University of Tunis El Manar

National Engineering School of Tunis

Department of Information and Communication Technologies

INTERNSHIP REPORT

Third-Year Computer Engineering

AI-Powered Penetration Testing Command Framework

Host Company

KEYSTONE

Tunis, Tunisia

Prepared by: Wael HAMMALI

Class: Third-Year Computer Engineering 1

Supervised by: Mr. Firas CHAOUCH

Academic Year: 2026–2027

Acknowledgements

I would like to express my sincere gratitude to my internship supervisor, Firas CHAOUCH, for supporting my work on this project. I am grateful for the opportunity to develop my engineering skills and to examine the practical challenges of AI-assisted cybersecurity assessment.

I also thank Keystone for hosting this internship and providing the professional setting in which the project was carried out. This experience allowed me to connect my academic studies with a concrete engineering problem involving security tools, artificial intelligence, and software integration.

Finally, I express my appreciation to the National Engineering School of Tunis for the academic education that provided the foundation for this work.

Abstract

Cybersecurity assessments rely on specialized tools whose outputs differ in structure and level of detail. When reconnaissance and subsequent investigation operate as separate processes, useful context can be lost, results require repeated interpretation, and discovery tasks may be duplicated. This internship project examines how an AI-assisted workflow can connect these stages while retaining explicit boundaries between observations, hypotheses, and reviewed findings.

The proposed framework combines IntelForge, which collects reconnaissance data and produces structured intelligence, with Voidhawk, which supports iterative tool selection, persistent operational memory, analysis, and finding review. DarkIntel provides the integration layer: it transforms selected reconnaissance results into observations, reasoning memories, and initial planning context, then coordinates downstream execution and report generation. Dragon provides an additional narrative synthesis. The principal contribution is the engineering integration of complementary subsystems built on existing security tools and software frameworks, rather than a new vulnerability-detection model.

Evaluation relies on offline and mocked software tests and controlled local checks of implementation behavior. This evidence supports selected data transformations, persistence operations, workflow transitions, and reporting paths; it does not establish real-world detection accuracy or a comparative performance advantage. Remaining limitations concern validation status, evidence provenance, session isolation, and the completeness of follow-up analysis. The project provides an inspectable integration prototype and identifies concrete improvements needed before broader security effectiveness can be assessed through controlled experiments.

Keywords: AI-assisted security assessment; reconnaissance; multi-agent orchestration; persistent memory; system integration; evidence traceability.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **v**

List of Tables **vi**

List of Abbreviations **vii**

General Introduction **1**

1 Host Organization and Internship Context **2**
1.1 Presentation of Keystone 2
1.2 Areas of Expertise and Cybersecurity Activities 2
1.3 Internship Context and Assigned Mission 3
1.4 Project Context and Motivation 3

2 Project Context, Problem Statement and Objectives **5**
2.1 Context and motivation 5
2.2 Problem statement 6
2.3 Requirements, scope and objectives 6

3 State of the Art and Technical Foundations **8**
3.1 Automated and AI-assisted security assessment 8
3.2 LLM agents, LangGraph and multi-agent orchestration 10
3.3 Retrieval-Augmented Generation and persistent memory 11
3.4 Security assessment, validation and vulnerability scoring 12
3.5 Limitations of existing approaches and identified integration gap 14

4 System Architecture and Design **15**
4.1 Overall architecture and data flow 15
4.2 IntelForge: reconnaissance and intelligence synthesis 17

4.3 Voidhawk: adaptive orchestration, memory, and finding review 19
4.4 Design choices and integration requirements 21

5 DarkIntel: Integration and Implementation 23
5.1 IntelForge-to-Voidhawk data transformation 23
5.2 Memory seeding and reconnaissance handoff 24
5.3 End-to-end execution lifecycle 26
5.4 Reporting and Dragon synthesis 27
5.5 Failure handling and implementation constraints 28

6 Evaluation and Critical Analysis 31
6.1 Testing strategy and controlled evaluation protocol 31
6.2 Results and comparative evaluation 33
6.3 Technical limitations and threats to validity 36

7 Discussion and Perspectives 37
7.1 Discussion of the Proposed Architecture 37
7.2 Technical Limitations 38
7.3 Improvement Perspectives and Future Extensions 38

General Conclusion 41

Bibliography 43

List of Figures

4.1 System architecture and main data flow. DarkIntel coordinates the two graphs through in-process calls and an explicit reconnaissance handoff. . . 16
4.2 IntelForge's implemented LangGraph pipeline. The dashed branch goes to reporting when LLM stages are skipped. 17
4.3 Voidhawk's iterative graph. Routing after analysis and validation is conditional; the diagram omits minor state fields for clarity. 19
4.4 Voidhawk's operational memory. Vector search is optional; persistent records remain in SQLite, and retrieved text is context rather than verified knowledge. 21

5.1 Implemented transformation channels. Raw IntelForge files and full page analyses stay outside the transferred Voidhawk state. 25
5.2 Integrated lifecycle and output paths. Reports and Dragon use different record selections; Dragon's narrative is returned separately. 27

6.1 Proposed controlled evaluation setup. No comparative security results from this protocol are claimed in this report. 33

List of Tables

4.1 Principal contracts visible in the implemented workflows. 16
4.2 Component responsibilities and source attribution. 17

5.1 IntelForge-to-Voidhawk field mapping performed by the DarkIntel bridge. . 24

6.1 Offline/mocked pytest outcomes verified on 4 October 2026. No live security-effectiveness claim follows from these results. 34
6.2 Reproduced local behaviors using synthetic data. Targets and findings in these checks were fixtures, not security discoveries. 35

List of Abbreviations

AI	Artificial intelligence
API	Application programming interface
BERT	Bidirectional Encoder Representations from Transformers
CLI	Command-line interface
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DNS	Domain Name System
FIRST	Forum of Incident Response and Security Teams
GRC	Governance, Risk and Compliance
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LLM	Large language model
NIST	National Institute of Standards and Technology
OWASP	Open Worldwide Application Security Project
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
WSTG	Web Security Testing Guide
XML	Extensible Markup Language

General Introduction

Security assessment requires continuity between discovery, investigation, and reporting. A scanner may identify an exposed service, but deciding whether that service presents a vulnerability requires further evidence. The National Institute of Standards and Technology (NIST) places technical testing within a planned process of analysis and reporting [16]. The practical challenge is therefore not only to collect information, but to carry it between stages without losing its source or overstating its meaning.

This project addresses that challenge by connecting IntelForge and Voidhawk through DarkIntel. IntelForge collects reconnaissance and produces structured intelligence. Voidhawk uses an iterative workflow to select tools, analyze observations, and review candidate findings. DarkIntel transforms selected reconnaissance results into persistent memory and initial planning context for Voidhawk, then coordinates execution and reporting. Dragon provides an optional narrative synthesis after the reports have been generated.

The main contribution is the engineering integration between these complementary subsystems. The project reuses security tools, external libraries, and an existing assessment framework: Voidhawk extends the open-source Watchtower project. Its contribution must therefore be distinguished from the capabilities of those dependencies. Artificial intelligence (AI), principally through large language models (LLMs), supports interpretation, planning, and review; the project does not develop a new vulnerability-detection model.

The report follows the implemented system rather than its component names or intended capabilities. It distinguishes observations from hypotheses, model review from technical verification, and software test outcomes from security effectiveness. This distinction is essential when judging both the value of the integration and its current limitations.

Chapter 1 presents Keystone and the internship context. Chapter 2 defines the problem, scope, and objectives. Chapter 3 establishes the relevant research and technical foundations. Chapter 4 explains the architecture and subsystem responsibilities, and Chapter 5 examines DarkIntel's integration. Chapter 6 evaluates the available evidence. Chapter 7 interprets the architectural trade-offs and improvement priorities, before the General Conclusion synthesizes the contribution.

Chapter 1

Host Organization and Internship Context

1.1 Presentation of Keystone

Keystone is the host organization for this engineering internship. Its official website presents it as an international group specializing in cybersecurity and states that it has more than fifteen years of experience in the field. Its activities combine technical assessment, security operations, governance, and advisory services [6]. The experience figure is reported here as the company's description of its expertise, rather than as a separately established date of incorporation.

This profile provides the organizational setting for the project. The relevant connection is between cybersecurity service delivery and the engineering work required to collect, interpret, and communicate assessment evidence. The report examines that connection through the internship project, without extending the company's public description into assumptions about its internal organization.

1.2 Areas of Expertise and Cybersecurity Activities

Keystone identifies red teaming and blue teaming among its cybersecurity activities. Its red-team service describes objective-based adversarial exercises that also examine an organization's ability to detect and respond to attacks [7]. These complementary offensive and defensive perspectives make the relationship between a test action, its observed result, and its interpretation relevant to the internship.

The company also presents managed security services covering monitoring and incident response. Its Governance, Risk and Compliance (GRC) activities address security policies, risk management, and regulatory requirements, while strategic advisory concerns the alignment of cybersecurity decisions with organizational objectives [6]. These activ-

ities provide a broader setting in which technical evidence must be understandable and usable. The internship focused on assessment workflow engineering; it did not attempt to implement this entire service portfolio.

1.3 Internship Context and Assigned Mission

The project was undertaken during a third-year computer engineering internship at Keystone, from 15 June to 31 July. Its stated focus was the design and development of an AI-assisted cybersecurity assessment framework. The mission described for this report was to explore how LLM-based agents could support penetration-testing workflows, particularly where interpretation and coordination connect the outputs of traditional security tools.

The work addressed several related needs: processing reconnaissance results, selecting and coordinating tools, and maintaining context between assessment stages. It also investigated structured reasoning, finding review, reporting, and integration. These activities form a coherent engineering scope because the usefulness of an agent's decision depends on both the evidence it receives and the operations the surrounding software permits. The report therefore examines the framework as a connected workflow rather than evaluating model responses in isolation.

The implemented project provides evidence of the work carried out, but it is not a substitute for the original internship specification. The mission presented here follows the supplied internship description and the documented project scope. It does not establish that every implemented component was an initial contractual requirement, or that the prototype was deployed within Keystone or used in client engagements.

1.4 Project Context and Motivation

The technical motivation follows from the variety of information involved in an assessment. A network scanner reports ports and services, a web discovery tool identifies locations and response codes, and page analysis exposes forms, links, or possible points of interest. These outputs differ in structure and meaning. Turning them into a useful next action requires interpretation, while retaining enough detail to explain why that action was selected.

When the stages are disconnected, the assessor may need to reconstruct context manually or repeat discovery because earlier results are not available in a usable form. Introducing LLM agents does not remove this coordination problem. The software must still decide which observations to retain, how to distinguish a hypothesis from a recorded result, and what information should reach the next decision. Reporting also depends on preserving the connection between a claim and the tool output behind it.

This provided the motivation for DarkIntel: connecting reconnaissance intelligence to a later assessment process while making the transfer inspectable. The organizational relevance lies in Keystone's assessment activities and the internship's interest in AI-assisted workflows; the detailed subsystem design is developed later. Chapter 2 now formulates the engineering problem, defines the scope, and states the objectives against which the integration is examined.

Chapter 2

Project Context, Problem Statement and Objectives

2.1 Context and motivation

Security tools produce different accounts of the same target. Network scanners report reachable hosts, ports, and services; web discovery tools return paths and response codes; fetched pages expose forms, links, and software identifiers. These outputs help an assessor choose subsequent tests, but they do not share a common format or level of certainty. Combining them requires interpretation and a record of how each conclusion was reached.

The Open Worldwide Application Security Project (OWASP) Web Security Testing Guide (WSTG) treats web testing as a structured process and discusses the limits of automated detection [13]. Its distinction between a potential issue and a supported finding is particularly relevant when language models interpret tool output. A model can summarize observations or suggest a weakness, but the suggestion still needs evidence from the assessed system.

The project brings together two workflows with complementary responsibilities. IntelForge conducts reconnaissance and synthesizes its results into structured target records. Voidhawk conducts an iterative assessment in which earlier observations inform later tool choices. DarkIntel connects the two so that the assessment can begin with reconnaissance context already available. The motivation is to make that transfer explicit and usable, while preserving the distinction between collected information and inferred security claims.

These responsibilities also define the contribution's provenance. IntelForge and DarkIntel are separately packaged projects by Wael Hammali. Voidhawk is a Keystone Groupe extension of Watchtower, whose original author is credited in the license. The report examines the integration work without attributing the underlying framework or external security tools to DarkIntel. Chapter 4 establishes those boundaries in detail.

2.2 Problem statement

The central engineering problem is to translate reconnaissance into useful context for a later assessment whose interfaces were designed independently. IntelForge accumulates ports, command records, and page analyses. Voidhawk expects graph state and operational memory. Connecting them requires decisions about which information to transfer, how to represent it, and how to start the second workflow consistently with the work already performed.

The difficulty extends beyond field conversion. A page containing an upload form is an observation; a claim that the server accepts unsafe file types is a hypothesis unless tested. If an integration transfers both as equivalent facts, later stages may report more than the original evidence supports. Conversely, transferring only a short summary may remove the detail needed to select or reproduce a relevant test.

Three terms guide the report. An *observation* is recorded tool output or response content. A *candidate finding* is a possible security issue inferred from observations. A *reviewed finding* has undergone the implemented review process, which does not necessarily include independent reproduction. Technical verification requires evidence of an appropriate test. These meanings take precedence over software field names that may use stronger terms.

Continuity also requires identity. A result must remain associated with the target and assessment that produced it. Persistent storage can preserve useful context, but an older observation is not automatically applicable to a new run. Likewise, excluding a reconnaissance action is justified only if its required coverage has already been obtained. Context transfer therefore combines representation, execution control, and evidence management.

The guiding question is: *How can a reconnaissance pipeline and an iterative assessment workflow be connected so that useful context reaches later stages and the meaning of the evidence remains clear?* The analysis addresses this through three questions: what the transfer preserves or omits, how the combined workflow behaves under success and failure, and what evidence supports its generated findings and reports.

2.3 Requirements, scope and objectives

The functional scope begins with one operator-supplied target. DarkIntel runs IntelForge, prepares a Voidhawk session, seeds selected reconnaissance data, and starts the iterative graph. It persists emitted observations and finding records, invokes the existing report generators, and optionally requests Dragon's narrative synthesis. The integration must make these stages interoperable without requiring either subsystem to adopt the other's complete internal state.

Four objectives guide the work. The first is interoperability: transform reconnaissance records into downstream observations, memory entries, and planning context. The second is execution coherence: coordinate the handoff and lifecycle under defined tool and configuration constraints. The third is traceability: examine whether observations, commands, findings, and report entries remain attributable to their source and session. The fourth is maintainability: assess the benefits and costs of connecting separate packages through their existing interfaces. These are evaluation criteria, not a claim that the current prototype satisfies every requirement.

The scope is narrower than the combined features of the standalone projects. DarkIntel accepts one target per run and exposes only part of each dependency's configuration. It does not provide every authentication option available in standalone Voidhawk, and tool execution still depends on installed binaries and suitable input. Its operation is intended for authorized assessment; permission and scope remain conditions established by the assessor, as required by planned security testing [16].

The project neither trains a new language model nor maintains a new vulnerability database. Reconnaissance synthesis supplies leads for investigation. Finding review evaluates those leads through the implemented model-based process. Dragon supplies an additional narrative without executing its proposed steps. The report treats these as distinct outputs and examines their limits where the relevant implementation is discussed.

Evaluation must match the evidence available. Source analysis and controlled software tests can establish data mappings, state transitions, persistence behavior, and reachable failure paths. They cannot establish real-world detection accuracy. Chapter 6 therefore separates the observed software evidence from a proposed comparative security protocol. This provides a basis for judging the integration without assuming that a coherent workflow necessarily produces more accurate findings.

Chapter 3

State of the Art and Technical Foundations

Security automation, agent orchestration, and retrieval address different parts of an assessment. This chapter examines their capabilities and limits, then identifies the integration requirements that motivate the architecture. The emphasis is on how existing methods support a traceable assessment, rather than on a general survey of security tools.

3.1 Automated and AI-assisted security assessment

Security assessment is a structured investigation of a defined target. NIST distinguishes planning, discovery, attack-oriented testing, and reporting within a technical assessment; it also emphasizes rules of engagement and the interpretation of results [16]. For web applications, the WSTG organizes tests around information gathering, configuration, identity, authorization, input validation, and other security concerns [13]. These references do not prescribe a single tool chain. They establish a more important principle: a test has meaning only in relation to its objective, scope, method, and observed result.

Conventional automation is valuable because it can perform repeatable tasks over many hosts, ports, pages, or requests. A network scanner can identify reachable services; a web crawler can enumerate paths; a rule-based vulnerability scanner can compare responses with known signatures. Such tools reduce the manual effort needed to obtain an initial view of the target. Their outputs are nevertheless measurements or alerts rather than complete security judgments. A detected software version may be inaccurate, and the presence of a version associated with a public vulnerability does not establish that the relevant feature is enabled or reachable. Similarly, an unusual Hypertext Transfer Protocol (HTTP) response may justify further investigation without proving a specific weakness.

The limitations are partly technical and partly contextual. A scanner can miss behavior that requires authentication, a particular sequence of requests, or knowledge of application-specific business rules. Conversely, a broad signature may flag a harmless response. OWASP warns that false positives should be removed before findings are reported and that automated testing cannot cover every application path [13]. The practical implication is that discovery and verification must remain distinguishable. Automation can produce candidates efficiently; it cannot make every candidate credible without additional evidence.

LLMs add a different kind of assistance. They can read heterogeneous text, summarize lengthy tool output, relate observations to possible weaknesses, and propose a next investigative step. This is attractive in security work because command output and web content do not share a convenient schema. A model can help turn a mass of observations into a smaller set of hypotheses for the assessor to examine. The benefit is interpretive flexibility, not a guarantee of correctness. The same flexibility can produce plausible but unsupported explanations, especially when source output is incomplete or ambiguous.

PentestGPT provides a relevant research example. Deng and colleagues studied LLM use in penetration-testing tasks and found that models could assist with tool use, output interpretation, and proposed next steps, while maintaining the full context of an assessment remained difficult. Their response was a modular framework that separated subtasks and managed information between them [2]. This result supports the use of explicit context management, but its benchmark outcomes are properties of that study's targets, models, and evaluation protocol. They do not demonstrate that any other agent-based system has the same effectiveness, nor do they remove the need to verify individual findings.

It is useful to distinguish levels of automation. A scripted pipeline runs a predetermined sequence and can be tested against expected inputs and outputs. An AI-assisted pipeline preserves that sequence but uses a model for tasks such as classification or summarization. An adaptive agent selects later actions from earlier observations. Each level offers a different trade-off. Fixed pipelines are easier to reproduce and audit, but can be inefficient when the next useful test depends on what was discovered. Adaptive systems can respond to new evidence, but their behavior becomes harder to predict, constrain, and compare across runs. The correct choice depends on the assessment objective and on which decisions can safely be delegated.

Published results must be interpreted within their evaluation conditions. A parser test establishes behavior for a fixture; a benchmark with reference findings can estimate detection performance; a case study establishes only what occurred in that case. Models, prompts, target configurations, and allowed actions can all affect outcomes. The WSTG accordingly treats automated coverage and accuracy as properties to evaluate, not assume [13]. Chapter 6 applies this distinction to the project.

3.2 LLM agents, LangGraph and multi-agent orchestration

An LLM agent is a program that uses a model to choose or describe actions while interacting with external tools and observations. The model may suggest a command, but the surrounding software determines which tools exist, how inputs are checked, how commands run, and which outputs return to the model. This division is essential in a security setting. A generated action is only a proposal until application code accepts and executes it; a generated explanation is only an interpretation until it is supported by evidence.

The ReAct approach illustrates the basic feedback loop. It interleaves model-generated reasoning with actions that obtain new information from an environment, so later reasoning can take account of observations rather than rely solely on the initial prompt [17]. In assessment work, the analogous cycle is to inspect current evidence, select a test, observe its output, and revise the next step. This mechanism can be more responsive than a fixed list of tests. It also creates failure modes: a mistaken interpretation can lead to an irrelevant action, and a misleading tool result can influence all later choices. The loop needs explicit stopping conditions, bounded tool access, and a record of what was actually executed.

Agent workflows are often divided into roles such as planner, tool executor, analyst, and reviewer. This separation can make interfaces clearer. For example, the executor can return exact tool output while the analyst produces a tentative interpretation. It can also make errors easier to locate if each stage records its input and output. Yet several model roles do not automatically create independent checks. If the analyst and reviewer see the same incomplete evidence or use closely related prompts, they may agree on an unsupported conclusion. A review role becomes meaningful only when its criteria, inputs, and response to uncertainty are specified.

LangGraph provides a programming model for such workflows. Its graph consists of shared state, nodes that perform work, and edges that determine the next node. Edges may be fixed or conditional, allowing a workflow to branch, repeat a tool call, or stop according to state [9]. A node may call an LLM, invoke a tool, or run ordinary deterministic code. This matters because "agentic" behavior is not a property of the whole graph: some transitions can be fixed even when one node asks a model to make a choice. An architecture should identify precisely where the model influences control flow and where the program enforces it. Named fields and visible transitions support software testing, but structural validity does not establish that a stored claim is true or sufficiently supported.

Persistence introduces another distinction. LangGraph documents checkpointing as a way to save graph state during execution and resume a thread [10]. An application may instead, or additionally, store observations and findings in its own database. These are different functions: a checkpoint captures workflow state, whereas an application

database can organize durable evidence across runs. Neither mechanism by itself defines whether records are isolated by target, whether stale results are rejected, or whether a resumed workflow reflects a changed external system. Those rules must be designed at the application boundary.

The agent's inputs can also be adversarial. A target page or scanner output may contain text that looks like an instruction to the model. OWASP identifies prompt injection and excessive agency as distinct risks for LLM applications: untrusted content can influence model behavior, while overly broad tool permissions can turn a bad decision into an unintended action [12]. The relevant controls are concrete: limit available functions, validate parameters, preserve the distinction between data and instructions, and log the exact tool calls made. A prompt alone is an insufficient authorization boundary. In an assessment framework, these controls protect both the test environment and the integrity of the resulting evidence.

3.3 Retrieval-Augmented Generation and persistent memory

An assessment may produce more information than fits usefully into a single model prompt. Earlier scans, web pages, previous observations, and review notes may all matter when deciding what to test next. Retrieval-Augmented Generation (RAG) addresses this information problem by selecting relevant external material and supplying it as context for a generation step. The original RAG study combined a parametric language model with a non-parametric document index and investigated how retrieved passages could inform generated answers [11]. For an engineering application, the central idea is an explicit knowledge store that can be queried at the time an answer or decision is produced.

In application-level retrieval, indexed text is selected and added to a model's input; the model's learned parameters remain unchanged. Updating the store therefore differs from training or fine-tuning the model. The original RAG research included fine-tuning in its experiments [11], but this is not a necessary property of every deployed retrieval workflow. The relevant distinction is between updating available context and updating model weights.

Semantic retrieval commonly represents text as numerical embeddings. Sentence-BERT, based on Bidirectional Encoder Representations from Transformers (BERT), showed how sentence-level embeddings could support similarity search without comparing every query and candidate as a separate pair through a cross-encoder [15]. A vector store can search for records whose embeddings are close to a query and return their text and metadata. Chroma, for example, documents storage of embeddings with metadata and filtering during retrieval [1]. Such tools make it feasible to retrieve a compact set of

relevant observations from a larger collection. Similarity remains a ranking signal, however, not a statement that two records describe the same target, the same vulnerability, or the same time period.

Security assessments need more than one kind of memory. Short-lived workflow state records what the agent is doing now. A structured database can preserve exact commands, outputs, findings, timestamps, and session identifiers. A vector index can help find semantically related prior material. A cache may reuse the output of a repeated operation. These stores answer different questions. A vector search is useful for "What previous observation resembles this one?"; a structured query is better for "Which command produced this finding in this session?" A cached response answers "Have these inputs been seen before?" but not "Is the target still in the same condition?" Treating all four mechanisms as one undifferentiated "memory" obscures their different reliability requirements.

Retrieval quality depends on representation and scope. Entire logs preserve detail but can be noisy; summaries are easier to retrieve but may omit the request or error line needed for verification. Compact descriptions linked to full source records provide a useful separation between search and evidence. Target, session, timestamp, source, and review status should constrain interpretation: a semantically similar record may belong to another host or precede a patch. Historical observations should remain inspectable without being presented as fresh measurements. Retrieval supplies context for a decision, not proof of its conclusion.

There is also a security boundary around the index. Retrieved records may contain attacker-controlled web content or previously generated model text. If either is promoted into authoritative instructions, retrieval can carry prompt injection forward into later stages. OWASP explicitly includes vector and embedding weaknesses among the risks of LLM applications [12]. A sound design therefore separates source data from control instructions and records how each retrieved item entered the store. The goal is not merely to remember more, but to preserve enough provenance to know what has been remembered and whether it still applies.

3.4 Security assessment, validation and vulnerability scoring

Evidence can have different strengths. A service banner is a direct observation but may be unreliable as a version identifier. A scanner signature is a test result under a particular rule set. A model's suggested exploit path is a hypothesis. A successful, authorized reproduction under controlled conditions offers stronger support for exploitability, provided the record includes the exact test and response. These levels should not be collapsed into

a single "validated" flag. When a model reviews a candidate by reading the same text that generated it, the process is a consistency or plausibility check. It is not an independent retest of the target. An incomplete or failed review must remain distinguishable from an accepted result, even if the surrounding workflow continues.

Traceability allows an assessor to examine the strength of a claim. The relevant chain connects target and scope, tool invocation, time, raw or faithfully preserved output, any transformation of that output, candidate interpretation, review decision, and final report entry. These links establish applicability, allow inspection of the original result, and show why a candidate was accepted or rejected. Stable identifiers can preserve that chain without reproducing entire logs in the final report. If a stage fails, that failure must also be recorded. Otherwise, a later consumer may mistake an unreviewed candidate for a reviewed finding.

Evidence-aware reporting is more than attaching a screenshot or a quotation. The WSTG recommends describing the issue, its cause, testing method, impact, and remediation in a report that both technical and non-technical readers can use [14]. The report should state the limits of what was established. For instance, identifying a potentially vulnerable component justifies investigation, but a report should not claim code execution unless an appropriate test or other reliable evidence supports it. A concise conclusion with a clear source trail is more useful than an elaborate narrative that outruns the observations.

Severity scoring is a separate step from verification. The Common Vulnerability Scoring System (CVSS), specified by the Forum of Incident Response and Security Teams (FIRST), defines in version 3.1 Base, Temporal, and Environmental metric groups and a vector notation for the selected metric values [3]. The Base score describes intrinsic technical characteristics under the specified assumptions; it does not, by itself, express the full business risk for a particular organization. A numerical score is interpretable only when the underlying metrics and version are known. If those inputs are absent, a model-generated number may look precise while hiding uncertainty about attack complexity, required privileges, scope, or impact.

The same distinction applies to references to known vulnerabilities. A component name and version can suggest a match with a published advisory, but products may carry backported fixes, optional modules, or vendor-specific builds. A model's recollection of a Common Vulnerabilities and Exposures (CVE) identifier is particularly weak evidence unless it can be checked against an authoritative source and linked to the observed configuration. The proper output at this stage may be a research lead rather than a reportable vulnerability. Scoring should follow verification of the issue and the affected context; it should not be used to make a speculative match appear established.

3.5 Limitations of existing approaches and identified integration gap

The reviewed approaches offer complementary capabilities. Scanners produce observations; LLM agents interpret them and select actions; graph frameworks make transitions explicit; retrieval preserves access to earlier context. Their combination leaves an engineering gap: the stages may use incompatible representations and different assumptions about evidence. Transferring only prose can discard source details, while transferring every log can overwhelm planning. A useful handoff needs selective context with traceable links to its origins.

Adaptive orchestration also requires a record of what actually ran. A proposed action, an executed command, a returned observation, and a reviewed finding have different meanings. Persistence must retain those distinctions across targets and sessions, while review must expose uncertainty rather than reproduce an earlier model judgment. These requirements concern the interaction between subsystems; adding another prompt or index does not satisfy them automatically.

The resulting design requirements connect five capabilities: reconnaissance intelligence, persistent context, adaptive orchestration, explicit translation between subsystems, and evidence-aware finding review. They do not presume improved detection accuracy. They establish what an integrated architecture must make inspectable: the context transferred, the actions taken, and the evidence retained. Chapter 4 uses these requirements to explain the roles of IntelForge, Voidhawk, and DarkIntel.

Chapter 4

System Architecture and Design

IntelForge and Voidhawk have different control patterns and data contracts. This chapter explains their implemented roles and the design decisions that allow DarkIntel to connect them. Detailed field conversion and lifecycle handling follow in Chapter 5.

4.1 Overall architecture and data flow

The projects are separate Python packages with their own entry points and configurations. DarkIntel calls their application programming interfaces (APIs) within one process: it runs reconnaissance, prepares the assessment context, streams the iterative graph, and coordinates reporting. The graphs remain in IntelForge and Voidhawk. Their different state schemas and persistence conventions make an explicit handoff necessary [5, 8, 4].

Figure 4.1 combines the subsystem boundaries with the main data flow. IntelForge owns collection and intelligence synthesis. Voidhawk owns the adaptive assessment loop and operational memory. DarkIntel controls the sequence and transforms the reconnaissance record before starting Voidhawk. Both subsystems depend on external security tools and configured model providers; application code determines which actions and graph transitions can occur.

IntelForge validates the supplied target and fills a `TargetState` with ports, services, web locations, command results, and page analyses. It writes JavaScript Object Notation (JSON) state and raw-output files. DarkIntel consumes the in-memory record, seeds Voidhawk memory, and starts its graph. Emitted observations and finding records enter SQLite. Reporters then generate Portable Document Format (PDF), HyperText Markup Language (HTML), or Markdown files. Optional Dragon synthesis follows as a separate output; Section 5.4 explains that distinction.

Configuration remains distributed. IntelForge controls collector selection, scan profiles, directories, and model roles. Voidhawk controls its provider, timeouts, iteration limit, memory, and review settings. DarkIntel controls its integration directory and optional

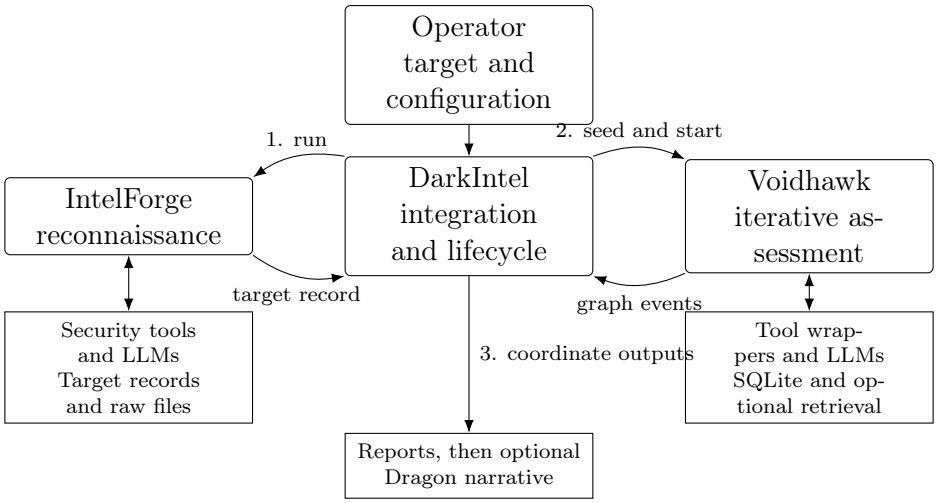


Figure 4.1: System architecture and main data flow. DarkIntel coordinates the two graphs through in-process calls and an explicit reconnaissance handoff.

Dragon model. This preserves standalone use but makes integrated behavior dependent on defaults in all three packages.

Table 4.1 distinguishes durable target data from transient graph state. IntelForge's `TargetState` manages the `TargetData` record; its `GraphState` also carries temporary page content. Voidhawk's `AgentState` connects graph nodes, while its database stores selected records beyond the invocation.

Contract	Main contents	Architectural role
`TargetData`	Target, ports, services, locations, `CommandResult`, `PageAnalysis`	IntelForge's accumulated reconnaissance record, serialized to JSON.
IntelForge `GraphState`	`TargetState`, options, fetched and cleaned pages, analyses, report path	Temporary state connecting its fixed graph nodes.
Voidhawk `AgentState`	Allowed tools, plan, observations, findings, review lists, phase, session	Shared state of the iterative graph.
Voidhawk memory rows	Sessions, observations, cleaned commands, findings, reasoning steps	Durable operational history and retrieval source.

Table 4.1: Principal contracts visible in the implemented workflows.

Table 4.2 records source attribution. The ownership of a package is distinct from the originality of every dependency it uses; DarkIntel's contribution is the integration between the two workflows.

Package	Responsibility in this system	Provenance in reviewed source
IntelForge	Collect and synthesize reconnaissance into target records and a JSON report.	Separately packaged project by Wael Hammali.
Voidhawk	Execute the iterative tool and finding-review workflow; persist and report results.	Keystone Groupe extension of Watchtower; original author `fzn0x` credited in license.
DarkIntel	Run the two packages in sequence and translate selected outputs between them.	Separate integration package by Wael Hammali.

Table 4.2: Component responsibilities and source attribution.

4.2 IntelForge: reconnaissance and intelligence synthesis

IntelForge's implemented graph contains seven nodes in a fixed sequence: reconnaissance, command cleaning, page fetching, HTML cleaning, analysis, research, and reporting. After command cleaning, `skip_llm` can route directly to reporting (Figure 4.2). Options can also disable individual collectors or reconnaissance. The graph exposes the pipeline's stages; it does not normally choose new security tools in response to each observation.

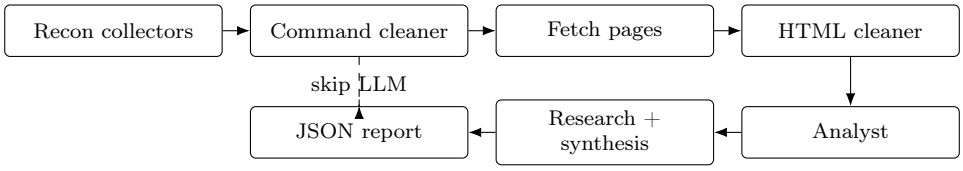


Figure 4.2: IntelForge's implemented LangGraph pipeline. The dashed branch goes to reporting when LLM stages are skipped.

The reconnaissance node can run three collectors concurrently. Nmap executes configured Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) profiles, then parses Extensible Markup Language (XML) output into hosts, ports, services, and versions. Profiles remain sequential within that collector. FinalRecon is an external Python installation whose output and dump files supply selected fields. IntelForge's own web collector probes directories through HTTP requests, tests subdomains through Domain Name System (DNS) lookups, and checks virtual hosts using the request host header. It runs for targets classified as names rather than Internet Protocol (IP) addresses; directory requests can run concurrently, whereas its other loops are sequential.

Concurrent collection permits independent network operations to overlap, but does not establish complete coverage. The reconnaissance node catches and logs collector failures, then continues with available data. Each result remains attributable to its collection path: a response code or service fingerprint is a discovery signal, not a vulnerability conclusion.

The collectors also differ in the form of their evidence. Nmap supplies structured probe results, FinalRecon supplies mixed command output and external files, and web

discovery supplies response codes or name-resolution results. IntelForge combines these in a target record while retaining distinct fields and raw references. This makes the record useful for synthesis without implying that each source establishes the same kind of fact. A returned web response, for example, can identify a location worth examining without showing that its content is sensitive.

IntelForge records command output in two forms. The raw text is written under a raw-output directory and queued for the command-cleaning node. A model then condenses it into `CommandResult` entries with a command, purpose, cleaned output, and reference to the raw file. Nmap sweeps become individual entries; FinalRecon output may be divided into logical sections. If no model is configured or cleaning fails, a deterministic text-strip fallback produces a shortened result. This choice keeps the pipeline moving and preserves a raw reference, although the shortened text can omit details that a later analyst might need. The target model also deduplicates ports, services, and selected lists.

The web-analysis stages use a different path. The graph builds Uniform Resource Locators (URLs) from the base target and discovered subdomains, virtual hosts, and directories, then fetches their pages. The HTML cleaner supplies selected page content to a model after truncation. The analyst processes cleaned pages in small batches and creates `PageAnalysis` records containing such fields as technologies, authentication pages, upload points, downloadable files, possible parameters, and suspicious items. This schema makes page interpretations comparable while retaining a URL for each record. It does not itself verify that an inferred parameter is exploitable.

Page fetching uses ordinary HTTP requests to constructed locations, not an authenticated browser session. Failed requests are logged and omitted; absent pages produce empty downstream outputs. IntelForge therefore analyzes successfully fetched content rather than claiming complete application coverage.

The research node takes each page analysis together with Nmap and cleaned command summaries. A model proposes research tuples and an attack order, and the analyst performs a final synthesis that can add priority vectors and a verdict to the page record. In the reviewed source, this "research" step is prompt-based reasoning over the supplied context. It does not query an authoritative CVE database or independently confirm exploitability. Its output is therefore intelligence for prioritization, not a verified vulnerability inventory. IntelForge's final node saves the accumulated `TargetData` and emits a timestamped JSON report; the command-line presentation also renders console tables. The durable output available to another package is the target record, not an automatically shared LangGraph state.

4.3 Voidhawk: adaptive orchestration, memory, and finding review

Voidhawk provides the second style of workflow. Its graph starts at a Planner, moves through a Worker and Cleaner to an Analyst, optionally passes through Logic Analysis, and reaches a Validator. The Validator routes to a Reporter, a Discard node, or directly back to the Planner for a requested retest. The Reporter and Discard nodes also return to the Planner. The loop ends when the Planner marks the assessment finished or an iteration guard is reached. Figure 4.3 shows this control flow. The label "Reporter" here names a graph node that records a summary message; formatted PDF, HTML, and Markdown reports are generated after the graph ends.

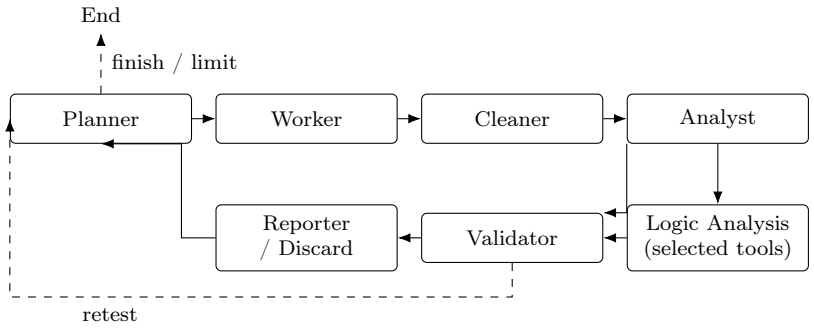


Figure 4.3: Voidhawk's iterative graph. Routing after analysis and validation is conditional; the diagram omits minor state fields for clarity.

The Planner receives the target, permitted tools, recent observations, findings, review outcomes, completed tools, and memory context. It requests a structured `PlannerOutput` containing a plan, one or more tool names, a completion flag, and optional phase or retry information. Phases guide planning rather than define separate graphs. Several state lists use append semantics, preserving history, while the prompt includes only selected recent material. The Planner can request additional memory when no actionable step is available; freshness and deduplication depend on node logic.

The Worker accepts only names in `available_tools`, imports the corresponding `watchtower.tools` module, and invokes its `run` function. A tool list can be executed through a bounded thread pool. Most wrappers construct commands from the target and fixed flags: the Planner generally selects a tool rather than its endpoint, parameter, or payload. The shared runner checks arguments and applies timeouts and output truncation. This bounds model-directed execution but limits targeted follow-up and still depends on installed executables.

The SQLMap wrapper illustrates the boundary between selection and parameter adaptation: it passes the target as a URL with predetermined crawling and risk options. A Planner decision to use that wrapper does not itself encode a discovered parameter or a

tailored payload. The application controls the invocation, while the model controls the permitted tool choice. This division is relevant when comparing the system with approaches that generate arbitrary commands: the implemented flexibility is narrower and more inspectable, but not sufficient for every targeted verification task.

The Cleaner uses tool-specific parsers and, when needed, LLM interpretation to extract structured fields, summaries, and raw-output previews. The Worker normally attaches the cleaned result to its observation; the graph's Cleaner node reuses it or performs cleaning. The Analyst reads the latest observation, proposes findings with titles, severity, descriptions, and evidence, and deduplicates by title. A parallel batch does not automatically receive one analysis pass per observation. Section 5.5 examines this association problem alongside the integration's other execution limits.

Logic Analysis follows selected web-discovery tools and proposes tests for insecure direct object references or business-logic weaknesses. Its informational output is a plan, not an observed access-control failure. The Validator then reviews candidates against recent raw observations through an LLM. Its schema requests `confirmed`, `rejected`, or `needs_retest`, with confidence, remediation, and an estimated CVSS 3.1 score. Low-confidence confirmations become retest requests. These are model-review decisions; the node does not independently execute a reproduction.

The router gives retest requests precedence; otherwise, accumulated validated or rejected lists lead to Reporter or Discard before returning to planning. Reporter records a graph summary rather than creating the final document. Disabled or failed review has a separate pass-through behavior, examined in Chapter 5; the state field name alone is insufficient to identify the evidence status.

Voidhawk's `MemoryAgent` separates persistent operational records from graph state. SQLite stores sessions, observations, cleaned commands, findings, and reasoning steps. A time-limited command cache supports reuse, and session reasoning can be recalled for planning. These records document assessment activity rather than form an independently curated cybersecurity knowledge base.

The stored record types have different roles. A session identifies a run; an observation records tool output; a cleaned-command record supports structured recall and cache reuse. Findings record claims and status, while reasoning entries preserve decisions or summaries. This separation allows reporting to distinguish findings from ordinary tool results. However, a session column provides only the capacity for isolation: insertion and query methods must consistently use it. The bridge's dependence on those methods is examined in the next chapter.

For broader recall, `MemoryAgent` can create a persistent Chroma collection. It indexes textual descriptions of reasoning steps and some cleaned observations for similarity search. If Chroma is unavailable or a query fails, the code can compare locally stored Sentence Transformers embeddings with NumPy; if that path is unavailable, it falls back to a

Structured Query Language (SQL) keyword search in SQLite. Figure 4.4 distinguishes these stores and retrieval routes. Chroma and Sentence Transformers provide retrieval mechanisms, not new model training or independent verification.

The retrieval routes do not search identical material. Chroma can return indexed reasoning and observation descriptions with limited metadata; local embedding comparison and keyword fallback search reasoning records. Dependency or index availability can therefore change what is recalled. Retrieved text supplies optional planning context, while exact evidence remains in the operational record. The current cache, retrieval, and reporting boundaries also require explicit attention to session isolation, as evaluated in Chapter 6.

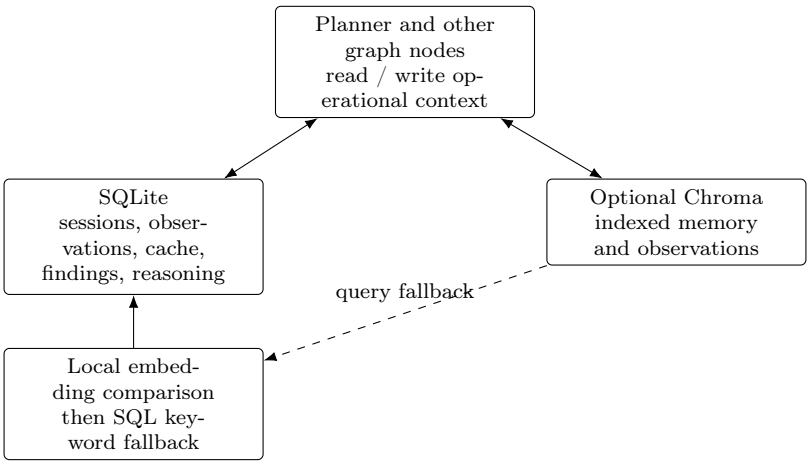


Figure 4.4: Voidhawk's operational memory. Vector search is optional; persistent records remain in SQLite, and retrieved text is context rather than verified knowledge.

4.4 Design choices and integration requirements

Keeping the packages separate preserves their distinct control patterns and standalone interfaces. IntelForge offers predictable reconnaissance stages; Voidhawk offers observation-dependent tool selection. DarkIntel avoids rewriting both graphs around one shared state machine, at the cost of maintaining compatibility with their field names, memory methods, and tool identifiers.

This boundary also determines how changes should be assessed. An IntelForge field change or a Voidhawk tool-name change can leave both standalone entry points functional while altering the integrated workflow. Testing each package is therefore necessary but insufficient for maintaining the connection. The bridge is an explicit compatibility layer, and its contract must be reviewed when either dependency evolves. This explains why integration behavior, rather than the combined feature inventories of the packages, is the focus of the implementation chapter.

Local persistence supports inspection across process runs without a remote database service. However, JSON files, SQLite rows, and vector records have separate lifecycles. The integration must preserve target identity and source links across those stores. A successful write does not by itself show that a retrieved record remains applicable to the current assessment.

Structured records make the boundary explicit. IntelForge uses Pydantic models for validation and serialization; Voidhawk uses typed graph state and Pydantic schemas for selected model responses. These choices constrain data shape while leaving evidence status to application logic. The bridge must therefore distinguish source observations from model hypotheses when preparing memory and Planner context.

The handoff has three resulting requirements: select useful reconnaissance without losing its evidential meaning, start the downstream workflow according to actual coverage, and coordinate persistence and reporting under a consistent target and session. Controlled wrappers and partial-failure handling impose additional limits on what later actions can establish. Chapter 5 explains how DarkIntel implements these requirements and where its current contract remains incomplete.

Chapter 5

DarkIntel: Integration and Implementation

The architecture requires a boundary between IntelForge's target record and Voidhawk's session, operational memory, and graph state. DarkIntel implements that boundary through selective transformation, memory seeding, execution control, and output coordination. This chapter follows those operations and their consequences for coverage and evidence [4].

5.1 IntelForge-to-Voidhawk data transformation

DarkIntel receives IntelForge's in-memory `TargetState` after its graph completes. It neither reloads the JSON report nor passes the complete object into Voidhawk. Instead, bridge functions select fields from `TargetData` to create persistent records and a compact Planner digest. Temporary graph values, including fetched page content, are outside this transfer. The selective contract lets Voidhawk consume reconnaissance without understanding IntelForge's complete schema.

Each IntelForge `CommandResult` contains a command, purpose, cleaned output, and raw-file reference. The bridge logs `clean_output` as a Voidhawk observation and prefixes its cleaned summary with `purpose`. It infers the collector name from the raw-file stem, removing the target suffix where recognized and falling back to filename or command information. This convention helps identify collectors launched through an interpreter, but does not provide an explicit source-tool field. The exact command and raw-file path have no direct destination in the seeded observation.

Page-level interpretations follow a separate route. The bridge reads `priority_exploit_vectors` from the `exploit_report` of each page analysis and stores each vector as an IntelForge reasoning entry, retaining its dictionary in the details. It also renders severity, keyword, proposed action, and any final page verdict into a text digest.

IntelForge's Nmap summary supplies the digest's port and service context. This text becomes the initial `memory_context`; it is prepared at handoff, separately from the persistent entries available for later recall.

Table 5.1 identifies retained and omitted information. Source material remains in IntelForge's files, but the downstream record cannot reconstruct an omitted command or raw-file link from the cleaned summary alone.

IntelForge source	Voidhawk mapping and information retained
Target string	`TargetData.target` becomes the session target and graph scope. No further target identity or authorization metadata is added.
Nmap summary	`get_nmap_summary()` supplies readable ports and services to the planner digest and Dragon; individual port objects are not inserted into Voidhawk state.
Cleaned command output	`CommandResult.clean_output` becomes a SQLite observation and summary. Raw output stays in IntelForge's files.
Command purpose	`CommandResult.purpose` prefixes the cleaned summary; it does not record full execution settings.
Raw-file reference	The stem of `raw_ref` supplies an inferred collector label; the path itself is not transferred.
Recorded command	`CommandResult.command` has no direct destination in the seeded observation or Dragon's later history string.
Priority exploit vectors	Selected `PageAnalysis` hypotheses become reasoning memories and planner-digest text, not findings.
Final page verdict	`final_verdict` appears as prose in the planner digest, not as a separate finding.
Other page and graph data	Fetched HTML, full page analysis, discovered location lists, and intermediate graph outputs are not systematically transferred.

Table 5.1: IntelForge-to-Voidhawk field mapping performed by the DarkIntel bridge.

The three transfer channels in Figure 5.1 serve distinct purposes. Observations preserve selected collector output, reasoning entries preserve hypotheses, and the digest supplies immediate planning context. Priority vectors are not inserted as confirmed findings.

The source and downstream records consequently serve different review tasks. IntelForge retains the raw-file reference and JSON target record for manual inspection. Voidhawk receives a target, inferred collector, and cleaned output, without a direct link back to that original file. This is information omitted by the transfer, not deleted source evidence. Reconstructing a seeded observation may therefore require consulting IntelForge's artifacts separately rather than following a complete provenance chain within Voidhawk.

5.2 Memory seeding and reconnaissance handoff

After IntelForge returns, DarkIntel constructs Voidhawk's `MemoryAgent` using Voidhawk's configured SQLite path, vector-search flag, embedding model, cache flag, and cache lifetime. It creates a session for the target before transferring any reconnaissance material. The session provides an identity for the later graph execution and for reasoning memories inserted by `seed_memory()`. Each cleaned IntelForge command result is logged through the same observation method used for Voidhawk tool output. Each priority exploit vector

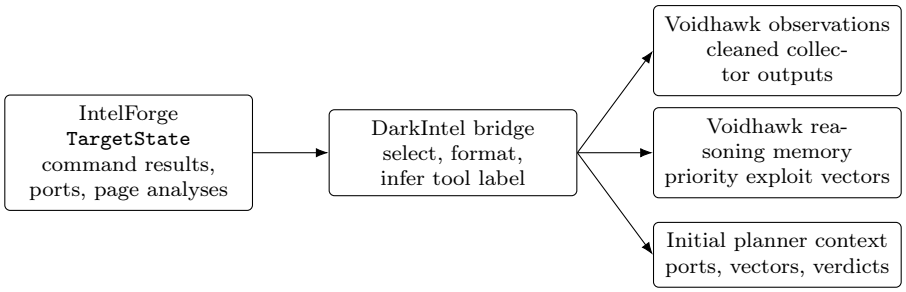


Figure 5.1: Implemented transformation channels. Raw IntelForge files and full page analyses stay outside the transferred Voidhawk state.

is logged through the agent-memory method with `agent=IntelForge` and the new session identifier. This places reconnaissance in Voidhawk's operational history before its Planner chooses a tool.

Session attribution differs between memory methods. Reasoning insertion receives `session_id` explicitly; observation insertion relies on the memory object's active session. This supports the ordinary sequential path, while making attribution dependent on that mutable state. Optional vector indexing can expose some text to retrieval, but the retrieval boundary described in Chapter 4 does not consistently restrict results to this target or session.

DarkIntel prepares a separate initial `AgentState` for the Voidhawk graph. It sets `scope_targets` to the supplied target, attaches the active memory object and session ID, leaves observation and finding lists empty, and places the formatted reconnaissance digest in `memory_context`. The initial phase is `scan`. The graph's Planner can read that digest immediately and can later request session reasoning memory. The seeded observations are persisted, but they are not inserted into the initial graph state's `observations` list. Consequently, downstream nodes that examine only recent graph observations do not automatically receive those earlier IntelForge outputs. Their principal immediate route into planning is the digest and retrieved memory, not a replay of IntelForge observations through the Voidhawk graph.

The handoff changes context rather than generating a test for every hypothesis. A stored priority vector can influence the Planner's next choice, but it is not converted into a command with a selected endpoint, parameter, or payload. Nor does storing an IntelForge observation replay it through Voidhawk's Cleaner and Analyst. These distinctions explain the contribution's precise scope: DarkIntel prepares the information available to assessment, while the existing graph and wrappers determine how that information can affect execution.

The bridge copies a fixed set of reconnaissance tool names from Voidhawk's planning conventions, removes them from `available_tools`, and inserts the entire set into `completed_tools`. Because the Worker accepts only permitted names, this enforces exclusion rather than merely suggesting that the model avoid repetition. It prevents those

tool calls in the ordinary integrated path; whether the policy reduces genuinely redundant work without losing useful coverage remains unmeasured.

The initial `scan` phase and completed-tool list express phase ownership, not verified collector success. DarkIntel does not derive them from an execution ledger or reconcile the twelve excluded tools with IntelForge's actual coverage. A successful Nmap collector does not establish that other excluded discovery functions ran. Partial and complete reconnaissance can therefore produce the same downstream starting state. This matters when an omitted discovery action would have supplied a host, endpoint, or technology needed for follow-up.

The exclusion also has a less obvious graph consequence. Voidhawk enters its Logic Analysis node after particular web-discovery tools, including `httpx`, `kiterunner`, `arjun`, `gobuster`, and `ffuf`. All of those names are in DarkIntel's excluded set. With the ordinary bridge state and normal Worker selection, the run cannot produce a fresh observation from one of those triggers. The route to Logic Analysis is therefore effectively unavailable in the integrated path, although the node remains part of Voidhawk's graph. IntelForge's page analyses are seeded as context and memories; they are not converted into the specific graph observations that would activate this route. This is a concrete example of how an integration decision changes behavior without changing the upstream graph.

5.3 End-to-end execution lifecycle

The command-line interface (CLI) accepts a target and report format; an interactive console invokes the same bridge through its `burn` command. Both reach `run(target, report_format=...)`. Integration occurs within one Python process through package APIs, with execution determined by subsystem settings, model providers, and installed security tools.

DarkIntel constructs `TargetState` in its IntelForge data directory, sets the target, and invokes the reconnaissance graph with default `ScanOptions`. It consumes the resulting in-memory state. IntelForge's JSON report and raw files remain separate artifacts; its report path is not included in DarkIntel's final return value.

After session creation and seeding, the bridge obtains Voidhawk's canonical tool list, excludes reconnaissance names, and builds the initial state. It does not first restrict the list to executables discovered on the host. It creates the existing Voidhawk graph and streams updates under the configured recursion limit, without replacing its agents.

The event loop writes emitted observations to memory and to a run-local list. Entries emitted under `validated_findings` are persisted and collected separately, with title-based deduplication within the bridge invocation. This avoids repeated writes of the same title but cannot distinguish different issues that share one. The bridge also retains the latest nonempty `validation_summary`. Graph state drives subsequent decisions, SQLite

supports persistence and reporting, and the local lists provide Dragon's selected input; they are not interchangeable records.

Figure 5.2 shows execution order and the two output paths. Reports read the database, while Dragon later receives run-local findings, an observation summary, and IntelForge's Nmap summary. The dashed arrow denotes ordering rather than a transfer of report contents.

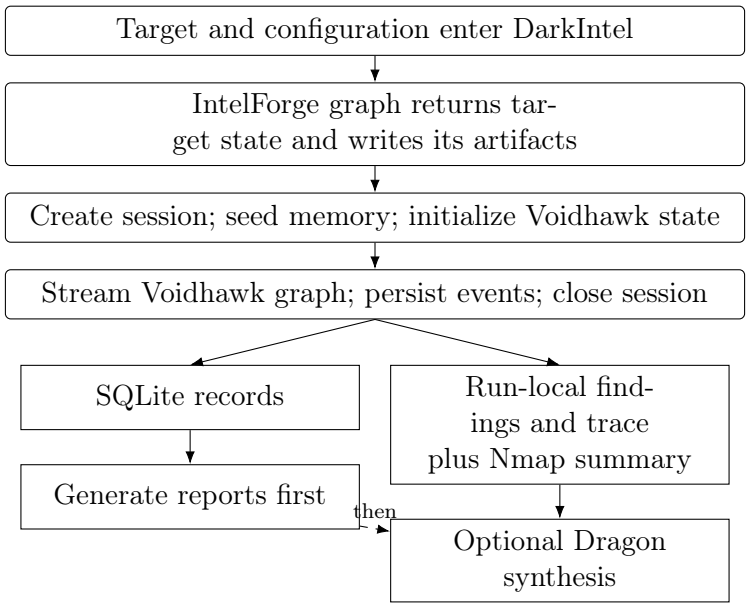


Figure 5.2: Integrated lifecycle and output paths. Reports and Dragon use different record selections; Dragon's narrative is returned separately.

On normal graph completion, DarkIntel closes the session and calls the requested PDF, HTML, or Markdown generator, or all three. Filenames use a session prefix. Optional Dragon synthesis follows, and the bridge returns the session identifier, report paths, and attack-path object. The console displays progress and Dragon's outcome. Earlier graph or reporting exceptions do not universally produce a partial successful return.

Configuration remains selective. DarkIntel passes Voidhawk's SQLite path, vector flag, embedding model, and cache settings to `MemoryAgent`, but omits its configured vector-database path, leaving the constructor default. IntelForge runs with default scan options. DarkIntel's CLI does not expose collector selection or authentication metadata; the bridge initializes the latter empty. These limits belong to the integrated interface even where standalone packages offer more settings.

5.4 Reporting and Dragon synthesis

DarkIntel reuses Voidhawk's report generators after graph completion and session closure. They receive the database path, output path, and latest validation summary, then assemble reports from persisted records rather than from the bridge's run-local findings.

Reusing the renderer avoids another reporting implementation, but makes report scope dependent on database queries.

Those queries read all findings; Markdown also reads all observations. Although the generator interfaces accept a session identifier, the relevant reads are not session-filtered, and DarkIntel does not pass one. A current-session filename can therefore contain older database records. The latest validation summary is likewise not an audit of every reported row. Session naming and evidence isolation must be assessed separately.

The latest validation summary introduces a related reporting boundary. DarkIntel retains the last nonempty summary observed in graph updates, whereas report generators can select a broader set of database rows. That summary is not a complete review history for every row. A report and a Dragon narrative can likewise contain different selections, because the former reads persistent storage and the latter uses the bridge's current-run lists. Their agreement cannot be assumed from the fact that both were produced during the same invocation.

Dragon receives run-local findings, IntelForge's Nmap summary, and a string of Voidhawk tool names, targets, and outputs. The string omits actual wrapper arguments and IntelForge command strings; caching or truncation can remove further detail. The prompt nevertheless describes exact commands from both systems and independently confirmed findings. These premises exceed the supplied evidence. The generated narrative must be interpreted according to its actual inputs, not the prompt's description of them.

With a configured model and nonempty findings, Dragon requests a JSON object containing an entry point, narrative, steps, and confidence. It parses the response but neither checks it against a command schema nor executes the steps. Missing configuration or findings produce a low-confidence placeholder without a model call. Parsing failures return a failure placeholder, and other Dragon exceptions are caught after report generation. Its output is returned and displayed separately; it does not modify the report files already written.

5.5 Failure handling and implementation constraints

Failure handling determines whether partial progress remains useful and whether the record exposes what went wrong. IntelForge can continue after collector or model-stage errors, and Voidhawk wrappers can return error text as observations. DarkIntel closes memory through a `finally` block once graph execution begins and isolates late Dragon failures. These mechanisms support continuation, but do not consistently encode whether a step failed, was skipped, or completed without a finding.

The most consequential status ambiguity lies in validation. Voidhawk passes candidates into `validated_findings` when review is disabled or raises an exception. DarkIntel persists entries under that key with `validated=True`. That flag therefore cannot

distinguish a model-confirmed candidate from bypassed or failed review. Continuing with unresolved candidates can preserve useful information, but the stored status does not preserve those different paths.

The implication is not that every retained candidate is false. Rather, the same database status can represent different evidential histories, preventing a consumer from distinguishing them without additional records. The bridge faithfully responds to the emitted state key but does not strengthen the underlying review. This separates two responsibilities: retaining candidate information after a failure can be useful, while presenting it as uniformly validated can mislead downstream reporting. The local validator checks in Chapter 6 examine these reachable paths.

Observation association has another boundary. Parallel Worker execution can append several results while the ordinary Analyst reads only the latest. The Cleaner may reuse a state-level `clean_result` when an observation lacks its own; DarkIntel's event loop has a similar update-level fallback. Without an explicit observation link, stale cleaned content can be associated with another output. Persisting all observations is therefore different from giving each an equivalent analysis pass.

A `needs_retest` route also expresses intent more readily than a fresh procedure. Retest requests accumulate in graph state, while fixed wrappers usually cannot accept a new endpoint, parameter, or payload from the Planner. Repeating a tool choice does not necessarily change the tested condition.

The cache further qualifies repeated execution. Its key represents tool and target rather than the complete invocation, environment, and session; it can return an earlier cleaned result or raw-output preview. Worker exit-code and duration fields are initialized to fixed values rather than reliably reporting subprocess measurements. Consequently, a repeated graph step cannot be assumed to be a fresh successful test, and this metadata cannot support performance claims without separate instrumentation.

Severity has a distinct evidential limit. The Validator supplies an estimated CVSS 3.1 number, which DarkIntel stores with the finding. The reviewed path does not calculate it from a recorded metric vector. The value can assist prioritization, but should not be presented as a reproducible standards-based calculation, particularly when review was bypassed or failed.

Identity can weaken before or after the handoff. IntelForge reloads an existing JSON state from its data directory; setting another target does not clear every accumulated list. Voidhawk's cache and some retrieval and report queries are not consistently session-restricted. A current target label consequently does not establish the origin of every record. Chapter 6 reports controlled reproductions of these boundaries.

Additional lifecycle limits concern naming, dependencies, and cleanup. Tool-label recovery depends on raw-file conventions, and target path separators can complicate file creation. The canonical allowed-tool list can include unavailable executables. Memory

construction, session creation, and seeding occur before DarkIntel's connection-closing `try/finally`; exceptions there are outside that cleanup boundary. Later graph or reporting exceptions close the connection but do not necessarily close the session row or return partial artifacts. These are specific failure paths, rather than evidence of a measured failure rate.

The integration thus provides explicit data transformation and execution coordination while leaving coverage, evidence status, and isolation partially enforced. These mechanisms define what can be tested: field preservation, memory insertion, routing, persistence, and failure outcomes. The next chapter examines that software evidence separately from the security effectiveness that would require a controlled target benchmark.

Chapter 6

Evaluation and Critical Analysis

The evaluation separates software correctness from security effectiveness. The first concerns transformations, state transitions, persistence, and failure handling. The second requires genuine vulnerabilities and an independently established reference. This chapter presents the available software evidence and a controlled security-evaluation protocol whose comparative outcomes remain unmeasured.

6.1 Testing strategy and controlled evaluation protocol

The packages have separate test suites. IntelForge tests target handling, configuration, data models, persistence, Nmap parsing, cleaning, agent responses, CLI behavior, and graph transitions; its fuller graph path uses fake collectors, HTTP responses, and models. Voidhawk tests cleaning, guardrails, provider selection, memory, caching, validation, and scripted graph execution. DarkIntel tests collector-label recovery, context formatting, memory seeding, console forwarding, output display, and Dragon with fake or absent model output. No DarkIntel test runs the complete bridge across both real graphs and final reporting.

The existing suites were rerun against the current local sources on 4 October 2026 using Python 3.13.12. IntelForge used its own environment; DarkIntel and Voidhawk used DarkIntel's environment with local source imports. Plugin autoload was disabled. An offline harness blocked network and assessment subprocess execution and supplied fixed output for the Voidhawk graph test's otherwise executable HTTP tool call. The suites' existing mocks and fixtures supplied model responses, target records, and temporary storage. These conditions define offline/mocked software testing, not a live security assessment.

Each test layer supports a different inference. Unit fixtures can establish parsing and cache behavior; scripted graph tests can establish routing and termination; bridge fixtures can establish memory insertion and context formatting. Failure injection can reveal status and cleanup paths. Persistence requires at least two sessions to test isolation. A canned

finding reaching Reporter demonstrates that it can be parsed and routed, not that a model would discover it on an unknown target. This follows the assessment requirement to relate conclusions to documented tests and evidence [16, 14].

A security comparison would require authorized targets with documented vulnerabilities, benign controls, and reproduction procedures. An independent reviewer should match candidate findings to that reference. Each run should use isolated IntelForge state, SQLite storage, and vector indexes. Tool and model versions, prompts, wordlists, target snapshots, network conditions, scope, and run budgets should be recorded. Repeated trials and varied execution order would help distinguish workflow effects from model variability and target drift.

Figure 6.1 presents the proposed setup, not an experiment already completed. IntelForge alone provides a reconnaissance baseline; its output is not directly equivalent to Voidhawk's reviewed findings. Voidhawk and DarkIntel can be compared under equivalent assessment scope and budgets. Validation can already be disabled by configuration, while separate seeding and handoff variants require test-only bridge changes that must be documented.

Each contrast should isolate its intended question. Standalone Voidhawk versus DarkIntel measures the combined effect of reconnaissance, seeding, and exclusion. Suppressing seeding while holding reconnaissance and allowed tools constant tests context transfer. Permitting repeated reconnaissance tests the exclusion policy's effect on duplicated actions and coverage. Comparing validation settings measures changed review decisions; only independent adjudication can establish whether those decisions reduce false positives.

A fair comparison must also distinguish a changed workflow from a changed opportunity to test. Repeating reconnaissance after IntelForge may produce additional information, not merely redundant calls. Conversely, preventing a call can lower runtime while reducing coverage. The comparison therefore needs identical target snapshots and an explicit record of allowed actions. The current CLI does not expose every proposed experimental switch, so test-only variants must be described as instrumented configurations rather than existing user features.

The observation unit must be fixed before scoring. One weakness can produce several alerts, and one title can describe different endpoints. Match findings by target, affected location, vulnerability condition, and supporting response, with ambiguous matches reviewed independently. Count attempted, completed, failed, and cached tool calls separately so that omitted work is not mistaken for efficiency.

Collect raw observations and counts before derived metrics: tool invocations, repeated reconnaissance, candidates, review decisions, cache hits, retrieval and model calls, and report entries with traceable evidence. Precision and recall require the reference finding set and a declared matching rule. Runtime and cost require consistent execution boundaries

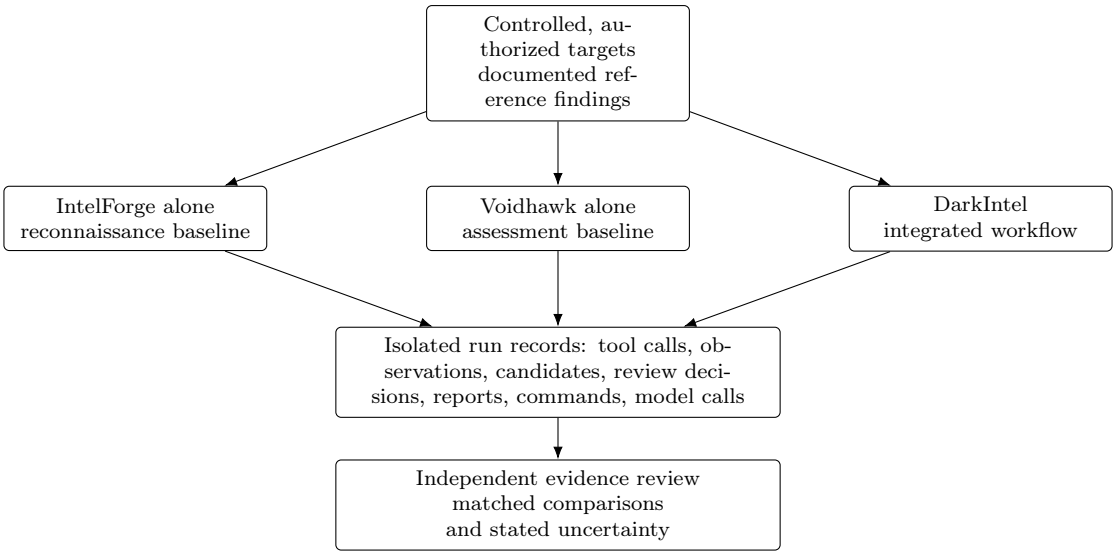


Figure 6.1: Proposed controlled evaluation setup. No comparative security results from this protocol are claimed in this report.

and recorded usage; pytest durations are not assessment runtimes. Report completeness should check source links, exact invocations, review status, and remediation. Missing provenance must remain missing rather than be inferred from a summary.

Evidence traceability is a useful measure even before detection accuracy can be established. For each accepted finding, an evaluator can check the target and session, exact invocation or request, source response, review decision, and link between them. A rendered report section alone does not meet that criterion. Similarly, cache reuse and retrieval usage should be counted separately from fresh collection: their presence demonstrates reuse, while their relevance and effect require additional inspection. This keeps software activity from becoming an unsupported proxy for assessment quality.

Failure trials should be analyzed separately from detection trials. Removing a tool, injecting a provider error, changing cache state, switching targets, or interrupting reporting can reveal which artifacts survive and whether failures remain visible. The recorded outcomes should include return status, session closure, written artifacts, and finding status. These trials measure resilience and consistency, not vulnerability absence.

6.2 Results and comparative evaluation

All 157 tests passed in the offline/mocked rerun (Table 6.1). These are test-case counts, not coverage percentages, vulnerabilities detected, or independent security trials. The result supports the assertions exercised by each suite under the stated harness; it does not establish a comparative advantage for DarkIntel.

Six additional local checks were reported during the implementation analysis, using synthetic records and temporary storage. They exercised the actual target-state, bridge-

Suite	Verified result	What the result supports, and its boundary
IntelForge	64 passed	Parsing, state, CLI, agents, and a fake-driven graph path behave as asserted. The result does not measure reconnaissance completeness on real targets.
Voidhawk	78 passed	Memory, cleaning, guardrails, validation paths, and a scripted graph path behave as asserted. A canned finding and review response do not establish detection accuracy.
DarkIntel	15 passed	Selected bridge mappings, memory seeding, CLI forwarding, output display, and Dragon behavior with fake input work as asserted. The full integrated `run()` path is not exercised by this suite.

Table 6.1: Offline/mocked pytest outcomes verified on 4 October 2026. No live security-effectiveness claim follows from these results.

memory, cache, validator, and Markdown reporting methods without an external target or live model. Table 6.2 retains their setups and observed outcomes separately from the current suite rerun.

The checks used two synthetic targets or sessions where isolation was relevant, a temporary database with vector search disabled, and a temporary IntelForge directory. Validator tests either disabled review or injected a provider exception. The provenance check seeded a command with distinct cleaned output and inspected the resulting observation and Dragon history. These minimal setups isolate reachable behavior; they do not estimate its frequency during an assessment.

For the reporting check, two fixture findings were stored in different sessions and the Markdown generator was given the second session identifier; the resulting file was inspected for both titles. The cache check requested the same stored command under a second session. In the validation error check, the structured-model invocation raised a synthetic provider exception. These conditions are important to reproduction: they test the actual methods under controlled inputs without relying on a scanner finding, a live provider response, or a security event.

The isolation checks show why a comparison cannot casually reuse default storage. Earlier target lists can alter the apparent reconnaissance input, a cache can substitute old output for fresh execution, and a report can include another session's findings. This does not prove that a particular historical report is incorrect. It establishes the need to isolate storage and verify row attribution before comparing assessment outcomes.

The provenance and validator checks establish different boundaries: exact command information is lost in the downstream representation, and a validated-field entry does not identify a uniform review path. The source mechanisms were explained in Chapter 5; these probes demonstrate their reachability with controlled input. No true-positive or false-positive rate can be inferred from them.

Local check and setup	Observed result and interpretation
Reload IntelForge state for target A, then set target B in the same directory	A port stored for A remained after `set_target(B)`. The target label changes without clearing earlier reconnaissance lists; a fresh directory is needed for clean-run comparisons.
Store a cleaned command in session A; request the same command with session B	The cache returned A's row to B. The lookup is keyed by command and lifetime, not restricted by the supplied session identifier.
Store one synthetic finding in each of two sessions; render Markdown with B's session ID	The generated file contained both findings. Passing a session ID to this reporter did not isolate its database read.
Seed a command result with an exact Nmap command and cleaned output	The Voidhawk observation contained the inferred tool and cleaned output; the bridge's Dragon history string did not contain the recorded Nmap flags. The probe establishes information loss, not an incorrect security result.
Disable Voidhawk's Validator for one synthetic candidate	The candidate appeared in `validated_findings` without review. This confirms the disabled-review pass-through path.
Raise a synthetic provider exception during Validator review	The candidate again appeared in `validated_findings`, with a summary stating that validation failed and the finding passed through unvalidated.

Table 6.2: Reproduced local behaviors using synthetic data. Targets and findings in these checks were fixtures, not security discoveries.

No matched benchmark of standalone and integrated assessment is available. There are no adjudicated detection results or comparative runtime, cost, or duplicate-work measurements. The supported result is software-level interoperability in tested paths, together with reproduced evidence-management limitations. Complete integrated execution and security effectiveness require the protocol described in Section 6.1.

6.3 Technical limitations and threats to validity

Implementation limitations. The constraints in Section 5.5 affect measurement as well as operation. Static reconnaissance exclusion can omit coverage, fixed wrappers can limit follow-up, and incomplete analysis of parallel observations can leave collected data unused. Cached or stale content can weaken the connection between a graph step and fresh evidence. These properties must be accounted for when defining completed tests and reportable findings; implementation labels cannot serve as experimental ground truth.

Experimental limitations. The evidence consists of existing suite assertions and local synthetic probes. It includes neither a benchmark-scale target set nor a complete bridge test spanning both graphs and reporting. Mocks establish responses to specified inputs, while the probes establish reachable branches rather than prevalence or operational impact. The passing count does not measure branch coverage or overall test quality. These limits constrain the conclusions that can be drawn from a successful software run.

Threats to internal validity. A comparison could attribute differences to integration when they actually arise from model variability, unavailable tools, changed targets, old state, or cache reuse. Error text can enter observations, and incomplete execution metadata can misclassify a graph step as a successful test. Matching findings only by title risks double counting or false equivalence. Isolated storage, exact traces, explicit failure states, repeated runs, and independent adjudication would reduce these confounders.

Threats to external validity. Local fixtures do not represent arbitrary applications, authentication flows, network defenses, or rate limits. Different wordlists, installed tools, provider behavior, and network permissions can change what the workflows observe. Even successful controlled testing would apply first to its declared targets and conditions. Broader effectiveness would require varied authorized environments and independent replication. The current evidence supports targeted software claims and identifies what subsequent security evaluation must establish.

Chapter 7

Discussion and Perspectives

The evaluation establishes tested software behavior while leaving security effectiveness unresolved. The architectural question is what those findings mean for the prototype's maturity and evolution. This chapter interprets the principal trade-offs and prioritizes improvements without repeating the implementation or test results.

7.1 Discussion of the Proposed Architecture

The separation of IntelForge, Voidhawk, and DarkIntel is appropriate to their different control needs. A predictable reconnaissance sequence can inform an observation-dependent assessment loop without forcing both into one graph. DarkIntel makes their boundary inspectable and preserves standalone use. However, the bridge depends on specific fields, memory methods, and tool names, so package separation does not eliminate compatibility costs. The implementation supports modular composition; stable semantic contracts remain an architectural responsibility.

In-process integration is proportionate to a prototype: it avoids separate services and makes transformations directly accessible. The cost is shared dependence on package configuration and failures across several persistence mechanisms. JSON state, database records, indexes, and reports do not form one transaction. A reliable assessment unit therefore requires explicit identity and completion status across those artifacts, even if the deployment remains a single process.

Controlled wrappers impose another useful but incomplete boundary. They restrict which actions a model can request, while their fixed inputs limit focused testing. Persistent memory likewise provides continuity while requiring rules for applicability and freshness. In both cases, more flexibility is not automatically better: the architectural objective is an appropriate action grounded in relevant evidence. The current design provides recognizable control points, but the observed status and isolation issues show that those controls do not yet support uniform assurance about every finding.

7.2 Technical Limitations

The most significant limitations concern the meaning of stored state. A review flag that represents several distinct paths cannot reliably communicate confidence to a report reader. Likewise, a severity estimate without its metric vector is difficult to reproduce. These are contract weaknesses at the point where interpretations become conclusions. Their importance extends beyond a particular model's accuracy: even a correct interpretation becomes difficult to audit if its review path is unclear.

Provenance and isolation determine whether an assessment can be reconstructed. Missing command links and inconsistent session boundaries can leave a plausible record whose origin is uncertain. The distinction between database reports and Dragon's selected inputs further complicates comparison of the outputs. More detailed prose would not repair this ambiguity; the underlying records must identify their sources and transformations consistently.

Coverage and execution constraints also qualify adaptability. The findings in Chapters 5 and 6 show why another graph iteration is not necessarily a new test, and why persisted observations are not necessarily fully analyzed. The architectural implication is that completion should refer to a defined action and its outcome, rather than a tool name or phase alone. This would let the system distinguish useful reuse from stale evidence and deliberate exclusion from missing coverage.

External dependencies and limited experimental scope constrain maturity claims. The prototype has no demonstrated throughput or isolation guarantee for concurrent multi-target use, and software tests do not establish operational detection performance. These are unresolved properties to measure, not evidence of a quantified scalability defect. Improvements should first make the execution record dependable enough for such measurements.

7.3 Improvement Perspectives and Future Extensions

A. Short-term engineering improvements. The first priority is explicit finding status. Separate unreviewed, accepted, rejected, failed-review, and technically reproduced states would prevent disabled or unsuccessful validation from silently strengthening a claim. CVSS values should retain documented metric vectors when evidence supports calculation and otherwise remain labeled estimates. Transition tests and injected provider errors could verify that reports preserve the actual review path.

Observation handling should then be corrected within each run. Commands, execution status, raw-output links, and target/session identity should remain attached through cleaning and analysis. Every parallel result should receive its own analysis pass, and retesting should require fresh execution when cached evidence cannot answer the ques-

tion. Two-observation graph fixtures and instrumented retests could check both complete processing and the source of each returned result.

Bounded wrapper parameters could support discovered endpoints without permitting unrestricted shell commands. Immediate session filtering in report generation would address cross-run leakage, while Dragon should receive the same evidence selection with its unexecuted proposals clearly identified. Parameter-boundary tests and two-session report fixtures would evaluate these changes without assuming improved detection.

B. Medium-term architectural improvements. A versioned handoff contract should describe observations, uncertainty, source links, and per-collector coverage. Exclusion could then depend on successful equivalent work rather than a static completed-tool list. This would make partial reconnaissance visible to planning. Compatibility tests and scripted collector failures could establish whether changes in either package preserve the intended handoff.

A shared run identity and evidence manifest could connect IntelForge state, Voidhawk memory, cache, findings, reports, and Dragon. Each derived claim would link to its source and transformation history, while stage outcomes would distinguish completion from partial failure. This addresses fragmentation without requiring one storage technology. Multi-run regression tests and report-to-command audits could evaluate isolation and reconstructability.

Orchestration and retrieval policies should use those records. Coverage summaries, cache status, and scoped retrieval would help planning distinguish a useful next action from repeated or unrelated work. Controlled variants could measure duplicate actions, missed follow-up, and evidence traceability with and without those policies. This would evaluate the value of context quality rather than merely the amount stored.

C. Longer-term research perspectives. The protocol in Section 6.1 provides the next research step: determine whether integration improves assessment outcomes under controlled conditions. IntelForge serves as a reconnaissance baseline, while Voidhawk and DarkIntel support matched assessment comparisons. Seeding, review, and exclusion variants would help attribute differences to particular mechanisms. Independent adjudication and repeated runs would be necessary for defensible detection, cost, runtime, and robustness measurements.

Model and retrieval comparisons could then use the same frozen observations. No retrieval, scoped operational memory, and a separately curated security corpus would test different sources of context. Independently verified outcomes could reveal whether proposed findings and confidence levels are well grounded. This would distinguish improved reasoning from more fluent explanation and assess false acceptance as well as missed valid findings.

Finally, constrained parameter generation and larger multi-target studies could explore the limits of the architecture. Typed inputs, explicit scope, and execution logs would bound generated actions; independent evidence would measure their usefulness. Larger controlled runs could examine cost, latency, isolation, and failure recovery. These perspectives extend the prototype only after its evidence contracts are dependable, providing the basis for the report's final assessment of its contribution.

General Conclusion

Security assessment requires continuity between reconnaissance, investigation, and reporting. When these stages use separate tools and data representations, relevant context can be lost, work can be repeated, and conclusions can become detached from their supporting observations. This project addressed that engineering problem by connecting reconnaissance intelligence to an iterative assessment workflow.

IntelForge provides collection and structured intelligence generation. Voidhawk provides adaptive tool selection, operational memory, analysis, and finding review. DarkIntel connects them by transforming selected reconnaissance results into observations, reasoning memories, and initial planning context, then coordinating execution and reporting. Dragon supplies an optional narrative synthesis after report generation. The principal contribution is this integration of complementary subsystems, built on existing security tools, libraries, and upstream work.

The engineering value lies in making reconnaissance available to downstream decisions while retaining an identifiable boundary between collection, interpretation, and review. It also exposes assumptions that would be harder to examine if the two workflows were treated as one opaque process. The integration can therefore be assessed through its transformations and execution records, independently of claims about a model's intelligence.

The implementation makes the handoff explicit and inspectable. The available offline and mocked tests support selected mappings, memory operations, graph transitions, and output paths. Controlled local checks also reproduced important implementation behaviors. This evidence demonstrates software properties under defined conditions; it does not establish the accuracy of vulnerability detection or a measured advantage over standalone assessment. A complete comparative security benchmark remains necessary.

The prototype's limits concern the strength and continuity of evidence. Model-based review is not independent technical verification, and failed or disabled validation can produce ambiguous finding status. Incomplete provenance and session isolation affect reproducibility, while retesting, parallel-observation analysis, and fixed tool parameters constrain follow-up. These limitations identify concrete engineering priorities rather than invalidate the integration result.

DarkIntel establishes a solid architectural foundation for more coherent AI-assisted security assessment. Its value lies in connecting previously separate workflows while making their assumptions and evidence boundaries visible. Stronger validation states, consistent provenance, isolated operational memory, and controlled comparative evaluation would allow future work to determine whether that foundation also improves assessment reliability and effectiveness. The contribution is therefore both a working integration prototype and a clearly defined basis for further evidence-grounded development.

Bibliography

[1] Chroma. Introduction to Chroma. Official documentation, n.d. URL `https://docs.trychroma.com/docs/overview/introduction`. Accessed 2026-10-01.

[2] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. PentestGPT: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864. USENIX Association, 2024. URL `https://www.usenix.org/conference/usenixsecurity24/presentation/deng`.

[3] Forum of Incident Response and Security Teams. Common vulnerability scoring system version 3.1: Specification document. Official specification, n.d. URL `https://www.first.org/cvss/v3.1/specification-document`. Accessed 2026-10-01.

[4] Wael Hammali. DarkIntel: Source code. Git repository; local source snapshot inspected, 2026. URL `https://github.com/WaelHammali/DarkIntel`. Reviewed revision fe6dd8d. Implementation evidence is from the local checkout.

[5] Wael Hammali. IntelForge: Source code. Git repository; local source snapshot inspected, 2026. URL `https://github.com/WaelHammali/IntelForge`. Reviewed revision 81d5f33. Implementation evidence is from the local checkout.

[6] Keystone. Keystone: Cybersecurity, AI and digital trust. Official company website, n.d.. URL `https://www.keystone-corporation.com/`. Accessed 2026-10-04; destination of the official keystone.tn website.

[7] Keystone. Red team operations. Official service description, n.d.. URL `https://www.keystone-corporation.com/services/red-team`. Accessed 2026-10-04.

[8] Keystone Groupe. VoidHawk: Source code. Git repository; local source snapshot inspected, 2026. URL `https://github.com/WaelHammali/VoidHawk`. Reviewed revision 21aeadf. Extended fork of Watchtower; the license credits fzn0x for the original project. Implementation evidence is from the local checkout.

[9] LangChain. LangGraph: Graph API overview. Official documentation, n.d.. URL https://docs.langchain.com/oss/python/langgraph/graph-api. Accessed 2026-10-01.

[10] LangChain. LangGraph: Persistence. Official documentation, n.d.. URL https://docs.langchain.com/oss/python/langgraph/persistence. Accessed 2026-10-01.

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/6b493230-Abstract.html.

[12] OWASP Foundation. OWASP top 10 for LLM applications 2025. OWASP Generative AI Security Project, 2025. URL https://genai.owasp.org/llm-top-10/.

[13] OWASP Foundation. Web Security Testing Guide, version 4.2: Introduction. Versioned online documentation, n.d.. URL https://wstg.owasp.org/v4.2/2-Introduction/. Accessed 2026-10-01.

[14] OWASP Foundation. Web Security Testing Guide, version 4.2: Reporting. Versioned online documentation, n.d.. URL https://wstg.owasp.org/v4.2/5-Reporting/. Accessed 2026-10-01.

[15] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3982–3992. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1410. URL https://aclanthology.org/D19-1410/.

[16] Karen Scarfone, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. Technical guide to information security testing and assessment. Special Publication 800-115, National Institute of Standards and Technology, September 2008. URL https://csrc.nist.gov/pubs/sp/800/115/final.

[17] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.