mode produces distinct output fields. In `counts` mode, the command appends `pii_total`, per-type counters such as `nif_count` and `email_count`, and confidence-level tallies. In `detail` mode, `pii_detail` entries list each match with its type, matched text, and confidence level. In `redact` mode, detected values are replaced inline with `[REDACTED:TYPE]` markers.

A safety limit of one million characters per field prevents catastrophic regex backtracking on oversized events. Fields exceeding this threshold are skipped with a `pii_skipped` marker, and fields shorter than six characters bypass detection entirely as they cannot contain meaningful PII patterns.

5.7.2 Rust Core and Python Wrapper

The Python wrapper communicates with the Rust binary through a newline-delimited JSON protocol over standard input and output, using two protocol modes. In `counts` mode, an initialisation handshake configures the Rust process with the active detector list and minimum confidence level; subsequent messages send batches of 256 events as JSON objects, and the Rust process returns summary counts. In `detail` and `redact` modes, each request contains a complete batch with all configuration, and the response includes per-field detection hits.

The `RustBackend` class manages the subprocess lifecycle. At startup, it verifies binary existence and executability; if either check fails, or if the process crashes mid-session, the wrapper activates the pure-Python fallback path transparently. Beyond the `RegexSet` prefilter and Aho-Corasick optimisations discussed in Section 4.7.2, the Rust backend applies a further optimisation in `counts` mode: it avoids constructing individual hit structures and returns only aggregate counters, eliminating allocation overhead for the most common use case.

5.7.3 Detection Engine

The fourteen detectors defined in Section 4.7.2 are implemented as `Detector` dataclass instances, each specifying a compiled regex pattern, an optional validator function, context and negative keyword lists, a default confidence level, and a `requires_context` flag.

Per-detector negative keyword lists suppress known sources of noise; the phone detector, for instance, carries over forty negative keywords including session identifiers, build numbers, and trace identifiers. Quick-reject functions perform inexpensive pre-checks before invoking expensive regex evaluation; the email detector skips fields lacking an `@` character, and the credit card detector skips fields with fewer than thirteen digits. IPv4 detection is excluded from the default detector set because proxy and load-balancer logs generate excessive matches; analysts opt in explicitly with `types=ipv4`.

5.7.4 Dashboard and Scheduled Scanning

The PII Exposure tab on the unified dashboard presents detection results through four KPI panels (total PII hits, events containing PII, distinct PII types, and high-confidence findings), three trend charts (PII hits over time by type, confidence distribution, and index-level exposure), and a summary table listing all fourteen PII categories with per-index hit counts.

A scheduled saved search, `PII Exposure Scanner`, runs every thirty minutes, as defined in `savedsearches.conf`. It samples a bounded subset of events from each monitored index group, pipes them through `| piifinder mode=counts min_confidence=medium`, aggregates the counts by source index, records the applied `sample_cap` and `scan_time` in the output, and writes the results to the diagnostics summary index (`uso_diagnostics`) with sourcetype `uso_piifinder_summary`. The dashboard reads from this summary index rather than scanning raw data at render time, which keeps panel load times under one second. A dedicated `Last Scan` panel computes elapsed minutes from the most recent `scan_time`, and the positive-findings table surfaces `Last PII Seen` timestamps for each index and detector combination. Appendix C.6 presents the full saved search. Figure B.8 in Appendix B shows the complete tab.

5.8 Authentication and Security Implementation

5.8.1 Service Account Provisioning

Before token-based authentication could operate, the service account had to be configured in both Active Directory and Splunk's authentication layer. The dedicated monitoring service account, following the naming convention and capability requirements described in Section 4.3.3, already existed in the organisation's Active Directory as a machine-to-machine identity. Gaining access to it for use within Splunk required a formal request through the IT governance process, which granted the cybersecurity

operations team authorisation to bind it to the search head cluster's authentication configuration. Because service accounts reside in a different directory subtree from standard user accounts, the cluster's LDAP configuration also required adjustment so that Splunk could resolve the identity during authentication (Splunk Inc., 2026h).

A dedicated Splunk role, `sec-mon-access`, was created with a least-privilege capability set scoped to health monitoring and introspection endpoints. A role mapping stanza then linked the corresponding Active Directory group to this role, completing the authentication chain: group membership in Active Directory, LDAP resolution by Splunk, and role mapping to capabilities. Because all configuration resides in the deployer's cluster app, changes propagate consistently to every search head without per-node intervention. Appendix A.3.5 documents the LDAP configuration, capability list, and role mapping.

5.8.2 Token-Based Authentication Implementation

Building on the credential architecture designed in Section 4.3.3, the implementation uses bearer token authentication with JWTs. When the Splunk SDK module connects to a remote instance, it retrieves the target's token from the credential store and passes it to the SDK's `connect()` function via the `splunkToken` parameter. Remote queries therefore execute under the service account identity associated with the stored bearer token, rather than the invoking analyst's credentials.

Token detection relies on JWT format recognition. The `_is_jwt_token()` method examines credential values for the characteristic "eyJ" prefix that indicates Base64-encoded JSON header data. This heuristic distinguishes tokens from other credential types that might exist in the storage API.

Storage uses Splunk's native `storage/passwords` endpoint, which encrypts credential values within `passwords.conf` and replicates entries across search head cluster members. The username field encodes structured metadata following a defined format that includes the target IP address, an optional human-readable label, the creation timestamp, and a service identifier. This structure permits retrieval by IP address for programmatic access whilst also supporting label-based queries for analyst convenience.

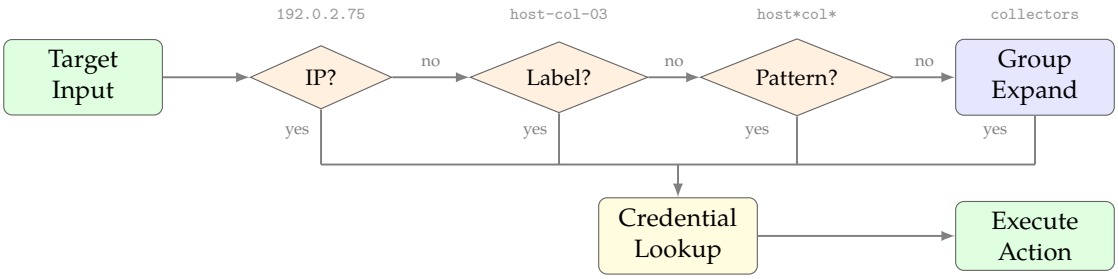


Figure 5.3: *Target Resolution and Credential Flow*

Three command-line tools manage application credentials: a token manager for Splunk bearer tokens, a Satellite credential manager for Red Hat Satellite username/-password credentials, and a Graph credential manager for OAuth 2.0 client credentials. Appendix A.3 documents these three credential-management tools alongside the separate SSH key procedure.

5.8.3 Error Handling and Security

Error messages balance diagnostic utility with security considerations. When authentication fails, the system reports that authentication was denied without revealing whether the target exists, the credential was wrong, or the account lacks permissions. Connection timeout messages indicate network or availability issues without exposing internal addressing.

Credential handling follows secure practices throughout execution. Tokens appear in memory only during active use and pass to SDK functions through parameters rather than command-line arguments that might appear in process listings. Log messages redact full token values, displaying only abbreviated prefixes.

5.9 Iterative Refinement

The implementation took shape incrementally as operational feedback exposed weaknesses in each subsystem.

Lifecycle Management While operational stability necessitates long-lived bearer tokens for uninterrupted monitoring, a formal revocation procedure mitigates the risk of credential compromise. Rather than relying on fixed-timer expiration which could disrupt critical observability workflows, token rotation is event-driven—triggered by personnel changes, suspected compromise, or periodic audits. Revocation is executed by deleting the compromised entry directly from the `storage/passwords` endpoint, immediately severing access without requiring service restarts.

5.9.1 Bearer Token Security Evolution

Authentication for remote Splunk instances began with a migration from a custom secrets workflow to the platform's native `storage/passwords` API. This initial implementation still accepted inline username and password parameters through the command interface, preserving backward compatibility with analyst workflows that predated token-based authentication. Operational review revealed that this flexibility was itself a vulnerability: analysts could inadvertently pass plaintext credentials through search history, job artefacts, or shared saved searches.

A subsequent refactoring enforced token-only authentication for remote targets. Inline authentication parameters were removed from the command syntax, and the search grammar definitions in `searchbnf.conf` were updated to reflect the narrower

interface. A final pass eliminated all remaining backward-compatibility code paths that had handled username and password credentials, leaving bearer tokens as the sole authentication mechanism. Each iteration removed an attack surface rather than adding defensive controls around it.

5.9.2 Mail and Graph Module Consolidation

Because the email pipeline spanned two concerns (querying Microsoft Graph API endpoints for mailbox health and checking internal ingestion pipelines), the initial design created separate modules: `graph` for OAuth2 client-credential flows and API queries, `mail` for pipeline status checks. In practice, the barrier proved artificial. Mail actions needed Graph API internals to verify mailbox folder statistics, whilst Graph actions depended on mail pipeline context to interpret their results meaningfully. Rather than introducing cross-module imports that would have created circular dependencies, the two modules were consolidated into a single `mail` module with the Graph client library and OAuth2 credential management retained as shared internal utilities in a sibling `graph_submodules` package. The merged module better reflected the actual dependency structure and eliminated the duplicated action registries, base classes, and constant definitions that the split had required.

5.9.3 Compliance Report Pipeline Modularisation

All report processing logic originally resided in a single processor file. This monolithic design was adequate for the initial scope of extracting, transforming, and uploading compliance data. When email notification capabilities were added, however, the processor grew to accommodate CSV attachment generation, recipient management, and delivery confirmation alongside the existing pipeline logic.

The resulting file mixed concerns at every level: SSH upload functions sat adjacent to notification formatting, and indexing logic sat alongside validation checks. A dedicated refactoring split the processor into focused modules for pipeline orchestration, SSH operations, notifications, indexing, constants, and shared utilities.

A bug in the anomaly detection logic for reports configured to skip the CSV header row required a targeted fix in a single pipeline module, a change that would have been considerably harder to isolate and verify in the original monolithic processor.

5.9.4 Diagnostics Retry Centralisation

Retry logic in the diagnostics subsystem started as a local concern. Job diagnostics, file diagnostics, and KV Store diagnostics each implemented their own exponential backoff when queries against the Splunk REST API failed due to transient errors. Whilst each implementation was correct in isolation, the three copies diverged in their backoff parameters, maximum attempt counts, and error classification logic.

Recognising this duplication, a refactoring introduced a dispatcher pattern with a shared library that centralised retry behaviour. Each diagnostic module registered itself with the dispatcher and delegated retry decisions to the common library, which applied consistent backoff timing and error classification across all modules. This centralisation then extended naturally to the notification transport layer: when webhook delivery to Mattermost encountered HTTP 403, 429, or 5xx responses, Splunk bulletin board writes failed due to transient REST errors, or email delivery encountered connection failures, the same shared retry logic handled recovery. The shared retry library now serves both diagnostic modules and the notification layer.

5.9.5 PII Detection Architecture Evolution

The PII detection command began as a monolithic Python script with inline regex patterns and a simple risk-scoring system. Production deployment against over 400 indexes revealed that the initial approach generated excessive false positives, particularly from name, NIF, and phone detectors. After disabling name detection entirely and applying targeted fixes, development paused for three months.

The second iteration redesigned the architecture around a detector registry pattern with fourteen independent detectors, three operating modes, and a three-tier confidence scoring system. Context gating and negative keyword suppression brought false-positive rates to production-acceptable levels across the full index catalogue. This rewrite also introduced a pytest suite covering individual detectors, confidence helpers, the detection engine, and end-to-end streaming behaviour.

The third iteration ported the detection logic to Rust for up to a fourfold throughput improvement for batches exceeding 50,000 events (Section 6.6.5). The Python wrapper retained identical detection logic as a fallback, and behavioural equivalence was verified against the existing test suite.

5.10 Challenges and Solutions

Tables 5.2 and 5.3 synthesise the implementation phase. Specifically, Table 5.2 categorises the technical challenges encountered, whilst Table 5.3 outlines the resulting design decisions and the trade-offs accepted to meet the system requirements.

The preceding sections presented the implementation of all system components, from the core API framework and its six integration modules to the compliance report pipeline, diagnostics subsystem, and PII detection engine, each deployed and validated in the production environment. Chapter 6 assesses the degree to which this implementation fulfils the requirements defined in Chapter 3, drawing on production metrics, usability observations, and an analysis of remaining limitations.

Table 5.2: *Implementation challenges and solutions*

Challenge	Problem	Solution	Section
Command type selection	`GeneratingCommand` vs `StreamingCommand` semantics determine API design; infrastructure queries fetch state rather than transform events	USO API uses `GeneratingCommand`; `piifinder` uses `StreamingCommand` for event enrichment	5.2.1, 5.7.1
Ansible return codes	`needs-restarting` returns exit code 1 for "reboot needed", which Ansible propagates as failure	Executor treats codes 1 and 2 as valid responses for specific actions	5.2.5
Data source mismatch	External APIs return nested structures incompatible with Splunk's flat table model	Per-action transformation layer isolates display from API interaction	5.3
SDK permissions	REST endpoints require dedicated capabilities (e.g., `list_health` vs `list_health_subset`) discovered through 403 errors	Service account role updated with monitoring capabilities	5.8.1
API rate limiting	AbuseIPDB daily limits exhausted under frequent dashboard refreshes	Lookup-table cache with staleness threshold and periodic maintenance	4.4.3
PII false positives	Name detection: 85% false-positive rate; NIF matched timestamps; phone matched trace IDs	Context-gated detectors, negative keyword lists (40+ terms), field-label requirements	5.7.3, 5.9.5
Python-to-Rust migration	Pure-Python throughput insufficient for scheduled scans within search time limits	Rust subprocess with JSON-line IPC; Python fallback retained; equivalence verified against the existing test suite	5.7.2

Table 5.3: *Technical decision matrix: design choices, rejected alternatives, and trade-offs*

Decision	Chosen	Rejected	Trade-off	Evidence
Integration model	Embedded Splunk search command (no middleware)	External Flask/FastAPI service	Increased Splunk coupling but eliminated external service dependency, network hop, and separate deployment	100% uptime over 5 months
Command type	`Generating-Command` for USO API	`Streaming-Command`	Structured table output vs raw event enrichment; composability with SPL pipelines preserved	5.2.1
Ansible concurrency	`FORKS=30` with 10s/120s timeouts	Higher fork counts (50+)	30 forks balance parallel speed against Search Head CPU/memory pressure for ~50 managed hosts	5.2.5
PII engine	Rust core with Python fallback	Pure Python; pure Rust	Hybrid adds IPC complexity but yields 4× throughput whilst preserving SDK integration and graceful degradation	Table 6.12
Dashboard loading	7-stage priority queue with tab-aware gating	Simultaneous panel dispatch	Added implementation complexity in JS token management; reduced initial load from >120s to ~15s	5.4.3
Data refresh	Polling (scheduled searches, autorefresh)	Real-time streaming	Higher latency (30s–5min) but predictable resource consumption and no persistent search slots consumed	5.4.3

6 Evaluation and Discussion

This chapter evaluates the implemented system against its requirements, assesses operational impact, and reflects on the lessons learned during development. The evaluation follows the Design Science Research (DSR) framework introduced in Section 1.4, and draws on case study methodology (Yin, 2018) as adapted for software engineering research by Runeson et al. (2009).

6.1 Evaluation Methodology

Hevner et al. (2004) identify seven guidelines for DSR, of which guideline three (design evaluation) requires demonstrating the artefact's utility, quality, and efficacy through well-executed methods. Peffers et al. (2007) position evaluation as the fifth phase of the DSR process model, where the artefact is observed and measured against the objectives defined during problem identification. Table 6.1 maps each DSR phase to the corresponding project chapter.

Table 6.1: *Mapping of DSR process phases to project structure*

DSR Phase	Chapter	Evidence Source
Problem identification	Section 1.2	Operational workflow analysis
Objectives definition	Section 1.3	Stakeholder requirements
Design & development	Chapters 4–5	Architecture decisions and implemented artefact
Demonstration	Sections 6.6.1–6.6.4	Production use scenarios
Evaluation	This chapter	Metrics below
Communication	This project	Documented artefact

The evaluation employs three complementary methods (Table 6.2), each targeting a distinct research question from Section 1.3:

- **Workflow comparison** (RQ1): analyst-attested time estimates for representative scenarios before and after deployment, supported by process decomposition of the manual steps involved.

- **System measurement** (RQ2): browser-observed time to first usable active-tab dashboard content and infrastructure-level telemetry from the priority queue controller during routine production use.
- **Production telemetry** (RQ3): structured diagnostic events from a dedicated Splunk summary index, measuring deviation rates against historical baselines over a four-week monitoring period.

Table 6.2: *Research question to evaluation evidence mapping*

RQ	Evidence	Section	Method
RQ1	Workflow comparison, context-switching reduction, usability assessment	6.6.1–6.6.2, 6.6.7	Analyst attestation, process decomposition, SUS
RQ2	Time to first usable active-tab content, priority queue metrics	6.5.1, 5.4.3	Production observation (browser timing)
RQ3	Diagnostics baseline comparison, deviation rates	6.6.4	Production telemetry (summary index)

RQ3 in particular rests on observational production telemetry from the diagnostics summary index; no controlled threshold-versus-baseline experiment was performed.

Three methodological constraints bound the validity of these findings. First, no controlled experiment with randomised participants was conducted; workflow time estimates derive from analyst attestation rather than timed trials. Second, no formal baseline measurement was recorded before deployment, so before-and-after comparisons rely on retrospective process analysis instead of pre-registered metrics. Third, production telemetry reflects a single organisational context with approximately 50 hosts, 20 Splunk instances, and a five-month observation window (November 2025 to April 2026). These constraints are discussed further in Section 6.7.3.

6.1.1 Requirements Traceability

Table 6.3 traces each functional requirement from its specification through architecture, implementation, and evaluation, mapping the end-to-end coverage across project chapters; Table 6.4 provides the same view for non-functional and supplementary security and privacy requirements, keeping the quality-attribute traceability explicit without repeating the fuller fulfilment discussion in Section 6.3.

6.2 Testing Approach

The project adopted a dual validation strategy that evolved during development. Early phases relied on production-centric validation: changes were tested in a dedicated Splunk development instance and then deployed to the production environment used by the team. This approach suited tools that query live infrastructure and process real security data.

Table 6.3: *Requirements traceability matrix across project chapters*

Req.	Component	(Ch. 4) Architecture	(Ch. 5) Implementation	(Ch. 6) Evaluation
FR1	ansible	4.3.2	5.3.1	6.6.1, 6.5.1
FR2	splunk	4.3.2	5.3.2	6.3.1
FR3	extractor	4.3.2	5.3.4	6.6.6
FR4	satellite	4.3.2	5.3.3	6.6.6
FR5	mail	4.3.2	5.3.6	6.3.1
FR6	identity	4.3.2	5.3.7	6.6.6
FR7	piifinder	4.7	5.7	6.6.5
FR8	dashboard	4.4	5.4	6.6.1
FR9	diagnostics	4.6	5.6	6.6.4

Table 6.4: *Compact traceability matrix for non-functional and supplementary requirements*

Req.	Implementation Anchor	Evaluation Evidence	Status
NFR1	5.2.5, 5.4.3, 5.4.3	6.5.1	Achieved
NFR2	4.8, 5.9.1	6.3.2	Achieved with residual risks
NFR3	5.5.2, 5.6.1	6.3.2, 6.6.3	Achieved
NFR4	5.2.5, 5.3.4	6.5.2	Partial
NFR5	5.2.2, 5.9.3	6.3.2	Achieved
NFR6	5.2.4, 4.4.1	6.6.7	Achieved
SPR1	5.7, 5.7.4	6.3.2, 6.6.5	Achieved
SPR2	4.3.3, 5.9.1	6.3.2	Achieved
SPR3	4.8.1, 4.8.5	6.3.2	Achieved

As the codebase grew, the need for regression protection became apparent. An automated unit test suite was introduced during the later stages of development to complement operational validation. Initial test scaffolding was accelerated by an AI coding assistant, after which each generated file was inspected, adjusted, and extended to cover project-specific edge cases and mocking patterns. The combination caught distinct defect classes: unit tests surfaced parsing regressions, whilst production testing exposed environment-dependent failures such as authentication mismatches and API response changes. A local development helper guards against this by refusing to proceed if any automated test fails.

6.2.1 Automated Test Suite

The automated test suite comprises 1,466 tests across four project components: 1,114 for the core USO API, 82 for the diagnostics subsystem, 43 for the compliance report pipeline, and 227 for the PII detection tool (222 Python unit tests plus five Rust integration tests). The combined Python suites execute in under five seconds, supporting rapid feedback during development.

The USO API tests follow a consistent per-module pattern. Four of the six integration modules (Ansible, Splunk, Satellite, and Extractor) have dedicated test files covering four aspects: registry integrity checks confirm that all actions are correctly reg-

istered and discoverable; base class tests validate shared functionality inherited from `CommonActionBase`; action parsing tests verify that each action correctly handles its expected output formats; and executor tests exercise the module's execution pipeline including error handling and dispatch. The Mail module covers three of these aspects, omitting executor tests because it aggregates results from multiple subsystems rather than following a single execution pipeline. Beyond the integration modules, the Graph library has targeted tests for the `GraphClient` class and credential manager. Root-level tests cover the custom search command class, CLI utilities, secret masking, and the self-documenting help system.

The mocking strategy isolates tests from external dependencies entirely. The Splunk SDK (`splunklib`) is mocked at import time through a shared `conftest.py` module, removing the need for a running Splunk instance. A `FakeBaseAction` stub supplies the authentication and service context that actions expect, whilst SSH connections, HTTP calls, and database queries are intercepted through `pytest-mock` patches. Each module's test fixtures supply representative response data modelled on production outputs, which ensures that parsing logic is validated against realistic formats instead of synthetic placeholders. This approach guarantees deterministic execution: the full suite passes regardless of network state or infrastructure availability.

The diagnostics subsystem has 82 tests across nine modules, covering the dispatcher and entry point, file monitoring, KV Store validation, job diagnostics, historical baseline comparison, results processing, and notification formatting. Twenty-eight of these tests target the notification classes (`WebhookNotifier`, `SplunkBulletinNotifier`, `EmailNotifier`, and `NotificationHandler`), mocking HTTP requests and Splunk service connections to verify message formatting, delivery logic, fan-out handling, and error recovery.

The PII detection tool (`uso_TA_piifinder`) has 227 tests: 222 Python unit tests exercising all 14 detectors, the confidence scoring system, false positive regression cases, and end-to-end streaming command behaviour across all three operating modes. Five Rust integration tests verify the binary's JSON protocol and batch processing. The Python tests cover detector-specific validation logic (Luhn checks for credit cards, mod-11 verification for Portuguese tax identifiers, mod-97 validation for IBANs) alongside context-gated detection, negative keyword suppression, and the graceful fallback path when the Rust backend is unavailable.

The test toolchain uses `pytest` (Pytest Contributors, 2026) with multiple extensions: `pytest-cov` for coverage measurement, `pytest-mock` for patching, `pytest-randomly` for execution order randomisation to surface hidden state dependencies, and `pytest-sugar` for improved output formatting.

6.2.2 Integration Testing

Each module underwent integration testing against its target systems. Tests for the Ansible module verified SSH connectivity and command execution against the production control node, along with output parsing. The Splunk and Satellite modules were vali-

dated in a similar fashion: Splunk tests covered both the development instance and a subset of production instances, exercising REST API access, token authentication, and response handling, whilst Satellite tests targeted the staging server before production deployment. Error conditions were simulated, including network timeouts, authentication failures, and malformed responses, to verify that the system reports the failing host, HTTP status code, and endpoint URL instead of exposing raw stack traces.

6.2.3 User Acceptance Testing

Analysts from the security operations team conducted acceptance testing by exercising the dashboard and search commands during their normal operational workflows. Their feedback addressed functionality, usability, and performance. Confusing output formats, unintuitive parameter names, and performance problems at realistic query volumes all emerged through operational use.

Issues identified during user acceptance testing were addressed iteratively; for instance, ambiguous parameters were renamed and inline error descriptions added after analysts reported confusion with raw exception output.

6.2.4 Testing Limitations

The testing approach carries four acknowledged limitations. The automated suite is dominated by unit tests and mocked component-level checks; no end-to-end tests verify behaviour against live systems. Whilst these tests confirm that individual actions parse expected output formats correctly, they cannot detect changes in external API responses or infrastructure configuration that would cause failures in production.

Coverage spans all four project components, though depth varies with component complexity. The core USO API has extensive test coverage across all six integration modules and the Graph library, and the PII detection tool covers all 14 detectors with dedicated false positive regression cases. The compliance report pipeline (`uso_TA_compliance`) covers entry-point dispatch, input validation and retry logic, pipeline orchestration, SSH verification, and notification formatting, with external dependencies (SSH File Transfer Protocol (SFTP) uploads, Mattermost webhooks, file system operations) isolated through mocking. The diagnostics subsystem's test suite covers all modules, spanning the dispatcher and entry point, file monitoring, KV Store validation, job diagnostics, historical baseline comparison, results processing, and notification formatting.

This allocation reflects deliberate prioritisation. The USO API contains the most complex parsing and dispatch logic across six integration modules, making it the highest-value target for regression testing. The compliance report pipeline's heavy dependence on external I/O (file system operations, SFTP transfers, webhook calls) is addressed through mocking, which isolates the pipeline logic from infrastructure variability. The diagnostics subsystem covers its full runtime path through mocked

unit tests; the primary opportunity for further coverage is integration-level validation against live Splunk summary indexes.

Development introduced the test suite late in the cycle, after the core implementation was substantially complete. Tests were written retrospectively rather than driving design decisions through test-driven development; consequently, certain architectural choices did not account for testability, and some internal interfaces are less amenable to isolated testing than they might otherwise be.

6.3 Feature Completeness

6.3.1 Requirements Fulfilment

Table 6.5 maps each functional requirement from Chapter 3 to its implementation status and evidence of completion.

The implementation satisfies all functional requirements identified during requirements analysis. Among the non-functional requirements, NFR1, NFR3, NFR5, and NFR6 are achieved, NFR2 is achieved with documented residual risks, and NFR4 is partially achieved because the 2× growth target has not been tested empirically (Section 6.5.2). The supplementary security and privacy requirements are also evaluated explicitly and are achieved through the implemented credential, permission, and PII-processing controls. Three requirements were introduced during the development cycle as operational priorities emerged: FR6 (Identity and Access Visibility), FR7 (PII Exposure Detection), and FR9 (Infrastructure Health Diagnostics).

6.3.2 Evaluation of Non-Functional, Security, and Privacy Requirements

Whilst Table 6.5 confirms that the performance, reliability, maintainability, and usability requirements are achieved, NFR2 (Security) carries documented residual risks and NFR4 (Scalability) remains only partially achieved. NFR1 and NFR4 receive dedicated analysis in subsequent sections. The remaining requirements, including the supplementary security and privacy requirements, are evaluated below with cross-references to the architectural and implementation evidence that supports each claim.

Security requirements are addressed through architecture and implementation controls rather than through ad hoc operational practice. The design specifies encrypted credential storage and least-privilege boundaries (Section 4.8), with dedicated treatment of credential protection (Section 4.8.3) and constrained role capabilities (Section 4.8.5). Implementation evidence further shows migration to token-only remote authentication, removing inline username and password paths and reducing credential exposure in routine workflows (Section 5.9.1). These measures satisfy the core requirement to avoid plaintext credential storage and to operate within permission boundaries. They do not, however, constitute a formal adversarial security assessment, and selected internal Transport Layer Security (TLS) paths still rely on an accepted residual risk where peer verification is disabled.

Table 6.5: *Requirements Fulfilment Assessment*

Requirement	Status	Evidence	Section
FR1: Infrastructure Querying	Achieved	14 Ansible actions	5.3.1
FR2: Splunk Integration	Achieved	11 Splunk actions	5.3.2
FR3: Database Monitoring	Achieved	8 Extractor actions	5.3.4
FR4: Vulnerability Management	Achieved	9 Satellite actions	5.3.3
FR5: Email Pipeline Monitoring	Achieved	4 Mail actions	5.3.6
FR6: Identity and Access	Achieved	11 Identity actions querying local lookups	3.2.6, 5.3.7
FR7: PII Exposure Detection	Achieved	14 detectors, 3 modes, scheduled scanning	3.2.7, 5.7
FR8: Unified Dashboard	Achieved	16-tab dashboard	5.4
FR9: Infrastructure Diagnostics	Achieved	Three diagnostic modules	4.6.1
NFR1: Performance	Achieved	Most `| usoapi` queries complete well below 20s in routine use	6.5.1
NFR2: Security	Achieved with residual risks	Token-based auth, encrypted storage, audit logging; selected internal TLS paths still rely on accepted residual risk	4.8
NFR3: Reliability	Achieved	99.67% search completion rate across approximately 29,000 invocations (30-day window); errors returned as structured rows by design	6.3.2
NFR4: Scalability	Partially Achieved	Batch operations supported; 2× scale untested	6.5.2
NFR5: Maintainability	Achieved	Modular architecture, registry pattern	5.2.2
NFR6: Usability	Achieved	Consistent syntax, built-in help, exploratory pilot SUS mean 93.3 ($N = 3$, census)	6.6.7
SPR1: Data Minimisation	Achieved	Stateless PII processing with no intentional local persistence of raw matches	5.7, 6.6.5
SPR2: Credential Rotation	Achieved	Stored tokens can be revoked or replaced through the credential store and Token Manager tooling	5.9.1
SPR3: Access Segregation	Achieved	Capability-based access checks and least-privilege service identities	4.8.5, 4.8

SPR1 (Data Minimisation) is satisfied by the PII subsystem's stateless processing model. The detector scans, annotates, and redacts event content in memory without introducing an intentional local persistence path for raw matches, and the implementation further constrains processing through per-field size limits and streaming output (Section 5.7). Evaluation of the subsystem therefore focuses on detection quality and throughput rather than on a separate disk-forensics experiment (Section 6.6.5).

SPR2 (Credential Rotation) is achieved. The implementation supports token deletion and replacement through Splunk's credential store and the Token Manager tooling, so administrators can revoke or rotate stored credentials without changing application code or exposing plaintext secrets in routine workflows (Section 5.9.1).

SPR3 (Access Segregation) is achieved through capability-based access control. Queries execute within the invoking user's Splunk permissions, sensitive remote operations rely on constrained service identities, and missing capabilities lead to denied execution rather than privilege escalation (Sections 4.8.5 and 4.8).

Both automation subsystems employ explicit retry and validation mechanisms. The compliance pipeline applies bounded exponential backoff and pre-upload data validation before transmission (Section 5.5.2), whilst the diagnostics dispatcher wraps module failures in configurable retry behaviour (Section 5.6.1). Together, these controls limit transient-failure impact and prevent silent propagation of incomplete outputs.

Production audit logs record over 28,000 completed searches out of approximately 29,000 total invocations over a 30-day observation window, yielding a 99.67% search completion rate that exceeds the 99% target specified in NFR3. The remaining outcomes comprise 77 cancellations and 17 failures at the Splunk scheduler level (timeouts, concurrency limits, or user-initiated cancellation), not framework-level crashes. At the application layer, the command framework returns errors as structured result rows with exit code zero by design, so a code-level defect would typically manifest as a malformed row rather than a search failure. This two-layer distinction separates platform-level search completion from application-level graceful degradation: the former is measured empirically, whilst the latter is an architectural property validated through the retry and error-handling mechanisms described above.

Maintainability is supported by a modular, registry-driven design that localises change impact. The core command and modules use registry patterns for discoverability and extension (Section 5.2.2), and shared abstractions are reused across dashboard and diagnostics components. The iterative refinements in Section 5.9 yield empirical evidence that this structure allowed targeted refactoring without destabilising unrelated functionality.

Usability is evidenced by a consistent command syntax and integrated guidance, supported by analyst feedback from production use. The command framework includes self-documenting help output with uniform parameter presentation (Section 5.2.4), whilst the requirements mandate a stable syntax across modules (Section 3.3.6). User acceptance testing confirmed that interface and naming issues were identified through iterative feedback and corrected during the deployment period

(Section 6.2.3). An exploratory usability pilot study conducted after two months of production use yielded a mean System Usability Scale (SUS) score of 93.3 ($N = 3$, census sample), placing the system in the "Excellent" to "Best Imaginable" range (Section 6.6.7).

6.4 Comparison with Alternatives

The decision to build a custom integration layer within Splunk, rather than adopting external SOAR platforms or middleware, was guided by the operational constraints of the security operations team. Table 6.6 compares the proposed framework against three primary alternatives based on the criteria identified during requirements analysis (Chapter 3).

Table 6.6: *Comparison with alternative platforms and integration strategies*

Criteria	Proposed	Splunk SOAR	MS Sentinel	Grafana
SPL-native	High (Direct)	Low (External)	N/A (Azure)	None
Maintenance	Low (Python)	High (Complex)	Medium (Cloud)	Medium
Security	High (JWT)	High (Vault)	High (Managed)	Low (Plugins)
Licence Cost	Included	High (Seat/Vol)	Volume-based	Low/OSS
Flexibility	High (Custom)	High (Playbooks)	Medium (Logic)	High (Visual)
Scalability	Medium (Linear)	High (Distributed)	High (Cloud)	High

The embedded approach eliminates the need for a separate platform, preserving the team's existing SPL skill set and avoiding the substantial licensing and maintenance overhead associated with enterprise SOAR solutions. Whilst platforms like Splunk SOAR or Microsoft Sentinel offer greater out-of-the-box scalability and playbook orchestration, the proposed framework's alignment with the team's "no-middleware" constraint (Section 3.5) made it the most viable choice for the target environment.

6.5 Performance Analysis

6.5.1 Response Time Characteristics

The non-functional requirement NFR1 specified that `| usoapi` command execution should remain within 20 seconds for typical interactive use. Observational data from production use confirms that typical queries stay well below this ceiling.

Ansible-based queries exhibit the highest latency because they traverse several network hops (search head to Ansible control node to target host). Simple status checks complete in 2–4 seconds, whilst disk usage queries that parse filesystem output require 4–6 seconds; the `find` action, which may scan large directory structures, can exceed the 10-second target for broad search patterns. By contrast, Splunk SDK queries involve

a single REST API call and return faster: instance information and health queries typically complete in 1–2 seconds, whilst KV store enumeration and search job history queries may take 3–5 seconds depending on the volume of results.

Dashboard panel loading benefits from the priority queue system (Section 5.4.3). Fast lookup queries complete first, allowing analysts to see partial results whilst longer-running queries execute. The staggered autorefresh intervals prevent search cluster congestion during periodic updates.

6.5.2 Scalability Observations

The current deployment monitors approximately 50 hosts and 20 Splunk instances. At this scale, sequential query execution yields acceptable performance. Extrapolating to larger deployments suggests that the 20-second target could be exceeded for substantially larger host groups, given the per-host overhead observed at the current scale.

The compliance report pipeline handles workloads of up to 3.2 million rows per report without performance degradation. File transformation and upload operations complete within seconds for typical report sizes, scaling to just over two minutes for the largest reports (Section 6.6.3).

6.6 Operational Impact

6.6.1 Workflow Comparison

Table 6.7 compares six operational scenarios that span daily monitoring, periodic reporting, and investigative tasks reflecting the team's routine workload. These scenarios capture three categories of improvement: consolidation of multi-tool workflows into single queries, full automation of previously manual pipelines such as compliance reporting, and introduction of monitoring for subsystems that were previously unobserved. To account for variability in network latency and infrastructure load, the post-deployment metrics include both the median (p_{50}) and 90th percentile (p_{90}) durations, derived from system logs over a 30-day observation window. Section 6.6.2 discusses the interface fragmentation that these consolidations address.

Process Decomposition: Daily Health Check

To validate the "Before" state estimates from Table 6.7, a process decomposition was performed for the Daily Health Check scenario. This task involves verifying the operational status of approximately 50 hosts and 20 Splunk instances. Table 6.8 breaks down the manual effort required before the implementation of the unified dashboard.

This decomposition confirms that the approximately 20-minute estimate for the pre-deployment state is evidence-based, accounting for multiple authentication steps, context switching between tools, and manual inspection of heterogeneous data formats.

Table 6.7: *Workflow comparison across representative operational scenarios*

Scenario	Before	After (Avg)	p50	p90	Reduction	Method[a]
Daily health check	~20 min	58 s	55 s	65 s	>95%	AA, SL
Vulnerability lookup	~3–5 min	24 s	22 s	31 s	>83%	AA, SL
Monthly compliance report	~1 hr	22 s[e]	13 s	105 s	100%	PD, SL
Anomaly detection	Hours–days	<3 min	N/A	N/A	N/A[b]	AA, PT
Database connectivity	~30+ min[f]	32 s	30 s	45 s	>98%	AA, SL
Email pipeline health	N/A	28 s	25 s	35 s	N/A[b]	PO, SL
Comparable daily subtotal[c]	**≥48–55 min**	**~5 min**	**–**	**–**	**>95%[d]**	

[a] AA = analyst attestation; PD = process decomposition; SL = system logs; PT = production telemetry; PO = production observation. Controlled timing experiments were not performed (Section 6.7.3).
[b] No pre-deployment baseline exists for anomaly detection or email monitoring.
[c] Covers health check, vulnerability lookup, and database connectivity. Excludes scenarios without pre-deployment baselines and the monthly compliance report (automated, not daily interactive).
[d] Estimated reduction derived from approximate time ranges, not controlled measurements.
[e] Average across 36 runs; duration varies significantly with report volume (Section 6.6.3).
[f] No proactive pre-deployment workflow existed; the cited duration estimates the systematic per-stanza checking that would have been required (Section 6.6.6).

Table 6.8: *Process decomposition: manual daily health check (Pre-deployment)*

Step	Description	Est. Time
1	Login to Ansible control node and inventory filter	1.5 min
2	Execute ad-hoc disk space checks across host groups	3.0 min
3	Manually review Ansible output for 50 hosts	4.0 min
4	Login to Satellite and filter for critical errata	4.0 min
5	Review separate dashboards for system resources and log health	4.0 min
6	Scan Extractor logs for script failures and data-load errors	4.0 min
Total		**20.5 min**

The "After" state reduces this to a single dashboard load, with the automated background queries completing in under one minute.

Time estimates in Table 6.7 derive from analyst attestation based on operational experience with the pre-existing workflow. The compliance report estimate of approximately one hour per monthly cycle was corroborated by process decomposition (N = 4 monthly cycles): executing several search queries, exporting each result set to CSV, running transformation scripts, uploading files via secure transfer, and verifying delivery. Pipeline execution data from the diagnostic index (36 production runs across seven report types) provides corroborating system-level timing. Some report types lacked saved searches, adding cognitive overhead from recalling the correct query syntax each month.

6.6.2 Context-Switching Reduction

Before the USO API deployment, a typical morning infrastructure check required switching between an SSH terminal on the Ansible host, several Splunk dashboards covering different operational concerns, and the Red Hat Satellite web User Interface

(UI) for vulnerability data. For some workflows, additional portals were needed for identity or email-pipeline data. For newer team members, part of the overhead lay in knowing which dashboard exposed a given metric and when a check required Ansible syntax rather than a saved search. Each context switch carried authentication overhead, navigation cost, and the cognitive burden of recalling tool-specific syntax such as Ansible ad-hoc command flags or Satellite API navigation paths.

The dashboard consolidates these data sources into a single tabbed interface within Splunk. An analyst's morning health check, previously distributed across multiple tools over approximately 20 minutes, now completes in roughly a minute on a single screen. This reduction from several operational views and external interfaces to one unified view addresses the tool fragmentation identified as a core problem in Chapter 1.

Consolidating multiple data sources into a single interface extends beyond individual lookup speed. During an investigation, an analyst can cross-reference a host's health status, outstanding vulnerability errata, pending reboot requirements, and database connectivity data within a single dashboard session, without switching tools or re-authenticating. This concurrent visibility allows analysts to correlate issues faster than sequentially consulting separate systems, each with its own query syntax and authentication overhead. The advantage grows with investigation complexity: the more sources an analyst needs to consult, the greater the cumulative overhead that a unified interface eliminates.

6.6.3 Compliance Pipeline Metrics

The compliance report pipeline was operational for roughly five months (November 2025 to April 2026) at the time of evaluation. During this period, the pipeline executed 36 runs across seven production report types. Table 6.9 summarises execution characteristics for each report type. Metrics were extracted from a dedicated diagnostic index that records structured events for each pipeline stage.

Table 6.9: *Compliance pipeline execution metrics by report type*

Report	Runs	Avg Rows	Avg Duration (s)	Steady-State Success Rate
Report A	5	3,017,000	104.5	100%
Report B	6	440,000	14.9	100%
Report C	5	350,000	12.8	100%
Report D	5	194,000	12.7	100%
Report E	5	141,000	6.0	100%
Report F	6	25,000	1.6	100%
Report G	4	58	1.1	100%

Table 6.9 reports run counts and average row volumes and durations across the full 36-run observation window, whereas the success-rate column refers only to the steady-state subset from January to April 2026.

The full observation window (November 2025 to April 2026) encompasses 36 pipeline runs across seven production report types, including the commissioning pe-

riod when report schedules and row-count thresholds were being established. From January 2026 onward, the pipeline operated at steady state with 30 runs; all seven report types achieved a 100% success rate across this period, confirming fully automated delivery. The validation mechanisms described in Section 5.5.2 were verified through the dedicated test infrastructure detailed in Section 5.5. These results confirm that the fault tolerance mechanisms function under production conditions.

Across a typical monthly cycle, the pipeline processes roughly 4.2 million rows. Execution durations range from 0.93 seconds for the smallest report (58 rows) to a peak of 135 seconds for the largest (approximately three million rows), confirming that processing time scales with data volume without exhibiting unexpected bottlenecks. No retry attempts were triggered during the steady-state period, which indicates stable network connectivity between the search heads and the partner upload server. Monthly event distribution confirms this trajectory: commissioning-phase failure and validation-block events, concentrated in November–December 2025, dropped to zero from January 2026 onward.

6.6.4 Diagnostics Subsystem Performance

The diagnostics subsystem, deployed in early February 2026, performed 358 diagnostic checks during the first 28-day production observation window, defined as the first 28 days after the earliest production diagnostics event, across six data stores belonging to an internal inventory management application. Once the schedules settled, the configured steady-state cadence was 12 checks per day, comprising one job health check and one file or KV Store validation per data store. Table 6.10 breaks down check activity by module.

Table 6.10: *Diagnostics subsystem check distribution (first four weeks of production operation)*

Module	Checks	Proportion
Job health	180	50.3%
File monitoring	116	32.4%
KV Store validation	62	17.3%
Total	**358**	100%

Of 358 checks, 357 passed and one failed. The single failure was an intentional negative test targeting a nonexistent KV Store collection to verify that the system correctly detects and reports errors; excluding this test, the operational pass rate was 100%. The historical comparison engine applies to file and KV Store diagnostics using a rolling 10-run lookback with a minimum of three historical records before activating; job diagnostics use static warning thresholds rather than learned baselines. During the first 28-day observation window, baseline-enabled comparisons covered 161 checks across six monitored entities, recording row-count deviations between 0.0% and 0.40% and size deviations between 0.0% and 0.42%. These figures, drawn from the `uso_diagnostics` summary index, fall well within the configured 20% deviation threshold and indicate stable data volumes across the monitored estate.

Across the full available window, 484 baseline-enabled comparisons continued to yield zero historical anomalies, with maximum row and size deviations of 1.61% and 0.54% respectively, which indicates that this low-noise behaviour persisted beyond the initial 28-day window. No genuine anomalies occurred during either observation period, so the system's ability to detect real failures remains unverified in production.

Should a genuine anomaly occur, the automated checks would surface it on the next scheduled run, which means it would normally become visible within minutes rather than after hours of ad-hoc manual review. This remains a design property rather than an empirically measured outcome, given the absence of real failures during the observation period.

Static Thresholds versus Historical Baselines

The diagnostics subsystem supports static threshold checks (`file_max_rows`, `file_max_size_bytes`, `file_age_max_seconds`, `job_max_duration_seconds`) configurable per saved search alongside the historical comparison engine. The two approaches address different failure modes and impose different operational costs. Table 6.11 contrasts them along the dimensions encountered during deployment.

Table 6.11: *Static thresholds versus historical baselines: qualitative comparison*

Dimension	Static threshold	Historical baseline
Calibration	Per-metric absolute value chosen by an operator	Derived from a rolling 10-run window with a configurable deviation percentage
Cold start	Effective from the first execution	Requires at least three historical records before activating
Drift handling	Absolute bound; manual recalibration when normal data shape evolves	Rolling window absorbs gradual drift; may mask slow degradation if change is below the deviation threshold
Sudden anomaly	Detected if the value crosses the bound	Detected through deviation percentage against the recent average
Per-resource sensitivity	Single value applies uniformly unless replicated per saved search	Each monitored entity (host, file path, or collection) computes its own baseline
Configuration burden	Recurring as the monitored estate or its data shape changes	One-time per saved search

This trade-off reflects findings in operational-monitoring research. D. Liu et al. (2015) report that hand-tuning detector parameters and thresholds for production key-performance indicators consumes more than ten days of operator effort per service in a large-scale internet deployment, identifying the calibration burden as the principal obstacle to wider adoption of static-threshold alerting. The historical baseline engine sidesteps this cost by deriving the reference from the metric's own recent history. Two

limitations apply: Katz et al. (2023) characterise the cold-start problem for newly created monitored objects, where insufficient historical data delays effective anomaly detection; and J. Liu et al. (2023) show that rolling baselines absorb gradual drift unless paired with explicit change-point detection. The current implementation addresses cold start through the `MIN_HISTORICAL_RECORDS` guard (Section 5.6.3) but does not include drift detection.

During the four-week observation window, both approaches would have produced identical alert counts: zero. Row-count deviations remained between 0.0% and 0.4%, far below either the 20% deviation threshold or any reasonable absolute bound. The qualitative difference emerges in calibration cost over time: activating the baseline engine for a new monitored resource requires only the historical-comparison flag and an identifier field, whilst static thresholds require operators to estimate appropriate maximum row counts, file sizes, and age limits for each new monitored artefact and to revise these estimates when the underlying data shape changes. The Data Health tab in the dashboard surfaces historical anomalies separately from current failures, allowing operators to distinguish baseline deviations from immediate data-delivery problems at a glance. A controlled comparison through synthetic anomaly injection was not performed within the project timeline; Section 6.8 discusses the resulting evidential constraints on RQ3.

6.6.5 PII Detection Metrics

The privacy-monitoring extension scans production Splunk indexes on a 30-minute schedule, applying 14 detector types against event data to identify exposed personal information. Table 6.12 summarises throughput benchmarks measured on the `uso_identity` index, comparing the pure Python implementation against the Rust hybrid backend.

Table 6.12: *PII detection throughput: Python baseline vs Rust hybrid backend*

Events	Python (s)	Rust Hybrid (s)	Speedup
10,000	11.1	4.8	2.3×
50,000	42.8	11.1	3.9×
100,000	73.9	18.7	4.0×

Peak observed throughput reaches approximately 5,400 events per second with the Rust backend, compared to roughly 1,400 events per second in the pure Python implementation. These benchmarks were measured on a production Splunk Search Head cluster node equipped with 24 vCPUs and 46 GB of RAM. Throughput was averaged over $N = 10$ runs for each data point, with a measured standard deviation of less than 3.5% across all trials. The test dataset was sampled from the `uso_identity` index with a typical event size of 800 bytes and a PII hit rate of approximately 15%. The speedup increases with volume because the Rust process amortises its startup cost and benefits from better cache locality during sustained processing. At 100,000 events the Rust

backend completes in under 19 seconds, well within acceptable limits for scheduled background scanning.

Detection coverage spans the fourteen PII types catalogued in Section 4.7.2, six targeting Portuguese-specific identifiers and eight handling universal patterns. Each detector applies the algorithmic and context-gated validation described in Section 4.7.2.

False positive management required over ten production tuning rounds against more than 400 indexes. Context-gated detection, where six high-noise detectors fire only when supporting keywords appear within a 50-character window, proved the most effective mitigation strategy. Negative keyword lists per detector suppress common false matches: UUIDs misidentified as credit cards, build numbers matching tax identifier patterns, and session identifiers resembling phone numbers. The three-tier confidence scoring system (low, medium, high) with context-based promotion and suppression gives analysts tunable sensitivity through the `min_confidence` parameter.

Detection Accuracy

The PII detection subsystem underwent significant refactoring to improve precision without sacrificing performance, following established patterns for improving software design (Fowler, 1999). To evaluate these improvements, a synthetic evaluation corpus of 25 labelled cases was constructed: 15 positive cases embedding known PII instances across all 14 detector types (18 gold-standard spans), and 10 negative cases designed to test false-positive patterns (timestamps matching tax identifiers, localhost addresses, placeholder phone numbers) and validator edge cases (invalid NIF checksums, non-standard date formats). The evaluation uses exact span-and-type matching: a detection counts as a true positive only when the predicted span boundaries and PII type match the gold label precisely.

Table 6.13 presents per-detector precision, recall, and F1-score on this corpus. The email detector recorded one false positive on an internal-domain address (`admin@internal.corp.example.com`), exposing a limitation in the domain rejection logic that checks for exact `.internal.` matches but not subdomain patterns. All other detectors achieved perfect precision and recall, with validators correctly rejecting invalid NIF checksums, non-standard date formats, and placeholder telephone numbers. The Rust and Python backends produced identical outputs across all 25 parity test cases, confirming behavioural equivalence between the two execution paths.

These results establish a correctness baseline for the detector implementations but carry important caveats. The corpus is small and synthetic, designed to exercise each detector's core pattern and false-positive guards rather than to characterise performance on the heterogeneous enterprise data described in Section 5.9.5. Production deployment against over 400 indexes revealed false-positive challenges that the synthetic corpus does not fully represent, particularly for detectors operating without context gating on unstructured log data. A sample-based review protocol has been developed for estimating precision and recall on real operational data; executing this protocol against production indexes remains as future work (Section 7.3). Appendix E presents the complete evaluation corpus with input texts and expected detections.

Table 6.13: *Per-detector precision, recall, and F1-score on the synthetic evaluation corpus ($n = 25$ cases, 18 gold spans)*

Detector	Scope	TP	FP	FN	Precision	Recall	F1
Email	Universal	2	1	0	0.67	1.00	0.80
Phone	PT	1	0	0	1.00	1.00	1.00
NIF	PT	1	0	0	1.00	1.00	1.00
Credentials	Universal	2	0	0	1.00	1.00	1.00
Address	PT	1	0	0	1.00	1.00	1.00
Structured Name	Universal	1	0	0	1.00	1.00	1.00
Credit Card	Universal	2	0	0	1.00	1.00	1.00
IBAN	PT/EU	2	0	0	1.00	1.00	1.00
NISS	PT	1	0	0	1.00	1.00	1.00
IPv4	Universal	1	0	0	1.00	1.00	1.00
Citizen Card	PT	1	0	0	1.00	1.00	1.00
Vehicle Plate	PT	1	0	0	1.00	1.00	1.00
Date of Birth	Universal	1	0	0	1.00	1.00	1.00
JWT Bearer	Universal	1	0	0	1.00	1.00	1.00
Micro Total		**18**	**1**	**0**	**0.95**	**1.00**	**0.97**

6.6.6 Visibility Improvements

The dashboard also surfaces data that was previously inaccessible or difficult to obtain. Pending reboot status, which requires SSH access to each host, is now visible at a glance, as are Splunk queue depths useful for diagnosing ingestion problems.

The integration of vulnerability data from Red Hat Satellite into the Splunk environment (Section 5.3.3) allows analysts investigating alerts to check whether the affected host has outstanding errata, informing their assessment of the incident. Database connectivity monitoring through the Extractor module (Section 5.3.4) replaces a previously reactive workflow with proactive visibility: before deployment, no systematic verification existed across the full set of extraction stanzas, and analysts detected failures only after error noise accumulated in Splunk logs; the Extractor dashboard tab now surfaces reachability issues across all stanzas in a single query, reducing the gap between a collection failure and its detection.

The identity module consolidates user, group, and host information from six previously separate data sources into queries accessible from the SPL search bar. These sources comprise Active Directory, the access governance platform, the HR system, the Splunk accounts table, the internal infrastructure inventory, and a host resolver lookup. Before this integration, resolving a user's group memberships, access rights, and associated host information required navigating separate web portals, each with its own authentication and query syntax. An analyst can now run a single command to retrieve a user's full organisational context, including Splunk role mappings and managed host assignments, without leaving the SIEM interface.

Email pipeline health, previously unmonitored, became observable through Microsoft Graph API integration that reports mailbox folder statistics and message counts. The PII Exposure dashboard tab and its scheduled search present privacy risk data that was also previously unmonitored. Production indexes are scanned every 30 minutes for exposed personal information, with results aggregated into KPI panels showing detection counts by type, confidence distribution, and affected indexes.

6.6.7 Exploratory Usability Pilot Study

To supplement the operational evidence presented in the preceding sections, an exploratory usability pilot study was conducted using the SUS (Brooke, 1996), a standardised 10-item questionnaire widely adopted for usability evaluation (reproduced in Annex L). The study adopted a census-sampling strategy: the Splunk infrastructure core team has four members, and the author was excluded to avoid self-assessment bias, leaving three eligible participants ($N = 3$, representing the complete eligible population). All three responded, yielding a 100% response rate. The questionnaire was administered after approximately two months of regular production use, during which each analyst incorporated the dashboard and `| usoapi` command into their daily workflows.

Table 6.14 presents individual SUS scores, adjective ratings per Bangor et al. (2009), and group statistics.

Table 6.14: *System Usability Scale results* ($N = 3$)

Participant	SUS Score	Adjective Rating
P1	85.0	Excellent (A)
P2	95.0	Best Imaginable (A+)
P3	100.0	Best Imaginable (A+)
Mean	**93.3**	Best Imaginable (A+)
SD	**7.6**	
Range	**85.0–100.0**	
Industry benchmark	**68**	(Sauro, 2011)

The mean score of 93.3 places the system in the "Best Imaginable" category on Bangor et al. (2009)'s adjective scale. For reference, the cross-study industry average is 68 (Sauro, 2011), though direct comparison is limited given the small sample size. Even the lowest individual score (85.0) qualifies as "Excellent." All ten questionnaire items received uniformly high ratings, with one exception: the learnability item ("I think that most people would learn to use this system very quickly"), where two participants scored 4 ("Agree") rather than 5. This pattern is consistent with the system's reliance on SPL syntax and infrastructure domain knowledge, which present an initial learning curve for analysts unfamiliar with custom search commands or the underlying managed systems.

As an exploratory pilot study, these results characterise the perceptions of the complete eligible expert population rather than a statistical sample drawn from a larger pool. Tullis et al. (2004) recommend 8–12 participants for stable SUS estimates, a threshold that exceeds the total team size and places conventional statistical generalisation out of reach. The 100% response rate from the constrained population nevertheless provides stronger indicative evidence than a partial response from a larger cohort would, because it eliminates non-response bias entirely (Runeson et al., 2009). The scores are best interpreted as triangulated evidence alongside the operational metrics and workflow comparisons presented in the preceding sections, consistent with the single-case DSR evaluation tradition (Wieringa, 2014).

6.7 Limitations and Constraints

6.7.1 Technical Limitations

Several technical boundaries constrain the current implementation.

Scale Constraints The scalability characteristics described in Section 6.5.2 constrain the current deployment to its target scale. The extractor module already addresses sequential execution through `ThreadPoolExecutor`-based parallelism for database connectivity tests, demonstrating that the same registry-based architecture can support concurrent execution with $O(1)$ amortised latency per host. Extending this pattern to the Ansible and Splunk modules is identified as a near-term enhancement in Section 7.3: the architectural prerequisites (independent per-host queries, stateless action execution) are already satisfied.

Platform Dependency The solution is tightly coupled to Splunk Enterprise. The custom search command framework, credential storage mechanism, dashboard XML format, and summary index patterns all depend on features native to Splunk (Section 4.2.1). Migration to an alternative SIEM platform would require substantial reimplementation, not simple adaptation.

Network Assumptions The implementation assumes reliable network connectivity between the Splunk search heads and target systems. Intermittent connectivity issues, whilst handled through retry logic in the compliance report pipeline, can still cause query failures for interactive dashboard use. Environments with restrictive firewall policies or unreliable network segments would require connection pooling, local caching, or circuit-breaker[1] patterns.

[1] A circuit-breaker is a software resilience pattern that temporarily halts requests to a failing service to prevent cascading failures (Nygard, 2018).

SSH Host Key Policy The Ansible executor establishes SSH connections using a TOFU policy (`StrictHostKeyChecking=accept-new`). Under this configuration, the system accepts an unknown host key on first connection and stores it locally; subsequent connections reject any key change, which detects potential man-in-the-middle attacks against previously contacted hosts. This approach trades initial trust against operational convenience: full host key pre-distribution would require a key management infrastructure beyond the project's scope. Mitigating factors include the restricted network environment (all targets reside on private subnets with controlled SSH access) and the limited set of approximately 50 managed hosts. Section 7.3 identifies migration to pre-provisioned host key verification as a future enhancement.

Rate Limiting The USO API does not implement rate limiting on query execution. In the current deployment, access is restricted to authenticated analysts within a small team, which bounds the risk of resource exhaustion. Scaling to a broader user base without rate controls could allow excessive concurrent queries to degrade search head performance.

Retry Backoff Jitter The compliance report pipeline's exponential backoff strategy uses a deterministic 2^n delay (Section 5.5.2) without random jitter. If concurrent pipeline executions retry after a shared upstream failure, their retry intervals align, which could amplify load on the upload endpoint.

6.7.2 Organisational Limitations

Change Management Overhead Production deployment required working through the organisation's change management process, which imposes scheduling constraints on updates. This made iterative deployment cycles longer than ideal for an agile development approach.

Credential Provisioning Overhead Adding new monitored Splunk hosts requires two manual steps: generating a bearer token for each target instance, then registering it in the credential store through the token manager CLI. This sequential process creates a per-host operational cost that accumulates as the monitored environment grows. Tokens configured with expiration periods introduce additional recurring regeneration and re-registration labour. Automating token generation and registration would reduce this friction but lies outside the project's scope.

6.7.3 Methodological Limitations

Informal Requirements Gathering Requirements emerged through ongoing conversations instead of formal elicitation techniques. This approach worked well given the close collaboration with stakeholders, but it means the requirements documentation reflects retrospective analysis rather than upfront specification.

Limited Formal Testing As discussed in Section 6.2.4, the automated test suite was introduced late in the development cycle and covers only unit-level behaviour through mocked dependencies. Integration and end-to-end testing against live systems was performed manually.

Formal security assessment of the USO API itself, such as penetration testing or structured code audit, was outside the scope of this project. Section 7.3 identifies thorough security analysis as a direction for future work.

A further limitation is the absence of formal peer review by domain experts external to the project. The architectural decisions and design trade-offs were validated through supervisor feedback and operational deployment, but a structured expert review by senior security engineers (such as the Likert-scale protocol described by Wohlin et al. (2012)) would strengthen confidence in the solution's design quality and applicability to other contexts.

The exploratory usability pilot (Section 6.6.7) is constrained by the small participant pool. Tullis et al. (2004) recommend 8–12 respondents for stable SUS estimates; the team size of four imposes a hard ceiling of three eligible participants. However, the 100% response rate from the complete eligible population (census sample) yields stronger indicative evidence than a partial response from a larger cohort. The resulting scores are treated as exploratory rather than statistically generalisable.

Beyond testing methodology, security operations work constrains how precisely efficiency gains can be measured. Most analyst tasks (host investigations, vulnerability triage, incident response) follow context-dependent paths that vary with the nature and severity of each case. Unlike the monthly compliance report pipeline, whose fixed sequence of steps permits direct before-and-after comparison, interactive workflows resist averaging because no two investigations traverse the same sequence of tools or require the same depth of analysis. Real incidents cannot be reproduced under controlled conditions, which further limits the feasibility of repeated-measures experimental designs.

6.8 Answering the Research Questions

This section synthesises the evidence presented in the preceding sections to address each research question from Section 1.3.

RQ1: Unified Query Interface and Context-Switching The workflow comparison in Table 6.7 shows that representative tasks previously requiring several operational views—most commonly an SSH terminal, several Splunk dashboards, the Satellite web UI, and additional portals for selected workflows—now complete through a single SPL query or dashboard tab. For the daily health check, analysts estimate the workflow reduced from approximately 20 minutes across multiple interfaces to approximately one minute on a single screen (Section 6.6.2). The identity module consolidates six previously separate data sources into SPL-accessible queries (Section 6.6.6), removing the need to navigate distinct web portals for user, group, and host information.

Table 6.15: *Research question evidence synthesis*

RQ	Evidence summary	Main finding	Key limitation	Answer status
RQ1	Workflow comparison (6 scenarios), process decomposition, context-switching analysis, SUS pilot ($N = 3$, census)	Daily health check reduced from ~20 min across multiple interfaces to ~1 min; mean SUS 93.3	Analyst attestation; no pre-deployment timed trials	Supported
RQ2	Browser-observed time to first usable active-tab content, priority queue telemetry	First usable active-tab content in ~10 s versus >120 s before optimisation	Informal browser measurement; uncontrolled network conditions	Supported
RQ3	358 diagnostic checks (initial 28-day window), 484 baseline-enabled comparisons (full window), zero historical anomalies	Stable low-noise baseline operation; deviations well below 20% threshold; lower calibration burden than static approach	No genuine anomalies observed; no controlled comparison performed	Qualified

Subject to the methodological constraints noted in Section 6.1, the current evidence satisfies the success criterion defined in Section 1.3. An exploratory usability pilot study using the SUS yielded a mean score of 93.3 from the complete eligible expert population ($N = 3$, census sample; Section 6.6.7), corroborating the operational evidence that the unified interface is well received by the target user population. Appendix C.2 presents a concrete invocation with anonymised representative output, illustrating how a single SPL expression replaces manual per-host inspection.

RQ2: First Usable Dashboard Content Production observations indicate that time to first usable active-tab dashboard content decreased from over 120 seconds (all panels loading simultaneously) to approximately 10 seconds in routine use after implementing the search priority queue (Section 5.4.3) and tab priority controller (Section 5.4.3). These measurements were obtained through browser developer tools during routine production use, not through systematic benchmarking with controlled network conditions.

These observations support the success criterion defined in Section 1.3, though load times vary with network latency, Splunk cluster load, and concurrent user count. A controlled evaluation would repeat measurements under varying conditions and report confidence intervals.

RQ3: Historical-Baseline Anomaly Detection Section 6.6.4 reports that 357 of 358 checks passed in the first 28-day observation window, with the sole failure corresponding to the intentional negative test described earlier in the section. Within the same window, row-count deviations remained between 0.0 and 0.40%, and size deviations remained between 0.0 and 0.42%, both well below the configured 20% deviation threshold. Across the full available window, 484 baseline-enabled comparisons yielded zero historical anomalies, with maximum deviations of 1.61% (rows) and 0.54% (size).

The evidence supports two specific claims. First, the historical comparison engine operated in a stable, low-noise manner throughout the observation period, accommodating natural data-volume variance without generating spurious alerts—the success criterion from Section 1.3. Second, activating the baseline engine for a new monitored resource requires only a single configuration flag and an identifier field, suggesting lower calibration effort than estimating and maintaining per-metric static bounds, which is consistent with the calibration-burden findings reported by D. Liu et al. (2015).

The evidence does not support a claim that baseline alerting outperforms static thresholds in detection sensitivity: no genuine anomalies occurred during either observation window, and no controlled comparison through synthetic anomaly injection was performed. The cold-start (Katz et al., 2023) and drift (J. Liu et al., 2023) limitations of rolling baselines therefore remain uncharacterised in this deployment. The qualitative comparison in Table 6.11 contrasts the two approaches along the dimensions encountered during deployment. The answer to RQ3 is qualified: stable low-noise operation and lower calibration burden are supported, but demonstrated detection improvement over static thresholds requires further investigation.

6.9 Threats to Validity

This section maps the methodological constraints discussed in Section 6.7.3 to the standard validity categories established by Wohlin et al. (2012) and Runeson et al. (2009).

Internal Validity. Internal validity concerns whether the observed effects can be attributed to the intervention rather than to confounding factors. The absence of a controlled baseline measurement before deployment prevents establishing a rigorous causal link between the framework and the reported efficiency gains. Workflow time estimates derive from analyst attestation rather than timed trials, introducing recall bias. The iterative development process, during which requirements, implementation, and evaluation overlapped, may further confound the attribution of specific improvements to specific design decisions. These threats are partially mitigated by the five-month observation window and the consistency of telemetry data across the entire deployment period.

External Validity. External validity concerns the extent to which findings generalise beyond the study context. All evidence originates from a single organisational unit (the organisation's security operations team) with a specific infrastructure profile (approximately 50 managed hosts, 20 Splunk instances). Organisations with substantially different scales, tool ecosystems, or operational cultures may experience different outcomes. The registry-dispatch architecture is not inherently Splunk-specific, but its portability to other SIEM platforms remains unvalidated. Replication in a second organisational context would strengthen external validity.

Construct Validity. Construct validity concerns whether the metrics and instruments measure what they claim to measure. The exploratory usability pilot ($N = 3$, census sample) employs a validated instrument (SUS), but the sample size precludes statistical generalisation. The "context-switching reduction" metric relies on workflow comparison rather than direct observation of analyst behaviour, and the "dashboard load time" metric was measured through browser developer tools during production use rather than through systematic benchmarking under controlled conditions. The diagnostics pass rate captures operational correctness but does not directly measure detection quality in terms of precision and recall against a ground-truth anomaly set.

Conclusion Validity. Conclusion validity concerns whether the relationship between intervention and outcome is statistically supported. The small population size ($N_{\text{eligible}} = 3$) and single-case deployment preclude statistical hypothesis testing. The quantitative evidence (approximately 29,000 invocations, 36 pipeline runs, 358 diagnostic checks in the initial 28-day observation window) provides operational confidence through volume and consistency rather than through inferential statistics. These constraints are inherent to applied DSR projects evaluated in production environments (Wieringa, 2014) and do not diminish the practical validity of the engineering contributions.

6.10 Lessons Learned

6.10.1 Technical Lessons

Command Type Selection in Splunk SDK The choice of `GeneratingCommand` over `StreamingCommand` (Table 5.2) proved essential for usability: streaming commands render results as raw events rather than structured table columns. Command chaining remains possible through Splunk's native `map` command, preserving composability without sacrificing output quality.

Non-Standard Return Code Semantics External tools sometimes assign non-standard meanings to return codes (Table 5.2). The broader lesson is that integration code must treat return code semantics as tool-specific instead of assuming Unix conventions; documenting such edge cases in action-level comments prevents repeated misdiagnosis.

Capability Requirements Discovery Splunk REST endpoints require particular capabilities that vary by endpoint and are often underdocumented (Table 5.2). The practical lesson is to record discovered capability requirements in deployment documentation immediately, preventing repeated troubleshooting when onboarding new monitored instances.

Action Composition for Email Pipeline The Mail module's evolution from independent integrations to shared library composition (Section 5.9.2) demonstrated that cross-module actions benefit from reusing existing client libraries, avoiding the duplication of authentication and connection handling. Bug fixes to the Graph client library now propagate automatically to all consumers, and adding a new email data source requires only a thin action wrapper around an existing library.

Graph API Authentication Patterns Microsoft Graph API authentication using client credentials required different handling than Splunk bearer tokens. Splunk tokens, once created with no expiration, remain valid indefinitely unless revoked. Graph access tokens expire after configurable periods, often one hour, requiring refresh logic that Splunk integrations never needed.

The GraphClient implementation caches tokens and checks for expiration before every API call. To avoid mid-load stalls, it refreshes tokens sixty seconds before they become invalid, rather than reacting to 401 errors. This preemptive strategy proved effective, as early versions only refreshed on failure, which caused multiple dashboard panels to hang simultaneously when a token expired.

Credential storage also differed: Splunk tokens are simple strings stored directly in the password field, whilst Graph credentials comprise three components (tenant ID, client ID, and client secret). Storing these as a JSON object in the password field works but requires parsing at retrieval time. The Graph Credential Manager CLI validates JSON structure on storage, preventing malformed credentials from causing cryptic errors during authentication attempts.

Performance-Driven Architectural Layering The PII detection tool's evolution from pure Python to a Rust hybrid illustrates when performance demands justify added architectural complexity. The initial Python implementation processed roughly 1,300 events per second, adequate for small ad-hoc queries but insufficient for scheduled scans across production indexes containing hundreds of thousands of events. Porting the compute-intensive regex scanning to Rust yielded a fourfold throughput improvement whilst preserving the Python wrapper for Splunk SDK integration. The IPC boundary between the two languages adds deployment and debugging complexity, but the graceful fallback mechanism ensures that the tool remains functional even when the Rust binary is absent. The broader lesson is that hybrid architectures become practical when a clear performance boundary separates the integration layer from the processing core.

False Positive Tuning at Scale Deploying PII detection against over 400 production indexes exposed a recurring challenge: patterns that performed well in isolation produced unacceptable false positive rates against heterogeneous enterprise data. Unstructured name detection, which scanned free text for capitalized name patterns, generated over 85% false positives and was disabled entirely in the initial version. Tax identifiers matched build numbers, credit card patterns matched UUIDs, and phone number regexes matched JSON array indices. Over ten tuning rounds, three mitigation strategies emerged as effective: context-gated detection, which requires supporting keywords near the match; negative keyword lists, which suppress known false-match patterns; and algorithmic validators such as Luhn and mod-11 checks, which reject structurally invalid matches. These strategies reduced false positives to operationally acceptable levels without sacrificing detection coverage. Following the major code rewrite (Section 5.9.5), a context-gated structured name detector was re-introduced: it fires only when a field-label prefix (e.g., `Nome=`, `name=`) precedes the match, restricting detection to structured key-value contexts where precision is adequate.

6.10.2 Process Lessons

Pre-Flight Connectivity Checks SSH connectivity to upload destinations failed when target hosts were missing from search heads' known hosts files. Each of the search head cluster members required manual SSH key verification. Adding pre-flight connectivity checks to the pipeline would surface such issues before processing begins, reducing troubleshooting time when failures occur late in the workflow.

Activation Requirements for Splunk Configuration New stanzas in `inputs.conf` require a Splunk restart to activate. During initial compliance report pipeline deployment, log monitoring entries appeared correct but produced no indexing because the service had not been restarted. The solution was to switch to direct SDK indexing via `index.submit()`, which requires no configuration changes and offers immediate visibility. Where configuration-based approaches are necessary, documenting restart requirements in deployment procedures prevents this class of issue.

Alert Classification Configuration Splunk classifies scheduled searches as either "Reports" or "Alerts" based on particular configuration settings. The `enableSched = 1` setting is required for alert action triggers; setting it to 0 causes Splunk to treat the search as a report, disabling the trigger mechanism entirely. This distinction is not immediately obvious from the web interface and required `savedsearches.conf` inspection to diagnose.

6.10.3 Development Setbacks

Beyond the implementation challenges described in Section 6.10.1, several development efforts consumed disproportionate time relative to their final contribution. These setbacks, whilst frustrating during development, revealed recurring patterns in platform behaviour and integration constraints.

Credential Architecture Iterations The path to the final credential architecture required four complete pivots: environment files, encrypted credential files, inline username-and-password pairs (added, then removed shortly afterwards), and finally secrets stored through Splunk's `storage/passwords` endpoint. Each pivot required new code and removal of the previous approach's configuration. The central difficulty was synchronising credentials safely across search head cluster members, a constraint that only became apparent during production deployment, not during local development. The key insight was that `storage/passwords` entries are encrypted at rest, replicated automatically across the search head cluster, and never exposed through search results or application logs. For remote Splunk access, bearer tokens also bind permissions to the dedicated service account that issued them rather than to the end users who invoke them.

Intermittent Chevron Attachment Failures The Splunk Web Framework's `TableView` class supports row expansion through a renderer pattern, but the implementation required several non-obvious steps. Clicking an expanded row must call `tableView.render()` to update the display; simply triggering search execution leaves the interface unchanged. This requirement is not emphasised in framework documentation and cost several hours of debugging.

Cache invalidation for expanded row data also required explicit handling. When the parent search refreshes, expanded content can become stale. The solution involved listening to the parent search's completion event and clearing cached row data, then collapsing any expanded rows to avoid displaying outdated information.

A far more persistent problem was the intermittent failure of expansion chevrons to appear at all. The root cause remained elusive: a race condition in Splunk's `addRowExpansionRenderer()` function causes chevrons to silently fail when the table DOM is not fully rendered at attachment time. Because the timing depends on search completion speed, network latency, and browser load, the defect appeared sporadically and resisted systematic reproduction.

Four successive fix strategies were attempted before the behaviour stabilised: timeout-based retries, manual chevron injection, interval polling, and finally a centralised utility library. The resulting `row_expansion_utils.js` module encapsulates all retry and lifecycle management logic, reducing what had been scattered, duplicated logic across twelve dashboard scripts to a single shared utility.

Log Source Freshness Grid The Logs Status tab required displaying freshness indicators for 40 log sources, each with heterogeneous query patterns (index-level, sourcetype-filtered, wildcard, and lookup-based) and different staleness thresholds. Six implementation strategies were attempted before one proved viable: direct index queries were too coarse; per-source `map` subsearches achieved correctness but took over two minutes when combined with the dashboard's other panels; a `tstats` consolidation approach (Splunk's accelerated statistics command) was reverted the same

day after performing worse than targeted queries; and two hybrid approaches each introduced brittle dependencies.

The final solution, a CSV-driven scheduled materialiser running on a five-minute cron cycle, decouples query logic from the dashboard entirely. Adding a new log source requires only a CSV row rather than dashboard modifications, at the cost of a brief staleness window acceptable for operational monitoring.

Dynamic Panel Width Controller An attempt to automatically adjust dashboard panel widths based on a naming convention ultimately produced zero net lines of code. The implementation used a `MutationObserver` (a browser API that fires callbacks when DOM elements change) on `document.body` with `childList`, `subtree`, and `attributes` options to persist widths across tab switches. This created a feedback loop: applying width changes triggered further observer callbacks, which re-applied widths, rendering the dashboard unusable through extreme lag. Replacement attempts using Splunk's token model and CSS `:has()` selectors proved equally fragile. The feature was deleted entirely once the tab priority controller (Section 5.4.3) subsumed its responsibilities.

Diagnostics Form Complexity The diagnostics alert action's configuration form initially presented all controls simultaneously: nearly 40 checkboxes, dropdowns, and text inputs on a single surface. As features were added, the form grew rapidly, and usability degraded to the point where operators struggled to locate relevant options. Restructuring the form using HTML5 `<details>`/`<summary>` elements, grouping related controls into collapsible sections, restored usability. Dark mode compatibility required CSS `:has()` selectors rather than JavaScript, as Splunk's alert action framework executes form HTML in a restricted context.

Graph Module Separation Shortly after creating a standalone Graph API module, it became clear that the module duplicated authentication logic already present in the mail module. Consolidating Graph functionality into the mail module (Section 5.9.2) eliminated this redundancy, but the abandoned files illustrate the tension between upfront architectural planning and the exploratory reality of integration development.

The recurring lesson across these setbacks is that validating platform-level behaviour early, not just application logic, reduces rework well before the first production deployment.

The scale constraint on sequential query execution and the testing gaps surfaced in this evaluation motivate concrete enhancements. Chapter 7 outlines contributions and future directions.

7 Conclusions and Future Work

Extending a team's primary SIEM to query heterogeneous infrastructure proved a practical strategy for reducing the context switching that slowed routine security operations. This chapter frames that conclusion through the project's contributions, limitations, and future directions.

7.1 Summary of Work

This project addressed operational inefficiency within the organisation's security operations team. The core deliverable is the USO API, a modular integration framework that extends Splunk's query capabilities to heterogeneous infrastructure systems.

The six artefacts outlined in Section 1.6 and validated in Table 6.5 have been deployed to production: the core USO API with its six integration modules, the unified dashboard, the diagnostics subsystem, the compliance report pipeline, the PII detection tool, and the token-based authentication scheme. The project centres on the first three as the main observability and security-operations contribution. The compliance report pipeline and the PII tool broaden that platform as secondary operational extensions. Together, they consolidate infrastructure visibility within the team's existing SIEM environment, so analysts can perform routine checks without leaving Splunk.

All functional requirements were met in full. Of the non-functional requirements, NFR1, NFR3, NFR5, and NFR6 were achieved, NFR2 was achieved with documented residual risks, and NFR4 was partially achieved because the projected 2× growth target has not been empirically validated (Section 6.3.2). The supplementary security and privacy requirements were also achieved through the implemented credential, permission, and PII-processing controls. Provisioning a synthetic 100-host test environment fell outside the project's operational scope, as the production infrastructure comprises a fixed set of managed hosts. The architectural prerequisites for scaling are nonetheless in place: actions execute independently per host with no shared state, and one module already queries several hosts concurrently, keeping total execution time roughly constant as the host count grows (Sections 5.3.4 and 6.5.2). Table 7.1 maps the core requirements to their implementation evidence and validation results.

Table 7.1: *Requirement-to-Evidence Traceability Matrix*

ID	Requirement	Implementation Evidence / Artefact	Validation
FR1–4	Infrastructure Integrations	42 actions across Ansible, Splunk, Database, and Satellite modules.	6.5
FR5	Email Pipeline Monitoring	Mail module; ~2,000 production executions.	5.3.6
FR6	Identity and Access Visibility	Identity module; eleven actions querying six directory-consolidated local lookups.	6.6.6
FR7	PII Exposure Detection	Accelerated pattern-scanning command; 100% recall on synthetic corpus.	6.6.5
FR8	Unified Dashboard	16-tab interface; SUS 93.3 from census population.	6.6.7
FR9	Health Diagnostics	Diagnostics subsystem; 358 automated checks (357 passed; one intentional negative test confirmed detection); 100% operational pass rate.	6.6.4
NFR1	Performance	`| usoapi` command response within the 20-second acceptance threshold.	6.5
NFR2	Security	Token-based auth; encrypted credential store; audit logging; documented residual risks for selected internal TLS paths.	4.8
NFR3	Reliability	99.67% search completion rate; structured error rows; retry and validation logic in automation subsystems.	6.3.2
NFR4	Scalability	Batch operations supported, but projected 2× growth remains untested empirically.	6.5.2
NFR5	Maintainability	Registry-driven modular architecture with targeted refactoring across subsystems.	6.3.2
NFR6	Usability	Consistent syntax, integrated help, and exploratory SUS evidence.	6.6.7
SPR1	Data minimisation	Stateless PII processing without intentional local persistence of raw matches.	5.7
SPR2	Credential rotation	Operational token deletion and replacement through the credential store and Token Manager tooling.	5.9.1
SPR3	Access segregation	Capability-based access checks and least-privilege service identities.	4.8

7.2 Contributions

7.2.1 Practical Contributions

The integration framework was deployed in a production telecommunications environment, confirming that the module-action pattern supports production-scale operation whilst preserving the analyst's familiar SPL query workflow. The framework accommodates both external systems queried over SSH and REST and internal data sources such as identity lookups consolidated from six directory sources.

The replicated credential management architecture supports distributed authentication. The structured token storage approach and Token Manager CLI together handle bearer token lifecycle across a search head cluster.

The sixteen-tab operational dashboard consolidates visibility from diverse infrastructure sources into a single interface. The row expansion pattern, priority queue system, and staggered refresh strategy reduced time to first usable active-tab content from over two minutes to approximately 10 seconds during routine production use.

The Splunk-native health-monitoring subsystem was built as an alert action, combining scheduled baseline comparison with static threshold checks where appropriate.

Alongside the main observability deliverables, the compliance report pipeline removes a deadline-sensitive manual reporting workflow through scheduled unattended execution. A secondary privacy-monitoring capability scans Splunk indexes for fourteen categories of personally identifiable information. Scheduled scanning and a dedicated dashboard tab surface these detections for analyst review.

7.2.2 Organisational Contributions

The project delivers direct operational benefits to the security operations team. Compliance reporting now runs as a scheduled unattended process, removing a deadline-sensitive manual workflow from the team's monthly routine (Section 6.6.3).

Documented lessons learned and deployment procedures support onboarding of new team members. The extensibility discussed in Section 4.2 allows new integrations to be added.

7.2.3 SIEM Integration Pattern

The registry-dispatch architecture presented in Section 4.3.1 and implemented in Section 5.2.2 demonstrates that a SIEM-embedded command interface can unify heterogeneous infrastructure queries behind a single invocation syntax, without external middleware. Six distinct access mechanisms converge through one `| usoapi module=X action=Y` interface, and iterative deployment confirmed that adding a new integration requires only a single class definition and one registration call.

Production use revealed trade-offs between portability and platform integration. Coupling to Splunk's SDK reduces portability, but grants access to the platform's

scheduling engine, alerting framework, dashboard renderer, and role-based access control—capabilities that would otherwise require substantial reimplementation. The module-action indirection brings auditability (every query appears in Splunk's search history) and composability (action output can be piped through aggregation and filtering operators). Portability to other SIEM platforms remains an open question, discussed further in Section 7.3.3.

Two further applications confirm the pattern's breadth beyond external integrations. The identity module applies registry-dispatch to local Splunk lookups, unifying six directory sources through the standard `| usoapi` interface without external API calls. The PII detection tool extends the approach to streaming commands, enriching events in the search pipeline with detection metadata. Together, these implementations show that the pattern accommodates both data-sourcing and data-enrichment workloads within a single abstraction.

7.2.4 Privacy Exposure Monitoring Pattern

The PII detection tool demonstrates a second design approach: a high-throughput pattern-scanning component for compute-intensive analysis within a SIEM pipeline. The command layer handles event processing and pipeline integration, whilst an accelerated scanning backend performs the regex-intensive analysis at throughputs that a scripting-only implementation could not sustain (Chapter 5). A graceful fallback path maintains availability when the accelerated backend is absent. This design is applicable to other detection or enrichment tasks where per-event processing cost would otherwise limit the volume of data a search-time implementation can handle. Applicability beyond PII detection, however, remains unvalidated.

7.2.5 Engineering Contribution and Limitation Matrix

Four engineering contributions are transferable beyond this particular deployment. The first three align directly with the project's primary objectives, whilst the fourth emerged as a secondary privacy-monitoring extension. Table 7.2 formalises these contributions by linking them to the research questions from Chapter 1, the validation evidence from Chapter 6, and their explicit scope limitations. Corresponding future directions are discussed in Section 7.3.

7.2.6 Engineering Reflections on Project Objectives

The following paragraphs map engineering evidence to the project objectives from Chapter 1 without repeating the detailed evaluation already presented in Chapter 6.

Objective 1: Unified Observability Across Heterogeneous Tools The registry-based dispatch mechanism abstracts six backend systems behind a consistent command syntax, and the sixteen-tab dashboard consolidates this information into a single opera-

Table 7.2: *Project Contribution and Limitation Matrix*

Contribution	Description & Justification	Validation Evidence	Scope Limitation	Future Work
SC1: Registry-Based Integration Pattern (RQ1)	Unifies heterogeneous infrastructure queries behind a single `usoapi` interface using the Strategy and Registry patterns. Decouples module implementation from dispatch logic.	Mean SUS 93.3 ($N = 3$); health-check duration reduction from approximately 20 minutes to approximately 1 minute.	Validated in a single SIEM (Splunk) with ≈50 hosts.	Parallel query execution; read–write operations.
SC2: Multi-Level Priority Architecture (RQ2)	Separates panel-level search weights from tab-level orchestration so that the active view becomes usable quickly even when slower searches continue in the background.	Production observations measured at approximately 10 seconds for first usable active-tab content, within the 20-second success criterion.	Measurements taken during routine use in one deployment; varies with cluster load.	Result caching to reduce repeated external API calls.
SC3: Historical Baseline Diagnostics (RQ3)	Uses statistical baseline comparison over lookback windows to support low-noise infrastructure diagnostics without manual threshold calibration.	100% operational pass rate across 358 checks (357 passed; one deliberate negative case, included to verify detection, behaved as expected); observed baseline deviations remained low throughout production operation.	No controlled comparison with static thresholds; rests on single intentional error and a low-noise observation window.	Diagnostics module expansion to additional subsystems.
SC4: High-Throughput PII Pattern Scanning (Secondary Extension)	High-throughput pattern-scanning pipeline for PII detection within the SIEM search path.	4× throughput improvement (5,400 vs 1,400 EPS) at scale.	Applicability beyond pattern-based PII detection unvalidated.	PII detector expansion and real-time stream detection.

tional interface. Production evidence in Chapter 6 shows that workflows once spread across multiple tools now complete through one dashboard session and one query syntax. The exploratory usability pilot (mean SUS 93.3) further indicates that this consolidation matched the target users' working practices.

Objective 2: Fault-Tolerant Automation of Compliance Reporting As a secondary operational extension, the compliance report pipeline converted a deadline-sensitive manual workflow into a scheduled unattended process. During the January–April 2026 steady-state period, it achieved a 100% delivery success rate across 30 runs; across the full 36-run window, failures were confined to the commissioning period before schedules and validation thresholds stabilised.

Objective 3: Proactive Infrastructure Health Monitoring The diagnostics subsystem replaced manual, ad-hoc health checks with automated baseline comparison across three diagnostic modules (Section 6.6.4). Production telemetry showed stable low-noise operation, with one deliberate negative test confirming failure reporting and no genuine anomalies observed during the evaluation window. Scheduled PII scanning complements this monitoring layer as a complementary privacy-monitoring extension by surfacing sensitive-data exposure across production indexes every thirty minutes.

Cross-Cutting Insight: Security Implications of Centralised Credential Management Centralised token storage creates a high-value target. Mitigations include Splunk's encrypted credential storage, cluster replication through the search head captain, suppression of full token values in logs, and audit logging of credential access. The migration to token-only remote authentication (Section 5.9.1) reduced credential exposure by eliminating inline username and password paths from routine workflows. This credential lifecycle architecture is classified as a practical rather than scientific contribution because it applies well-established token-based authentication patterns (Jones et al., 2015; Hardt, 2012) to the Splunk platform. The contribution integrates JWT-based bearer tokens with Splunk's encrypted credential storage and search head cluster replication; the novelty lies in their integration within a distributed search head environment, not in the authentication primitives themselves. A broader trust boundary analysis appears in Section 4.8.6.

7.3 Future Work and Recommendations

7.3.1 Short-Term Enhancements

Automated Testing The existing unit test suite covers action parsing, module dispatch, and notification delivery. Expanding this coverage to include integration tests against live infrastructure would strengthen validation further. As discussed in Section 6.2.4, the compliance report pipeline and the diagnostics subsystem each have dedicated test suites. End-to-end integration tests across all components would increase confidence in cross-subsystem behaviour.

Error Reporting and Configuration Validation Two analyst-facing improvements would reduce the troubleshooting burden. Error messages could include more diagnostic context, such as which specific network hop failed or which capability is missing, shortening the time from symptom to root cause. Complementing this, a pre-flight validation command could check connectivity to all configured targets and report issues before analysts encounter failures during normal use, shifting error discovery from runtime to pre-execution.

Diagnostics Module Expansion The diagnostics subsystem currently monitors search jobs, file system state, and KV Store health. Additional modules for network connectivity diagnostics and index capacity tracking would extend automated health coverage with minimal architectural changes, since each new module conforms to the same base class contract.

PII Detector Expansion The current fourteen detectors cover the most common Portuguese and international PII categories. Additional detectors for Portuguese health system numbers and non-Portuguese IBAN formats would broaden coverage. Machine-learning-based confidence calibration could replace the rule-based scoring with empirically trained thresholds, reducing false positive rates further. One identified limitation is the internal domain rejection logic for email detection: the current implementation checks for exact `.internal.` domain matches but does not catch subdomain patterns such as `internal.corp.example.com`, resulting in one false positive on the synthetic evaluation corpus. Fixing this pattern to handle both exact and subdomain internal addresses would improve precision on enterprise data.

7.3.2 Medium-Term Enhancements

Parallel Query Execution The extractor module already uses thread-pool-based concurrency to test database connectivity across collector hosts simultaneously (Section 5.3.4). Extending this pattern to Ansible and Splunk modules would reduce response times for larger infrastructure deployments, where sequential host traversal currently dominates execution time. Careful handling of result aggregation and error collection across heterogeneous data sources would be required, since distinct backends return results in varying formats and with separate failure modes. The architectural groundwork exists in the executor framework, which already separates dispatch from result handling.

Read-Write Operations All 57 actions in the current system are strictly read-only: modules query host metrics, retrieve health data, and fetch errata listings, but none modify target systems. Introducing write-capable actions would extend the platform from an observability tool to an operational control interface, where analysts could restart services on managed hosts or apply Satellite errata. Abstracted commands that

alter production infrastructure carry inherent risk. Any write path would therefore require confirmation prompts, role-based authorisation, and audit logging before execution proceeds. The existing module-action architecture already separates dispatch from execution, so write actions could be registered within the same pattern whilst enforcing stricter validation gates at the executor layer.

Result Caching Database connectivity results in the extractor module are already cached with a 30-minute Time To Live (TTL) using file-based JSON storage, with expiry-on-read semantics and a force-bypass option for on-demand refresh. Applying this cache pattern to other modules, particularly Satellite errata queries and Splunk health checks, would reduce load on external systems and improve dashboard responsiveness.

Additional Integrations The modular architecture supports adding new integration modules for other systems relevant to security operations, such as ticketing systems, vulnerability scanners, or cloud infrastructure platforms. The existing Graph client library (Section 5.3.5) is domain-agnostic and could support additional Microsoft 365 integrations without reimplementing authentication or API communication.

Real-Time PII Detection Extending PII scanning from scheduled batch searches to index-time or real-time streaming detection would identify sensitive data exposure as events arrive, not after a thirty-minute batch cycle. This shift introduces additional engineering constraints: real-time operation must keep pace with incoming data without slowing the indexing process, and memory usage must remain stable under continuous load. Integration with data masking workflows could then automate redaction of detected PII before it reaches dashboards or long-term storage, though such workflows require organisational policy alignment alongside technical implementation.

7.3.3 Long-Term Enhancements

Cross-Platform Portability Abstracting the Splunk-specific dependencies would make the framework portable to other SIEM platforms, though this would represent a substantial reimplementation effort.

PII Correlation and Data Lineage Cross-index PII correlation could link detections of the same individual's data across multiple indexes, supporting data lineage tracking and breach impact assessment. This would require entity resolution across detection results and integration with data governance frameworks.

7.3.4 Research Extensions

Integration Pattern Generalisation Building on the portability direction outlined in Section 7.3.3, the registry-based dispatch mechanism and credential management architecture could be generalised into a reference architecture for SIEM-infrastructure integration, applicable beyond the specific platforms used in this project.

Quantitative Impact Assessment The exploratory usability pilot conducted with the target user population (Section 6.6.7) yielded a mean SUS score of 93.3 ($N = 3$, census sample), though the constrained team size limits generalisability. Controlled measurement of analyst efficiency before and after dashboard deployment, ideally with a larger cohort of security analysts across comparable teams, would produce more rigorous quantitative evidence of operational impact and support the case for similar integration projects.

Extended Security Analysis The STRIDE threat model and NIST SP 800-53 control mapping presented in Sections 4.8.6 and 4.10 provide a structured foundation for security assessment. Further work could extend this analysis through formal penetration testing of the SSH relay chain and REST API endpoints, and through independent verification of the credential storage architecture under adversarial conditions.

7.4 Final Remarks

The central finding is that a SIEM-embedded integration layer, built on registry-dispatch and token-based authentication, can consolidate heterogeneous infrastructure monitoring without external middleware. In this organisational setting, observational production evidence supports that conclusion: daily health checks collapsed from a fragmented multi-tool routine to a single dashboard session, the compliance pipeline reached a 100% steady-state delivery rate across 30 runs after commissioning, and historical-baseline diagnostics maintained stable low-noise operation during the evaluation window.

Operational simplicity governed every design trade-off. Embedded commands avoided external service dependencies; Ansible orchestration abstracted SSH complexity; polling favoured reliability over real-time latency. These choices reflect a constraint that many security teams share: solutions must remain maintainable without dedicated development resources.

The framework and its companion tools now form part of the security operations team's daily operational toolkit. The modular architecture, backed by automated test suites across all four deliverables, allows teams to add integrations without modifying core components, though the strength of this claim remains bounded by observation in one production deployment rather than by multi-site replication.

Whilst the framework has been successfully validated within a production environment, its generalisation to other organisational contexts requires further replication. The architectural patterns, credential management approach, and evaluation methodology documented in this project report provide a foundation for such replication, but the specific findings remain bounded by the infrastructure profile, team composition, and operational culture of the deployment site. Extending validation to additional SOC teams with different SIEM platforms, infrastructure scales, and operational maturity levels would strengthen the evidence base for the integration pattern proposed in this work.

Bibliography

AbuseIPDB (July 2024). *AbuseIPDB Splunk Integration*. URL: https://splunkbase.splunk.com/app/7040 (visited on 2025-11).

AbuseIPDB (2025). *AbuseIPDB: IP address abuse database for system administrators and security professionals*. URL: https://www.abuseipdb.com/ (visited on 2025-11).

Agyepong, Enoch et al. (2020). "Challenges and Performance Metrics for Security Operations Center Analysts: A Systematic Review". In: *Journal of Cyber Security Technology* 4.3, pp. 125–152. DOI: 10.1080/23742917.2019.1698178.

Aho, Alfred V. and Margaret J. Corasick (June 1975). "Efficient String Matching: An Aid to Bibliographic Search". In: *Communications of the ACM* 18.6, pp. 333–340. DOI: 10.1145/360825.360855.

Allodi, Luca and Fabio Massacci (2014). "Comparing Vulnerability Severity and Exploits Using Case-Control Studies". In: *ACM Transactions on Information and System Security* 17.1, pp. 1–20. DOI: 10.1145/2630069.

Astral Software Inc. (2026). *Astral Python Development Toolchain: Ruff, Ty, and uv*. URL: https://docs.astral.sh/ (visited on 2026-04).

Ban, Tao et al. (Aug. 2021). "Combat Security Alert Fatigue with AI-Assisted Techniques". In: *Proceedings of the 14th Cyber Security Experimentation and Test Workshop (CSET '21)*. Association for Computing Machinery, pp. 9–16. DOI: 10.1145/3474718.3474723.

Banco de Portugal (2024). *International Bank Account Number (IBAN)*. URL: https://www.bportugal.pt/sites/default/files/anexos/documentos-relacionados/international_bank_account_number_en.pdf (visited on 2026-04).

Bangor, Aaron, Philip T. Kortum, and James T. Miller (2009). "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *Journal of Usability Studies* 4.3, pp. 114–123.

Billoir, Eddie et al. (2024). "Enhancing Secure Deployment with Ansible: A Focus on Least Privilege and Automation for Linux". In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. DOI: 10.1145/3664476.3670929.

Brooke, John (1996). "SUS: A 'Quick and Dirty' Usability Scale". In: *Usability Evaluation in Industry*. Ed. by Patrick W. Jordan et al. Taylor & Francis, pp. 189–194. ISBN: 9780748404605. DOI: 10.1201/9781498710411-35.

Buschmann, Frank et al. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley. ISBN: 978-0-471-95869-7.

Cao, Lan and Balasubramaniam Ramesh (2008). "Agile Requirements Engineering Practices: An Empirical Study". In: *IEEE Software* 25.1, pp. 60–67. DOI: 10.1109/MS.2008.1.

Center for Internet Security (Apr. 2022). *Getting to Know the CIS Benchmarks*. URL: https://www.cisecurity.org/insights/blog/getting-to-know-the-cis-benchmarks (visited on 2025-11).

CERT Coordination Center (2024). *Stakeholder-Specific Vulnerability Categorization (SSVC) v2024.3*. URL: https://certcc.github.io/SSVC/ (visited on 2026-04-20).

Cloud Native Computing Foundation (Sept. 2020). *CNCF End User Technology Radar: Observability, September 2020*. URL: https://www.cncf.io/blog/2020/09/11/cncf-end-user-technology-radar-observability-september-2020/ (visited on 2026-01).

Crowley, Christopher (May 2024). *SANS 2024 SOC Survey: Facing Top Challenges in Security Operations*. Tech. rep. SANS Institute. URL: https://www.sans.org/white-papers/sans-2024-soc-survey-facing-top-challenges-security-operations.

Cybersecurity and Infrastructure Security Agency (2024). *CISA Vulnrichment: A project to conduct vulnerability enrichment*. GitHub. URL: https://github.com/cisagov/vulnrichment.

Davies, Andrew et al. (Oct. 2025). *Magic Quadrant for Security Information and Event Management*. Tech. rep. G00822919. Gartner. URL: https://www.gartner.com/en/documents/5415763.

Dempsey, Kelley et al. (Sept. 2011). *Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations*. Tech. rep. SP 800-137. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-137. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-137.pdf.

Deng, Mina et al. (2011). "A privacy threat analysis framework: motivating the cvss-based approach". In: *Proceedings of the 2011 international workshop on Security and trust management*, pp. 143–158.

Elastic N.V. (2021). *FAQ on Software Licensing*. URL: https://www.elastic.co/pricing/faq/licensing (visited on 2026-01).

Elastic N.V. (2025a). *Elastic Security: SIEM, XDR, and Endpoint Security*. URL: https://www.elastic.co/security (visited on 2025-11).

Elastic N.V. (2025b). *Elastic Stack Pricing*. URL: https://www.elastic.co/pricing (visited on 2025-11).

Elastic N.V. (2025c). *Manage Saved Objects in Kibana*. URL: https://www.elastic.co/guide/en/kibana/8.19/managing-saved-objects.html (visited on 2026-01).

ENISA (Oct. 2024). *ENISA Threat Landscape 2024*. Tech. rep. European Union Agency for Cybersecurity. URL: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024.

European Parliament and Council of the European Union (2016). *Regulation (EU) 2016/679 of the European Parliament and of the Council (General Data Protection Regulation)*. Official Journal of the European Union, L 119, 4 May 2016, pp. 1–88. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 2026-03).

European Parliament and Council of the European Union (2022). *Directive (EU) 2022/2555 on measures for a high common level of cybersecurity across the Union (NIS 2 Directive)*. Official Journal of the European Union, L 333, 27 December 2022, pp. 80–152. URL: https://eur-lex.europa.eu/eli/dir/2022/2555/oj (visited on 2026-03).

Few, Stephen (2013). *Information Dashboard Design: Displaying Data for At-a-Glance Monitoring*. 2nd. Analytics Press. ISBN: 978-1938377006.

FIRST.org (2023). *Exploit Prediction Scoring System (EPSS) v3.0 Documentation*. URL: https://www.first.org/epss/ (visited on 2026-04-20).

FIRST.org (2025). *Common Vulnerability Scoring System SIG*. URL: https://www.first.org/cvss/ (visited on 2025-11).

Forsberg, Joonas and Tapio Frantti (2023). "Technical Performance Metrics of a Security Operations Center". In: *Computers & Security* 135, p. 103529. DOI: 10.1016/j.cose.2023.103529.

Fowler, Martin (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. ISBN: 978-0201485677.

Gamma, Erich et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN: 978-0-201-63361-0.

González-Granadillo, Gustavo, Susana González-Zarzosa, and Rodrigo Diaz (2021). "Security Information and Event Management (SIEM): Analysis, Trends, and Usage in Critical Infrastructures". In: *Sensors* 21.14, p. 4759. DOI: 10.3390/s21144759. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC8309804/.

Grafana Labs (2024a). *Grafana Alerting*. URL: https://grafana.com/docs/grafana/latest/alerting/ (visited on 2026-01).

Grafana Labs (2024b). *Grafana Loki Documentation*. URL: https://grafana.com/docs/loki/latest/ (visited on 2026-01).

Gürses, Seda, Carmela Troncoso, and Claudia Diaz (Jan. 2011). "Engineering Privacy by Design". In: *Proceedings of the 2011 International Workshop on Computing, Privacy and Data Protection (CPDP)*. URL: https://software.imdea.org/~carmela.troncoso/papers/Gurses-CPDP11.pdf.

Hardt, Dick (Oct. 2012). *The OAuth 2.0 Authorization Framework*. Request for Comments RFC 6749. Internet Engineering Task Force. DOI: 10.17487/RFC6749. URL: https://datatracker.ietf.org/doc/rfc6749.

Hevner, Alan R. et al. (Mar. 2004). "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1, pp. 75–105. DOI: 10.2307/25148625.

Hohpe, Gregor and Bobby Woolf (Oct. 2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN: 978-0-321-20068-6.

IBM (2026). *IBM QRadar SIEM*. URL: https://www.ibm.com/products/qradar-siem (visited on 2026-01).

Icinga GmbH (2024a). *Icinga 2 API*. URL: https://icinga.com/docs/icinga-2/latest/doc/12-icinga2-api/ (visited on 2026-01).

Icinga GmbH (2024b). *Nagios Alternatives: Why Icinga Is the Strongest Choice*. URL: https://icinga.com/blog/nagios-alternatives/ (visited on 2026-01).

IEEE (2018). *ISO/IEC/IEEE 29148:2018 — Systems and Software Engineering — Life Cycle Processes — Requirements Engineering*. IEEE Standard. IEEE. DOI: 10.1109/IEEESTD.2018.8559686. URL: https://standards.ieee.org/standard/29148-2018.html.

International Organization for Standardization (Aug. 2019). *Security techniques — Extension to ISO/IEC 27001 and ISO/IEC 27002 for privacy information management — Requirements and guidelines*. Tech. rep. ISO/IEC 27701:2019. ISO/IEC.

Islam, Chadni, Muhammad Ali Babar, and Surya Nepal (Apr. 2019). "A Multi-Vocal Review of Security Orchestration". In: *ACM Computing Surveys* 52.2, 37, pp. 1–45. DOI: 10.1145/3305268. URL: https://doi.org/10.1145/3305268.

ISO/IEC (2022). *Information security, cybersecurity and privacy protection — Guidance on managing information security risks*. Tech. rep. ISO/IEC 27005:2022. International Organization for Standardization. URL: https://www.iso.org/standard/80585.html.

Jalalvand, Fatemeh et al. (Nov. 2024). "Alert Prioritisation in Security Operations Centres: A Systematic Survey on Criteria and Methods". In: *ACM Computing Surveys* 57.2, 42:1–42:36. DOI: 10.1145/3695462.

Joint Task Force Interagency Working Group (Sept. 2020). *Security and Privacy Controls for Information Systems and Organizations*. Tech. rep. SP 800-53 Rev. 5. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-53r5. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf.

Jones, Michael B., John Bradley, and Nat Sakimura (May 2015). *JSON Web Token (JWT)*. Request for Comments RFC 7519. Internet Engineering Task Force. DOI: 10.17487/RFC7519. URL: https://datatracker.ietf.org/doc/rfc7519.

Jung, Ralf et al. (Apr. 2021). "Safe Systems Programming in Rust". In: *Communications of the ACM* 64.4, pp. 144–152. DOI: 10.1145/3418295.

Katz, Yonatan and Danny Raz (2023). "Cold Start for Cloud Anomaly Detection". In: *2023 IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, pp. 1–8. DOI: 10.1109/NOMS56928.2023.10154340.

Khan, Saiful, Yulei Fan, and Min Chen (2023). "Dashboard Design Patterns". In: *IEEE Transactions on Visualization and Computer Graphics* 29.1, pp. 342–352. DOI: 10.1109/TVCG.2022.3209448.

Kokulu, Faris Bugra et al. (Nov. 2019). "Matched and Mismatched SOCs: A Qualitative Study on Security Operations Center Issues". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, pp. 1955–1970. DOI: 10.1145/3319535.3354239.

Kumara, Indika et al. (2021). "The Do's and Don'ts of Infrastructure Code: A Systematic Grey Literature Review". In: *Information and Software Technology* 137, p. 106593. DOI: 10.1016/j.infsof.2021.106593.

Larman, Craig and Victor R. Basili (June 2003). "Iterative and Incremental Development: A Brief History". In: *IEEE Computer* 36.6, pp. 47–56. DOI: 10.1109/MC.2003.1204375.

Lavrijsen, Wim T. L. P. and Aditi Dutta (Nov. 2016). "High-Performance Python-C++ Bindings with PyPy and Cling". In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, pp. 27–35. DOI: 10.1109/PyHPC.2016.008.

Liu, Dapeng et al. (2015). "Opprentice: Towards Practical and Automatic Anomaly Detection Through Machine Learning". In: *Proceedings of the 2015 Internet Measurement Conference (IMC '15)*. Association for Computing Machinery, pp. 211–224. DOI: 10.1145/2815675.2815679.

Liu, Jiayi et al. (2023). "Anomaly and Change Point Detection for Time Series with Concept Drift". In: *World Wide Web* 26.5, pp. 3229–3252. DOI: 10.1007/s11280-023-01181-z.

López Velásquez, Juan Miguel et al. (June 2023). "Systematic Review of SIEM Technology: SIEM-SC Birth". In: *International Journal of Information Security* 22.3, pp. 691–711. DOI: 10.1007/s10207-022-00657-9.

Luhn, Hans Peter (Aug. 1960). *Computer for Verifying Numbers*. US Patent 2,950,048. URL: https://patents.google.com/patent/US2950048A/en.

Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. ISBN: 978-0132350884.

Mattermost, Inc. (2026). *Editions and Offerings*. URL: https://docs.mattermost.com/product-overview/editions-and-offerings.html (visited on 2026-04).

Microsoft Corporation (2026). *Overview of Microsoft Graph*. URL: https://learn.microsoft.com/en-us/graph/overview (visited on 2026-03).

Morris, Kief (Jan. 2021). *Infrastructure as Code: Dynamic Systems for the Cloud Age*. 2nd ed. O'Reilly Media. ISBN: 978-1-0981-1467-1.

Nagios Enterprises, LLC (2024). *Nagios Core Plugin API*. URL: https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/pluginapi.html (visited on 2026-01).

National Institute of Standards and Technology (2025). *National Vulnerability Database*. URL: https://nvd.nist.gov/ (visited on 2025-11).

Nelson, Annie et al. (Apr. 2025). *Incident Response Recommendations and Considerations for Cybersecurity Risk Management: A CSF 2.0 Community Profile*. Tech. rep. SP 800-61r3. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-61r3. URL: https://doi.org/10.6028/NIST.SP.800-61r3.

Nygard, Michael T. (2018). *Release It! Design and Deploy Production-Ready Software*. 2nd ed. Pragmatic Bookshelf. ISBN: 978-1-68050-239-8.

OWASP Foundation (2023). *OWASP API Security Top 10*. Tech. rep. OWASP Foundation. URL: https://owasp.org/www-project-api-security/.

OWASP Foundation (May 2025). *OWASP Application Security Verification Standard (ASVS) 5.0.0*. Tech. rep. OWASP Foundation. URL: https://asvs.dev/.

Pascoe, Cherilyn, Stephen Quinn, and Karen Scarfone (Feb. 2024). *The NIST Cybersecurity Framework (CSF) 2.0*. Tech. rep. National Institute of Standards and Technology. DOI: 10.6028/NIST.CSWP.29. URL: https://doi.org/10.6028/NIST.CSWP.29.

PCI Security Standards Council (2025). *Official PCI Security Standards Council Site*. URL: https://www.pcisecuritystandards.org/ (visited on 2025-11).

Peffers, Ken et al. (2007). "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3, pp. 45–77. DOI: 10.2753/MIS0742-1222240302.

Prometheus Authors (2024). *Overview — Prometheus Documentation*. URL: https://prometheus.io/docs/introduction/overview/ (visited on 2026-01).

Pytest Contributors (2026). *pytest Documentation*. URL: https://docs.pytest.org/ (visited on 2026-03).

Qualys, Inc. (2025). *VMDR and PA Help*. URL: https://docs.qualys.com/en/vm/latest/ (visited on 2026-01).

Red Hat, Inc. (2024a). *Administering Red Hat Satellite*. URL: https://docs.redhat.com/en/documentation/red_hat_satellite/6.13/html/administering_red_hat_satellite/accessing_server_admin (visited on 2026-01).

Red Hat, Inc. (2024b). *Using the Red Hat Satellite REST API*. URL: https://docs.redhat.com/en/documentation/red_hat_satellite/6.14/html/api_guide/chap-red_hat_satellite-api_guide-using_the_red_hat_satellite_api (visited on 2026-01).

Red Hat, Inc. (2025a). *Controlling Playbook Execution: Strategies and More*. URL: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_strategies.html (visited on 2026-01).

Red Hat, Inc. (2025b). *Getting Started with Ansible*. URL: https://docs.ansible.com/projects/ansible/latest/getting_started/index.html (visited on 2025-11).

Red Hat, Inc. (2025c). *Security Automation with Red Hat Ansible Automation Platform*. URL: https://www.redhat.com/en/technologies/management/ansible/security-automation (visited on 2025-11).

Runeson, Per and Martin Höst (2009). "Guidelines for Conducting and Reporting Case Study Research in Software Engineering". In: *Empirical Software Engineering* 14.2, pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

Sauro, Jeff (2011). *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. Measuring Usability LLC. ISBN: 9781461062707.

Scarfone, Karen, Tim Kent, and Murugiah Souppaya (Sept. 2006). *Guide to Computer Security Log Management*. Tech. rep. SP 800-92. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-92. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-92.pdf.

Shneiderman, Ben (Sept. 1996). "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations". In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. IEEE, pp. 336–343. DOI: 10.1109/VL.1996.545307.

Shostack, Adam (2014). *Threat Modeling: Designing for Security*. John Wiley & Sons. ISBN: 978-1-118-80999-0.

Souppaya, Murugiah and Karen Scarfone (Apr. 2022). *Guide to Enterprise Patch Management Planning: Preventive Maintenance for Technology*. Tech. rep. SP 800-40 Rev. 4. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-40r4. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r4.pdf.

Spiekermann, Sarah and Lorrie Faith Cranor (Jan. 2009). "Engineering Privacy". In: *IEEE Transactions on Software Engineering* 35.1, pp. 67–82. DOI: 10.1109/TSE.2008.88.

Splunk Inc. (Feb. 2024a). *Splunk Common Information Model (CIM) Add-on 5.3.x Manual*. URL: https://docs.splunk.com/Documentation/CIM/5.3.3/User/Overview (visited on 2025-11).

Splunk Inc. (Oct. 2024b). *Splunk Enterprise Security 8.0 Documentation*. URL: https://help.splunk.com/en/splunk-enterprise-security-8/user-guide (visited on 2025-11).

Splunk Inc. (June 2024c). *Splunk Python SDK 2.0.2*. URL: https://dev.splunk.com/enterprise/docs/devtools/python/sdk-python/ (visited on 2025-11).

Splunk Inc. (2026a). *About configuring role-based user access*. URL: https://help.splunk.com/en/splunk-enterprise/administer/manage-users-and-security/9.4/manage-splunk-platform-users-and-roles/about-configuring-role-based-user-access (visited on 2026-02).

Splunk Inc. (2026b). *About distributed search*. URL: https://help.splunk.com/en/splunk-enterprise/administer/distributed-search/10.2/overview-of-distributed-search/about-distributed-search (visited on 2026-02).

Splunk Inc. (2026c). *App architecture and object ownership*. URL: https://help.splunk.com/en/splunk-enterprise/administer/admin-manual/10.0/meet-splunk-apps/app-architecture-and-object-ownership (visited on 2026-02).

Splunk Inc. (2026d). *Configure access control for secret storage*. URL: https://dev.splunk.com/enterprise/docs/developapps/manageknowledge/secretstorage/secretstoragerbac/ (visited on 2026-02).

Splunk Inc. (2026e). *Create custom search commands for apps in Splunk Cloud Platform or Splunk Enterprise*. URL: https://dev.splunk.com/enterprise/docs/devtools/customsearchcommands/ (visited on 2026-02).

Splunk Inc. (2026f). *Federated Search for Splunk platform*. URL: https://help.splunk.com/en/splunk-cloud-platform/get-started/splunk-validated-architectures/splunk-platform-indexing-and-search/federated-search-for-splunk-platform (visited on 2026-02).

Splunk Inc. (2026g). *Manage secret storage in apps for Splunk Cloud Platform or Splunk Enterprise*. URL: https://dev.splunk.com/enterprise/docs/developapps/manageknowledge/secretstorage/ (visited on 2026-02).

Splunk Inc. (2026h). *Splunk Enterprise User and Admin Documentation: Search, Dashboards, and Data Management*. URL: https://help.splunk.com/en/splunk-enterprise/ (visited on 2025-11).

Strom, Blake E. et al. (July 2018). *MITRE ATT&CK™: Design and Philosophy*. Tech. rep. The MITRE Corporation. URL: https://apps.dtic.mil/sti/trecms/pdf/AD1108016.pdf.

Sundaramurthy, Sathya Chandran et al. (June 2016). "Turning Contradictions into Innovations or: How We Learned to Stop Whining and Improve Security Operations". In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association. URL: https://www.usenix.org/conference/soups2016/technical-sessions/presentation/sundaramurthy.

Tenable, Inc. (2025). *Tenable Nessus Vulnerability Scanner*. URL: https://www.tenable.com/products/nessus (visited on 2025-11).

The MITRE Corporation (2025). *CVE — Common Vulnerabilities and Exposures*. URL: https://www.cve.org/ (visited on 2025-11).

The MITRE Corporation (2026). *MITRE ATT&CK®*. URL: https://attack.mitre.org/ (visited on 2026-03).

Tilbury, Jack and Stephen Flowerday (July 2024). "Humans and Automation: Augmenting Security Operation Centers". In: *Journal of Cybersecurity and Privacy* 4.3, 20, pp. 388–409. DOI: 10.3390/jcp4030020. URL: https://doi.org/10.3390/jcp4030020.

Tran, Anh-Duy et al. (2024). "TerrARA: Automated Security Threat Modeling for Infrastructure as Code". In: *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy*. DOI: 10.1145/3714393.3726494.

Tullis, Thomas S. and Jacqueline N. Stetson (2004). "A Comparison of Questionnaires for Assessing Website Usability". In: *Usability Professionals Association (UPA) 2004 Conference*, pp. 1–12.

Van Landuyt, Dimitri and Wouter Joosen (2021). "A descriptive study of assumptions in STRIDE security threat modeling". In: *Software and Systems Modeling*. DOI: 10.1007/s10270-021-00941-7.

Vielberth, Manfred et al. (2020). "Security Operations Center: A Systematic Study and Open Challenges". In: *IEEE Access* 8, pp. 227756–227779. DOI: 10.1109/ACCESS.2020.3045514.

Wieringa, Roel J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer. ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8.

Wohlin, Claes et al. (2012). *Experimentation in Software Engineering*. Springer. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2.

Wuyts, Kim (2015). "Privacy threat modeling: engineering privacy in software". PhD thesis. KU Leuven.

Yin, Robert K. (2018). *Case Study Research and Applications: Design and Methods*. 6th ed. SAGE Publications. ISBN: 978-1-5063-3616-9.

Appendices

A

Extended Implementation Details

This appendix records implementation details omitted from Chapter 5. The sections below document selected dashboard subsystems, row expansion mechanics, and credential-management tooling that are easier to inspect in a reference appendix than in the main narrative.

A.1 Dashboard Tab Descriptions

The unified dashboard described in Section 5.4 organises monitoring data across sixteen tabs. This section documents the panel layout and data sources for five tabs whose implementation details complement the architectural overview in the main chapter.

A.1.1 Vulnerabilities Tab Implementation

The Vulnerabilities tab integrates Satellite errata data with infrastructure context. A summary row displays total errata counts segmented by type, using single-value visualisations with coloured indicators for critical findings. Below, a detailed table lists individual errata with severity, type, affected host count, and CVE references.

Filtering controls allow analysts to focus investigation. A severity dropdown constrains results to security, bugfix, or enhancement categories. A text filter allows keyword search across errata titles and descriptions. The time range selector, though errata data is point-in-time rather than temporal, allows comparison across different scan dates when investigating remediation progress.

A.1.2 Chart Configuration

Charts follow consistent styling conventions. Licence usage displays as horseshoe gauges showing percentage utilisation against allocation limits. Queue depths render

as bar charts with threshold lines indicating warning and critical levels. Time-series charts track trends in metrics such as index growth or licence consumption.

Colour coding conveys status semantically. Green indicates healthy states, amber signals warnings that warrant attention, and red highlights critical conditions requiring immediate response. These colours apply to gauges, table cell backgrounds, and status indicator icons throughout the dashboard.

A.1.3 Forwarder Monitoring Implementation

The Forwarders tab integrates data from multiple sources to present complete forwarder fleet status. Unlike tabs that query a single data source, the Forwarders tab combines deployment server metadata, internal telemetry, and external enrichment to build a full picture.

The primary data source is the Splunk deployment server, accessed through Splunk's federated search feature rather than the `usoapi` command. A federated provider configured on the search head cluster authenticates against the deployment server with a dedicated service account, where a saved search queries the local client status REST endpoint. The dashboard retrieves these results via `| from federated:forwarders_ds`, obtaining forwarder check-in records, installed application inventories, assigned server classes, and last communication timestamps. Because the underlying REST endpoint returns only *deployment clients* that actively poll the deployment server, any forwarder not registered with that server does not appear in these results.

Version distribution and operating system charts derive from deployment server data. The implementation extracts version strings from client metadata, groups by major/minor version, and counts instances per group. Operating system data comes from the forwarder's reported platform, supporting segmentation by Windows, Linux, and other platforms.

Missing forwarder detection derives from the polling timestamps in the deployment server data. The base search computes the elapsed time since each forwarder's last check-in and classifies forwarders that have not reported within 15 minutes as missing. This threshold-based approach avoids maintaining a separate expected-inventory list. Figure B.4 in Appendix B shows the resulting view with version distribution, OS breakdown, and missing forwarder panels.

Row expansion for the Forwarders table retrieves detailed information through a secondary search pipeline. When an analyst expands a forwarder row, the corresponding script (`forwarder_row_expansion.js`) executes queries for installed applications, server class memberships, and DNS resolution data from the deployment server metadata. Result caching at the JavaScript level reduces API load for repeated expansions. Figure B.2 in Appendix B illustrates this pattern applied to the Infrastructure tab; the Forwarders implementation follows the same interaction model with different content.

A.1.4 Performance Tab Implementation

The Performance tab visualises host resource utilisation using Splunk's internal metrics infrastructure. Unlike USO API modules that query external systems, this tab queries the local Splunk instance's `_introspection` index and metrics data using the `mstats` command.

CPU utilisation charts aggregate processor metrics across cores, distinguishing between user, system, and idle time. The implementation uses `mstats` with appropriate metric filter specifications, grouping by host and calculating percentage utilisation. Multi-host views display each host as a separate series; single-host views offer finer per-core breakdowns. Figure B.3 in Appendix B illustrates the resulting layout with time-series CPU, memory, disk I/O wait, and swap panels.

Memory monitoring tracks both usage percentages and absolute values. Metrics include total memory, used memory, cached and buffered amounts, and swap utilisation. The implementation calculates percentage values dynamically, as the underlying metrics store absolute byte counts. Swap usage receives particular attention, as any non-trivial swap activity may indicate memory pressure affecting performance.

Disk I/O metrics reveal read and write rates by device. The implementation filters to relevant disk devices, excluding virtual and loop devices that generate noise. Rate calculations use `mstats` aggregation functions to compute bytes per second from cumulative counters. High I/O rates may indicate backup activity, database operations, or potential ransomware behaviour depending on context.

Network traffic charts show bytes in and out by interface. The implementation allows filtering to specific interfaces when analysts need to focus on particular network segments. Combined with process count metrics, which reveal how many processes are running on each host, these visualisations aid capacity planning and anomaly detection.

A.1.5 Firewall Tab

Six base searches power the Firewall tab. These searches query firewall indexes matching the `fw_*` pattern, covering two enterprise firewall platforms. Index selection relies on a multiselect input that populates its choices dynamically from a `firewall_devices.csv` lookup file, listing predefined device groups alongside individual appliance entries. Analysts can combine selections to compare traffic across vendor boundaries or isolate a single device for focused investigation.

Four of the six searches prefix their queries with the literal terms `(drop OR deny)` before any field extractions or transformations: `fw_infra_category`, `fw_infra_blocked`, `fw_blocked_sources`, and `fw_blocked_ports`. Because Splunk's indexer maintains a term-level index for each data bucket, placing literal terms at the start of a search allows the indexer to skip entire buckets that cannot contain matching events, reducing the volume of raw events the indexers must scan. The remaining two searches omit this prefix. Because `fw_action_trend` and `fw_coverage` aggregate across mixed action

types including accepted connections, pre-filtering on denial terms would exclude relevant data.

Trend visualisations split across two complementary panels: a stacked area chart breaks down connection actions (accept, block, start, other) over time, whilst a separate line chart isolates blocked traffic trends for pattern identification. A bar chart in the same row categorises infrastructure drops by target group, drawing on the `infrastructure_targets_lookup` to enrich raw destination addresses with managed infrastructure labels. This enrichment lets analysts see which operational categories (servers, network devices, security appliances) attract the most unsolicited connections, rather than only individual IPs.

Ranking tables in subsequent rows surface the most active blocked sources. Source-to-destination correlation tables apply colour-coded status cells: red for purely blocked sources, yellow for mixed sources that generate both accepted and blocked connections. A companion table ranks source IPs by destination coverage, which identifies sources that probe broadly across the infrastructure rather than targeting single hosts.

The Connection Details panel is the most detailed view, filtering raw connection records through seven inline input controls: an action dropdown, source IP, destination IP, source port, destination port, protocol, and a time range picker. This filter set is the most extensive among all dashboard tabs, as firewall data carries both high volume and wide variety in its fields. All query logic remains inline within the dashboard XML; the tab requires no dedicated JavaScript files, saved searches, or macros.

A.2 Row Expansion Implementation

Section 5.4.2 introduces the row expansion pattern used across the dashboard. This section details the JavaScript architecture, caching mechanisms, and visual indicators that underpin the expansion functionality.

A.2.1 Row Expansion Architecture

The dashboard implements row expansion through thirteen specialised JavaScript files, each extending Splunk's `TableView.BaseRowExpansionRenderer` class. A shared utility module, `row_expansion_utils.js`, supplies common functionality that the table-specific implementations consume. This separation isolates presentation logic from framework interaction; each tab defines its own expansion content without duplicating infrastructure code.

The shared utility module exports four functions:

`registerRowExpansion()` Accepts a table identifier, a renderer class, and optional configuration, then handles the registration lifecycle including retry logic for dynamically loaded tables.

`injectChevronCSS()` Adds style rules for the expandable row indicator column.

`addChevronColumn()` Inserts clickable chevron cells into table rows after data loads.

`createChevronClickHandler()` Returns an event handler that manages expansion state and icon rotation.

Each table-specific renderer extends the base class and overrides three methods: `canRender()` returns true unconditionally since all rows support expansion, `initialize()` creates cache structures and search manager dictionaries, and `render()` extracts cell values from the row data, checks the cache, and either displays cached content or initiates secondary searches.

The Infrastructure tab renderer demonstrates the pattern's complexity. When a host row expands, the renderer spawns six parallel `SearchManager` instances: one queries Satellite for host metadata, another retrieves applicable errata, three retrieve disk usage, service state, and open ports through Ansible, and a sixth obtains Splunk health metrics if the host runs Splunk. Each search has a fifteen-second timeout to prevent indefinite loading states. A completion counter tracks pending searches; when all complete or timeout, the renderer assembles the composite view.

Listing 10 shows the trimmed core of the registration function, including the state object that coordinates retry logic, CSS injection, and table component discovery.

Listing 10: *Core registration function in* `row_expansion_utils.js`

```
function registerRowExpansion(tableId, RendererClass, options) {
  options = options || {};
  if (options.singleExpand === undefined) options.singleExpand = true;

  injectChevronCSS(tableId, options.hiddenColumnSelector);

  const state = {
    registered: false,
    tableViewRef: { view: null },
    renderer: null,
  };

  function ensureChevrons() {
    setTimeout(function () {
      addChevronColumn(tableId, state.tableViewRef, options);
    }, 200);
  }

  function tryRegister() {
    if (state.registered) return;
    const tableElement = mvc.Components.get(tableId);
    if (tableElement) {
      tableElement.getVisualization(function (tableView) {
        if (state.registered) return;
        state.registered = true;
        state.tableViewRef.view = tableView;
        state.renderer = new RendererClass();
        tableView.addRowExpansionRenderer(state.renderer);
        tableView.render();
      });
    }
  }

  tryRegister();

  // Retry if table not yet in DOM
  if (!state.registered && typeof MutationObserver !== "undefined") {
    const wrappedTryRegister = () => { tryRegister(); };
    pendingRegistrations.set(tableId, wrappedTryRegister);
    startSharedObserver();
  } else if (!state.registered) {
    setInterval(tryRegister, 2000);
  }

  ensureChevrons();
}
```

A.2.2 Caching Strategy

Repeated expansion of the same row should not trigger redundant API calls. Each renderer maintains a `_dataCache` dictionary keyed by a unique identifier, typically the hostname or errata identifier. Before initiating searches, the `render()` method checks this cache; if data exists, rendering proceeds immediately from cached values. This approach reduces repeated SSH and REST requests to managed systems and improves perceived responsiveness.

Active search tracking prevents duplicate requests. A `_activeSearches` dictionary records rows currently undergoing data retrieval. If an analyst collapses and re-expands a row while its searches are in flight, the renderer detects the pending state and waits rather than spawning parallel duplicates. Search managers persist across expansions, keyed by sanitised identifiers, which avoids the overhead of repeated instantiation.

Listing 11 illustrates this pattern in the Infrastructure renderer, where the cache key, cache lookup, active search guard, and timeout operate together.

Listing 11: *Cache guard pattern in the Infrastructure renderer*

```
const cacheKey = hostname;
const hasSplunk = splunkWeb.indexOf("OK") === 0
              || splunkWeb === "Down";

if (this._dataCache[cacheKey]) {
  this._renderContent($container, {
    hostname, ip, os, role, status, erratastatus,
    splunkWeb, uptime, reboot, hasSplunk,
    ...this._dataCache[cacheKey],
  });
  return;
}

if (this._activeSearches[cacheKey]) return;

this._activeSearches[cacheKey] = true;
setTimeout(() => {
  if (this._activeSearches && this._activeSearches[cacheKey])
    delete this._activeSearches[cacheKey];
}, 60000);
```

A.2.3 Cache Invalidation

Static caching would display stale data when the parent table refreshes. Each renderer listens to the `search:done` event on its parent table's search manager. When this event fires, both `_dataCache` and `_activeSearches` clear, forcing subsequent expansions to fetch current data. This invalidation couples cache lifetime to the dashboard's autorefresh cycle.

The registration function establishes this binding during setup. After obtaining the table's associated search manager through the component registry, it attaches a listener that clears caches and re-injects chevron cells after each data refresh. Continuous interval checking, running every second for up to two minutes, handles cases where the table component registers after the initial page load completes.

Listing 12 shows the `search:done` handler that disposes cached `SearchManager` instances and clears both data and active-search dictionaries.

Listing 12: *Cache invalidation on parent search refresh*

```
searchManager.on("search:done", function () {
  ensureChevrons();
  if (state.renderer) {
    // Dispose cached SearchManagers to free memory
    if (state.renderer._searchManagers) {
      Object.keys(state.renderer._searchManagers)
        .forEach(function (key) {
          const sm = state.renderer._searchManagers[key];
          if (sm && typeof sm.dispose === "function")
            sm.dispose();
        });
      state.renderer._searchManagers = {};
    }
    if (state.renderer._dataCache)
      state.renderer._dataCache = {};
    if (state.renderer._activeSearches)
      state.renderer._activeSearches = {};
  }
});
```

A.2.4 Chevron Injection

Splunk's Simple XML tables do not natively display expansion indicators. The JavaScript implementation manually prepends a header cell and corresponding data cells to each row. The header cell contains an info icon; data cells contain a right-pointing triangle that rotates downward upon expansion.

Cell injection runs on multiple triggers: after initial registration, after each data refresh, and through a continuous interval during page load. This redundancy accounts for Splunk's asynchronous rendering, where table content may not exist at the moment JavaScript first executes. The interval ceases after two minutes or upon successful registration, whichever occurs first.

Click handling manages visual state. When an analyst clicks the chevron, the handler checks whether the row is already expanded by examining adjacent row classes and the presence of expansion-row markers. For collapsed rows, it calls `tableView.expandRow()`, which triggers the renderer's `render()` method. For expanded rows, it removes the expansion content and resets the icon orientation.

Listing 13 shows the column injection logic that prepends a header cell and attaches click handlers to each data row.

Listing 13: *Chevron column injection in* `addChevronColumn()`

```
const $headerRow = $table.find("thead tr");
if ($headerRow.length && !$headerRow.find("th.expands").length) {
  $headerRow.prepend('<th class="expands col-info"></th>');
}

$table.find("tbody tr").each(function () {
  const $row = $(this);
  if ($row.hasClass("expanded")
      || $row.find("td.expands").length > 0) return;

  const $chevronCell = $(
    '<td class="expands row-expansion-toggle">' +
    '<a><i class="icon-triangle-right-small"></i></a></td>'
  );
  $row.prepend($chevronCell);
  $chevronCell.on("click",
    createChevronClickHandler($row, tableViewRef, options));
});
```

A.3 Credential Management Tools

Section 5.8 describes the token-based authentication architecture. This section documents the command-line interfaces that administrators use to manage the three application credential types: Splunk bearer tokens, Red Hat Satellite username/password credentials, and OAuth 2.0 client credentials for the Microsoft Graph API. SSH key management for the Ansible control node is an internal operational responsibility of the Splunk infrastructure team and is documented in Section A.3.4 below.

A.3.1 Token Manager CLI Implementation

The `token_manager.py` script provides administrative operations for credential lifecycle management. The tool implements six commands: `add` stores a new token with metadata, `list` displays stored credentials with deterministic ordering, `get` retrieves a specific credential for verification, `delete` removes credentials from storage, `test` validates that a stored token successfully authenticates against its target, and `edit` updates an existing token whilst preserving host metadata. Each credential is stored under a username key of the form `<host>:<label>:<timestamp>:uso_api_service`. The host segment identifies the target by address, the optional label gives a human-readable mnemonic when present, and the Unix timestamp makes recreated keys unique whilst recording when the credential was added.

The `list` command sorts labels alphabetically and IP addresses numerically by octet. This ordering prevents the counterintuitive output that would place addresses ending in 100 between those ending in 10 and 11.

Token display balances security with usability. By default, the list command shows only the first eight characters of each token, sufficient to distinguish credentials without exposing the full secret. The `--reveal` flag with the `get` command displays complete token values when needed for verification, with such accesses noted in the application logs.

Listing 14 shows the `add` command, which authenticates to the local Splunk instance and delegates storage to `CredentialManager`.

Listing 14: *Token Manager `add` command implementation*

```
def cmd_add(args: argparse.Namespace) -> None:
    service = get_splunk_service(args)

    target_ip = args.ip or input("Remote server IP address: ").strip()
    label = args.label if args.label else input(f"Label for {target_ip} (optional):
    ↪  ").strip() or None

    if args.target_token:
        target_token = args.target_token
    else:
        display_name = f"{target_ip} ({label})" if label else target_ip
        target_token = getpass.getpass(f"Token from {display_name}: ").strip()

    cred_mgr = CredentialManager(service=service)
    result = cred_mgr.store_credential(target_ip, target_token, service=service,
    ↪  label=label)
```

A.3.2 Satellite Credential Management

Red Hat Satellite access uses a dedicated command-line tool, `satellite_credential_manager.py`, which stores a service-account username and password in Splunk's credential store under a dedicated Satellite integration realm. At runtime, the Satellite module retrieves this credential pair and authenticates its API requests against the configured Satellite server.

The tool follows the same pattern as the other credential managers, providing `add`, `list`, `get`, `delete`, and `test` commands. To rotate credentials, administrators update the stored username/password pair and re-run the connectivity check through the `test` command.

A.3.3 Graph Credential Management

Microsoft Graph API access requires OAuth 2.0 client credentials rather than the bearer tokens used for Splunk instance authentication. These credentials consist of a tenant identifier, client identifier, and client secret issued through Azure Active Directory application registrations. Because the credential structure differs fundamentally from the single-value JWT tokens managed by the token manager, Graph credentials occupy a separate realm within Splunk's credential store.

The implementation stores Graph credentials under a dedicated Graph integration realm, distinct from the default realm used for Splunk bearer tokens. Each credential entry serialises its components as a JSON object containing the tenant identifier, client identifier, client secret, and a list of monitored mailbox addresses. This structured format lets the Graph client library deserialise a single credential store entry into all parameters required for OAuth 2.0 token acquisition, without distributing related values across multiple storage entries.

A companion command-line tool, `graph_credential_manager.py`, offers administrative operations that mirror the token manager's interface. The tool exposes five commands: `add` stores a new credential set with associated mailbox addresses, `list` displays configured tenants with masked secrets, `get` retrieves full credential details for a specific tenant, `delete` removes a credential entry, and `test` validates credentials by authenticating against the Graph API and optionally verifying mailbox access. The `test` command instantiates the `GraphClient` class directly, exercising the same authentication path that production queries follow and confirming both token acquisition and API connectivity.

Realm-based separation permits both credential types to coexist within the same `storage/passwords` namespace. The Splunk module filters by the default realm when retrieving bearer tokens, whilst the Graph client filters by `graph_credentials` when retrieving OAuth 2.0 credentials. This approach avoids namespace collisions and allows each credential manager to operate independently without awareness of the other's entries.

Listing 15 shows how the `add_credential` function serialises OAuth 2.0 parameters into a JSON string and stores them under a dedicated realm.

Listing 15: *Graph API credential storage with realm separation*

```
CREDENTIAL_REALM = "graph_credentials"

def add_credential(args: argparse.Namespace,
                   service: Any) -> int:
    credential_data = {
        "tenant_id": args.tenant_id,
        "client_id": args.client_id,
        "client_secret": args.client_secret,
        "mailboxes": (args.mailboxes.split(",")
                      if args.mailboxes else []),
    }
    password = json.dumps(credential_data)

    for pwd in service.storage_passwords:
        if (pwd.realm == CREDENTIAL_REALM
                and pwd.username == args.tenant):
            pwd.delete()
            break

    service.storage_passwords.create(
        password=password,
        username=args.tenant,
        realm=CREDENTIAL_REALM,
    )
```

A.3.4 SSH Key Management

Unlike the three application credential types above, the SSH key used for the Ansible control node is maintained through the team's operational SSH procedures rather than these credential-manager tools.

Ansible execution depends on SSH key-based authentication between the search head and the Ansible control node. A dedicated key pair establishes this connection, with the private key stored in the Splunk user's SSH directory on the search head and the matching public key deployed to the control node's `authorized_keys` file.

Known hosts configuration requires maintenance across the search head cluster. Each member must have entries for both the Ansible control node and any rsync upload destinations, so that automated operations do not encounter interactive host-key prompts.

The application assumes that the operational team provisions and maintains the key pair outside the credential-manager tools, and that the matching public key remains present in the control node's `authorized_keys` file.

A.3.5 Service Account Configuration

The search head cluster authenticates users through LDAP queries against Active Directory, resolving accounts by searching the directory subtrees listed in the `userBaseDN` parameter of `authentication.conf` (Splunk Inc., 2026h). Because the existing configuration referenced only the organisational unit containing analyst accounts, and the monitoring service account resides in a separate subtree reserved for service accounts, a second path was appended to the `userBaseDN` value. This change was applied in the shared cluster configuration app on the deployer node and replicated to all search head cluster members during the next bundle push.

The custom role `sec-mon-access` carries sixteen capabilities, identified through iterative testing. The platform documentation does not consolidate which capabilities each monitoring endpoint requires, so the final set was determined empirically:

- `search` and `rest_properties_get` — core query and metadata access
- `list_health` and `list_introspection` — health endpoint and queue monitoring
- `list_inputs` and `list_search_scheduler` — data input and scheduler status
- `list_settings` and `list_all_objects` — server configuration visibility
- `list_dist_peer` — distributed peer enumeration
- `admin_all_objects` and `edit_own_objects` — administrative and ownership-scoped object access
- `license_edit` — licence pool information
- `edit_messages` — bulletin message management
- `run_collect` and `run_mcollect` — summary and metric index collection
- `schedule_rtsearch` — saved search scheduling

Index access is restricted to `_internal` and `_introspection`, and the search quota is set to ten concurrent jobs with zero real-time searches permitted.

A role mapping stanza in `authentication.conf` associates the corresponding Active Directory security group with `sec-mon-access`, so that any account belonging to that group inherits the role upon login (Splunk Inc., 2026h). This mapping completes the authentication chain described in Section 5.8.1: group membership in Active Directory, LDAP resolution by Splunk, and role mapping to capabilities.

B

Dashboard Screenshots

This appendix presents full-resolution screenshots of the unified security operations dashboard, reproduced at landscape scale to preserve legibility. Hostnames and network addresses have been anonymised.

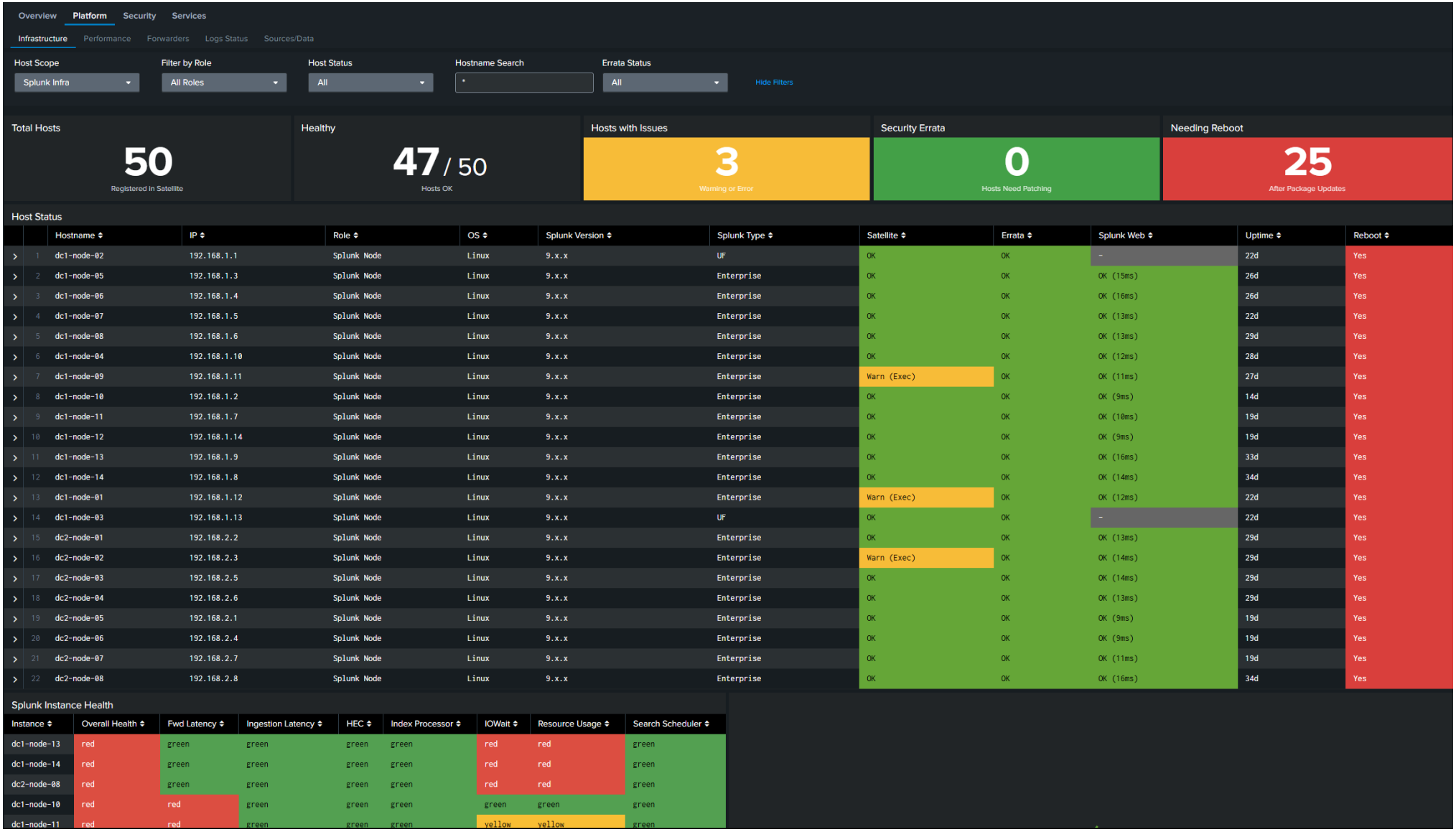


Figure B.1: *Infrastructure tab with host inventory and health indicators*

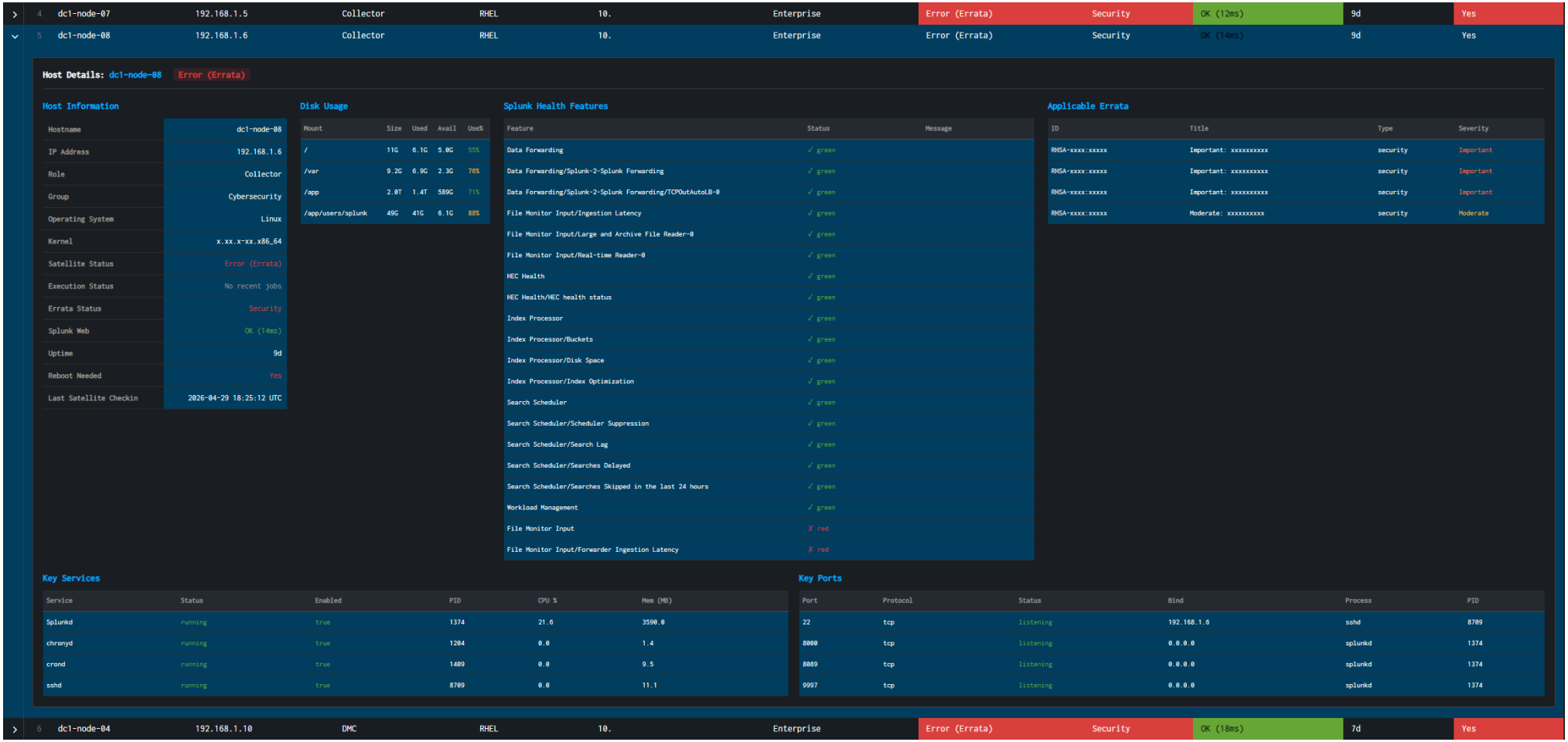
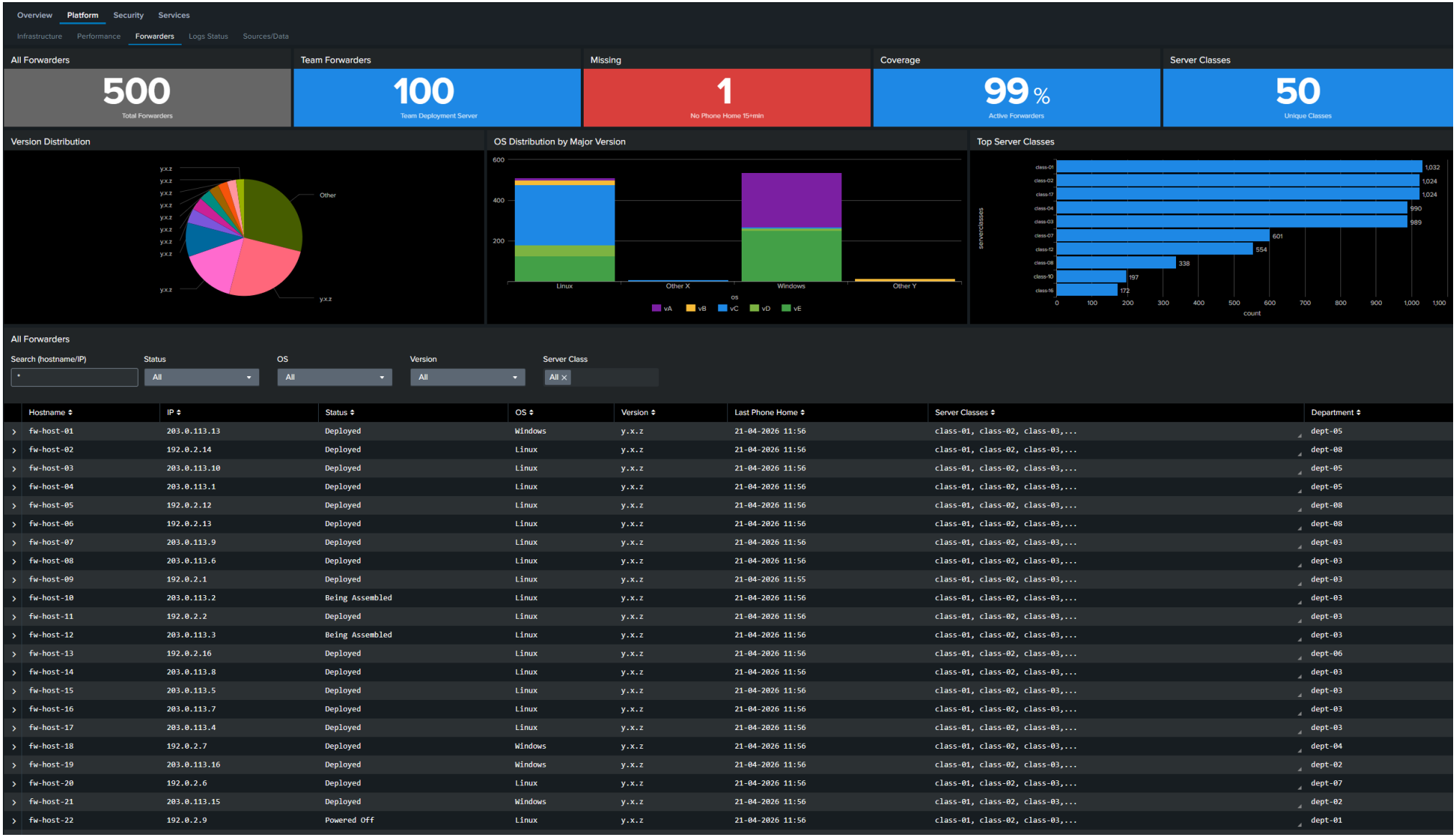


Figure B.2: *Row expansion detail panel for a selected host in the Infrastructure tab*

Figure B.3: *Performance tab with time-series CPU, memory, disk, I/O wait, and swap monitoring*



Hostname	IP	Status	OS	Version	Last Phone Home	Server Classes	Department
fw-host-01	203.0.113.13	Deployed	Windows	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-05
fw-host-02	192.0.2.14	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-08
fw-host-03	203.0.113.10	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-05
fw-host-04	203.0.113.1	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-05
fw-host-05	192.0.2.12	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-08
fw-host-06	192.0.2.13	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-08
fw-host-07	203.0.113.9	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-08	203.0.113.6	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-09	192.0.2.1	Deployed	Linux	y.x.z	21-04-2026 11:55	class-01, class-02, class-03,...	dept-03
fw-host-10	203.0.113.2	Being Assembled	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-11	192.0.2.2	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-12	203.0.113.3	Being Assembled	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-13	192.0.2.16	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-06
fw-host-14	203.0.113.8	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-15	203.0.113.5	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-16	203.0.113.7	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-17	203.0.113.4	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-03
fw-host-18	192.0.2.7	Deployed	Windows	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-04
fw-host-19	203.0.113.16	Deployed	Windows	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-02
fw-host-20	192.0.2.6	Deployed	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-07
fw-host-21	203.0.113.15	Deployed	Windows	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-02
fw-host-22	192.0.2.9	Powered Off	Linux	y.x.z	21-04-2026 11:56	class-01, class-02, class-03,...	dept-01

Figure B.4: *Forwarders tab showing deployment server application inventories, server class memberships, and polling recency*

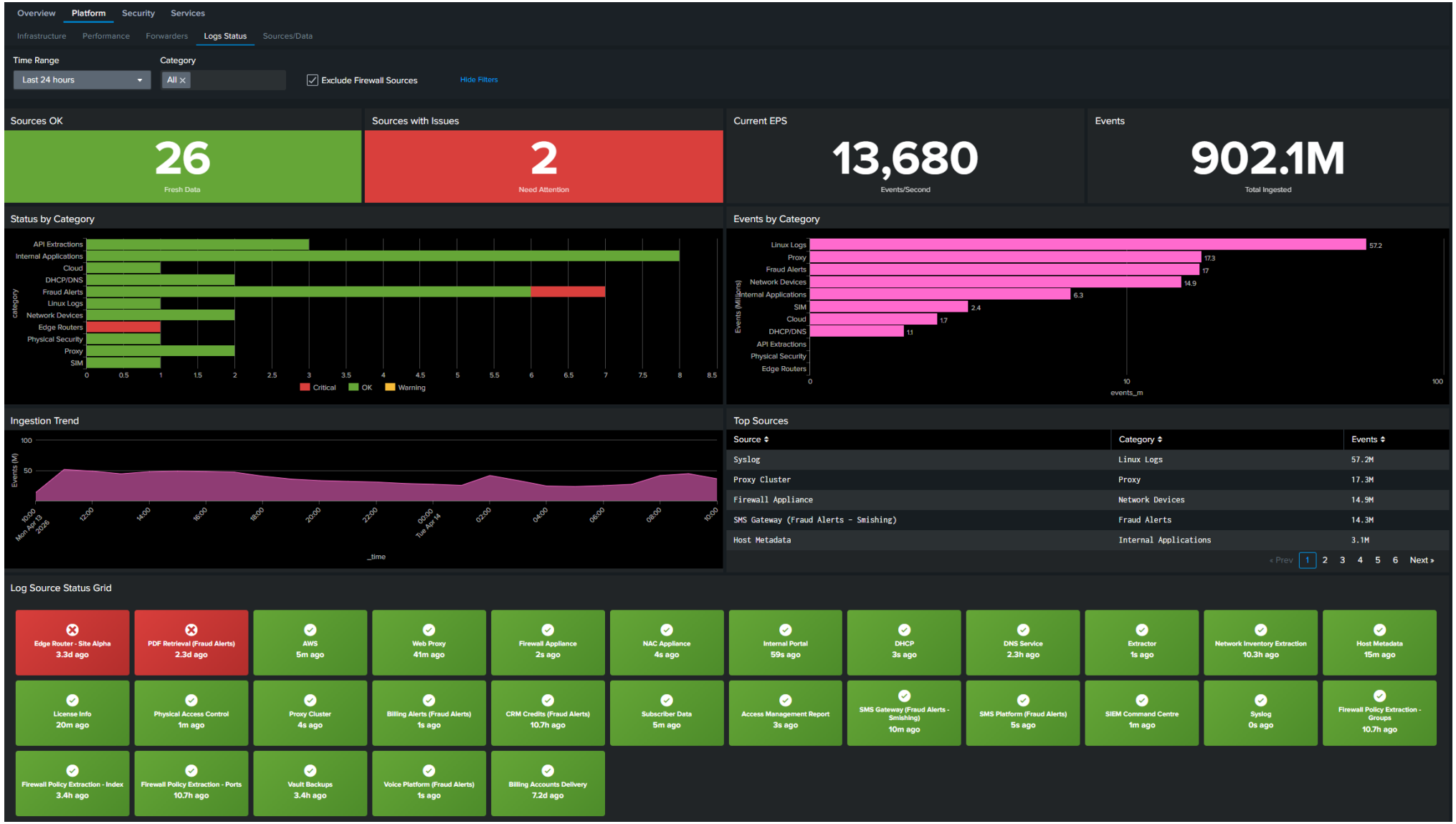


Figure B.5: *Logs Status tab. The bottom section labelled "Log Source Status Grid" is the custom* `logs_status_grid.js` *component described in Section 5.4.4; the remaining panels are native Splunk visualisations*

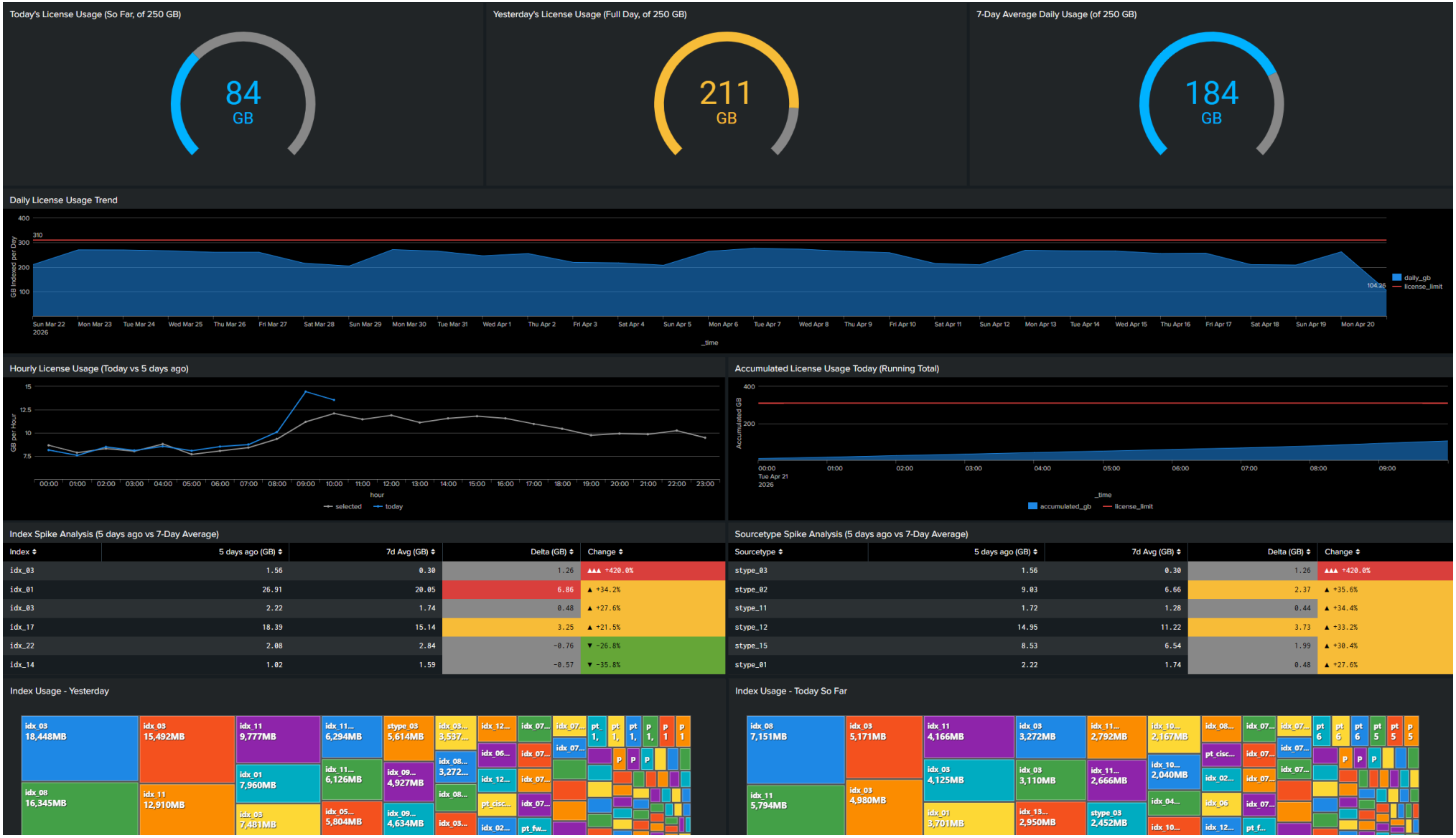


Figure B.6: *Licenses tab with current, previous-day, and seven-day licence-usage gauges, trend panels, and per-index and per-sourcetype breakdowns*

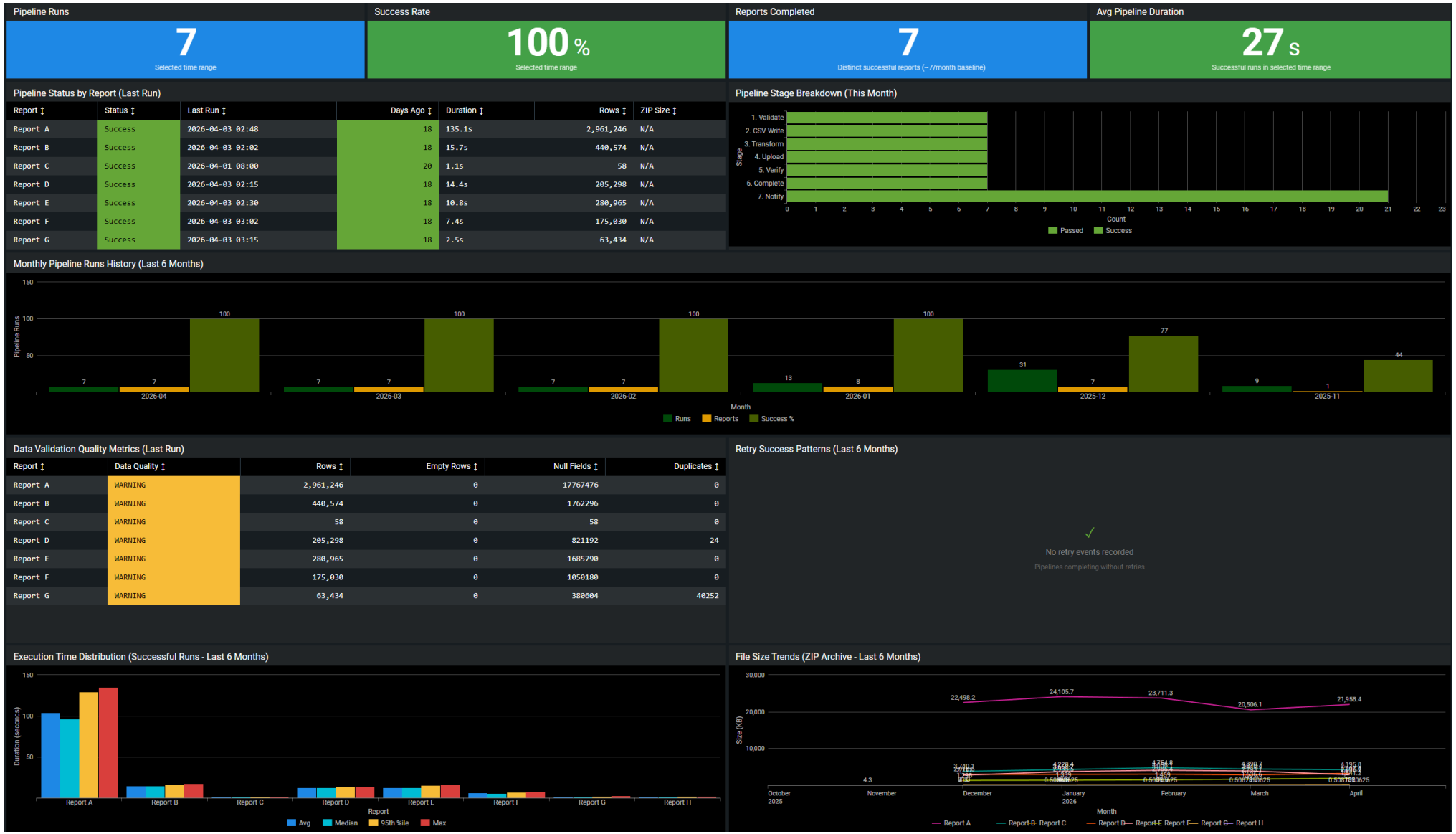
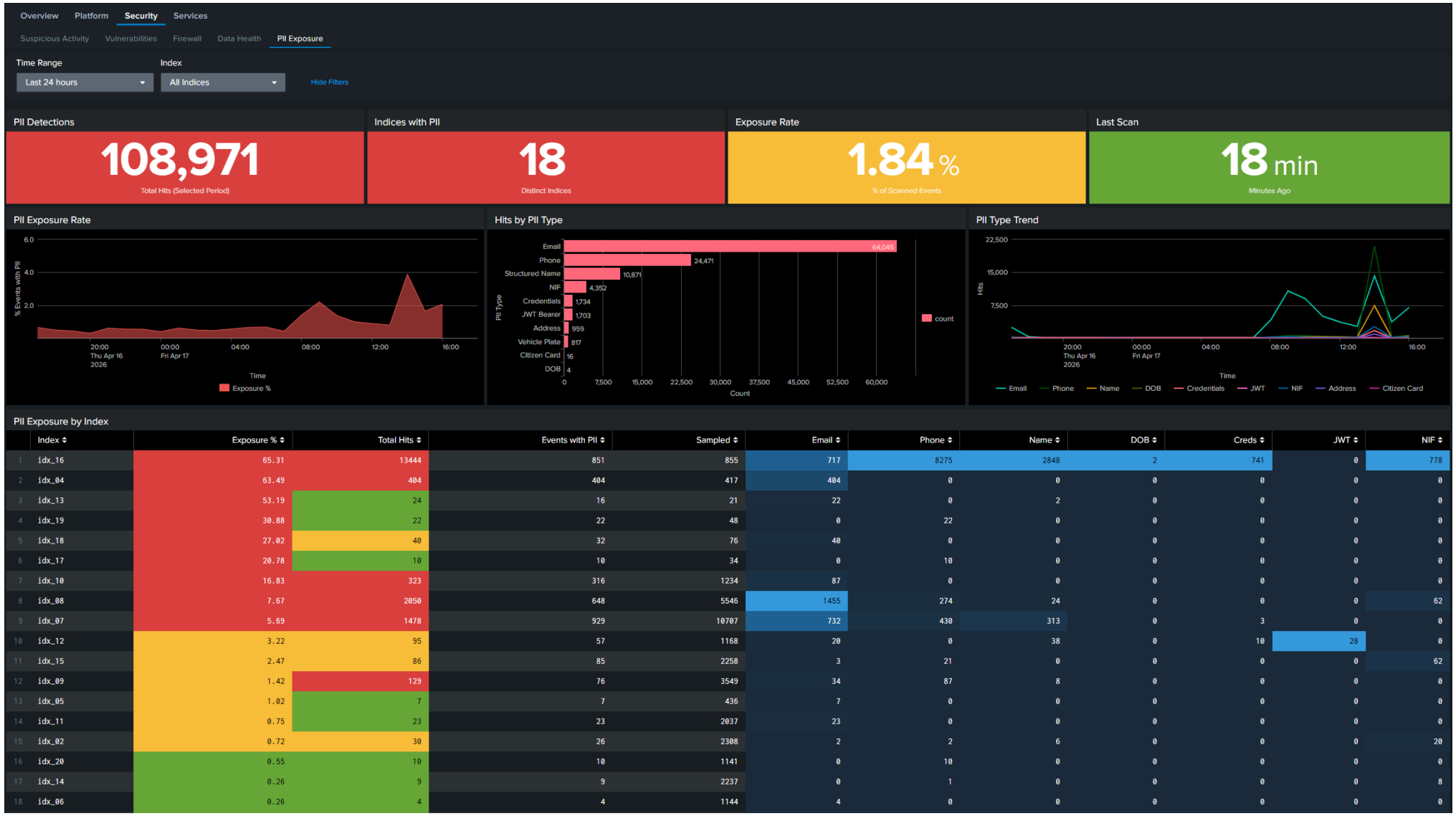


Figure B.7: *Automation tab displaying compliance report pipeline execution history, colour-coded by outcome*



PII Exposure by Index

	Index	Exposure %	Total Hits	Events with PII	Sampled	Email	Phone	Name	DOB	Creds	JWT	NIF
1	idx_16	65.31	13444	851	855	717	8275	2848	2	741	0	778
2	idx_04	63.49	404	404	417	404	0	0	0	0	0	0
3	idx_13	53.19	24	16	21	22	0	2	0	0	0	0
4	idx_19	30.88	22	22	48	0	22	0	0	0	0	0
5	idx_18	27.02	40	32	76	40	0	0	0	0	0	0
6	idx_17	20.78	10	10	34	0	10	0	0	0	0	0
7	idx_10	16.83	323	316	1234	87	0	0	0	0	0	0
8	idx_08	7.67	2050	648	5546	1455	274	24	0	0	0	62
9	idx_07	5.69	1478	929	10707	732	430	313	0	3	0	0
10	idx_12	3.22	95	57	1168	20	0	38	0	10	28	0
11	idx_15	2.47	86	85	2258	3	21	0	0	0	0	62
12	idx_09	1.42	129	76	3549	34	87	8	0	0	0	0
13	idx_05	1.02	7	7	436	7	0	0	0	0	0	0
14	idx_11	0.75	23	23	2037	23	0	0	0	0	0	0
15	idx_02	0.72	30	26	2308	2	2	6	0	0	0	20
16	idx_20	0.55	10	10	1141	0	10	0	0	0	0	0
17	idx_14	0.26	9	9	2237	0	1	0	0	0	0	8
18	idx_06	0.26	4	4	1144	4	0	0	0	0	0	0

Figure B.8: *PII Exposure tab showing KPI panels, trend charts, and per-index hit-count summary for the most common detector categories*

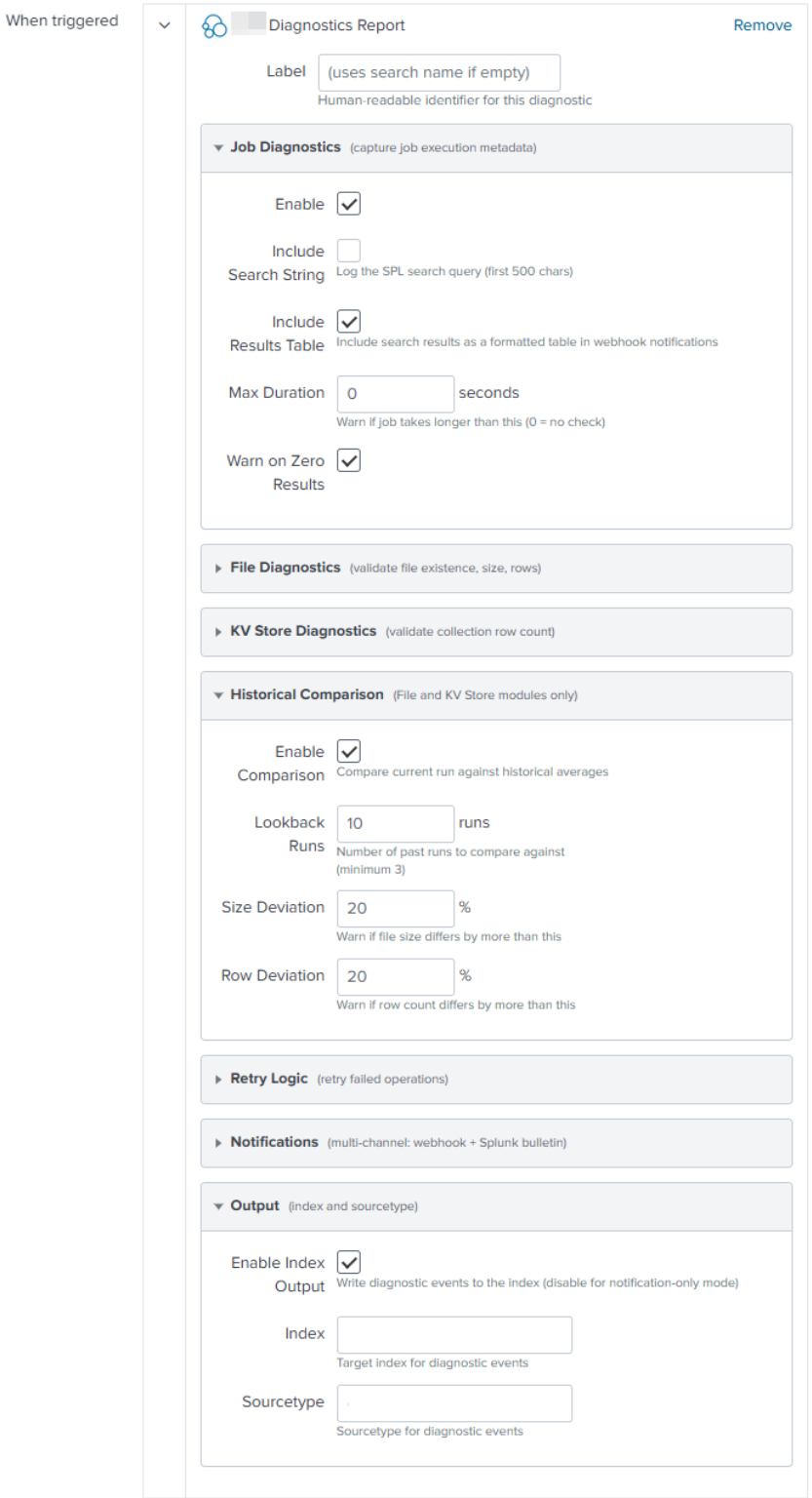


Figure B.9: *Custom alert action configuration interface for* `uso_TA_diagnostics`*, showing the seven parameter groups: job diagnostics, file diagnostics, KV Store diagnostics, historical comparison, retry logic, notifications, and output configuration*

C

Usage Examples

This appendix presents annotated SPL examples drawn from the production dashboard and saved searches described in Chapter 5. Each listing illustrates a distinct integration pattern across the dashboard and its supporting automation. Host names, network addresses, and internal identifiers have been anonymised throughout.

C.1 Basic Command Invocation

The simplest usage of the command framework consists of a single invocation specifying a module and action. Listing 16 shows the base search that populates the Extractor tab with log-check status rows.

Listing 16: *Basic `usoapi` invocation for extractor log monitoring*

```
<search id="base_extractor_log_check" depends="$extractor_visited$">
  <query>| usoapi module=extractor action=log_check</query>
</search>
```

C.2 Splunk Module Query and Output

The following example illustrates a direct `splunk` module query against a search-head. Listing 17 shows the invocation; Table C.1 presents a representative eight-row result. Hostnames and operational identifiers have been anonymised.

Listing 17: *`usoapi` Splunk summary query against a search-head*

```
| usoapi module=splunk action=summary target=sh
| table hostname component status message
| head 8
```

Table C.1: *Representative output of the Splunk summary query (anonymised)*

hostname	component	status	message
host-sh-01	server	ok	Splunk X.X.X operational
host-sh-01	health	critical	Overall health: red
host-sh-01	kvstore	ok	KV Store operational (port 8191)
host-sh-01	queues	ok	All queues flowing normally
host-sh-02	server	ok	Splunk X.X.X operational
host-sh-02	health	critical	Overall health: red
host-sh-02	kvstore	ok	KV Store operational (port 8191)
host-sh-02	queues	ok	All queues flowing normally

C.3 Cross-Module Data Fusion

The host-status table on the Infrastructure tab fuses data from three modules and a lookup file into a single view. The search begins with a Satellite host inventory query, then enriches each row with Ansible web-UI checks, reboot status, and Splunk version information through successive left joins. Listing 18 presents an abridged version of this search.

Listing 18: *Cross-module host-status search (abridged)*

```
| usoapi module=satellite action=hosts
    target="name~groupA or name~legacyGroup" count=200
| eval sat_status=case(
    global_status="OK", "OK",
    global_status="Warning" AND status_reason!="",
        "Warn (".status_reason.")",
    global_status="Warning", "Warning",
    status_reason!="", "Error (".status_reason.")",
    1==1, "Error"
)
| join type=left ip [
    | usoapi module=ansible action=web_ui target="$ansible_target$"
    | rename ip_address as ip
    | eval splunk_web=case(
        web_ui_status="up", "OK",
        web_ui_status="down", "Down", 1==1, "N/A")
    | table ip, splunk_web
]
| join type=left ip [
    | usoapi module=ansible action=needs_restarting
        target="$ansible_target$"
    | rename ip_address as ip
    | eval reboot_needed=case(
        needs_restarting="Yes", "Yes", 1==1, "No")
    | table ip, reboot_needed
]
| join type=left ip [
    | usoapi module=splunk action=info target=all
    | rename ip_address as ip
    | table ip, rest_version
]
| join type=left name [
    | inputlookup forwarder_assets_lookup
    | stats latest(version) as dmc_version by hostname
    | rename hostname as name
    | table name, dmc_version
]
| eval splunk_version=coalesce(rest_version, dmc_version, "N/A")
| table name, ip, os_short, splunk_version, sat_status,
    splunk_web, uptime_display, reboot_needed
| rename name as "Hostname", ip as "IP", sat_status as "Satellite",
    splunk_web as "Splunk Web", reboot_needed as "Reboot"
```

C.4 Search Priority Queue

The dashboard uses custom JavaScript optimisations built on top of Simple XML's token mechanism to sequence searches into seven priority stages, from fast lookups through to slow vulnerability scans (described in detail in Section 5.4.3). When an analyst selects a tab, the controller releases the tokens that tab's panels require ahead of the natural queue, staggering them at 1.5-second intervals to avoid concurrent overload. Unvisited tabs preload in priority order after a 20-second initial delay, with 5-second intervals between stages. Listing 19 shows two consecutive tiers: a fast lookup completes and sets a token that gates a moderate-cost search.

Listing 19: *Token dependency chain for search priority sequencing*

```
<search id="priority_fast_lookup" depends="$workorders_ready$">
  <query>
| inputlookup vulnerability_lookup
| search department="PROJECT"
| stats count
  </query>
  <done>
    <set token="fast_lookups_done">1</set>
  </done>
  <fail>
    <set token="fast_lookups_done">1</set>
  </fail>
</search>

<search id="priority_moderate_stats" depends="$fast_lookups_done$">
  <query>
| rest /services/server/info
| head 1
  </query>
  <done>
    <set token="moderate_stats_done">1</set>
  </done>
</search>
```

The `<fail>` handler ensures the dependency chain continues even if a gated search fails, preventing downstream panels from stalling indefinitely.

C.5 Materialised Snapshot Pipeline

Saved searches materialise point-in-time snapshots into lookup tables, decoupling dashboard rendering from live queries. The index-status snapshot in Listing 20 runs every thirty minutes, computing freshness metrics and writing results to a CSV lookup.

Listing 20: *Materialised index-status snapshot saved search*

```
| tstats latest(_time) as last_event,
    dc(host) as host_count,
    dc(sourcetype) as sourcetype_count
    where index=project_app_* earliest=-7d latest=now by index
| join type=left index
    [| tstats count as event_count_1h
        where index=project_app_* earliest=-1h latest=now by index]
| fillnull value=0 event_count_1h
| eval freshness_min=round((now()-last_event)/60,1),
    snapshot_time=now(),
    health=case(
        freshness_min<60, "healthy",
        freshness_min<360, "delayed",
        1=1, "missing")
| outputlookup index_status_snapshot.csv
```

C.6 PII Detection Integration

The PII detection pipeline samples events from monitored indexes and writes aggregated exposure counts to a summary index. Listing 21 presents the saved search that executes on a thirty-minute schedule, invoking the `piifinder` streaming command to scan sampled events.

Listing 21: *PII detection saved search with per-index sampling (abridged)*

```
| union
    [search index=project_app_auth
        | head 20000
        | eval source_index="project_app_auth"]
    [search index=project_app_waf
        | head 20000
        | eval source_index="project_app_waf"]
    [search index=project_app_*
        NOT index=project_app_auth
        NOT index=project_app_waf
        NOT index=project_app_internal_*
        | head 20000
        | eval source_index=index]
| piifinder mode=counts min_confidence=medium
| where pii_total > 0
| stats
    sum(pii_total) as total_pii_hits
    sum(email_count) as email_hits
    sum(phone_count) as phone_hits
    sum(nif_count) as nif_hits
    sum(credentials_count) as credentials_hits
    dc(_raw) as events_with_pii
    by source_index
| eval scan_time=now(), events_sampled=20000
| collect index=project_diag_monitoring
    sourcetype=pii_summary
```

C.7 API Cache Materialisation

The Suspicious Activity tab enriches external IP addresses with reputation data from AbuseIPDB using the official Splunk integration (AbuseIPDB, 2024). To prevent exceeding daily API limits during dashboard refreshes, a cache-update pipeline materialises results into a KV Store collection. Listing 22 presents the maintenance search that identifies new or stale IP addresses and updates the cache.

Listing 22: *AbuseIPDB cache-update saved search*

```
| inputlookup log_sources_status.csv
| search category="perimeter"
| tstats count WHERE (index=firewall_*) by src_ip
| lookup abuseipdb_cache ip as src_ip OUTPUT cachedAt
| where isnull(cachedAt) OR (now() - cachedAt) > 86400
| head 500
| abuseipdbcheck ip=src_ip
| eval cachedAt=now()
| outputlookup append=true abuseipdb_cache
```

D

Data Protection and Operational Security

This appendix documents the operational security controls and anonymisation practices used to protect sensitive information during system execution and project report preparation.

D.1 Anonymisation Methodology

To protect the confidentiality of the organisation's infrastructure, telemetry, logs, screenshots, and exported artefacts were sanitised through a deterministic but artefact-specific scheme rather than a single universal masking rule:

- **IP Addresses**: Export scripts replace addresses with stable placeholders chosen for the target artefact. Some outputs use RFC 5737 example ranges such as `192.0.2.x` or `198.51.100.x`, whilst others retain private-range-shaped placeholders such as `192.168.1.x` where this better preserves the visual structure of dashboards and alerts.
- **Hostnames and Domains**: Replacement values preserve the original artefact's semantics. Fully qualified names become synthetic domains such as `*.example.com`, whilst infrastructure-oriented views use simplified labels that retain grouping cues without exposing production naming conventions.
- **Personally Identifiable Information (PII)**: Detected PII has been redacted using the `[REDACTED:TYPE]` format as implemented by the PII detection module.

D.2 Sanitised Log Examples

The following sanitised snippets illustrate the system's logging structure without reproducing verbatim production events. They show how execution context can be recorded without disclosing operational secrets.

D.2.1 Audit Log (Success)

```
time="2026-04-15T14:32:01Z" level=INFO component=executor
module=ansible action=test target=ansible-01.example.com status=success
user=a.smith_admin duration=1.2s bytes_out=450
```

D.2.2 Error Log (Sanitised)

```
time="2026-04-15T14:35:12Z" level=ERROR component=rest_client
module=splunk action=health target=idx-01.example.com status=failed
error_code=HTTP_403 message="Forbidden: check token capabilities"
```

D.3 Operational Security Checklist

Administrators maintaining this system should periodically review the following controls to confirm that the deployed configuration still matches the architecture described in Chapters 4 and 5:

- **Token Rotation and Revocation:** When a monitored Splunk instance rotates its bearer token or a service account changes, the superseded entry should be removed from `storage/passwords` through the token manager so that obsolete credentials cannot be reused accidentally.
- **Role Review:** The `sec-mon-access` role should retain only the capabilities required by the implemented monitoring actions, matching the least-privilege design described in Section A.3.5.
- **SSH Trust Model:** Search head cluster members should retain the expected known-host entries for the Ansible control node and other outbound SSH destinations. The `accept-new` policy is a deliberate TOFU trade-off: it allows first contact without manual pre-distribution, but any subsequent host-key change should still be investigated.
- **PII Review:** The PII Exposure dashboard should be reviewed for new detector categories, confidence shifts, or repeated matches in the same index, since these changes may indicate either genuine data exposure or detector drift.

E

PII Detection Evaluation Corpus

The synthetic evaluation corpus quantifies detector correctness for the PII detection subsystem described in Section 6.6.5. The corpus contains 25 test cases designed to exercise both positive detection (valid PII patterns) and negative rejection (false-positive patterns and validator edge cases). All examples use synthetic or clearly masked data.

E.1 Positive Test Cases

Positive cases embed known PII instances across all 14 detector types. Each case specifies the detector type and expected match span.

Case ID	Input Text	Expected Detection
email_basic	Contact user@example.com for support.	Email: user@example.com
phone_basic	Subscriber phone: 912345678 answered the call.	Phone: 912345678
nif_basic	Cliente com NIF 123456789 validado no portal.	NIF: 123456789
credentials_basic	password=MySecret123 was rotated yesterday.	Credentials: password=MySecret123
address_basic	Morada: Rua da Liberdade, 123, Lisboa.	Address: Rua da Liberdade, 123
structured_name_basic	Nome=João Silva.	Structured Name: Nome=João Silva
cc_basic	credit_card: 4111111111111111.	Credit Card: 4111111111111111
iban_basic	IBAN PT5000020123123456789O154 confirmado.	IBAN: PT5000020123123456789O154
niss_basic	niss: 12345678901 processado sem erros.	NISS: 12345678901
ipv4_basic	client_ip 192.168.1.1 acedeu ao portal.	IPv4: 192.168.1.1
citizen_card_basic	cartao cidadao 123456789AB1 validado.	Citizen Card: 123456789AB1
vehicle_plate_basic	Viatura com matrícula AA-00-AA entrou no parque.	Vehicle Plate: AA-00-AA
dob_basic	date_of_birth: 1990-01-15 registada no formulário.	DOB: 1990-01-15
jwt_basic	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 expirou.	JWT: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
mixed_positive	Contact masked.person@example.com, payment card_number 4111111111111111, IBAN PT50000201231234567890154 and client_secret=UltraSecret987 in one ticket.	Email, CC, IBAN, Credentials
Total	**15 positive cases**	**18 gold spans**

Table E.1: *Positive test cases: expected PII detections*

E.2 Negative Test Cases

Negative cases contain patterns that should NOT be detected as PII. These test common false-positive patterns (to verify rejection logic) and validator edge cases (to verify algorithmic validation).

Case ID	Input Text	Expected Result
negative_clean	System health OK with no personal data present.	No detection
negative_nif_timestamp	timestamp=1710345678 level=INFO msg=hello	No detection (timestamp, not NIF)
negative_ipv4_localhost	remote_addr 127.0.0.1:8089	No detection (localhost)
negative_phone_placeholder	msisdn: 910000000	No detection (placeholder)
negative_cc_build	version 4111111111111111 build	No detection (build number)
negative_nif_invalid_checksum	NIF 123456788 processed successfully.	No detection (invalid checksum)
negative_phone_placeholder_explicit	msisdn: 910000000 registered in system.	No detection (placeholder)
negative_email_internal_subdomain	Contact admin@internal.corp.example.com for help.	No detection (internal domain)
negative_dob_non_standard_format	birth: 14.2.1981	No detection (non-standard format)
negative_iban_wrong_checksum	IBAN PT5000020123123456789015 5 para pagamento.	No detection (invalid checksum)
Total	**10 negative cases**	**0 expected detections**

Table E.2: *Negative test cases: expected rejections*

E.3 Evaluation Results

The corpus was evaluated using exact span-and-type matching: a detection counts as a true positive only when the predicted span boundaries and PII type match the gold label precisely. Table 6.13 in Section 6.6.5 presents the per-detector results. The email detector recorded one false positive on the internal-domain test case (`admin@internal.corp.example.com`), exposing a limitation in the domain rejection logic.

Annexes

L

System Usability Scale Questionnaire

This annex reproduces the System Usability Scale (SUS) instrument (Brooke, 1996) as administered to participants during the usability assessment described in Section 6.6.7. Respondents rated each statement on a five-point Likert scale ranging from "Strongly Disagree" (1) to "Strongly Agree" (5).

Table L.1: *System Usability Scale questionnaire items (Brooke, 1996)*

#	Statement	SD	D	N	A	SA
1	I think that I would like to use this system frequently.	○	○	○	○	○
2	I found the system unnecessarily complex.	○	○	○	○	○
3	I thought the system was easy to use.	○	○	○	○	○
4	I think that I would need the support of a technical person to be able to use this system.	○	○	○	○	○
5	I found the various functions in this system were well integrated.	○	○	○	○	○
6	I thought there was too much inconsistency in this system.	○	○	○	○	○
7	I would imagine that most people would learn to use this system very quickly.	○	○	○	○	○
8	I found the system very awkward to use.	○	○	○	○	○
9	I felt very confident using the system.	○	○	○	○	○
10	I needed to learn a lot of things before I could get going with this system.	○	○	○	○	○

Scoring. For odd-numbered items, the score contribution is the response value minus 1. For even-numbered items, the contribution is 5 minus the response value. The sum of all ten contributions is multiplied by 2.5 to yield a final score on a 0–100 scale (Brooke, 1996).