International Journal of Creative and Open Research in Engineering and Management
ISSN: 3108-1754 (Online)
Volume 02 Issue 09 September-2026 | Impact Factor: 3.5

SecureXon: Design, Architecture, and Evaluation Methodology for an AI-Augmented Web Reconnaissance and Cybersecurity Threat-Intelligence Platform

Sahil Bagde[1], Swapnil Meshram[2], Harish Dange[3], Mansi Mate[4], Prof. Komal Tiwari[5], Dr. Sushama Telrandhe[6]

[1,2,3,4] U.G. Students, Department of Computer Science and Engineering, Guru Nanak Institute of Engineering and Technology, Nagpur, India

[5] Assistant professor, Department of Computer Science and Engineering, Guru Nanak Institute of Engineering and Technology, Nagpur, India

[6] Associate professor, Department of Computer Science and Engineering, Guru Nanak Institute of Engineering and Technology, Nagpur, India

How to Cite this Article:

Bagde, S., Meshram, S., Dange, H., Mate, M. & Tiwari, K. (2026). SecureXon: Design, Architecture, and Evaluation Methodology for an AI-Augmented Web Reconnaissance and Cybersecurity Threat-Intelligence Platform. International Journal of Creative and Open Research in Engineering and Management, 02(9), 1-9. https://doi.org/10.55041/ijcope.v2i9.004

https://doi.org/10.55041/ijcope.v2i9.004

***Abstract*—** The rapid expansion of internet-facing web applications has widened the attack surface available to automated scanners, botnets and malicious actors, while common weaknesses such as misconfigured servers, unpatched software, obsolete transport-layer encryption and missing HTTP security headers continue to be exploited at scale. Commercial vulnerability scanners are costly and largely opaque, whereas open-source command-line utilities operate independently of one another and demand specialised expertise, offering little contextual or remediation guidance. This paper presents the design of SecureXon , a modular, full-stack security reconnaissance and threat-intelligence platform built around a Python/Flask backend that consolidates fifteen asynchronous reconnaissance modules with a large-language-model-driven Security Operations Center (SOC) assistant for false-positive vulnerability filtering and remediation guidance. A dedicated Zero-Trust defensive subsystem, the SSRF Guard, validates every outbound network request against loopback, private, link-local, multicast and encoded IP representations before it is dispatched. A companion log-analysis engine maps detected attack signatures in Nginx/Apache traffic to the MITRE ATT&CK knowledge base. Beyond the system design, this paper contributes a normalised risk-scoring formulation, an architecture and workflow specification, a structured SSRF bypass test-vector suite, and a precision/recall/F1-based evaluation protocol for the AI-assisted CVE triage stage. As the platform is currently at the design-and-development stage, the paper specifies evaluation protocols for quantitative validation rather than reporting unmeasured performance results.

***Index Terms*—** *Web reconnaissance, vulnerability assessment, artificial intelligence in security operations, Server-Side Request Forgery, CVE triage, MITRE ATT&CK, SSL/TLS auditing, log analysis, asynchronous scanning.*

I. INTRODUCTION

Web application security has become a foundational requirement of modern digital infrastructure [1]. As organisations migrate to cloud-hosted portals, REST APIs and micro-service deployments, the volume and sophistication of attacks directed at these surfaces has grown correspondingly. Security teams must continuously audit public-facing infrastructure for configuration flaws, open service ports, absent security headers, outdated dependencies and weak cryptographic settings before adversaries exploit them.

The present tooling landscape is fragmented: enterprise-grade scanners are expensive and closed-source, while open-source utilities such as Nmap [6] operate in isolation and produce raw, unorganised output that requires manual correlation. SecureXon is proposed as a web-based reconnaissance suite and AI-assisted SOC assistant that addresses this gap. Built on Python, Flask [7] and SQLite, it provides an interactive interface through which a researcher can evaluate the risk profile of a target host, receive an AI-generated executive summary, and obtain concrete

remediation snippets, while a defensive layer prevents the tool itself from being abused to reach internal or cloud-metadata endpoints.

The remainder of this paper is organised as follows. Section II reviews related work. Section III states the problem and objectives. Section IV describes the system architecture with reference to Figs. 1-2. Section V formalises the risk-scoring function. Section VI details the proposed methodology and Section VII the core algorithms. Sections VIII and IX define, respectively, the SSRF Guard test protocol and the AI CVE-triage evaluation protocol. Section X lists the implementation tools. Section XI reports implementation status against the project schedule. Section XII discusses limitations, and Section XIII concludes.

II. RELATED WORK

A. Active Network Reconnaissance

Network reconnaissance - the enumeration of hosts, open ports, running services and operating-system characteristics - is typically the first phase of a security assessment. Established utilities such as Nmap [6] are effective at collecting this technical information, but their raw output is difficult to interpret without additional manual analysis. This has motivated research into automated organisation of reconnaissance data so that it can feed subsequent vulnerability-intelligence stages.

B. Vulnerability Intelligence and CVE Correlation

Public vulnerability databases such as the National Vulnerability Database [4] assign standardised identifiers to known software weaknesses, and scanners routinely correlate detected software versions against these records. However, version-based matching alone is known to generate a substantial proportion of false positives, since a nominally vulnerable component may be unreachable, patched, or protected by compensating controls. Contemporary vulnerability-management practice therefore favours contextual analysis that accounts for the deployment environment rather than a bare version lookup.

C. Server-Side Request Forgery Mitigation

Server-Side Request Forgery (SSRF) allows an attacker to coerce a server into issuing requests on their behalf, potentially reaching internal services or cloud metadata endpoints. PortSwigger Research [14] documents that naive validation based on string or simple regular-expression matching is frequently bypassed by representing target addresses in decimal, hexadecimal, octal or IPv6-mapped form. Robust mitigation instead requires hostname resolution followed by structural inspection of the resolved address against the private-address ranges defined in RFC 1918 [10].

D. AI-Assisted Security Operations

Recent work has examined the use of large language models, including general-purpose models such as Gemini [5], to summarise findings, correlate multi-source telemetry and suggest remediation steps within security-operations workflows. Torkamani et al. [13] show that GPT-based triage of NVD vulnerability records can automate CWE identification and severity assessment, reducing manual analyst effort, while their reported accuracy remains partial (68% CWE-identification accuracy and 51.2% combined CWE-and-severity accuracy), underscoring that model output must be treated as an augmentation layer rather than an authoritative verdict - a concern this paper addresses directly through the evaluation protocol in Section IX.

E. Standardised Threat Frameworks

Industry frameworks such as the OWASP Top Ten [2] and the MITRE ATT&CK Enterprise matrix [3] provide shared vocabularies for categorising web risks and adversary behaviour respectively, allowing findings from disparate tools to be communicated in standardised terms.

F. Research Gap

Taken together, the reviewed literature indicates that reconnaissance, vulnerability scanning, threat classification and reporting are commonly implemented as separate, loosely-coupled tools, increasing operational overhead and leaving false-positive suppression and secure outbound probing as open problems. SecureXon (v2) is motivated by these gaps and integrates all four concerns within a single modular application.

III. PROBLEM STATEMENT AND OBJECTIVES

Four concrete technical problems guided the design of the system. First, synchronous handling of port and directory probes tends to block the serving process, degrading interface responsiveness. Second, unvalidated outbound requests expose the scanning host itself to SSRF abuse through obfuscated address representations, or through requests to the well-known cloud metadata address 169.254.169.254 [14]. Third, public CVE feeds such as the NVD API [4] are rate-limited and return broad keyword-based matches that are frequently inapplicable to the scanned environment. Fourth, storing third-party API credentials without encryption creates an avoidable data-exposure risk.

The corresponding objectives of this work are to: (i) unify DNS resolution, multi-threaded port probing, header auditing [9], TLS inspection [8] and technology fingerprinting within one Flask application; (ii) implement a non-blocking scan queue that reports live progress; (iii) construct a Zero-Trust SSRF defence, specified and tested in Section VIII; (iv) integrate a large-language-model pipeline for executive risk briefing, interactive remediation guidance and CVE false-positive suppression, evaluated in Section IX; (v) build a regex-based log-analysis engine that maps detected attack traffic to MITRE ATT&CK [3] tactics and techniques; and (vi) apply production-grade safeguards, including salted password hashing, symmetric encryption of stored credentials, rate limiting and multi-channel alerting.

IV. SYSTEM ARCHITECTURE

The system follows a layered architecture, shown in Fig. 1. A browser-based client communicates over HTTPS/AJAX with a Flask application instantiated through the application-factory pattern [7]. Requests are routed through a blueprint layer that separates authentication, dashboard rendering, scan-queue and reporting APIs, SOC log endpoints and account settings. A parallel security-and-middleware layer enforces the SSRF Guard, request-rate limiting, CSRF protection, PBKDF2-SHA256 password hashing and Fernet-based encryption of stored API keys. Scan requests that pass validation are handed to an asynchronous worker pool, which executes the fifteen reconnaissance modules against a SQLite persistence layer and, where configured, external services such as the NVD API [4], Shodan [15], and the Gemini [5] and OpenRouter language-model endpoints.

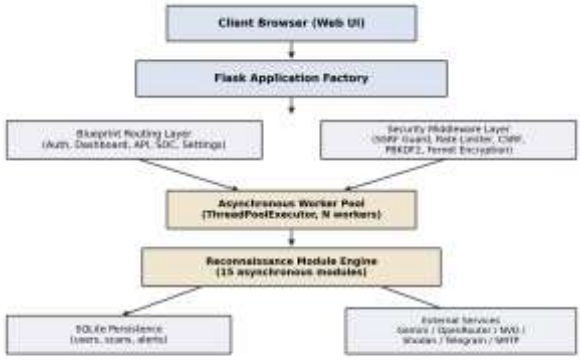

Fig. 1. System architecture of SecureXon (v2), showing the routing, middleware, asynchronous execution and persistence layers.

A. Reconnaissance Pipeline

A scan begins with URL normalisation and hostname extraction, followed by SSRF validation (Fig. 2). Once cleared, a scan record is created with a pending status and dispatched to the worker pool, which executes the enabled modules - IP resolution, HTTP header and technology analysis, threaded TCP port scanning with banner grabbing, subdomain and directory enumeration, TLS handshake inspection [8], DNS SPF/DMARC auditing [11], [12], robots.txt and WHOIS retrieval, CVE correlation, Shodan lookup [15], and cookie/CORS inspection. Each module contributes a bounded deduction to the normalised risk score defined in Section V, after which the record is marked complete and, if configured, an alert is dispatched by e-mail or Telegram.

TABLE I. RECONNAISSANCE MODULE INVENTORY

#	Reconnaissance Module
1	IP Resolution
2	HTTP Header Analysis
3	Technology Fingerprinting
4	TCP Port Scanning
5	Banner Grabbing
6	Subdomain Enumeration
7	Directory Enumeration
8	TLS Inspection
9	SPF Audit
10	DMARC Audit
11	robots.txt / WHOIS Retrieval
12	CVE Correlation
13	Shodan Lookup
14	Cookie Inspection
15	CORS Inspection

Note: this inventory is derived from the module descriptions given in this section and reflects the design-stage specification; if the eventual implementation names or groups modules differently, the implementation's actual module list should be treated as authoritative.

B. Log Analysis Pipeline

Independently of the scan pipeline, uploaded or simulated Nginx/Apache access logs are parsed line-by-line with a regular-expression pattern, URL-decoded, and evaluated against both user-defined and built-in signatures covering SQL injection, cross-site scripting, path traversal, brute-force and directory-fuzzing attempts. Matches are deduplicated, tagged with the corresponding MITRE ATT&CK [3] tactic and technique identifiers (for example, Initial Access / T1190 and Credential Access / T1110), and rendered on a SOC dashboard.

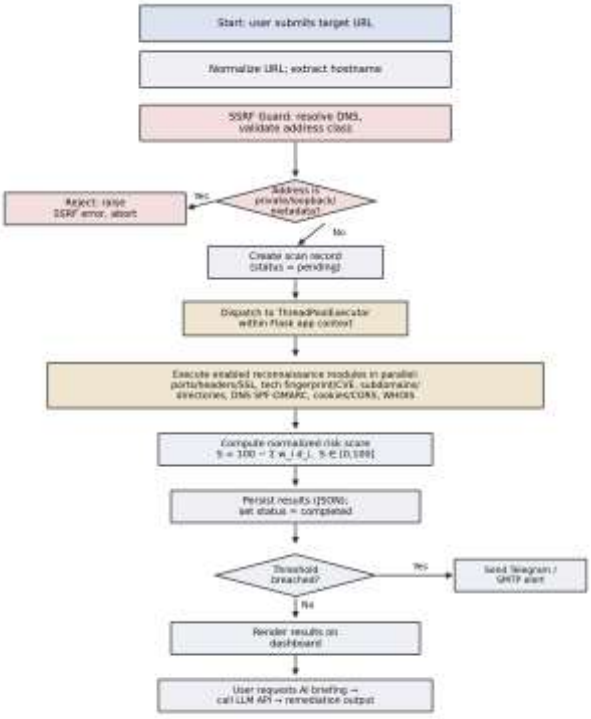

Fig. 2. Asynchronous scan-execution workflow, from URL submission through SSRF validation, parallel reconnaissance, risk scoring, alerting and AI-assisted remediation.

V. RISK-SCORE FORMULATION

Each scan is summarised by a single normalised risk score S so that heterogeneous module findings can be compared and thresholds applied consistently. Let M denote the set of k enabled reconnaissance modules m1 ... mk, and let di in [0, 1] denote the normalised severity of the findings returned by module mi (0 = no issue found, 1 = maximum severity for that module's finding class). Each module is assigned a design weight wi, reflecting its relative contribution to overall exploitability risk, subject to the sum of all wi equal to 100. S is defined as an aggregate Security Score rather than a residual-risk index: it starts at a ceiling of 100 and is reduced by each weighted deduction, so a higher value of S denotes a more secure target and a lower value denotes greater residual risk. The risk score is then computed as:

$$S = 100 - \Sigma i\ (wi \cdot di), \qquad S \in [0, 100] \quad (1)$$

and the score is subsequently mapped to a discrete risk level: Low for S >= 80, Medium for 50 <= S < 80, and High for S < 50. Table II lists the design-stage weighting scheme; these weights are treated as tunable parameters rather than fixed constants - Section XI proposes calibrating them against analyst-labelled scan outcomes once field data is available, rather than presenting them as already-validated coefficients.

TABLE II. DESIGN-STAGE RISK-DEDUCTION WEIGHTING SCHEME

Module Category	Weight wi (%)	Deduction Trigger (di = 1 condition)
Open high-risk ports / weak services	20	Sensitive service (e.g., DB port) reachable without auth
Missing OWASP security headers	15	All 7 audited headers absent
TLS/SSL misconfiguration	15	Expired cert, weak cipher, or deprecated protocol
Unmitigated CVE (post-AI filtering)	20	High-severity CVE retained after context filtering
SPF/DMARC absence	10	Both records absent for the domain
Exposed directories / subdomains	10	Sensitive path or unintended subdomain reachable
Cookie / CORS misconfiguration	10	Wildcard CORS with credentials, or insecure cookie flags

It is important to distinguish the fifteen reconnaissance execution modules described in Section IV-A from the seven risk-scoring categories of Table II: the fifteen modules are the operational units the worker pool actually runs against a target (IP resolution, header/technology analysis, port scanning, subdomain/directory enumeration, TLS inspection, SPF/DMARC auditing, robots.txt/WHOIS retrieval, CVE correlation, Shodan lookup, and cookie/CORS inspection, among others), whereas the seven categories in Table II are an aggregation layer that groups the findings of one or more related modules for the purpose of weighting and score deduction. For example, the "Open high-risk ports / weak services" category consumes output from the port-scanning module alone, while the "Exposed directories / subdomains" category aggregates findings from both the subdomain-enumeration and directory-enumeration modules. The module count (fifteen) and the category count (seven) therefore describe two different layers of the pipeline - execution granularity versus scoring granularity - and are not intended to correspond one-to-one.

VI. PROPOSED METHODOLOGY

1) Application factory and modularisation: The entry point instantiates the Flask object through the application-factory pattern [7], wires session, rate-limiting, form-protection and mail extensions, and registers blueprints for authentication, dashboard, API, SOC and settings functionality.

2) Persistence layer: A SQLite database, accessed exclusively through prepared statements, stores user accounts, encrypted credentials, scan queues, JSON scan results, SOC alerts, custom detection rules and scheduled tasks.

3) Credential protection: External API keys are encrypted at rest with symmetric Fernet encryption keyed from the application secret, so that a leaked database file alone does not expose usable credentials.

4) Asynchronous execution: A fixed-size thread-pool executor accepts scan tasks and runs them inside a preserved application context, updating a pending/running/completed lifecycle that the interface polls via AJAX without blocking the user session.

5) Zero-Trust SSRF guard: Every outbound probe is preceded by hostname resolution and structural validation against loopback, private [10], link-local, multicast, decimal- and hexadecimal-encoded address forms, blocking access to internal services and cloud metadata endpoints, per the protocol formalised in Section VIII.

6) Reconnaissance engine: Fifteen loosely-coupled modules perform IP resolution, HTTP and technology analysis, port scanning, subdomain/directory enumeration, TLS inspection [8], mail-authentication auditing [11], [12], WHOIS/robots retrieval, CVE correlation, threat-intelligence lookup [15], and cookie/CORS inspection.

7) Contextual AI pipeline: Scan output is serialised into structured prompts submitted to a hosted language model [5] for executive summaries, interactive remediation chat and configuration-snippet generation, while a second model service filters CVE matches against the detected technology profile, per the protocol in Section IX.

8) Log and threat-mapping engine: Access logs are parsed with regular expressions, matched to known attack signatures, and tagged with MITRE ATT&CK [3] identifiers for standardised reporting.

9) Reporting and alerting: Findings are exported as PDF reports containing risk scores and remediation notes, with configurable e-mail and Telegram alerts triggered when a risk threshold is crossed.

10) Verification: The implementation will be validated through module-level unit tests, the SSRF test-vector suite of Section VIII, end-to-end pipeline tests, and replay of simulated attack traffic through the log engine.

VII. ALGORITHMS

Algorithm 1 - Asynchronous Scan Execution with SSRF Protection

1. Receive the scan request containing the target URL and enabled-module flags.

2. Normalise the URL and extract protocol and hostname.

3. Resolve the hostname and validate the resulting address against loopback, private, link-local, multicast and encoded representations; abort with an SSRF error on any match.

4. Create a scan record with status pending and obtain a scan identifier.

5. Submit the scan task to the thread-pool executor within the application context.

6. Execute the enabled reconnaissance modules and accumulate deductions into the bounded risk score S of Eq. (1).

7. Classify risk level as Low (S >= 80), Medium (50 <= S < 80) or High (S < 50).

8. Persist the result JSON and mark the record completed.

9. Dispatch an alert if the configured threshold is breached.

Algorithm 2 - Log-Based Threat Detection and ATT&CK Mapping

1. Receive an uploaded or simulated access-log file.

2. Parse each line with the log-record regular expression and URL-decode the request path.

3. Evaluate the line against user-defined detection rules.

4. Evaluate the line against built-in signatures for injection, cross-site scripting, path traversal, brute-force and directory-fuzzing behaviour.

5. Tag matching lines with the corresponding MITRE ATT&CK [3] tactic and technique.

6. Deduplicate and store the resulting alerts for display on the SOC dashboard.

VIII. SSRF GUARD: THREAT MODEL AND TEST PROTOCOL

Because SecureXon issues outbound network requests on behalf of its users, the tool itself is a potential SSRF vector unless every probe is validated before dispatch [14]. The SSRF Guard therefore intercepts each outbound target, resolves it via hostname lookup, and rejects it if the resolved address falls within any reserved or internal range. Table III specifies the bypass-technique test-vector suite designed to exercise this guard; each vector encodes a loopback, private-network [10], or cloud-metadata address in a form that a naive string filter would typically miss. The "Required Outcome" column states the specification the implementation must satisfy - it is a pass/fail acceptance criterion for the test phase, not a reported measurement. Section XI records the actual pass rate once this suite has been executed against the implemented guard.

TABLE III. SSRF GUARD BYPASS TEST-VECTOR SUITE (PLANNED ACCEPTANCE CRITERIA)

Vector Class	Example Payload	Bypass Technique	Required Outcome
Decimal-encoded loopback	http://2130706433/	Integer-encoded IPv4	Rejected
Hexadecimal loopback	http://0x7f000001/	Hex-encoded IPv4	Rejected
Octal loopback	http://0177.0.0.1/	Octal-encoded octet	Rejected
IPv4-mapped IPv6	http://[::ffff:127.0.0.1]/	IPv6 mapping of loopback	Rejected
IPv6 loopback	http://[::1]/	Native IPv6 loopback	Rejected
Link-local range	http://169.254.1.1/	RFC 3927 link-local	Rejected
Cloud metadata endpoint	http://169.254.169.254/	AWS/GCP/Azure metadata IP	Rejected
RFC 1918 private range	http://10.0.0.5/, http://192.168.1.1/	Private-network address [10]	Rejected
DNS rebinding attempt	Domain resolving to a private IP at request time	Time-of-check/time-of-use gap	Rejected
Legitimate public host	http://example.com/	N/A (control case)	Accepted

A vector is scored as Pass only if the corresponding request is rejected before any socket is opened to the resolved address (for reject cases) or is correctly allowed through (for the control case). The full suite will be scripted so that it can be re-run automatically after any change to the validation logic, guarding against regression.

IX. AI-ASSISTED CVE TRIAGE: EVALUATION PROTOCOL

The CVE false-positive filtering stage forwards each candidate CVE, together with the detected target technology profile, to an LLM service and retains only the matches the model judges to be contextually applicable [13]. Because LLM output can be inaccurate, this stage must be evaluated against a human-labelled ground truth rather than accepted at face value.

A. Ground-Truth Construction

A stratified sample of targets spanning common web technology stacks will be scanned, and every CVE candidate returned by the raw NVD lookup will be manually classified by a reviewer with security-assessment experience as a true positive (genuinely applicable to the target's configuration) or false positive (inapplicable due to patching, non-exposure, or version mismatch). This manual label set forms the evaluation ground truth and is kept separate from the filtering model.

B. Metrics

Let TP be the count of CVEs correctly retained by the AI filter, FP the count of inapplicable CVEs incorrectly retained, and FN the count of applicable CVEs incorrectly discarded. The evaluation reports:

$$Precision = TP / (TP + FP) \quad (2)$$

$$Recall = TP / (TP + FN) \quad (3)$$

$$F1 = 2 \cdot (Precision \cdot Recall) / (Precision + Recall) \quad (4)$$

The false-positive reduction rate relative to the unfiltered NVD baseline is additionally reported as (FP_baseline - FP_filtered) / FP_baseline. The original project brief targets an approximate 60-70% reduction in false-positive CVE entries; this figure is stated here as a design target for the implementation phase and is explicitly not a measured result - Table IV shows the reporting template that will be populated once the labelled evaluation set has been scored.

TABLE IV. CVE-TRIAGE EVALUATION REPORTING TEMPLATE (TO BE POPULATED DURING TESTING)

Metric	Baseline (raw NVD)	With AI Filter	Status
Precision	—	—	*Pending evaluation*
Recall	—	—	*Pending evaluation*
F1-score	—	—	*Pending evaluation*
False-positive count / target	—	—	*Pending evaluation*
FP reduction vs. baseline	N/A	—	*Target: 60-70%*

X. IMPLEMENTATION TOOLS AND PLATFORMS

The prototype is implemented in Python 3.10+ for backend logic and ES6 JavaScript, HTML5 and CSS3 for the client interface. The Flask 3.x framework [7] is extended with session, rate-limiting, form-protection and mail extensions. Persistence uses SQLite3 accessed through prepared statements. Credential protection relies on the Python cryptography library for symmetric encryption and Werkzeug's PBKDF2-SHA256

hashing, together with the standard-library address module for SSRF validation. Reporting uses ReportLab for PDF generation, and outbound integrations include the requests, socket and ssl [8] libraries, a WHOIS client, a Shodan SDK client [15], and the Telegram Bot API. Development uses Visual Studio Code and Git/GitHub, with cross-platform target support for Linux, Windows and macOS. A recommended runtime environment is a dual-core processor at 2.0 GHz or above, 4-8 GB of RAM, under 500 MB of disk space, and a stable broadband connection.

XI. IMPLEMENTATION STATUS AND EVALUATION PLAN

At the time of writing this remains a design-and-development-stage project: the architecture, algorithms, risk-scoring formulation and evaluation protocols in Sections IV-IX are finalised, while implementation follows the phased schedule in Table V. No component has yet completed the full test protocols of Sections VIII and IX, and this paper accordingly reports no simulated or placeholder performance numbers in their place. Once each phase is complete, the following measurements will be taken and reported in a subsequent revision: (i) SSRF Guard pass rate against the full vector suite of Table II; (ii) precision, recall and F1-score of the CVE-triage stage against the labelled ground truth of Section IX, together with the realised false-positive reduction rate; (iii) end-to-end scan latency distribution (median and 95th-percentile) across the fifteen reconnaissance modules; and (iv) calibration of the module weights in Table II against analyst-labelled scan outcomes.

TABLE V. PROJECT IMPLEMENTATION SCHEDULE (2026-27)

Phase / Activity	Duration	Timeline
Requirement analysis & literature survey	3 weeks	Aug-Sep 2026
System design & architecture finalisation	2 weeks	Sep 2026
Backend development (Flask, SQLite, SSRF guard)	4 weeks	Oct 2026
Reconnaissance module implementation	4 weeks	Nov 2026
AI integration (LLM services)	3 weeks	Dec 2026
SOC log engine & ATT&CK mapping	2 weeks	Jan 2027
Interface development	2 weeks	Feb 2027
Testing against Sections VIII-IX protocols	3 weeks	Mar 2027
Final report, presentation & demonstration	2 weeks	Apr 2027

XII. LIMITATIONS

• Quantitative claims in Sections VIII and IX are, at the current stage, acceptance criteria and evaluation protocols rather than measured outcomes; the reported target figures (e.g., the 60-70% false-positive reduction goal) are design targets carried over from the initial project brief and require empirical confirmation.

• The AI-assisted stages depend on third-party LLM APIs (Gemini, OpenRouter); model outputs may be inconsistent across API versions, subject to rate limiting, and are not guaranteed to be free of hallucinated or unsupported claims, which is precisely why Section IX evaluates them against human-labelled ground truth rather than trusting them directly.

• The SSRF Guard's address-validation logic is defined for the vector classes in Table III; exhaustive coverage of every possible encoding (including future or platform-specific quirks in DNS or IP-address parsing) cannot be guaranteed and should be extended as new bypass techniques are documented in the literature.

• Scan throughput is bounded by the size of the thread pool and by SQLite's write-concurrency characteristics; the current design targets single-node, small-to-medium deployment rather than enterprise-scale concurrent scanning.

• The log-analysis engine relies on regular-expression signatures and is therefore inherently limited to known attack patterns; it is not designed to detect novel or heavily obfuscated payloads that fall outside the signature set.

• CVE and threat-intelligence freshness is bounded by the rate limits and update cadence of the underlying third-party feeds (NVD, Shodan).

• No formal usability study of the dashboard interface has been conducted; interface design choices are based on developer judgement rather than user testing.

XIII. CONCLUSION AND FUTURE SCOPE

This paper presented the design of SecureXon (v2), a modular web-security reconnaissance and SOC-assistant platform that unifies asynchronous multi-module scanning, a Zero-Trust SSRF defence, large-language-model-assisted vulnerability triage and remediation, and MITRE ATT&CK-aligned log analytics within a single lightweight application. Beyond the architecture, this paper contributes a formal risk-scoring function, an SSRF bypass test-vector suite, and a precision/recall/F1 evaluation protocol for AI-assisted CVE triage, so that the system's claimed benefits can be verified rather than asserted. Future work includes executing the protocols of Sections VIII and IX against the completed implementation, distributed scan execution across multiple worker nodes, integration of additional threat-intelligence feeds, and formal benchmarking of false-positive reduction and mean-time-to-remediation against baseline open-source tool chains.

REFERENCES

[1] M. E. Whitman and H. J. Mattord, Principles of Information Security, 6th ed. Boston, MA, USA: Cengage Learning, 2018.

[2] OWASP Foundation, "OWASP Top 10:2021 - The Ten Most Critical Web Application Security Risks," 2021.

[3] MITRE Corporation, "MITRE ATT&CK Enterprise Matrix," 2023.

[4] National Institute of Standards and Technology, "National Vulnerability Database API v2 Specification," NIST NVD, 2023.

[5] Google DeepMind, "Gemini 2.0 Flash Model Card and API Reference," Google AI, 2024.

[6] G. F. Lyon, Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Sunnyvale, CA, USA: Insecure.Com LLC, 2009.

[7] M. Grinberg, Flask Web Development: Developing Web Applications with Python, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2018.

[8] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF RFC 8446, Aug. 2018.

[9] R. Fielding et al., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," IETF RFC 7230, Jun. 2014.

[10] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address Allocation for Private Internets," IETF RFC 1918, Feb. 1996.

[11] S. Kitterman, "Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1," IETF RFC 7208, Apr. 2014.

[12] M. Kucherawy and E. Zwicky, "Domain-based Message Authentication, Reporting, and Conformance (DMARC)," IETF RFC 7489, Mar. 2015.

[13] M. J. Torkamani, J. Ng, N. Mehrotra, M. Chandramohan, P. Krishnan, and R. Purandare, "Streamlining Security Vulnerability Triage with Large Language Models," arXiv preprint arXiv:2501.18908, 2025.

[14] PortSwigger Research, "Introducing the URL Validation Bypass Cheat Sheet," PortSwigger Ltd., 2024. [Online]. Available: https://portswigger.net/research/introducing-the-url-validation-bypass-cheat-sheet

[15] Shodan.io, "Shodan REST API Developer Documentation," 2023.