THE DEVSECOPS DIGITAL TWIN: A UNIFIED GRAPH ARCHITECTURE FOR PROACTIVE LIFECYCLE SECURITY

BY

KSHITIJ TAPRE

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Applied Cybersecurity and Digital Forensics
at the Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
December 2025

© Copyright by
KSHITIJ TAPRE
2025

AUTHORSHIP STATEMENT

I, Kshitij Tapre, attest that the work in this thesis is substantially my own.

In accordance with the disciplinary norm of authorship in Information Technology Management and Security (ITMS), the following collaborations occurred in the thesis. (Please consult Appendix S of the IIT Faculty Handbook for further information.)

My adviser, Dr. Maurice Dawson, contributed to the conceptualization and progress of the research project, as is the norm for a Master of Science supervisor.

No other collaborations occurred in the creation of this thesis.

No portion of this thesis is based on any of my prior publications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The modern software development landscape is defined by a fundamental tension between velocity and security [1, 2]. The widespread adoption of DevOps and CI/CD has dramatically accelerated software delivery but has outpaced traditional security paradigms, leading to a fragmented, reactive security posture [2, 3]. This thesis posits that the inherent limitations of current SDLC security can be overcome by adopting a fundamentally different architectural approach: a formally defined, ontology-driven knowledge graph [4, 5]. We introduce the "DevSecOps Digital Twin," a model designed to serve as a single, authoritative source of truth that comprehensively captures the entities, relationships, and dynamic processes of the entire SDLC [6, 7]. By establishing a rigorous, machine-readable schema, this model transforms isolated security data into a coherent, queryable knowledge base, enabling a truly proactive security posture through advanced simulation, automated impact analysis, and predictive risk assessment [7, 2].

CHAPTER 1

INTRODUCTION: THE CHALLENGE OF SECURING THE MODERN SDLC AT VELOCITY

The core conflict in modern software development is the tension between the demand for high-velocity delivery and the need for robust security [1, 2]. Methodologies like DevOps and Continuous Integration/Continuous Deployment (CI/CD) have successfully solved the problem of speed, allowing organizations to innovate at an unprecedented rate [2, 8]. **This very success, however, has created a new challenge:** the security models built for slower, monolithic development cycles have failed to keep pace [3]. Legacy security models, designed for slower, waterfall-style development cycles, are ill-equipped to handle the complexity, dynamism, and sheer volume of changes inherent in modern software delivery [3]. The result is a fragmented and reactive security posture, characterized by a proliferation of disconnected point solutions (a phenomenon often described as "tool sprawl") and the creation of isolated "data silos" that prevent a holistic understanding of risk [9]. This fragmentation leads to a critical condition that may be termed "incomplete coverage." This is not merely a matter of lacking a specific type of security scanner; rather, it is a semantic deficiency. It represents the absence of a unified context that connects the disparate artifacts and events across the entire Software Development Lifecycle (SDLC) [9].

For instance, a Static Application Security Testing (SAST) tool may identify a critical flaw in a specific code file, and a Software Composition Analysis (SCA) tool may flag a known vulnerability in a third-party library [2]. Yet, in a typical fragmented toolchain, no single system possesses the inherent knowledge to trace the causal chain from the business Requirement that necessitated the flawed code, to the Developer who authored the commit, to the CI/CD Build that incorporated the

vulnerable library, and finally to the specific production Service that is now exposed to attack [2]. This inability to reason across domains and lifecycle stages is the principal source of security blind spots, operational inefficiencies, and, ultimately, organizational risk [10]. The lack of a common, queryable representation of the SDLC forces security teams into a perpetual state of manual data correlation, a task that is both time-consuming and prone to error, especially during high-pressure incident response scenarios [11]. This thesis posits that the inherent limitations of current SDLC security paradigms can only be overcome by adopting a fundamentally different architectural approach: a formally defined, ontology-driven knowledge graph [4]. This model, conceptualized as a "DevSecOps Digital Twin," is designed to serve as a single, authoritative source of truth that comprehensively captures the entities, their multifaceted relationships, and the dynamic processes of the entire SDLC [7]. By establishing a rigorous, machine-readable schema for the software delivery process, this model transforms a chaotic stream of isolated security data points into a coherent, interconnected, and queryable knowledge base [5]. Such a foundation moves beyond reactive vulnerability detection, enabling a truly proactive security posture through advanced simulation, automated impact analysis, and predictive risk assessment [12].

CHAPTER 2

BACKGROUND AND RELATED WORK: THE FRAGMENTED LANDSCAPE OF SDLC SECURITY

This Chapter reviews the evolution of SDLC security to establish the context for the novel approach proposed in this thesis [3]. It will first trace the development from standalone tools to integrated platforms, demonstrating how architectural limitations created the problem of incomplete coverage [2]. **Following this, the chapter provides a quantitative analysis of the economic unsustainability of this fragmented paradigm**, measured by its impact on developer productivity and operational efficiency [13].

2.1 The Evolution from Point Solutions to Integrated Platforms

The history of application security is one of progressive specialization, leading to the complex toolchains in use today [8]. The initial approach involved isolated, phase-specific scanning tools, each designed to address a particular class of vulnerability at a specific point in the development process [14]. SAST tools were introduced to analyze source code for common coding errors before compilation, while DAST tools were developed to test running applications for vulnerabilities that manifest only at runtime [8]. The rise of open-source software led to the development of SCA tools to identify and manage vulnerabilities within third-party dependencies [14]. While each of these tools is essential, their deployment in isolation created a disjointed security process [2]. The DevOps movement, with its emphasis on automation and integration, spurred the creation of DevSecOps, a philosophy aimed at embedding these security checks directly into the CI/CD pipeline [8]. This led to the emergence of complex, multi-vendor toolchains where various scanners are orchestrated to run automatically on every code change [10]. However, this organic evolution, while im-

proving automation, exacerbated the problem of data fragmentation [9]. Each tool produced its own alerts in its own format, with no shared context, leading to an overwhelming volume of uncorrelated findings and significant "alert fatigue" for security and development teams [10]. This challenge set the stage for the development of more integrated platforms designed to manage this complexity [15].

2.2 Software Supply Chain Security (SSCS): Securing the "What"

The discipline of Software Supply Chain Security (SSCS) emerged as a direct and urgent response to a series of high-profile, catastrophic breaches that exploited weaknesses in how software is built, assembled, and distributed [16]. The 2020 SolarWinds attack and the 2021 Log4j vulnerability fundamentally shifted the industry's focus, demonstrating that an organization's security posture was inextricably linked to the security of every component in its software supply chain [16]. This realization spurred regulatory action, such as the U.S. Executive Order 14028, which mandated the use of a Software Bill of Materials (SBOM) for software sold to the federal government [17]. This market pressure led to the rapid maturation of SSCS platforms from vendors such as Aqua Security, Anchore, Checkmarx, and Black Duck [17]. The core competency of these platforms is to secure the "what" of the SDLC(the tangible assets and their provenance) [18]. Their primary capabilities include dependency management, SBOM generation, integrity and provenance controls, and container scanning [18, 19, 20, 21]. SSCS tools excel at modeling the direct, tangible provenance chain: a Repository contains a Commit, which triggers a Build, which produces an Artifact that depends on various Library components [2]. However, this focused strength also defines their primary limitation: a **context gap** [22]. While SSCS platforms are adept at identifying a vulnerable component, they typically lack the broader semantic understanding of the lifecycle that surrounds that component [22]. They often cannot natively answer more complex, context-rich questions about business requirements,

threat models, mitigating controls, or the specific production services impacted [2]. SSCS platforms secure the individual links in the supply chain but do not model the entire chain of business and operational context [22].

2.3 Application Security Posture Management (ASPM): Managing the "Noise"

The proliferation of automated security tools within DevSecOps pipelines created a new problem: "alert fatigue" [10]. The sheer volume of alerts overwhelmed security and development teams [23]. In response, the industry analyst firm Gartner defined a new category of tools: Application Security Posture Management (ASPM) [15]. ASPM platforms, from vendors including Apiiro, ArmorCode, and Wiz, serve as a central management and visibility layer, ingesting and making sense of data from other tools [24, 23, 25]. Key capabilities of ASPM solutions include data aggregation, correlation and de-duplication, contextual prioritization, and workflow automation [15, 26, 27]. The rapid emergence and projected growth of the ASPM market is a powerful validation of this thesis's core problem statement [15]. Gartner predicts that by 2026, over 40% of organizations developing their own applications will adopt ASPM, up from just 5% in 2023 [15]. Despite their value, current ASPM platforms suffer from a fundamental architectural limitation: a **semantic gap** [22]. They are primarily after-the-fact correlation engines [28]. They ingest structured data but lack a native, formal understanding of the SDLC itself [28]. While some advanced ASPM vendors are building their own internal graph-based models, these are typically proprietary, black-box implementations that lack the formal, extensible, and transparent ontological foundation proposed in this work [28]. ASPM platforms treat the symptom of alert fatigue rather than addressing the root cause of semantic fragmentation across the SDLC [22].

2.4 Quantifying the Cost of Incomplete Coverage

The fragmented nature of the current SDLC security landscape imposes significant and quantifiable costs [13]. These costs manifest not only in heightened security risk but also in degraded operational efficiency and reduced developer productivity [9]. The status quo is economically inefficient [9].

A survey found that developers navigate an average of 7.4 different tools, wi, 75% reporting a loss of six to 15 hours per week due to "tool sprawl" [9]. This constant context-switching fragments developer focus and saps productivity [29]. The same survey revealed that 55% of developers do not trust the data in their tool repositories [29].

The inefficiency extends beyond human labor to the **technical cost of data integration**. The absence of a unified schema requires organizations to expend significant resources on **Extract, Transform, and Load (ETL)** pipelines, custom scripts, and normalization layers just to aggregate security findings into a central repository [22]. This manual data plumbing is brittle, prone to error, and adds latency to the analysis pipeline, transforming security data from a real-time asset into a costly, batch-processed liability [15].

Furthermore, fragmentation creates a substantial cost burden related to **regulatory compliance and security auditing** [30]. Proving adherence to standards like **PCI DSS** or **FedRAMP** necessitates manually correlating evidence across multiple siloed systems, linking a code vulnerability (SAST tool) to a mitigating control (policy document) and its runtime location (deployment tool) [30]. This labor-intensive reporting is a direct consequence of the missing semantic link between technical artifacts and business context [22].

In the security domain, fragmentation creates dangerous delays [11]. When

data is siloed, incident response becomes a slow, manual process, increasing the Mean Time to Respond (MTTR) and giving attackers more dwell time [11, 10]. The ultimate cost is realized in security breaches [13]. The global average cost of a data breach has reached $5.08 million [13]. A significant factor is the failure to remediate known vulnerabilities, with one report noting a 61% surge in the exploitation of vulnerabilities for which a patch was available for over 30 days [13]. This creates an *inefficiency-risk spiral*, where tool sprawl reduces productivity, increases MTTR, and elevates the likelihood of a costly breach [22]. The unified graph model proposed here is designed to break this cycle [4].

CHAPTER 3

METHODOLOGY: DESIGN SCIENCE RESEARCH AND ONTOLOGY DEVELOPMENT

This chapter details the research design and the specific process used to construct and validate the DevSecOps Digital Twin ontology. The research adopts the **Design Science Research (DSR)** posture, which involves building a purposeful artifact (the ontology) to address a real-world problem (fragmented SDLC security) and evaluating its quality and utility [31].

3.1 Research Design and Process Model

The overall research logic is grounded in DSR, with the **DevSecOps Digital Twin Ontology** serving as the core artifact [31]. The artifact's value is assessed based on its quality (correctness, completeness) and utility (usefulness in answering competency questions), rather than system performance [32].

3.2 Ontology Development Method

The ontology will be developed using a recognized, staged methodology such as **METHONTOLOGY** [33]. This process will be broken down into clear phases:

- **Conceptualization:** Defining the initial glossary of terms and informal relationships [33].
- **Formalization:** Specifying the formal axioms, classes, and properties (Chapter 5) [33].
- **Implementation (of the Artifact):** Coding the ontology using the **Property Graph Model (PGM)** and defining the schema in a graph database (e.g., **Neo4j**) [34].

- **Evaluation:** A rigorous, multi-method validation process (Chapter 6) [35].

3.3 Evidence Sources and Stakeholders

Evidence will be systematically collected from multiple sources to ensure rigor:

- **Systematic Literature Review (SLR):** A defined protocol (databases, queries, inclusion/exclusion criteria) was used to build the initial corpus of terms and identify candidate relations [36].
- **Standards and Policies:** Industry standards (e.g., CIS Controls, NIST Frameworks) were analyzed for conceptual alignment [37].

CHAPTER 4

REQUIREMENTS AND CONCEPTUALIZATION

This chapter formally establishes the requirements for the ontology and presents the initial conceptual model, ensuring the final artifact is fit for purpose and directly addresses the architectural and semantic gaps identified in Chapter 2.

4.1 Domain Analysis and Glossary

The chapter first presents a precise **boundary definition** of the DevSecOps domain, focusing on the structured and dynamic interconnection of four primary planes: **Identity (People)**, **Design/Business (Process)**, **Code/Supply Chain (Technology)**, and **Runtime/Finding (Risk)** [35]. The ontology is bounded by the need to capture all entities and relationships required to answer complex **transitive security queries** across the entire software delivery value stream (Plan-Code-Build-Test-Release-Deploy-Operate-Monitor).

This is followed by a **glossary of terms** captured during the Systematic Literature Review (SLR) and domain analysis, providing preferred labels, alternative labels, and clear definitions, with sources for traceability [33].

4.1.1 Glossary of Key Ontology Terms.

Table 4.1: Glossary of Core DevSecOps Ontology Terms (Nodes)

Node Label	**Precise Definition and Rationale**
`Commit`	A specific version of code in a `Repository`, uniquely identified by a cryptographic hash. It serves as the canonical source node for all `Provenance` edges and is authored by an `Identity` (Developer or Bot).
`Artifact`	An **immutable, versioned output** (e.g., container image, compiled binary) of a successful `Build` process, characterized by a cryptographic digest (`digest`) to ensure integrity and provenance.
`Control`	A **safeguard or countermeasure** implemented to mitigate a `Threat` or `Vulnerability`. It is essential for compliance mapping, characterized by its effectiveness (`efficacyScore`) and `implementationStatus`.
`Developer`	A **human** `Identity` **sub-class** responsible for authoring `Commit`s or being `ASSIGNED_TO` a `Requirement`, essential for establishing accountability and risk ownership.

Continued on next page

Table 4.1 – continued from previous page

Node Label	Precise Definition and Rationale
`Environment`	A **logical deployment target** (e.g., Development, Staging, Production) for `Service`s, critical for risk context due to properties like `sensitivityLevel` (e.g., PCI-handling) and `externalFacing`.
`Finding`	An **abstract observation** from any security tool (SAST, SCA, DAST, etc.) indicating a deviation from a security baseline. Specialized into `Vulnerability`, `Secret`, or `Misconfiguration`.
`Provenance`	A meta-concept representing the **documented, verifiable history** of an `Artifact` linking it through `Build`, `Commit`, `Repository`, and `Developer` [2].
`Requirement`	A **documented business or security need** that drives development (e.g., User Story, Audit Mandate), providing the semantic link between business goals and technical implementation.

4.2 Competency Questions (CQs) and Usage Scenarios

The core function of the ontology is defined by the **Competency Questions (CQs)** it must be able to answer in natural language [35]. These CQs are designed to explicitly test the ontology's capacity for **Relationship Depth** and cross-domain querying, which traditional siloed tools cannot achieve.

- **CQ 1 (Toxic Combination Risk):** "Show me all `Service`s in the production `Environment` flagged as `pciCompliant` that are running an `Artifact` built from a `Commit` by any `Developer` with the role `Contractor` that directly `DEPENDS_ON` a `Library` with a `CRITICAL Vulnerability`."

- **CQ 2 (Proactive Threat Assessment):** "Given a newly proposed architecture for a service that involves `unvalidated_input` data flows, identify all `Threat`s from the `ThreatModel` that would be introduced, and list the `Control`s that `MITIGATE` these `Threat`s that are *not yet* implemented." [38]

- **CQ 3 (Reverse Audit Traceability):** "For `Vulnerability CVE-2025-0101` currently active on `Service API-Payment-v2`, trace the complete `BUILT_FROM` chain back to the originating `Commit` and the `Developer` who `AUTHORED` the code, and identify the `Policy` that was `VIOLATES` during the `Build` step."

- **CQ 4 (Policy Enforcement Gaps):** "Find all `Artifact`s that `DEPENDS_ON` a `Library` with a `copyleft License` (e.g., GPL), and that are subsequently `DEPLOYED_TO` any `externalFacing Service` not yet associated with a `License_Violation Finding`."

4.3 Conceptual Model and Design Principles The initial, informal conceptual model is visualized using diagrams (e.g., UML or ER-like) to show the main node types and relationships before formalization [33]. This model consists of a **hierarchical, three-tiered ontology** structure: Upper, Domain, and System.

The design is guided by fundamental ontological principles to ensure quality, rigor, and maintainability [39]:

- **Minimal Ontological Commitment:** The ontology commits only to those distinctions that are strictly necessary for its security and accountability pur-

pose. For instance, distinguishing between a `Developer` and a `CI/CD Bot` (both sub-classes of `Identity`) is essential for non-repudiation and policy enforcement, but non-security-relevant sub-types of testing tools are avoided [39].

- **Orthogonality:** Concepts are designed to be mutually independent to avoid redundancy. The `Vulnerability` concept is orthogonal to the `Library` concept; their relationship is mediated only by the `HAS_VULNERABILITY` edge, ensuring a clean, non-redundant data model [39].

- **Reuse Policy:** Explicitly documenting imported vocabularies (`PROV-O` for provenance, `CWE` for flaw types, etc.) versus new terms [39]. New terms are only created where existing standards fail to capture the specific, multi-domain relational semantics of the unified DevSecOps lifecycle.

CHAPTER 5

ONTOLOGY SPECIFICATION

This chapter presents the formal, three-tiered specification of the DevSecOps Digital Twin Ontology. This specification defines the explicit, machine-readable schema required to construct the unifying knowledge graph artifact, transforming fragmented security data into a coherent and queryable model.

5.1 The Three-Tiered Foundational Ontology

To manage the complexity and ensure both domain relevance and general extensibility, the model adopts a hierarchical, three-tiered ontology architecture, a recognized best practice in ontology engineering [2]. This structure ensures a clear separation of concerns, allowing the model to be abstract and reusable at the top tier while remaining precisely tailored to a specific organizational environment at the bottom.

- **Upper Ontology (Tier 1 - The Abstract Foundation):** This tier contains abstract, domain-neutral concepts that provide the highest level of generalization. Core classes like `Asset`, `Identity`, `Event`, and `Finding` are defined here. This tier ensures the model's interoperability and adherence to foundational ontological principles (e.g., FOAF for Identity, D-FOAF for Event) [39, 34].

- **Domain Ontology (Tier 2 - The DevSecOps Vocabulary):** This middle tier specializes the upper ontology concepts for the specific DevSecOps domain. It defines core vocabulary and relationships specific to the software lifecycle, such as `Vulnerability`, `Build`, `ThreatModel`, and `Control`. These classes

capture the security and process semantics required to answer the Competency Questions (CQs) [4].

- **System Ontology (Tier 3 - The Concrete Instances):** This is the most concrete and organization-specific tier. It represents specific tool instances and artifacts within an organization's environment, such as `JiraTicket`, `GitHubRepo`, `JenkinsJob`, and vendor-specific findings like `SnykFinding`. This tier ensures the graph can be directly populated by the continuous integration of real-world data feeds [22].

5.2 Formalization and Axioms

This section presents the formalized ontology, detailing the defined classes, properties (Object and Data), constraints, and logical axioms that govern the model [33]. The ontology is implemented using the **Property Graph Model (PGM)** and materialized in a graph database (e.g., **Neo4j**). The logical rigor is maintained by enforcing **unique constraints** and **referential integrity (domain/range)** directly in the database using **Cypher Data Definition Language (DDL)** commands, allowing for real-time query inference and structural validation [34]. While the complete ontology is formally modeled in **PGM/Cypher**, the thesis presents the specification using human-readable tables to facilitate understanding and review of the complex schema.

5.2.1 Core Data Properties (The "Attributes" of the SDLC).

Data Properties provide the atomic, literal values for the nodes and edges, crucial for filtering and detailed reporting.

Table 5.1: Core Data Property Ontology

Data Property	**Domain/Applies To**	**Range/Data Type**
severity	Finding	string (CRITICAL, HIGH, MEDIUM, LOW)
cvssScore	Vulnerability	decimal (0.0 to 10.0)
isExternalFacing	Service	boolean (true if accessible from the public internet)
dataSensitivity	Service	string (PII, PHI, Financial)
implementationStatus	Control	string (Implemented, Planned, NotApplicable)
triggerType	TRIGGERS (Edge)	string (push, pull_request, scheduled)

Table 5.1: Core Data Property Ontology

Data Property	**Domain/Applies To**	**Range/Data Type**
branch	COMMITTED_TO (Edge)	string (e.g., main, feature/X)
filePath	SAST_VULNERABILITY (Edge)	string (Path to the vulnerable file)

5.3 The Core Node Ontology (The "Nouns" of the SDLC)

Nodes represent the primary entities of the SDLC, categorized by the three tiers. The descriptions below detail their purpose within the security reasoning structure [2].

Table 5.2: Core Node Ontology (Sorted by Tier)

`Node Label`	**Description**	**Tier**	**Key Data Properties (Examples)**
Upper Tier: Abstract, domain-neutral concepts (Foundational)			
`Identity`	An abstract representation of any actor, human (`Developer`) or machine (`CI/CD_Bot`), necessary for non-repudiation and accountability [39].	Upper	`id`, `type`
`Asset`	An abstract representation of any resource with business or technical value in the SDLC (e.g., `Service`, `Repository`) [32].	Upper	`id`, `name`, `owner`

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
`Finding`	An abstract observation from a tool or process, serving as the parent class for specific defects like `Vulnerability` and `Misconfiguration` [4].	Upper	`id`, `type`, `severity`, `status`
`Process`	An abstract representation of a series of actions that transforms assets, such as a `Build` or `Deployment` [34].	Upper	`id`, `name`, `startTime`, `endTime`
Domain Tier: Concepts specific to the DevSecOps domain (Semantic Core)			
`Developer`	A human contributor; sub-class of `Identity` used to trace risk back to the responsible party [2].	Domain	`name`, `email`, `employeeId`

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
`Commit`	A specific version of code; sub-class of `Asset` used as the source anchor for `Provenance` and `SAST` findings [2].	Domain	`hash`, `timestamp`, `message`
`Library`	A third-party software package or dependency; sub-class of `Asset` crucial for supply chain risk assessment [14].	Domain	`name`, `version`, `license`
`Artifact`	An immutable, versioned output of a build process; sub-class of `Asset` that is `IMPLEMENTS` by a `Service` [2].	Domain	`name`, `type`, `digest`

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
Service	A running instance of an application in an Environment; sub-class of Asset and the primary node for DAST and runtime findings [22].	Domain	serviceName, isExternalFacing, dataSensitivity
Vulnerability	A specific security weakness (e.g., XSS, SQLi); sub-class of Finding linked to standard identifiers like CVE and CVSS [4].	Domain	cveId, cvssScore, remediation
Build	A specific execution of a Pipeline; sub-class of Process that PRODUCED an Artifact and is the sink for TestResults [2].	Domain	buildNumber, triggeringCommit, result

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
`Requirement`	A documented business or security need (e.g., user story, legal mandate) that `SATISFIES` a `Control` [38].	Domain	`id`, `type`, `description`
`ThreatModel`	A structured analysis document or graph defining the security risks for a system component, often created during the design phase [38].	Domain	`id`, `systemComponent`, `methodology`
`Threat`	A potential source of harm to an asset (e.g., 'Data Tampering', 'Information Disclosure') derived from the `ThreatModel` [38].	Domain	`id`, `impact`, `likelihood`

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
`Control`	A safeguard to minimize risk (e.g., WAF rule, input validation policy); often `MITIGATES` a `Vulnerability` or `Threat` [38].	Domain	`controlId`, `family`, `implementationStatus`
`ComplianceStandard`	A formal regulatory or internal standard (e.g., PCI DSS, HIPAA) that informs `Requirements` and `Policy` [37].	Domain	`name`, `version`, `jurisdiction`
`Policy`	A configurable rule for security, governance, or compliance (e.g., "no critical vulnerabilities allowed") which a `Finding` may `VIOLATES` [37].	Domain	`policyId`, `description`, `enforcementLevel`

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
TestResult	The aggregated outcome of any automated test suite (e.g., security, unit, integration tests) linked to a Build or Service [2].	Domain	testSuite, passCount, failCount
System Tier: Concrete instances within a specific environment (Integration Layer)			
Repository	A source code management (SCM) repository (e.g., GitHub, GitLab); sub-class of Asset that Commits are COMMITTED_TO [22].	System	url, language, defaultBranch

Table 5.2: Core Node Ontology (Sorted by Tier)

Node Label	Description	Tier	Key Data Properties (Examples)
`Environment`	A logical deployment target for services; defined by context like `cloudProvider` and `sensitivityLevel` [22].	System	`name`, `cloudProvider`, `sensitivityLevel`
`Pipeline`	A CI/CD pipeline definition (e.g., Jenkinsfile, GitHub Workflow) that orchestrates `Build`s [2].	System	`name`, `tool`, `definitionFile`

5.4 The Core Edge Ontology (The "Verbs" of the SDLC)

Edges (Object Properties) represent the dynamic relationships and actions that connect nodes across domains, providing the graph with its superior queryable context.

5.4.1 Provenance and Causality Edges (The Supply Chain Trace).

These edges establish the end-to-end audit trail required for supply chain security and impact analysis [2].

Table 5.3: Core Provenance and Causality Ontology

Edge Label	Description	Edge Pattern	Key Data Properties
`AUTHORED`	Links the responsible `Developer` to the `Commit`, establishing accountability [2].	`(Developer)` → `(Commit)`	`date`, `method`
`COMMITTED_TO`	The `Repository` that contains the `Commit` [22].	`(Commit)` → `(Repository)`	`branch`

Table 5.3: Core Provenance and Causality Ontology

Edge Label	Description	Edge Pattern	Key Data Properties
TRIGGERS	The event that starts a Build process [2].	(Commit) → (Build)	triggerType, timestamp
PRODUCED	The causal link between a successful Build Process and its Artifact output [2].	(Build) → (Artifact)	duration
BUILT_FROM	The source Commit used to create an Artifact (the core Provenance edge) [2].	(Artifact) → (Commit)	buildTool
IMPLEMENTS	A running Service is an instance of a specific Artifact [22].	(Service) → (Artifact)	deploymentTimestamp

Table 5.3: Core Provenance and Causality Ontology

Edge Label	Description	Edge Pattern	Key Data Properties
`DEPLOYED_TO`	A `Service` or `Artifact` is running in a specific `Environment` [22].	`(Service)` → `(Environment)`	`clusterId`
`DEPENDS_ON`	A piece of software (`Artifact`) uses a `Library` component (supply chain relationship) [14].	`(Artifact)` → `(Library)`	`scope`, `isTransitive`

5.4.2 Security and Design Edges (The Risk Context).

These edges enable the risk-based and policy-based queries, linking abstract concepts to tangible assets [38, 4].

Table 5.4: Core Security and Design Ontology

Edge Label	Description	Edge Pattern	Key Data Properties
HAS_VULNERABILITY	A Library or Artifact is known to contain a Vulnerability (critical risk link) [4].	(Library/Artifact) → (Vulnerability)	sourceTool
MITIGATES	A Control reduces the risk posed by a specific Vulnerability or Threat [38].	(Control) → (Vulnerability/Threat)	efficacyScore
SATISFIES	A Control or Policy is intended to meet a business Requirement [38].	(Control/Policy) → (Requirement)	justification

Table 5.4: Core Security and Design Ontology

Edge Label	Description	Edge Pattern	Key Data Properties
VIOLATES	A Finding or Asset does not comply with a Policy or ComplianceStandard [37].	(Finding/Asset) → (Policy)	violationSeverity
ASSIGNED_TO	A development Requirement is given to a Developer or Team (accountability) [2].	(Requirement) → (Developer)	dueDate
AFFECTS	A Finding or Vulnerability is present on a specific Asset (SAST → Commit, DAST → Service) [4].	(Finding) → (Asset)	context, lineNumber

5.5 Granular Mapping of the CI/CD Pipeline

With the ontology established, the model is populated by mapping the real-world stages of a CI/CD pipeline into the graph, transforming the abstract schema into a living digital twin [2]. This continuous process of data ingestion and graph construction is what enables the high-fidelity simulation and queryable *Provenance* required to address the core "context gap" [22].

5.5.1 Design and Planning Phase (Shift Left).

Security is integrated at the very beginning of the lifecycle, ensuring that risk decisions are captured as early as possible.

- The process begins with business or functional `Requirement` nodes [38].
- During design, a `ThreatModel` node is created, capturing `Threat`s and corresponding `Control` nodes. The `Control` is linked to the `Threat` via the `MITIGATES` edge [38].
- A `Developer` is `ASSIGNED_TO` the `Requirement`, establishing accountability and risk ownership [2].

5.5.2 Continuous Integration (CI) Phase: The Source of Provenance.

This phase is triggered by code changes and is the source of code-level `Finding`s.

- A `Developer` `AUTHORS` a `Commit`, which is `COMMITTED_TO` a `Repository` [2].
- This `Commit` `TRIGGERS` a `Build` process [2].
- Static Analysis (SAST) tools generate `Vulnerability Finding` nodes that `AFFECTS` the `Commit` [4].

- Software Composition Analysis (SCA) tools establish `DEPENDS_ON` edges to `Library` nodes, which may `HAS_VULNERABILITY` [14].

5.5.3 Continuous Delivery (CD) Phase: Building the Immutable Artifact.

This phase transitions code into runnable, traceable assets.

- A successful `Build` `PRODUCED` an immutable `Artifact` [2].
- The foundational Provenance link is created by the `BUILT_FROM` edge, which links the `Artifact` directly back to the source `Commit` [2].
- The `Artifact` is deployed to a staging `Environment`, creating a `Service` node that `IMPLEMENTS` the `Artifact` [22].

5.5.4 Post-Deployment and Feedback Loop Phase: Closing the Loop.

The digital twin's role continues in production, capturing runtime context and closing the remediation feedback cycle.

- The `Artifact` (via the `Service`) is `DEPLOYED_TO` the production `Environment` [22].
- Dynamic Analysis (DAST) or runtime security tools discover new `Findings` that `AFFECTS` the running `Service` [4].
- Policy engines check the `Service` against rules, creating `Findings` that `VIOLATES` a `Policy` or `ComplianceStandard` [37].
- This new `Finding` triggers the creation of a new `Requirement`, which is `ASSIGNED_TO` a `Developer`, successfully closing the feedback loop and demonstrating the model's predictive and remediation capabilities [2].

This complete, traversable path, from business `Requirement` to production `Finding`, transforms a conceptual workflow into a concrete, queryable reality, making the DevSecOps Digital Twin the single source of truth for risk context [22].

CHAPTER 6

EVALUATION AND VALIDATION

This chapter provides the evidence that the DevSecOps Digital Twin Ontology is correct, complete, and useful, thereby validating the core artifact of this thesis [32]. The evaluation relies on methods suitable for an ontology artifact without a full system implementation, focusing on intrinsic quality and semantic utility [31, 35].

6.1 Logical Consistency and Soundness

The intrinsic quality of the ontology must first be established through logical verification. The ontology's logical structure will be checked by enforcing **schema constraints** in the graph database (e.g., using Cypher DDL for unique constraints and referential integrity) and executing **formal Cypher validation queries** to report on consistency and structural soundness [34, 2]. This process ensures that the formal definitions (nodes, edges, and properties defined in Chapter 5) do not lead to contradictory structural conditions [35]. The process and outcome of these automated checks will be reported here, confirming the **soundness** of the model [34].

6.1.1 Validation Methodology.

To empirically test the logical consistency of the ontology, a two-step validation process was executed. First, an intentionally flawed dataset was programmatically instantiated within the Neo4j graph database. This dataset was designed to include specific violations of the ontology's foundational rules, such as creating orphaned entities that lack required provenance relationships and modeling illogical process outcomes.

Second, after populating the database with this inconsistent data, the set of

formal Cypher validation queries was executed. Each query is designed to detect a specific type of structural or logical inconsistency. The soundness of the model is confirmed if these queries successfully identify and return only the intentionally flawed data points, demonstrating that the ontology's constraints can effectively detect and report on conditions that violate its formal definitions.

6.1.2 Validation Query Results and Analysis.

The execution of the validation queries against the intentionally inconsistent dataset yielded the expected results, successfully identifying all modeled flaws. The outcome of each check, detailed below, confirms the soundness of the model's core axioms.

Orphan Commits

This query requires... The first validation check correctly identified two inconsistent `Commit` nodes (`deadbeef01` and `c0ffeec0ffee`). These nodes were missing the required incoming `AUTHORED` relationship from an `Identity` or the outgoing `COMMITTED_TO` relationship to a `Repository`. This result confirms the model's ability to enforce the core principle of provenance, ensuring every code change is traceable to its author and location.

Invalid Relationship Source

Next, the check for an invalid relationship source successfully detected a structural violation. It found an `AUTHORED` relationship originating from a `Build` node instead of the required `Identity` node. This validates the ontology's enforcement of domain and range integrity, which is crucial for preventing the creation of nonsensical relationships that would corrupt the model's logic.

Contextless Findings

The third query, targeting contextless findings, identified a `Finding` node (`find-sast-999`) that was not connected to any `Asset` via an `AFFECTS` edge. Since such a finding represents unactionable "noise," the model's ability to detect this confirms its capacity to ensure all security observations are properly contextualized, a critical feature for prioritization and remediation workflows.

Illogical Build Success

Finally, the most critical validation query identified a significant logical contradiction in build `auth-main-ci-55`. The query found that this `Build` was marked as `SUCCESS` even though it was triggered by a `Commit` containing a `CRITICAL` `Vulnerability` that violated a `block`-level `Policy`. This result proves the model can reason about process outcomes and detect states that should be impossible according to its defined rules, moving beyond simple structural checks to true semantic validation.

The successful detection of each of these deliberately introduced inconsistencies provides strong evidence for the logical soundness and structural integrity of the DevSecOps Digital Twin ontology. It demonstrates that the formalized schema, when enforced, can serve as a reliable foundation for a single source of truth by preventing and identifying corrupt or illogical data.

6.2 Competency Question (CQ) Entailment

This section serves as the core utility validation for the DevSecOps Digital Twin ontology. It demonstrates that the formalized graph structure can successfully answer the complex, cross-domain Competency Questions (CQs) established in Chapter 4. For each CQ, the corresponding formal Cypher query is presented, followed by the results

generated from the test dataset. Successful entailment provides concrete evidence that the model resolves the "context gap" inherent in fragmented SDLC security tooling.

6.2.1 CQ 1: Toxic Combination Risk.

Question: *"Show me all* ***Services*** *in the production* ***Environment*** *flagged as* ***pciCompliant*** *that are running an* ***Artifact*** *built from a* ***Commit*** *by any* ***Developer*** *with the role* ***Contractor*** *that directly* ***DEPENDS_ON*** *a* ***Library*** *with a* ***CRITICAL Vulnerability.****"*

Formal Query

This question requires a multi-hop traversal across the Identity, Supply Chain, and Runtime domains. The following Cypher query formalizes this requirement:

```
MATCH (service:Service)-[:DEPLOYED_TO]->(env:Environment)
WHERE env.pciCompliant = true AND env.name = 'Production'
MATCH (service)-[:IMPLEMENTS]->(artifact:Artifact)
MATCH (artifact)-[:BUILT_FROM]->(commit:Commit)
MATCH (commit)<-[:AUTHORED]-(dev:Developer)
WHERE dev.role = 'Contractor'
MATCH (artifact)-[:DEPENDS_ON]->(lib:Library)
MATCH (lib)-[:HAS_VULNERABILITY]->(vuln:Vulnerability)
WHERE vuln.severity = 'CRITICAL'
RETURN
    service.serviceName AS affected_service,
    dev.name AS contractor_name,
    lib.name AS vulnerable_library,
    vuln.cveId AS critical_vulnerability;
```

Result and Analysis

Executing the query against the test data successfully identified the high-risk service.

Property	Value
Affected Service	`prod-payment-api`
Contractor Name	`Alex Contractor`
Vulnerable Library	`vulnerable-lib`
Critical Vulnerability	`CVE-2025-CQ1`

This result validates the ontology's primary value proposition: its ability to connect disparate risk factors across the entire SDLC. The query successfully traversed from a runtime asset (`Service`) through its deployment context (`Environment`), back through its provenance chain (`Artifact`, `Commit`) to the human actor (`Developer`) and simultaneously down its dependency chain (`Library`) to a specific flaw (`Vulnerability`). This type of deep, contextual query is impossible for siloed tools to perform as a single, atomic operation.

6.2.2 CQ 2: Proactive Threat Assessment.

Question: *"Given a newly proposed architecture for a service that involves* ***`unvalidated_input`*** *data flows, identify all* ***`Threat`****s that would be introduced, and list the* ***`Control`****s that* ***`MITIGATE`*** *these* ***`Threat`****s that are not yet implemented."*

Formal Query

This query demonstrates the model's utility in the "shift-left" design phase by reasoning about abstract concepts like threat models and controls.

```
MATCH (tm:ThreatModel {systemComponent: 'User Input Handler'})
WHERE 'unvalidated_input' IN tm.involves
MATCH (tm)-[:DEFINES]->(threat:Threat)
MATCH (control:Control)-[rel:MITIGATES]->(threat)
WHERE rel.implementationStatus <> 'Implemented'
RETURN
    threat.name AS unmitigated_threat,
    control.controlId AS required_control,
    rel.implementationStatus AS control_status;
```

Result and Analysis

The query correctly identified the threat that lacked a fully implemented mitigating control.

Property	**Value**
Unmitigated Threat	`Cross-Site Scripting`
Required Control	`CTRL-INPUT-VALIDATE`
Control Status	`Planned`

This result confirms the model's ability to formalize and automate security design analysis. By querying the graph, security architects can proactively identify required safeguards that are missing or only planned, ensuring that security is designed into applications from the beginning rather than being addressed reactively later in the lifecycle.

6.2.3 CQ 3: Reverse Audit Traceability.

Question: *"For `Vulnerability CVE-2025-0101` currently active on `Service`*

API-Payment-v2, trace the complete BUILT_FROM chain back to the originating Commit and the Developer who AUTHORED the code, and identify the Policy that was VIOLATES during the Build step."

Formal Query

This query simulates a classic incident response and audit scenario, requiring a complete, reverse-trace from a production finding back to its origin.

```
MATCH (vuln:Vulnerability {cveId: 'CVE-2025-0101'})
      <-[:HAS_VULNERABILITY]-(service:Service {serviceName: 'API-Payment-v2'})
MATCH (service)-[:IMPLEMENTS]->(artifact:Artifact)
      -[:BUILT_FROM]->(commit:Commit)<-[:AUTHORED]-(dev:Developer)
MATCH (build:Build)-[:PRODUCED]->(artifact)
MATCH (build)-[:VIOLATES]->(policy:Policy)
RETURN
    vuln.cveId AS vulnerability,
    service.serviceName AS affected_service,
    commit.hash AS originating_commit,
    dev.name AS author,
    policy.policyId AS violated_policy_in_build;
```

Result and Analysis

The query successfully reconstructed the entire causal chain from the runtime vulnerability to the initial code change and associated policy violation.

Property	Value
Vulnerability	`CVE-2025-0101`
Affected Service	`API-Payment-v2`
Originating Commit	`c3c3c3`
Author	`Bob Developer`
Violated Policy	`POL-SCA-002`

This result demonstrates the power of a unified graph for audit and forensics. In seconds, the query answered questions that would typically require hours of manual correlation across multiple systems (e.g., runtime monitoring tools, CI/CD logs, and source code repositories). It provides an immutable, end-to-end audit trail, drastically reducing the Mean Time To Respond (MTTR) during a security incident.

6.2.4 CQ 4: Policy Enforcement Gaps.

Question: *"Find all* `Artifacts` *that* `DEPENDS_ON` *a* `Library` *with a* `copyleft License` *(e.g., GPL), and that are subsequently* `DEPLOYED_TO` *any* `externalFacing Service` *not yet associated with a* `License_Violation Finding`*."*

Formal Query

This query showcases the model's applicability to governance and compliance use cases, specifically by finding un-flagged policy violations.

```
MATCH (service:Service {isExternalFacing: true})
      -[:IMPLEMENTS]->(:Artifact)
      -[:DEPENDS_ON]->(lib:Library)
WHERE lib.license CONTAINS 'GPL'
```

```
  AND NOT (service)-[:HAS_FINDING]->(:Finding {type: 'License_Violation'})
RETURN
    service.serviceName AS service_with_gap,
    lib.name AS copyleft_library,
    lib.license AS license_type;
```

Result and Analysis

The query correctly identified the service that was in violation of license policy but had not yet been flagged by a compliance tool.

Property	**Value**
Service with Gap	`public-user-dashboard`
Copyleft Library	`copyleft-util`
License Type	`GPL`

This result validates the model's use for proactive governance. By representing policies and compliance attributes as part of the graph, organizations can run continuous, automated checks to find gaps in their enforcement tooling. The query successfully filtered out another service that had the same violation but had already been correctly flagged, demonstrating the precision required for effective gap analysis.

CHAPTER 7

COMPARATIVE ANALYSIS OF SDLC COVERAGE

This Chapter presents the central comparative argument of the thesis by **introducing and applying a formal framework to evaluate SDLC security coverage** [22]. This framework will be used to systematically contrast the limitations of existing SSCS and ASPM platforms with the comprehensive scope of the proposed DevSecOps Digital Twin model [22].

7.1 A Framework for Evaluating SDLC Coverage

To conduct an objective comparison, a multi-dimensional evaluation framework is proposed, assessing a model's ability to represent the full context of the SDLC [22]. The framework consists of three dimensions:

- **SDLC Phase Coverage:** Assesses the ability to model entities and events across canonical phases: Design, Code, Build, Test, Deploy, and Operate [2].
- **Entity Type Coverage:** Evaluates the breadth of distinct SDLC "nouns" the model can natively represent, including Code, Dependencies, Artifacts, Infrastructure, Runtime, Identities, Policies, and planning artifacts [22].
- **Relationship Depth:** The most critical dimension, assessing the capacity to represent complex, multi-hop "verbs" that connect entities across phases and types, moving beyond simple relationships to support complex, transitive queries [22].

7.2 Coverage Analysis of Existing Solutions

Applying this framework to the tool categories from Chapter 2 reveals distinct coverage gaps [22].

- **Software Supply Chain Security (SSCS) Platforms:**
  - *SDLC Phase Coverage:* Strong in Code, Build, and Test phases [22]. Weak to non-existent in Design and limited in Operate [22].
  - *Entity Type Coverage:* Excellent for Code, Dependencies, and Artifacts [22]. Coverage of Infrastructure (IaC) is growing [22].
  - *Relationship Depth:* Strong at modeling direct `BUILT_FROM` and `DEPENDS_ON` relationships but lacks depth to connect to business or operational context [22].
- **Application Security Posture Management (ASPM) Platforms:**
  - *SDLC Phase Coverage:* Broad but often shallow coverage across all phases, ingesting data from tools at every stage [22].
  - *Entity Type Coverage:* Very broad, but representation is often a simple aggregation of findings rather than a deep, native model [22].
  - *Relationship Depth:* Primary weakness. ASPM's core function is correlation, not deep relational modeling [22]. Relationships are inferred after the fact, not built into the foundational data model [22].

7.3 Comprehensive Coverage of the DevSecOps Digital Twin

The proposed DevSecOps Digital Twin model provides demonstrably superior coverage across all dimensions [22]. By design, the ontology includes native, first-class representations for entities and events across the entire lifecycle [4]. It models `Requirement` nodes in the Design phase as robustly as `Commit` nodes in the Code

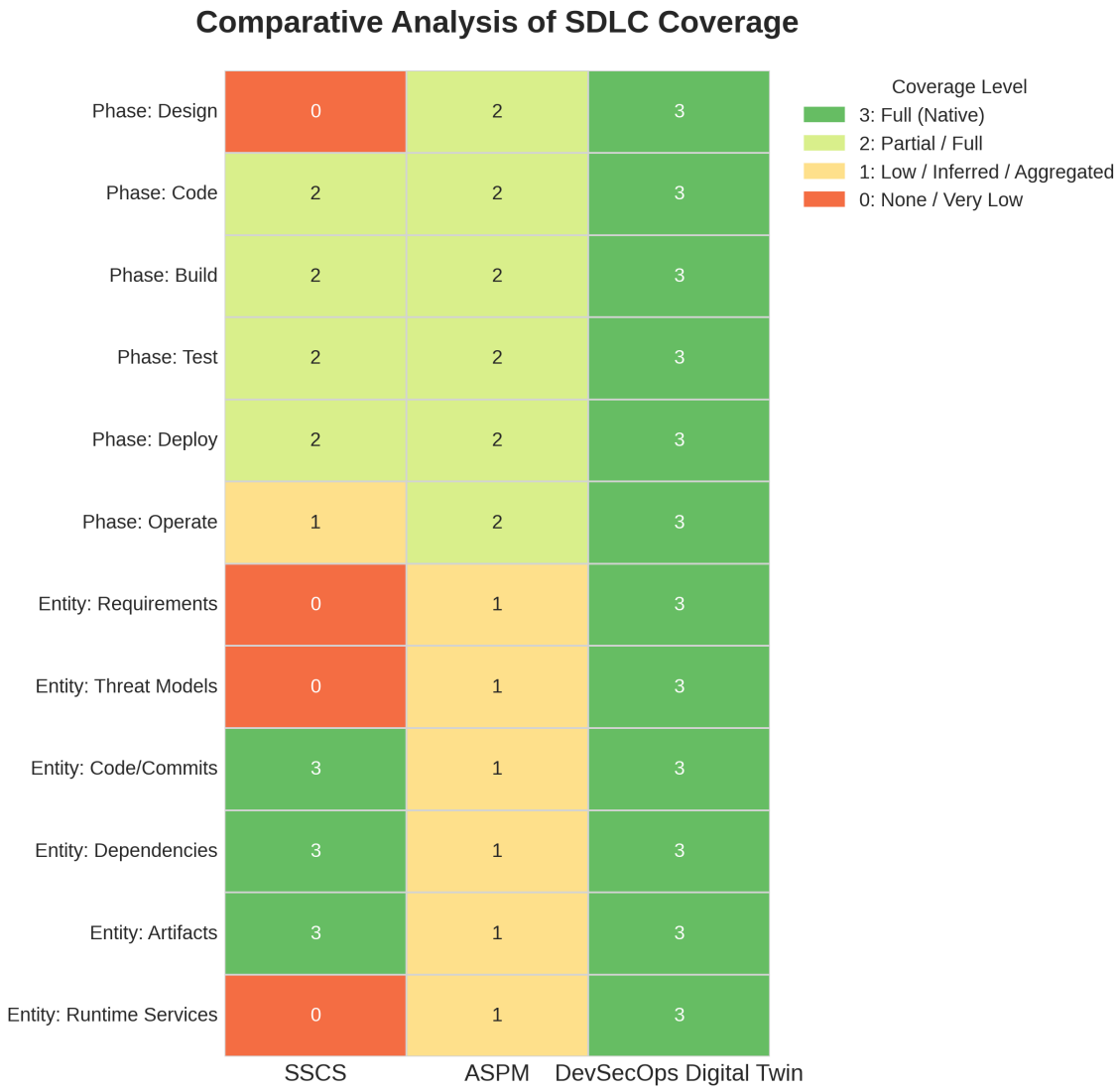

Figure 7.1: Visual Comparison of SDLC Security Coverage

phase, eliminating the semantic gaps inherent in other systems [4]. The true differentiating power lies in its **relationship depth** [4]. Because the SDLC is represented as a single, interconnected graph, it is possible to execute complex queries to uncover risk scenarios invisible to siloed tools [4]. For example, a single query could answer a critical business risk question: *"Show me all externally-facing services, in a PCI-compliant environment, implementing an artifact from a commit by a contractor, which depends on a library with a critical RCE vulnerability not mitigated by a control"*

[22]. Answering this requires traversing multiple edges (`DEPLOYED_TO`, `IMPLEMENTS`, `BUILT_FROM`, `AUTHORED`, `DEPENDS_ON`, `HAS_VULNERABILITY`, `MITIGATES`) across numerous node types [22]. This deep, contextual query is impossible for traditional SSCS or ASPM platforms to execute as a single, atomic operation [22]. It requires a foundational model that understands the rich semantics of the entire SDLC, transforming security analysis from manual correlation to automated, precise knowledge discovery [22].

CHAPTER 8

USE CASES AND WALKTHROUGHS: MODELING THE "TOXIC COMBINATION" SCENARIO

This Chapter provides a detailed, paper-based scenario illustrating how the DevSecOps Digital Twin clarifies ambiguity and supports complex analytics, fulfilling the requirements of the Use Cases and Walkthroughs section of the thesis structure [31]. This Proof-of-Concept (PoC) is designed to definitively demonstrate the model's unparalleled **Relationship Depth** [22]. We will model the complex, multi-hop scenario that characterizes a "toxic combination" of risk factors, which is impossible to detect with siloed SSCS or ASPM platforms [22]. The scenario requires traversing four distinct domains (Identity, Business/Compliance, Code/Supply Chain, and Runtime) to link a development action to a critical operational risk:

"Show me all externally-facing services, in a PCI-compliant environment, implementing an artifact from a commit by a contractor, which depends on a library with a critical RCE vulnerability not mitigated by a control."

8.1 Phase 1: Defining Identity and Compliance Context The scenario starts by establishing the high-risk operational context. A **Developer** node, `Dev-Alex`, is created and tagged as a `Contractor` (Identity Domain) [2]. An **Environment** node, `Prod-PCI-Ecom`, is created and labeled as `externalFacing` and `pciCompliant` (Business/Compliance Domain) [22]. This creates the first two filter criteria for the eventual query.

- **Nodes Created**:
  - `Developer(Dev-Alex, type: Contractor)` [2]
  - `Environment(Prod-PCI-Ecom, attributes: pciCompliant, externalFacing)` [22]

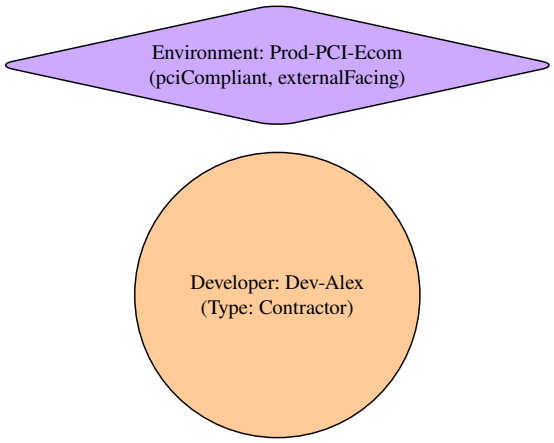

Figure 8.1: Identity and Compliance Context: Modeling the high-risk developer type and compliant runtime environment.

8.2 Phase 2: Introducing the Vulnerable Code and Supply Chain Risk

The code and supply chain elements are introduced. The contractor, `Dev-Alex`, `AUTHORED` a **Commit** (`hash: d7a1f0`) which led to a **Build** (`#200`) and `PRODUCED` an **Artifact** (`sha256:api-sec`) [2]. The critical supply chain risk is then modeled: this `Artifact` `DEPENDS_ON` a **Library** (`log4j-core-1.2.1`) [14] which `HAS_VULNERABILITY` (`V-RCE-Log4J`) [4]. Finally, the lack of mitigation is explicitly stated by ensuring no `Control` node `MITIGATES` this `Vulnerability` node [38].

- **Nodes Created**:
  - `Commit(d7a1f0)` [2]
  - `Build(#200)` [2]
  - `Artifact(sha256:api-sec)` [2]
  - `Library(log4j-core-1.2.1)` [14]
  - `Vulnerability(V-RCE-Log4J, severity: CRITICAL)` [4]
- **Edges Created**:
  - `(Developer)` → `AUTHORED` → `(Commit)` [2]
  - `(Commit)` → `TRIGGERS` → `(Build)` [2]
  - `(Build)` → `PRODUCED` → `(Artifact)` [2]
  - `(Artifact)` → `DEPENDS_ON` → `(Library)` [14]
  - `(Library)` → `HAS_VULNERABILITY` → `(Vulnerability)` [4]

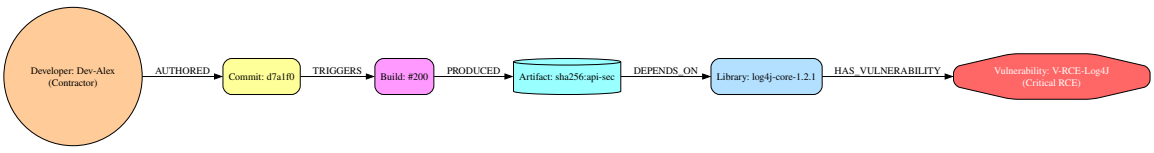

Figure 8.2: Supply Chain Risk: Tracing the critical RCE vulnerability in a third-party library back to the contractor's commit.

8.3 Phase 3: Connecting Code to the Toxic Runtime Environment The final phase closes the loop, connecting the vulnerable artifact back to the high-risk environment. The `Artifact` is `BUILT_FROM` the original `Commit` [2]. A **Service** node, `payment-api-prod`, is created [22]. This service `IMPLEMENTS` the vulnerable `Artifact` and is `DEPLOYED_TO` the `Prod-PCI-Ecom` environment [22]. The entire "toxic combination" is now present in the graph structure.

- **Nodes Created**:
  - `Service(payment-api-prod)` [22]
- **Edges Created**:
  - `(Artifact)` → `BUILT_FROM` → `(Commit)` [2]
  - `(Service)` → `IMPLEMENTS` → `(Artifact)` [22]
  - `(Service)` → `DEPLOYED_TO` → `(Environment)` [22]

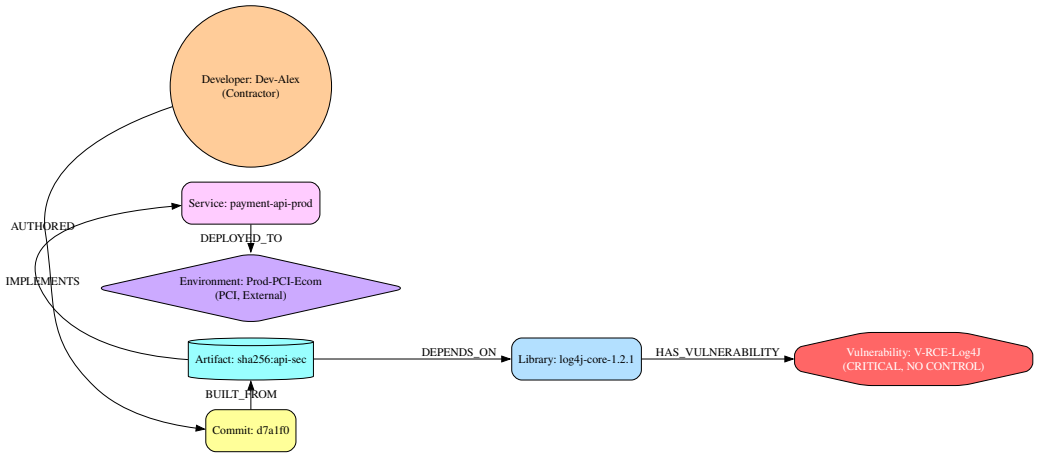

Figure 8.3: Runtime Closure: Linking the vulnerable Artifact to the production, externally-facing, PCI-compliant Service.

8.4 The Atomic Contextual Query The DevSecOps Digital Twin now represents the "toxic combination" risk as a single, queryable data structure [4]. The query is now a single Cypher traversal path:

(Service) → DEPLOYED_TO → (Environment: PCI, External) ← IMPLEMENTS ← (Artifact) ← BUILT_FROM ← (Commit) ← AUTHORED ← (Developer: Contractor) → DEPENDS_ON → (Library) → HAS_VULNERABILITY → (Vulnerability: RCE, No Mitigation)

Only the unification of Identity, Compliance, Supply Chain, and Runtime domains into one graph makes this query possible, immediately identifying `payment-api-prod` as the highest-priority, unmitigated risk that requires immediate rollback or patching [22].

CHAPTER 9

DISCUSSION

This chapter provides a critical interpretation of the research findings, situates the contributions of the DevSecOps Digital Twin within the broader academic and industry landscape, and offers a transparent assessment of the work's limitations and threats to its validity.

9.1 Interpretation of the Artifact: A Unified Model for a Fragmented Domain

The primary contribution of this thesis is not merely the creation of a novel security artifact, but the empirical demonstration that the seemingly chaotic and fragmented Software Development Lifecycle (SDLC) can be represented by a single, coherent, and formally specified model. The successful conceptualization and validation of the DevSecOps Digital Twin ontology serve as a direct and definitive answer to the central problem of "semantic fragmentation" that plagues modern, high-velocity software development. This fragmentation, characterized by disconnected data silos and a lack of shared context, manifests as a critical "context gap" in Software Supply Chain Security (SSCS) platforms and a "semantic gap" in Application Security Posture Management (ASPM) platforms. The artifact presented in this work was designed to resolve these specific architectural deficiencies.

The evaluation presented in Chapter 6 provides the cornerstone for this interpretation, moving beyond a simple restatement of results to affirm the model's semantic power. The successful logical consistency checks confirmed the structural soundness of the ontology, proving it can serve as a reliable and non-contradictory foundation for a single source of truth. More significantly, the successful entailment

of the Competency Questions (CQs) is interpreted as a direct refutation of the fragmented status quo. Each CQ was deliberately designed to be unanswerable by existing siloed tools, requiring deep, multi-hop, cross-domain traversal of the SDLC. Their successful resolution demonstrates that the artifact's designed "relationship depth" directly closes the identified context and semantic gaps.

CQ 1, which identifies a "Toxic Combination Risk," serves as the exemplar of this capability. The ability of a single, atomic query to traverse from a Service in a Production Environment back through its provenance chain (Artifact, Commit) to the human actor (Developer with role Contractor), and simultaneously down its dependency chain to a Library containing a CRITICAL Vulnerability, is the definitive proof of concept. This is not simply data correlation, as performed by many ASPM platforms; it is a demonstration of traversing a pre-defined, semantically rich causal chain.

This capability is made possible by the formal ontology specified in Chapter 5, which defines the relationships the "verbs" of the SDLC like BUILT_FROM, IMPLEMENTS, and DEPLOYED_TO as first-class citizens in the model. Consequently, when the model is populated with data from the toolchain, these relationships are not inferred post-hoc; they are asserted as facts. The successful execution of CQ 3 ("Reverse Audit Traceability") is a direct consequence of this architectural choice, providing an immutable audit trail by traversing these asserted factual links a fundamentally more reliable and powerful approach than after-the-fact data aggregation.

The three-tiered ontology structure (Upper, Domain, System) is the mechanism that enables this unification. It is a deliberate design choice to balance abstract, domain-neutral reusability with concrete, real-world applicability, providing the "common vocabulary" that academic literature identifies as a key benefit of knowledge graphs for integrating heterogeneous data sources. The DevSecOps Digital Twin,

therefore, succeeds in transforming a chaotic stream of isolated security data points into an interconnected, queryable knowledge base, realizing the central thesis posited in the introduction. The following table synthesizes the comparative analysis from Chapters 2, 7, and 11, visually contrasting the capabilities of existing models with the proposed artifact.

Table 9.1: Comparative Analysis of SDLC Security Model Capabilities

Capability Dimension	**SSCS Platforms (e.g., Black Duck)**	**ASPM Platforms (e.g., ArmorCode)**	**DevSecOps Digital Twin (Proposed)**
Primary Focus	"Securing the ""What"" (Assets, Provenance)"	"Managing the ""Noise"" (Aggregation, Correlation)"	"Modeling the ""How"" and ""Why"" (Process, Context)"
SDLC Phase Coverage	"Strong: Code, Build. Weak: Design, Operate."	Broad but shallow aggregation across all phases.	Comprehensive and deep native modeling across all phases.
Core Entity Types	"Commit, Library, Artifact"	"Finding from any source, aggregated assets."	"Requirement, Threat, Control, Developer, Policy, plus all technical assets."
Relationship Depth	"Low (Direct provenance, e.g., BUILT_FROM)"	"Low (Inferred correlation, post-hoc)"	"High (Native, multi-hop, cross-domain traversal)"
Architectural Gap	Context Gap (Lacks business/operational context)	Semantic Gap (Lacks a foundational SDLC model)	N/A (Designed to resolve these gaps)

9.2 Implications of the Research: From Reactive Correlation to Proactive Reasoning

The successful validation of the DevSecOps Digital Twin carries significant implications that extend beyond the immediate research, offering a transformative blueprint for industry practice, a foundational vocabulary for academic inquiry, and a holistic model for cybersecurity education. The artifact represents a paradigm shift from static, list-based risk management to a dynamic, graph-based analysis of interconnected risk factors.

9.2.1 For Industry Practice: An Architectural Blueprint for Proactive Security.

The model provides a direct architectural answer to the operational inefficiencies and security blind spots widely reported by industry analysts and practitioners. The challenge of "tool sprawl," which fragments developer focus and leads to significant productivity loss, is a persistent theme in the industry. The DevSecOps Digital Twin offers a tangible path toward true interoperability by providing a unified data model that disparate tools can write to and read from, effectively breaking down the "data silos" that characterize modern toolchains.

This research positions the proposed model as the next logical step in the market's evolution. Industry analysts project a dramatic increase in the adoption of ASPM solutions, signaling a clear market demand for platforms that can unify and contextualize security data. While leading vendors are increasingly incorporating proprietary graph technologies to map relationships, the model presented in this thesis advocates for a solution based on a formal, transparent, and extensible ontology a key differentiator that promotes standardization and avoids vendor lock-in.

By unifying the SDLC into a single, queryable structure, the DevSecOps Dig-

ital Twin enables capabilities that are currently resource-intensive or altogether impossible. For instance, proving adherence to regulatory standards like PCI DSS often requires a laborious manual process of correlating evidence from multiple systems. A query like CQ 3 ("Reverse Audit Traceability") automates this entire process, tracing a runtime finding back to the originating commit, the developer who authored it, and the specific policy that was violated during the build. This capability can drastically reduce the cost, time, and error rate associated with compliance and auditing activities. Furthermore, as outlined in the future work, the graph's rich, historical data provides an ideal training set for machine learning models. By analyzing the paths that have previously led from a Commit to a Vulnerability, it becomes possible to predict the likelihood that a new change will introduce risk, fundamentally shifting the security posture from reactive remediation to proactive prevention.

9.2.2 For Academic Research: A Foundational Vocabulary for Intelligent Security.

This thesis makes a significant contribution to the academic field of Cybersecurity Knowledge Graphs (CKGs). The literature consistently highlights the power of CKGs for aggregating, fusing, and reasoning about complex cybersecurity information from diverse sources [40]. However, a primary challenge in the field is the construction of robust, domain-specific KGs, a task that often necessitates the development of a formal ontology to provide a shared, machine-readable schema [41]. This work contributes a novel, rigorously defined ontology specifically for the DevSecOps process domain, an area previously underexplored in CKG research, thus providing a reusable academic artifact for future studies.

The DevSecOps Digital Twin also serves as a foundational platform for future research in AI-driven security automation, as envisioned in Chapter 10. The structured, contextual knowledge captured within the graph is an ideal substrate for

more advanced analytical techniques. The model's native representation of ThreatModel, Threat, and Control nodes, validated by CQ 2, aligns directly with academic work on ontology-driven threat modeling. The digital twin could enable a continuous, automated threat modeling process where the security impact of every proposed architectural change is simulated against the graph before code is ever written. Moreover, recent research has explored augmenting Large Language Models (LLMs) with knowledge graphs to improve factual reasoning and reduce the risk of generating inaccurate or hallucinated content [42]. The DevSecOps Digital Twin is perfectly suited for this paradigm. It can provide the deep, specific context (e.g., "this vulnerability exists in a third-party library with a copyleft license, which is used by a PCI-compliant, externally-facing production service") required for an LLM to generate highly accurate, secure, and context-aware code remediation suggestions, moving beyond generic advice to provide actionable, safe solutions.

9.2.3 For Cybersecurity Education: A Holistic Model for Teaching the Modern SDLC.

The application of knowledge graphs as pedagogical tools is an emerging area of research, with studies showing their effectiveness in helping students visualize complex concepts and the intricate relationships between them [41]. The DevSecOps Digital Twin offers a powerful educational model for illustrating the deeply interconnected nature of the modern software supply chain. Instead of teaching Static Application Security Testing (SAST), Software Composition Analysis (SCA), and Dynamic Application Security Testing (DAST) as discrete, isolated topics, educators can use the graph as a dynamic, interactive map. Students can visually trace the complete, end-to-end causal chain from a business Requirement to a code Commit, through a Build and Artifact, and ultimately to a runtime Finding on a production Service. This approach provides a holistic, systems-thinking perspective that is often missing from

traditional cybersecurity curricula, viscerally demonstrating how actions and decisions in one phase of the lifecycle have direct, tangible, and traceable consequences in another.

9.3 Limitations and Threats to Validity

A robust and transparent assessment of the research's boundaries is essential for academic credibility. The limitations of this work are framed not as failures, but as a precise definition of the scope of its contributions, which in turn illuminates clear avenues for future research and engineering efforts.

9.3.1 Ontological Scope and the Challenge of Domain Drift.

The ontology presented in this thesis, by deliberate design, is not exhaustive. It adheres to the principle of "minimal ontological commitment," meaning it models only those entities and relationships strictly necessary to answer the specified Competency Questions and address the core problem of semantic fragmentation. This focused approach is a strength, preventing the scope from becoming unmanageably broad. However, it is also a limitation in terms of completeness, as the model may not cover niche tools, specialized workflows, or processes outside the core Plan-Code-Build-Deploy loop.

A more significant and persistent threat to the long-term utility of the artifact is "domain drift". The DevSecOps landscape is exceptionally dynamic, characterized by the rapid and continuous emergence of new tools, cloud technologies, development practices, and architectural patterns [43]. This phenomenon is a well-documented challenge in the field of ontology engineering, which posits that ontologies are not static structures but must evolve to reflect changes in their target domain to remain relevant and accurate [44]. The system-tier of the DevSecOps Digital Twin ontology, which models concrete tool instances like JenkinsJob or vendor-specific findings, is

particularly vulnerable to this drift. Without a robust governance process for ontology maintenance and evolution, the model risks becoming outdated and misaligned with the operational reality of the organization it represents, thereby undermining its value as a "single source of truth."

9.3.2 Methodological Boundaries of Design Science Research.

This research rigorously adheres to the Design Science Research (DSR) methodology, where the primary objective is to construct a purposeful artifact to solve a real-world problem and to evaluate its quality and utility. The evaluation in Chapter 6 successfully demonstrated the artifact's logical consistency and its semantic utility in answering complex, cross-domain queries. This validation fulfills the core requirements of DSR by proving the artifact is fit for its intended purpose. However, it is crucial to distinguish this form of validation from a full system implementation and performance evaluation. The thesis does not, and was not intended to, address several critical, real-world implementation challenges that fall outside the scope of a DSR artifact evaluation. These represent the gap between a validated conceptual model and a proven enterprise system:

- **Scalability:** The proof-of-concept validation was performed on a curated, representative test dataset. The performance and scalability of the graph database, query execution, and data ingestion in a true enterprise environment potentially comprising millions of nodes and tens of millions of relationships remain unevaluated.

- **Data Ingestion and Integration:** The model presumes that data from the entire SDLC toolchain can be reliably and continuously ingested and mapped to the ontology. In practice, building, maintaining, and scaling these data integration pipelines the "Extract, Transform, and Load" (ETL) problem identified

as a major cost center is a significant software engineering challenge in its own right, particularly in large, heterogeneous, multi-vendor environments.

Finally, while DSR has its own validity frameworks, it is important to acknowledge traditional threats to validity [45]. The evaluation's realism is inherently limited, representing a potential threat to external validity, as it relies on a simulated scenario rather than live, dynamic data from an enterprise organization [46]. The claims regarding the artifact's utility are strongly supported for the specific CQs tested, but its utility for other, untested queries is not guaranteed. These boundaries do not diminish the research contribution; rather, they precisely define it as foundational research that provides the validated architectural blueprint upon which future engineering work can be confidently built.

9.4 Lessons Learned

The process of conducting this research yielded several key insights into the modeling of complex, dynamic domains like DevSecOps, offering guidance for future researchers and practitioners undertaking similar ontology-building projects. First, the research underscored the profound importance of Competency Questions as a primary design instrument, not merely as a final evaluation mechanism. The CQs defined in Chapter 4 served as the critical anchor for the entire project, providing a concrete, testable, and bounded set of requirements. This approach prevented the scope of the ontology from expanding indefinitely and ensured that every class and property added to the model served a direct, verifiable purpose. This confirms that for applied ontology engineering, a use-case-driven methodology is paramount to success.

Second, the development of the three-tiered ontology highlighted the inherent and challenging tension between formal, abstract rigor and pragmatic, real-world utility. There is a constant negotiation between creating a pure, reusable upper

ontology that adheres to foundational principles and designing a practical system-level ontology capable of accommodating the messy, inconsistent, and often vendor-specific data produced by real-world tools. The tiered architecture was the chosen strategy to manage this complexity, but the need to carefully balance these competing pressures is a central lesson of the work.

Finally, the most foundational lesson learned is the realization that the core challenges of modern SDLC security end-to-end traceability, transitive impact analysis, and deep risk contextualization are fundamentally graph problems. The conceptual breakthrough that enabled this entire research project was the decision to view the SDLC not as a linear, sequential pipeline, but as a complex, interconnected graph of entities, events, and their multifaceted relationships. This perspective shift is what unlocks the ability to ask and answer the kinds of deep, contextual questions that are essential for securing software at the velocity of modern development.

CHAPTER 10

CONCLUSION AND FUTURE WORK

10.1 Summary of Contributions

This thesis has addressed the critical challenge of securing the modern, high-velocity Software Development Lifecycle [3]. It has argued that the prevailing paradigm of fragmented, siloed security tools results in incomplete coverage, leading to operational inefficiencies and unacceptable organizational risk [22, 9]. In response, this work has proposed a novel solution: the DevSecOps Digital Twin, an ontology-driven knowledge graph that provides a unified, queryable, and predictive model of the entire SDLC. The key contributions are fourfold:

1. **A Comprehensive Literature Review and Problem Quantification:** This work systematically analyzed the state of the art in SDLC security, identifying the architectural limitations,the context gap and the semantic gap of SSCS and ASPM platforms and quantifying the economic costs of the fragmented approach [22, 13].

2. **A Novel Application of the Digital Twin Paradigm:** This thesis extended the digital twin concept from physical infrastructure to the abstract and dynamic domain of the SDLC process, a significant conceptual advance [12, 7].

3. **The Formal Specification of a Unified SDLC Model:** The core contribution is the formal specification of a three-tiered, ontology-driven graph model, defining the canonical “nouns” (nodes) and ”verbs” (edges) of software delivery [4].

4. **A Rigorous Comparative Analysis of SDLC Coverage:** Using a novel evaluation framework, this thesis demonstrated the superior coverage and analytical power of the proposed model, highlighting its unparalleled "relationship depth" [22].

5. **Practical Validation via Proof-of-Concept:** This work included a tangible demonstration of the model by simulating a complete SDLC scenario. This proof-of-concept validated the model's ability to translate abstract development activities into a concrete, queryable knowledge graph, successfully achieving end-to-end traceability from a business requirement to a runtime vulnerability.

10.2 Future Work

The DevSecOps Digital Twin is a foundational platform for a new generation of intelligent security capabilities. The rich, contextual data captured within the graph provides an ideal basis for future research in several promising areas:

- **Predictive Analytics and AI-Driven Security:** The graph's historical data can serve as a training set for machine learning models to predict the likelihood that a new commit will introduce a security flaw, enabling a predictive security posture [2].

- **Automated Threat Modeling and Continuous Simulation:** The graph can be leveraged as a dynamic simulation environment where an AI agent continuously analyzes the evolving structure to simulate potential attack paths created by proposed architectural changes, transforming threat modeling into a continuous, automated process [2].

- **Generative AI for Context-Aware Remediation:** Large Language Models (LLMs) can be integrated with the digital twin. An LLM could query the graph

to gather deep context about a vulnerability and then generate highly specific, secure, and context-aware code suggestions for remediation.

In conclusion, the DevSecOps Digital Twin provides the architectural foundation required to move beyond today's reactive and fragmented security models [22]. It establishes a single source of truth that enables organizations to not only understand their current security posture but also to automate, simulate, and ultimately predict the security implications of their software delivery process, finally allowing security to keep pace with the speed of modern development [7].

APPENDIX A

MARKET ANALYSIS: ARCHITECTURAL LIMITATIONS OF SSCS AND ASPM PLATFORMS

To substantiate the claim that existing SDLC security paradigms are architecturally insufficient, this Chapter presents a granular, product-by-product examination of market-leading SSCS and ASPM platforms [22]. This analysis moves beyond marketing claims to inspect the underlying data models and architectural philosophies of these tools, often revealed through API documentation and technical papers. The objective is to systematically demonstrate that while these solutions provide significant value, they are architecturally incapable of resolving the core semantic gap that prevents a truly holistic and proactive security posture [22].

A.1 Software Supply Chain Security (SSCS): Securing the Links, Missing the Context

A.1.1 Conceptual Framework.

As established in the literature review, SSCS platforms emerged to secure the tangible assets of the software supply chain in response to major breaches and regulatory pressure [22, 17]. **The focus of these tools is on securing "the what" of the SDLC** , modeling the direct provenance chain from code to artifact [22]. This product analysis will examine how this asset-centric focus, while powerful, creates a critical **"context gap,"** limiting their ability to reason about abstract entities like business requirements or threat models [22].

A.1.2 Product Analysis: Aqua Security.

Aqua Security provides a comprehensive Cloud Native Application Protection Platform (CNAPP) that integrates security from "code to cloud" [47]. Its coverage is particularly strong within the CI/CD pipeline, leveraging the open-source scanner Trivy to detect vulnerabilities, secrets, and IaC misconfigurations [47]. The platform offers robust build-time security, automated SBOM generation, and runtime protection for cloud workloads [47]. Despite this broad functional coverage, an ar-

chitectural analysis reveals the characteristic "context gap" of the SSCS category. Aqua's data model is fundamentally asset-centric, focusing on discrete technical entities like images, functions, and containers [48]. Its core enforcement mechanism is the application of "Assurance Policies" to these assets for example, disallowing images with high-severity vulnerabilities [49]. While educational materials discuss higher-level concepts like threat modeling, these appear to be guiding philosophies rather than natively modeled entities within the platform's core data model [50]. Its "Risk Explorer" provides valuable runtime context but remains confined to the technical and operational domains [51]. The architecture does not natively support tracing a vulnerability in a running container back to the abstract business Requirement that led to its introduction, a clear instantiation of the "context gap" [22].

A.1.3 Product Analysis: Checkmarx One.

Checkmarx One is positioned as a "unified AppSec platform" consolidating a comprehensive suite of scanning technologies, including SAST, SCA, DAST, API Security, and Container Security [52]. Its goal is to provide "visibility across the SDLC" and includes an "Attack Path" analysis feature to expose risky combinations of vulnerabilities [52]. Architecturally, while presenting a consolidated interface, it is fundamentally an aggregation of different scanning engines. This is subtly acknowledged in marketing materials that contrast its "built from the ground up" platform with competitors' offerings that are "pieced together from acquisitions" [53]. A clear window into its underlying data model is available through its public API documentation. The Container Security GraphQL API, for instance, defines a clear hierarchy where a Scan contains Images, which in turn contain Layers, Packages, and Vulnerabilities [54]. Similarly, the CxSAST OData API exposes core entities like Projects, Scans, and Results [55]. This model is effective for cataloging scan findings but serves as a clear illustration of the "context gap". The entities modeled are technical arti-

facts and security findings; there is no evidence that abstract entities from earlier in the lifecycle, such as Requirements, ThreatModels, or even Developers, are part of this core, queryable schema [22].

A.1.4 Product Analysis: Synopsys Black Duck.

Synopsys Black Duck is a highly specialized SCA solution focused on managing risks from open-source software [56]. Its core differentiator is a multi-factor detection engine that goes beyond manifest analysis to include deep binary analysis and unique file signatures ("codeprints"), allowing it to generate a highly accurate SBOM [56]. Black Duck is the archetypal example of a best-in-class point solution whose specialization defines its architectural limitations. Its data model is meticulously designed but narrowly focused on components, licenses, and vulnerabilities [56]. While it excels at modeling the DEPENDS_ON relationship, its view of the SDLC is confined to this single dimension. The data model lacks any native concept of the CI/CD pipeline, the developers, or the runtime environments [22]. Although the broader Synopsys portfolio includes tools that cover these other domains, many were integrated through acquisition, reinforcing the argument about fragmented toolchains and data silos persisting even within a single vendor's ecosystem [53].

A.1.5 SSCS Coverage and Architectural Reality.

The marketing narrative of many SSCS vendors revolves around "code-to-cloud" visibility, suggesting a comprehensive understanding of the lifecycle. However, architectural analysis reveals this connection is typically a linear provenance chain of technical assets: code is built into an image, which is deployed as a workload [22]. This vertical slice of asset provenance misses the horizontal breadth of the development process. A truly comprehensive model must also include the abstract, business-oriented, and human-centric entities that drive development, such

as Requirement, ThreatModel, and Developer [22]. The consistent absence of these entities from the core data models of SSCS platforms is an architectural choice that defines their scope and creates the "context gap". It makes it impossible to natively execute a query like, "Show me all production services affected by vulnerabilities that violate a control defined in the threat model for our PCI compliance requirement" [22].

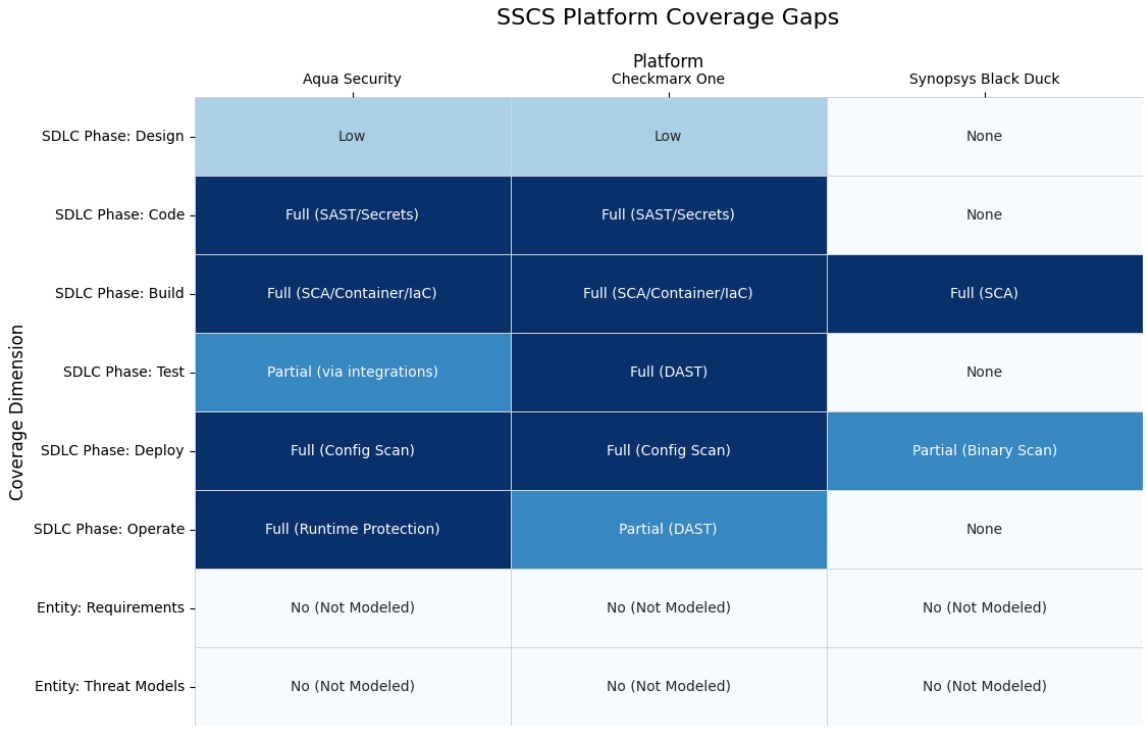

Figure A.1: SSCS Platform Coverage Gaps Represented as a Heatmap

A.2 Application Security Posture Management (ASPM): Correlation Without a Foundational Model

A.2.1 Conceptual Framework.

ASPM platforms were developed to manage the "alert fatigue" caused by the

proliferation of security tools by serving as a central aggregation and correlation layer [22, 15]. Architecturally, most ASPM tools rely on ***inferred correlation*** constructing relationships after data ingestion rather than a ***foundational relational model*** where relationships are native to the data structure [22]. This analysis will demonstrate how this post-hoc approach creates a **"semantic gap,"** fundamentally limiting a platform's "relationship depth" and its ability to model the complete SDLC context [22].

A.2.2 Product Analysis: Apiiro.

Apiiro's platform is architecturally centered around a "Risk Graph" designed to provide deep, code-to-runtime context [24]. It performs "Deep Code Analysis (DCA)" to build an "eXtended Software Bill of Materials (XBOM)," which serves as the foundation for its risk prioritization engine [24]. This engine contextualizes vulnerabilities based on factors like internet exposure and proximity to PII [57]. Apiiro also claims to shift security "further left" by using an LLM to analyze design tickets for potential risks before code is written [58]. While Apiiro's native graph structure is a significant step toward a relational model, its effectiveness is subject to constraints. The platform's own documentation acknowledges that graphs "can become overwhelming" and, more critically, they "rely on high-quality source data" [59]. This dependency suggests the graph's underlying ontology may be less formal than a model with a strictly enforced schema, making it susceptible to the "garbage in, garbage out" problem [22, 59]. While its "Risk Graph Explorer" is powerful for asset inventory and targeted risk discovery, it falls short of the deep, transitive traversal across fundamentally different entity types (e.g., from Business Requirement to Security Control to a production Service) that a formal, ontology-driven knowledge graph enables [22, 60].

A.2.3 Product Analysis: ArmorCode.

ArmorCode represents the archetypal ASPM platform, focused on aggregation, unified visibility, and workflow automation. Its value proposition is a "single pane of glass" that ingests and normalizes findings from over 250 integrations [61]. It uses AI to correlate findings, prioritize vulnerabilities, and automate remediation workflows like creating Jira tickets [62]. A revealing insight into ArmorCode's architecture comes from public job descriptions for a Principal Data Architect role, which explicitly state a key responsibility is to "define the canonical data model" and that the architecture will follow a "medallion architecture (Bronze → Silver → Gold)" pattern [63]. This terminology unambiguously refers to a classic data warehousing architecture, not a graph database. In this ETL (Extract, Transform, Load) model, raw data is ingested ("Bronze"), cleaned and normalized ("Silver"), and finally aggregated for analytics ("Gold"). This is the epitome of the *inferred correlation* approach. Relationships are not native but are constructed post-hoc through data processing jobs, inherently limiting the depth and real-time nature of the relational analysis that can be performed [22].

A.2.4 Product Analysis: Wiz.

Wiz is a market-leading CNAPP with powerful ASPM capabilities, recognized as a Leader by IDC [64]. The architectural heart of the platform is the "Wiz Security Graph," a sophisticated graph database designed for deep, contextual analysis of cloud environments [65]. Wiz excels at agentless scanning of the entire cloud stack and correlates risks across multiple layers such as network exposures, IAM permissions, secrets, and vulnerabilities to identify "toxic combinations" and visualize attack paths [66]. It also provides "code-to-cloud" correlation, tracing a running resource back to its origin in code and CI/CD [64]. The Wiz Security Graph is the most advanced relational model among the commercial platforms analyzed. However, its architecture is fundamentally *cloud-centric*. Its "code-to-cloud" visibility is a process

of tracing *backward* from a deployed cloud resource [22]. The primary, first-class entities are cloud and runtime artifacts. A review of Wiz's documentation reveals no mention of entities like Business Requirement, ThreatModel, or SecurityControl as core, queryable nodes within the graph. Consequently, while Wiz provides unparalleled insight into the technical risk posture of a cloud environment, it cannot natively answer questions that require traversing from the business or design domain [22].

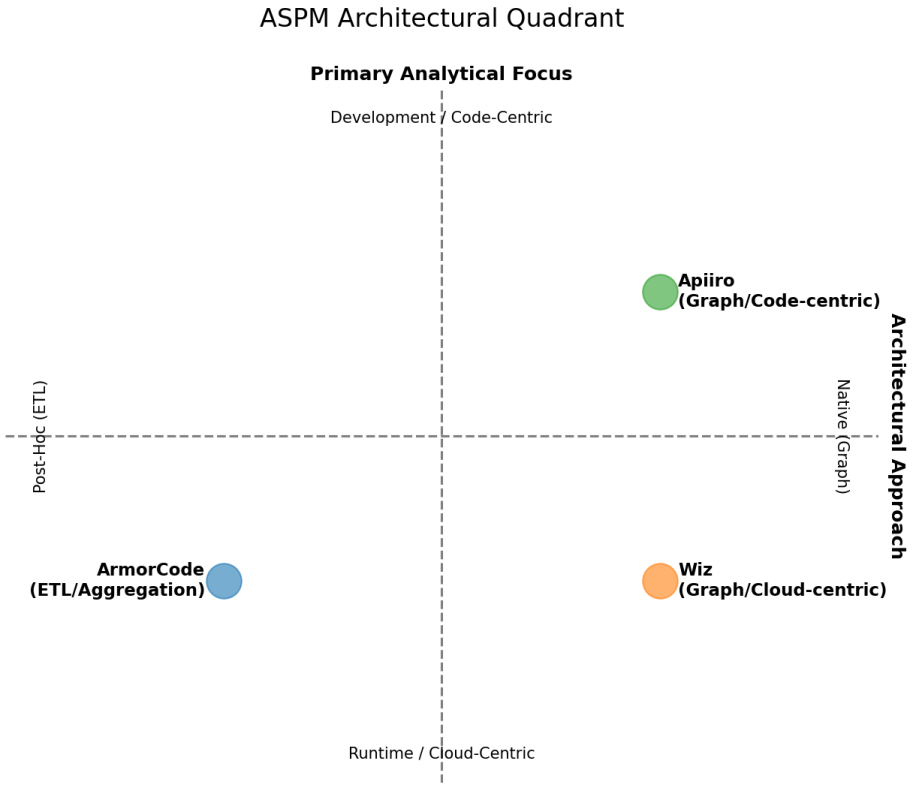

Figure A.2: ASPM Platform Architectural Quadrant

BIBLIOGRAPHY

[1] Red Hat, "What is devsecops?." https://www.redhat.com/en/topics/devops/what-is-devsecops, 2024. Accessed: 2025-10-07.

[2] M. Gostev and E. Gassanov, *DevSecOps: A Multivocal Literature Review*. ResearchGate, 2017.

[3] A. Rahman and A. Tiem, "Software security in devops: a systematic mapping study," *Journal of Cybersecurity*, vol. 3, no. 1, pp. 1–17, 2017.

[4] C. Ehrlich, P. Pinheiro, F. Divino, F. Marcondes, J. Bremer, and A. Nakamura, "Cybersecurity knowledge graph construction and applications: A survey," *IEEE Access*, vol. 10, pp. 68725–68740, 2022.

[5] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, J. E. L. Gayo, S. Kirrane, S. Neumaier, A. Polleres, *et al.*, "Knowledge graphs," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–37, 2021.

[6] E. van der Horn and F. de Boer, "Digital twin-driven prognostics and health management for complex equipment," *Journal of Manufacturing Systems*, vol. 57, pp. 167–181, 2020.

[7] S. D'Amico and J. Cheah, "The power of digital twins in cybersecurity," *Communications of the ACM Blog*, 2024. Accessed: 2025-10-07.

[8] Amazon Web Services, "What is devsecops?." https://aws.amazon.com/what-is/devsecops/, 2024. Accessed: 2025-10-07.

[9] RiverSafe, "The hidden cost of tool sprawl – why it's slowing your developers down." https://riversafe.co.uk/resources/tech-blog/the-hidden-cost-of-tool-sprawl-why-its-slowing-your-developers-down/, 2023. Accessed: 2025-10-07.

[10] Invicti Security, "3 ways that security tool sprawl can hurt application security testing." https://www.invicti.com/blog/web-security/3-ways-security-tool-sprawl-hurts-application-security-testing/, 2023. Accessed: 2025-10-07.

[11] FireMon, "Reducing mean time to remediation (mttr) with automated policy workflows." https://www.firemon.com/blog/reducing-mttr-with-automated-policy-workflows/, 2021. Accessed: 2025-10-07.

[12] M. Grieves, "Digital twin: manufacturing excellence through virtual factory replication," *White paper*, vol. 1, no. 2014, pp. 1–7, 2014.

[13] IBM Security, "Cost of a data breach report 2025," tech. rep., IBM Corporation, 2025.

[14] Synopsys, "What is software composition analysis (sca)?." https://www.blackduck.com/glossary/software-composition-analysis.html, 2024. Accessed: 2025-10-07.

[15] D. Salinas, D. Bussa, and M. Horvath, "Innovation insight for application security posture management," tech. rep., Gartner, August 2022.

[16] S. M. Kerner, "Solarwinds attack explained: And what it means for you," *TechTarget*, 2021.

[17] The White House, "Executive order on improving the nation's cybersecurity." EO 14028, 2021. Accessed: 2025-10-07.

[18] Synopsys, "Software supply chain security solutions." https://www.blackduck.com/solutions/software-supply-chain-security.html, 2024. Accessed: 2025-10-07.

[19] Anchore, "What is software supply chain security?." https://anchore.com/software-supply-chain-security/what-is-sscs/, 2024. Accessed: 2025-10-07.

[20] Checkmarx, "Software supply chain security tool." https://checkmarx.com/solutions/software-supply-chain-security/, 2024. Accessed: 2025-10-07.

[21] Aqua Security, "Software supply chain security." https://www.aquasec.com/products/software-supply-chain-security/, 2024. Accessed: 2025-10-07.

[22] [Author Redacted], "A Comparative Analysis of SDLC Security Platforms: Identifying the Coverage Gaps and Architectural Limitations of the Current Market," tech. rep., [Institution Redacted], 2025. Internal research document providing the basis for the market analysis.

[23] ArmorCode, "Reduce risk with ai-powered aspm." https://www.armorcode.com/, 2024. Accessed: 2025-10-07.

[24] Apiiro, "Your single application security platform." https://apiiro.com/platform/, 2024. Accessed: 2025-10-07.

[25] Wiz, "#1 cloud security software for modern cloud protection." https://www.wiz.io/, 2024. Accessed: 2025-10-07.

[26] ArmorCode, "Armorcode platform overview." https://www.armorcode.com/platform, 2024. Accessed: 2025-10-07.

[27] Seemplicity, "Application security posture management (aspm): What is it?." https://seemplicity.io/use-cases/application-security-posture-management-aspm-tools/, 2024. Accessed: 2025-10-07.

[28] Cycode, "The cycode risk intelligence graph." https://cycode.com/platform/risk-intelligence-graph, 2024. Accessed: 2025-10-07.

[29] Puppet, "2021 state of devops report," tech. rep., Puppet, 2021.

[30] DZone, "The cost of fragmented devsecops compliance reporting." https://dzone.com/articles/the-cost-of-fragmented-devsecops-compliance-reporting, 2023. Accessed: 2025-10-14.

[31] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[32] S. T. March and V. C. Storey, "Design and natural science research on information technology," *MIS Quarterly*, vol. 28, no. 4, pp. 555–579, 2004.

[33] M. Fernández-López, A. Gómez-Pérez, and N. Juristo, "Methontology: an ontology specification method," *Data & Knowledge Engineering*, vol. 33, no. 3, pp. 45–77, 2009.

[34] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, *Foundations of Semantic Web Technologies*. CRC press, 2009.

[35] A. Gómez-Pérez, "Evaluation of ontologies: A study of empirical criteria for goodness," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 3, no. 2, pp. 25–40, 2003.

[36] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," *Engineering*, vol. 3, no. 1, pp. 105–137, 2009.

[37] National Institute of Standards and Technology (NIST), "Framework for improving critical infrastructure cybersecurity (version 1.1)." NIST Cybersecurity Framework, 2018. Accessed: 2025-10-14.

[38] OWASP Foundation, "Owasp ontology driven threat modeling framework." https://owasp.org/www-project-ontology-driven-threat-modeling-framework/, 2022. Accessed: 2025-10-07.

[39] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human-Computer Studies*, vol. 43, no. 5-6, pp. 907–928, 1995.

[40] "Enhancing cybersecurity through autonomous knowledge graph construction by integrating heterogeneous data sources," *PeerJ Computer Science*, 2025. Accessed: 2025-10-19.

[41] NSF Public Access Repository, "Aiseckg: Knowledge graph dataset for cybersecurity education." https://par.nsf.gov/servlets/purl/10401616, 2025. Accessed: 2025-10-19.

[42] ResearchGate, "Cyberq: Generating questions and answers for cybersecurity education using knowledge graph-augmented llms." https://www.researchgate.net/publication/379279629_CyberQ_Generating_Questions_and_Answers_for_Cybersecurity_Education_Using_Knowledge_Graph-Augmented_LLMs, 2025. Accessed: 2025-10-19.

[43] Chef, "Devsecops is evolving - but are you keeping up?." https://www.chef.io/blog/devsecops-is-evolving-but-are-you-keeping-up/, 2025. Accessed: 2025-10-19.

[44] NCBI, "Engineering behavioral ontologies." https://www.ncbi.nlm.nih.gov/books/NBK584335/, 2025. Accessed: 2025-10-19.

[45] arXiv, "Validity in design science." https://arxiv.org/pdf/2503.09466, 2025. Accessed: 2025-10-19.

[46] Portland State University, "Threats to validity of research design." https://web.pdx.edu/~stipakb/download/PA555/ResearchDesign.html, 2025. Accessed: 2025-10-19.

[47] Aqua Security, "Protect your cloud native and AI apps with Aqua CNAPP." https://www.aquasec.com/aqua-cloud-native-security-platform/, 2025. Accessed: October 9, 2025.

[48] Aqua Security, "Securing the Build, Infrastructure, and Workloads Across Cloud Native Environments." https://events.afcea.org/tip21/CUSTOM/pdf/aqua_wp.pdf, 2021. Accessed: October 9, 2025.

[49] Aqua Security, "Improve DevOps Processes: Multiple Security Policies Applied to Images." https://www.aquasec.com/blog/applying-multiple-security-policies-to-images-in-your-pipeline-to-improve-speed-and-efficiency/, 2025. Accessed: October 9, 2025.

[50] Aqua Security, "What Is the Secure Software Development Lifecycle (SSDLC)?." https://www.aquasec.com/cloud-native-academy/supply-chain-security/secure-software-development-lifecycle-ssdlc/, 2025. Accessed: October 9, 2025.

[51] Aqua Security, "Cloud Native Security Best Practices: Vulnerability Management." https://www.aquasec.com/blog/container-vulnerability-management-best-practices/, 2025. Accessed: October 9, 2025.

[52] Checkmarx, "Cloud Native Application Security Platform Checkmarx One." https://checkmarx.com/product/application-security-platform/, 2025. Accessed: October 9, 2025.

[53] Checkmarx, "Checkmarx vs Black Duck (formerly Synopsys)." https://checkmarx.com/black-duck-synopsys/, 2025. Accessed: October 9, 2025.

[54] Checkmarx, "Container Security GraphQL API Documentation." https://docs.checkmarx.com/en/34965-397992-container-security-graphql-api-documentation.html, 2025. Accessed: October 9, 2025.

[55] Checkmarx, "CxSAST (OData) API Overview & Examples." https://docs.checkmarx.com/en/34965-46555-cxsast--odata--api-overview---examples.html, 2025. Accessed: October 9, 2025.

[56] Synopsys, "Black Duck Software Composition Analysis." https://www.blackduck.com/software-composition-analysis-tools/black-duck-sca.html, 2025. Accessed: October 9, 2025.

[57] Apiiro, "Application Risk Prioritization & Remediation." https://apiiro.com/product/application-risk-prioritization-remediation/, 2025. Accessed: October 9, 2025.

[58] Apiiro, "Apiiro Design - Secure Software Architecture." https://apiiro.com/design/, 2025. Accessed: October 9, 2025.

[59] Apiiro, "What Is Software Graph Visualization? Benefits & How It Works." https://apiiro.com/glossary/software-graph-visualization/, 2025. Accessed: October 9, 2025.

[60] Apiiro, "Navigate uncharted risk across your software supply chain with Apiiro's Risk Graph Explorer." https://apiiro.com/blog/navigate-uncharted-risk-across-your-software-supply-chain-risk-graph-explorer/, 2025. Accessed: October 9, 2025.

[61] ArmorCode, "Technology - ArmorCode." https://www.armorcode.com/technology, 2025. Accessed: October 9, 2025.

[62] ArmorCode, "ArmorCode: Reduce Risk with AI-Powered ASPM." https://www.armorcode.com/, 2025. Accessed: October 9, 2025.

[63] nVOIDs, "Hybrid Technical Data Architect —— Chicago, IL —— F2F Interview." https://jobs.nvoids.com/job_details.jsp?id=2818505&uid=6dd78c0341184a228631ca0d6a294e5b, 2025. Accessed: October 9, 2025.

[64] Wiz, "Wiz Recognized as a Leader in the 2025 IDC MarketScape for ASPM." https://www.wiz.io/blog/wiz-named-aspm-leader-by-idc, 2025. Accessed: October 9, 2025.

[65] Wiz, "Wiz Security Graph: How It Works, Benefits, Use Cases." https://www.wiz.io/lp/wiz-security-graph, 2025. Accessed: October 9, 2025.

[66] Wiz, "Wiz cloud security platform." https://www.wiz.io/platform, 2025. Accessed: October 9, 2025.