ThreatCompute: Leveraging LLMs for Automated Threat Modeling of Cloud-Native Applications

Anna Wimbauer
BIFOLD & TU Berlin
Berlin, Germany
Technical University of Munich
Munich, Germany
a.wimbauer@tu-berlin.de

Luca Muscariello
Cisco Systems
Paris, France
lumuscar@cisco.com

Jacques Samain
Cisco Systems
Paris, France
jsamain@cisco.com

Lion Steger
Technical University of Munich
Munich, Germany
stegerl@net.in.tum.de

Kilian Glas
Technical University of Munich
Munich, Germany
glask@net.in.tum.de

Max Helm
Technical University of Munich
Munich, Germany
helm@net.in.tum.de

Georg Carle
Technical University of Munich
Munich, Germany
carle@net.in.tum.de

Abstract

The increasing complexity of cloud-native applications has necessitated advanced methodologies for threat modeling and security analysis. This paper presents ThreatCompute, a novel framework that combines LLMs with attack graphs to automate the generation of threat hypotheses and the quantification of risk in Kubernetes environments. While traditional approaches to attack graph generation require significant manual effort from security experts, ThreatCompute leverages LLMs to extract security insights from system information, reducing reliance on manual intervention while maintaining high accuracy and generating context-specific, system-aware threat insights. The framework utilizes the MITRE ATT&CK Matrix and the Microsoft Threat Matrix for Kubernetes as structured domains of possible attack techniques. Based on LLM-generated threat hypotheses and a quantitative risk metric, ThreatCompute constructs detailed attack graphs that illustrate potential attack paths and assess their associated risks. This enables both qualitative and quantitative evaluations of application security across varying levels of granularity. Through real-world examples of Kubernetes applications, we demonstrate the effectiveness of our approach in identifying and quantifying security risks.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Threat Modeling, Attack Graphs, LLM, Kubernetes

CCSW '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1901-1/25/10
https://doi.org/10.1145/3733812.3765533

ACM Reference Format:
Anna Wimbauer, Luca Muscariello, Jacques Samain, Lion Steger, Kilian Glas, Max Helm, and Georg Carle. 2025. ThreatCompute: Leveraging LLMs for Automated Threat Modeling of Cloud-Native Applications. In *Proceedings of the 2025 Cloud Computing Security Workshop (CCSW '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3733812.3765533

1 Introduction

Modern applications are increasingly deployed in complex, distributed cloud environments. As cyber attacks grow in frequency and sophistication, there is a critical need for automated methods to assess and quantify security risks at scale. Security risk assessment plays a key role in protecting sensitive data, maintaining service availability, and preserving stakeholder trust. While existing frameworks provide structured approaches to modeling and mitigating risk, they often struggle with unseen or evolving attack strategies, especially in dynamic settings like Kubernetes.

Threat modeling generally involves two steps: First we identify potential threats in the system. Second, we estimate the likelihood of those threats being exploited. However, generating meaningful threat hypotheses typically requires deep system insight and extensive domain expertise. Johnson et al. [12] identify three core challenges in this process: (1) identifying and formalizing relevant system information, (2) structuring and analyzing security-relevant data, and (3) detecting and reasoning about potential weaknesses. Attack graphs, which are a widely used representation of attack paths, depend on this modeling process. Formal threat modeling languages such as the Meta Attack Language (MAL) provide a structured, automatable way to represent threats but still rely on manual input from security professionals.

Large language models (LLMs) have recently shown promise in cybersecurity, including for parsing system documentation and generating security insights [7, 36]. LLMs can synthesize unstructured and structured information, making them a strong candidate for

automating threat modeling in Kubernetes environments, where rich system metadata is available through APIs and scanning tools.

We present ThreatCompute, a novel framework that combines the formal modeling capabilities of MAL with the analytical power of LLMs. Our approach automates threat model generation and attack graph construction by extracting and analyzing Kubernetes system data using modular LLM prompting. Afterwards we let the LLM generate threat hypotheses using a curated set of attack techniques drawn from the Microsoft Threat Matrix for Kubernetes. These hypotheses are integrated with vulnerability scan results to generate structured attack graphs, guided by a refined time-to-compromise (TTC) metric. For evaluation, we tested ThreatCompute on two intentionally vulnerable Kubernetes applications. Against the ground truth in one application, ThreatCompute correctly identified 29 out of 30 known attack techniques and demonstrated high semantic accuracy in its generated threat descriptions. The implementation of the framework is designed for practical use, and we published the code on GitHub [1].

Contribution. We make the following contributions:

- **Threat Modeling (Section 4):** We introduce a framework that automatically generates system-aware threat models for Kubernetes using LLMs. The generated models are accurate and repeatable, as they are grounded in a structured system representation that deterministically captures the state and topology of the Kubernetes environment.
- **Time-To-Compromise (Section 5):** We adapt and extend a TTC model that incorporates vulnerability data and attacker skill to enable quantitative risk scoring.
- **Attack Graph Generation (Section 6):** We combine the system information, with the threat model and the computed TTC to simulate attacker behavior and generate probabilistic, data-driven attack graphs that reflect real exploit paths.

Together, these components address the core challenges of automated, scalable threat modeling in complex cloud-native systems.

2 Background

To support our approach to automated attack graph generation, this section introduces the foundational concepts and technologies on which our framework is built. We begin with an overview of Kubernetes security, threat modeling, and attack graphs, followed by a introduction to the MITRE ATT&CK framework.

Kubernetes Security. Cloud-native applications differ from traditional systems by using microservices that allow independent development, deployment, and scaling [21]. Containers are the main deployment units, ensuring consistency across environments. Kubernetes automates the deployment, scaling, and management of containers [4]. It features automated scheduling, self-healing, and service discovery, and is structured around master and worker nodes. Applications run in Pods, the smallest deployable units, which may contain one or more containers. Key components like the API server, scheduler, and controller manager maintain the cluster's state [28]. Despite these capabilities, Kubernetes introduces notable security challenges [28]. According to the Cloud Native Computing Foundation, 40% of users express security concerns. Exposed API servers, vulnerable containers, and misconfigurations increase risk. Best practices include role-based access control (RBAC), timely patching, and enforcing strict pod and network policies.

Threat Modeling and Attack Graphs. As computer systems grow in size and complexity, identifying and assessing security threats becomes increasingly difficult. Threat modeling methodologies such as STRIDE and PASTA offer structured approaches that help security professionals to systematically evaluate potential risks [15, 20]. To formalize and streamline this process, threat modeling languages can be used to describe system components, attack steps, and defensive mechanisms in a reusable, rule-based format. Unlike isolated threat models, these languages support the reuse of predefined logic and constraints, improving efficiency and consistency across analyses [12]. While threat modeling focuses on identifying vulnerabilities and possible threats within a system, attack graphs visualize how these threats might be exploited. Specifically, this work adopts the concept of host-based attack graphs, where nodes represent system components and edges denote feasible attack steps between them [39]. To facilitate the formal generation of such graphs, Johnson et al.[12] introduced MAL, a modeling framework that encodes threat logic into a domain-specific language (section 3.1). This enables the automated construction of attack graphs grounded in formal semantics and domain knowledge.

MITRE ATT&CK Framework. The MITRE ATT&CK framework is a widely adopted taxonomy and knowledge base of adversarial behavior across the attack life-cycle [31]. It classifies these attack steps by *tactic* (the adversary's goal), *technique* (the method used), and *procedure* (a specific implementation), commonly referred to as TTPs. Microsoft's Threat Matrix for Kubernetes adapts this framework to focus specifically on Kubernetes environments [19], offering a targeted reference for mapping security risks in containerized cloud-native systems.

3 Related Work

This section reviews relevant work in two key areas: (1) attack graphs and risk assessment, a foundational area for modeling adversarial behavior and quantifying system-level security risks; and (2) the application of LLMs in cybersecurity, including their use in threat modeling, security operations, and emerging automation efforts. Together, these threads highlight the potential and limitations of current methods, and motivate our goal of fully automating threat model generation and attack graph construction using LLMs.

3.1 Attack Graphs and Risk Assessment

Attack graphs have long been used to model adversarial behavior, support security assessments, and quantify system risk [9, 32]. Foundational contributions include early formalizations by Sheyner et al. [29] and Wang et al. [34], which introduced techniques for automated attack path enumeration and visualization. A notable advancement in this space is the Meta Attack Language (MAL) by Johnson et al. [12], which formalizes threat modeling as reusable, composable domain-specific languages (DSLs). MAL enables semi-automated generation of attack graphs by encoding attack logic and relationships in a structured, probabilistic format. Wideł et

[1] https://github.com/ThreatCompute/ThreatCompute

al.[35] later provided a formal specification of MAL, further establishing its value as a foundation for automation in attack graph generation. MAL's impact is evident in its adoption across domains such as industrial control systems [24], corporate IT environments [13], vehicular systems [14], cloud platforms like AWS [8], and the power grid [11]. However, while MAL reduces manual effort, DSL creation still requires significant domain expertise and system knowledge. Other efforts have aligned attack graph generation with real-world adversary behavior by integrating frameworks like MITRE ATT&CK [2, 22], or by modeling risk through probabilistic estimation [37]. Recent approaches explore automation using system telemetry or vulnerability data, yet many remain dependent on static rules or human input [16]. Building on MAL's two-stage process—first defining threat models, then generating graphs—our work seeks to automate both steps using LLMs. By extracting structured threat models directly from system data, we eliminate the need for manual DSL authoring and enable end-to-end attack graph generation for Kubernetes environments.

3.2 LLMs for Cybersecurity

LLMs are increasingly applied across cybersecurity domains, including threat modeling, incident response, and vulnerability analysis. Elsharef et al. [7] explore LLM-driven threat modeling by embedding design documents into a vector database to support analyst queries, offering flexible retrieval. Wu and Yang et al. [36] fine-tune LLMs to identify threats in banking system descriptions using the STRIDE framework. Similarly, Saha et al. [25] introduce ThreatLens, a multi-agent LLM framework for hardware security verification, which assists with threat identification and test plan generation but incorporates human feedback. While these efforts demonstrate the potential of LLMs in security analysis, they do not generate structured threat models as attack graphs or provide mechanisms to quantify risk. Other works explore broader applications of LLMs in cybersecurity, such as the Microsoft Security Copilot [10], which integrates LLMs into SOC workflows for triage and remediation using real incident data. CTI-focused research has also applied LLMs to entity extraction and TTP mapping [1, 3, 17], while tools like PentestGPT [6] simulate attacker behavior for offensive testing. One recent work touches on LLM-driven attack graph generation [23], though they typically focus on linking existing vulnerabilities rather than generating threat models from system data.

In contrast, ThreatCompute offers an end-to-end framework that fully automates both structured threat modeling and attack graph generation for Kubernetes systems. It builds formal system graphs, applies modular prompting, and grounds its reasoning in real vulnerability and misconfiguration data, effectively bridging the gap between raw system telemetry and actionable security models.

4 Threat Model Generation Framework

Writing threat modeling rules requires detailed knowledge of a system's architecture, components, and vulnerabilities. Johnson et al. [12] showed that once these rules are formalized, structured attack graphs can be generated automatically. However, authoring such rules remains a manual, expert-driven task.

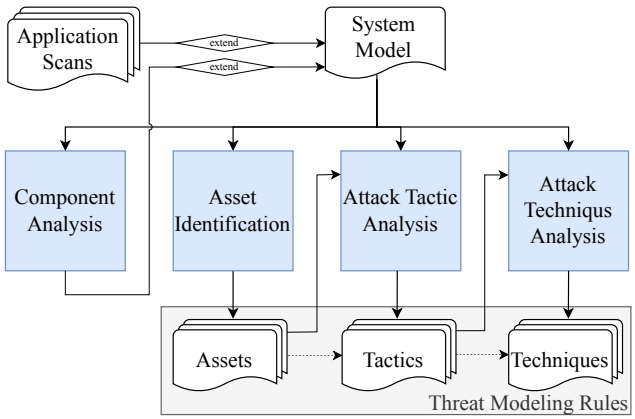

Figure 1: Framework overview.

ThreatCompute eliminates this bottleneck by using LLMs to automatically generate system-specific threat models for Kubernetes-based applications. Unlike generic prompting, our approach uses a structured graph representation of the target Kubernetes environment to guide the LLM. This graph captures the system's topology and configuration, with nodes representing entities like pods or services and edges encoding containment relationships. By aligning this system model with the Microsoft Threat Matrix for Kubernetes [19], we constrain and focus the LLM's reasoning on realistic, context-aware threats. In essence, ThreatCompute functions as an early AI agent [26] where the LLM dynamically directs its own reasoning over structured inputs, laying the groundwork for future integration with security scanners and orchestration tools via Kubernetes-native MCP servers.

Figure 1 provides an overview of the framework. Our approach applies a *divide and conquer* strategy, breaking down threat modeling into four key steps:

- **Component Analysis:** Analyzing and summarizing system architecture.
- **Asset Identification:** Categorizing system assets for threat modeling.
- **Attack Tactic Analysis:** Determine plausible attacker goals per asset (tactics).
- **Attack Technique Analysis:** Identify techniques attackers could use to achieve the attack tactics.

This modular pipeline enables precise LLM prompts at each step. Rather than presenting the entire system state, we filter scan data (e.g., SBOMs, vulnerabilities, and misconfigurations) based on the graph structure to include only the most relevant context. Each step builds on the last, allowing the LLM to generate focused, high-quality threat modeling rules.

The framework focuses on detecting Kubernetes-specific threats, leveraging Microsoft's Threat Matrix for Kubernetes [19]. It does not analyze source code or interact with application frontends, making techniques like SQL injection out of scope.

4.1 System Model as Input

The first and most essential step in threat modeling is gathering all security-relevant data. To enable a fully automated process, we systematically collect and structure this data for further analysis.

We extract relevant system information from the Kubernetes API and security scanners, namely Trivy[2] and Kubescape[3], and use it to construct a structured system model (*SM*). This model, represented as a directed graph $SM = (N_{SM}, E_{SM})$, captures the architecture of the Kubernetes environment and serves as the foundation for threat modeling. To retrieve this data, we query the Kubernetes API using the Python Kubernetes client, extracting system resources such as namespaces, pods, containers, and their containment relationships. We then enrich this topology with scanner output capturing vulnerabilities (CVEs), misconfigurations, exposed secrets, exposed ports, and privilege-related issues such as overly permissive RBAC roles or default service accounts with elevated access. Shell availability and root access are determined by inspecting each container individually. The system model graph consists of the following elements:

- **Nodes** (N_{SM}) represent system components such as clusters, namespaces, nodes, pods, containers, and shells. Each node is annotated with security-relevant metadata, including kernel versions, SBOM contents, known vulnerabilities, misconfigurations and compliance violations, access controls (e.g., RBAC role bindings), and network exposure details (e.g., open ports or public-facing services).
- **Edges** (E_{SM}) define containment relationships (e.g., a container belonging to a specific pod).

This system model aggregates critical security data, providing a structured and comprehensive representation of the environment while enabling architecture-based filtering. Listing 1 illustrates an example system model node representing a container in the Kubernetes cluster. For downstream processing, we serialize the graph in Graph Modeling Language (GML) format and use it as structured input for the LLM. The completeness and accuracy of this model are crucial: missing system components, privileges, or vulnerabilities in the system model would exclude them from the generated threat model and attack graph, directly impacting the quality of the security assessment.

Listing 1: Example system model node for a container.

```
NAME "hunger-check"
CONTAINER "hunger-check"
IMAGE "madhuakula/k8s-goat-hunger-check"
TYPE "Container"
NAMESPACE "big-monolith"
POD_INFOS [
    can_use_k8s_api 1
    roles "secret-reader"
]
SBOM "['ubuntu', 'adduser', ..., 'zlib1g']"
CHECKS [
    name "Non-root containers"
    severity "Medium"
    scoreFactor 6
]
CHECKS ...
CVEs [
  id "CVE-2024-8096"
  title "curl: OCSP stapling bypass with GnuTLS"
  resource "curl"
  severity "MEDIUM"
  cvss [
    version 3.1
    vector "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N"
    metrics [
        baseScore 6.5
        exploitabilityScore 3.9
        impactScore 2.5
    ]
  ]
]
CVEs ...
```

4.2 Threat Modeling Steps

We now describe how ThreatCompute uses a general-purpose LLM to transform a graph-based system model of a Kubernetes application into a structured, system-aware threat model. The process consists of four modular stages: component analysis, asset identification, attack tactic analysis, and attack technique analysis. Each step builds on the previous one, enabling a progressive refinement of threat knowledge. To perform these steps, ThreatCompute employs a general-purpose LLM capable of understanding and generating structured security-related content. We used Mistral-NeMo, but other open-source models, such as LLaMA 3 or Mixtral, can also be used, provided they do not restrict security-relevant outputs. The model is prompted to generate structured outputs, like asset summaries, tactic lists, and technique descriptions, formatted in JSON and aligned with established taxonomies, namely the MITRE ATT&CK Matrix for Kubernetes. Our prompting approach emphasizes deterministic reasoning and system-specific context, and outputs are explicitly grounded in the system graph and vulnerability scan data.

Component Analysis. In the initial preparatory step, the system information is parsed, and the LLM is used to analyze and summarize the information of the system components. For containers, this includes aggregating the SBOM and inferring the potential purpose of the container. If an instance contains other components (e.g., a pod with multiple containers), the analysis of those encapsulated components is included in the prompt (instance analysis prompt: "`You are provided with the attributes of an 'asset_type': 'node_attributes'. The 'asset_type' contains instances 'successor_asset_type' which were analyzed as follows: 'analyses'. Describe the 'asset_type' and its possible use in 3 sentences.`"). Additionally, the LLM is prompted to consolidate the analyses of all instances within a category (e.g., all pods). These summaries and analyses are an essential foundation for the subsequent steps.

Asset Identification. After the component analysis phase, the next step identifies relevant asset categories for threat modeling. To support high-level threat reasoning in Kubernetes-based applications, assets are abstracted as general categories rather than individual instances (e.g., specific pods or containers). These categories are derived from the system model (*SM*) and include foundational types such as *Cluster*, *Namespace*, *Node*, and *Pod*. Given the heterogeneity of container configurations—reflected in their SBOMs and roles—containers are further classified using the results of LLM-based component analysis. The LLM assigns each container

[2] https://trivy.dev/
[3] https://kubescape.io/

to a functional category such as *Application Container*, *Networking*, *Storage Container*, or *Health Probe*, accompanied by a concise, LLM-generated description for downstream threat modeling. To ensure completeness, all containers are checked for classification. Unassigned instances are placed in a fallback *General* category. This structured asset identification process supports the development of a threat model that operates at a meaningful level of abstraction, while preserving the critical security-relevant characteristics of system components.

Attack Tactic Analysis. Having identified the relevant attack assets, the subsequent step involves prompting the LLM to propose potential attack tactics applicable to each asset. The tactics of the MITRE ATT&CK Matrix serve as a well defined set of high-level attack objectives. For each identified asset, the LLM is instructed to "`Analyse the security, possible misuse or exploitation of the asset 'asset'.`" and to "`List tactics from the MITRE ATT&CK Matrix that can be performed on 'asset' instances.`". The foundation of the attack tactic analysis is the set of asset descriptions generated in the previous step, ensuring a context-aware evaluation. To further enhance accuracy and consistency, the prompt explicitly includes the complete list of tactics from the MITRE ATT&CK Enterprise Matrix. This approach constrains the LLM's output to a predefined taxonomy, reducing ambiguity and aligning the results with a widely accepted cybersecurity framework.

Attack Technique Analysis. The final stage of the threat modeling process focuses on identifying specific attack techniques that could be used to achieve the previously determined attack tactics for each asset. At this point, we explicitly integrate information about misconfigurations and vulnerabilities associated with each asset, which were not directly considered in the tactic analysis phase. To determine the applicable techniques, we must first process the potentially extensive list of vulnerabilities and misconfigurations identified by security scanners within the system model. To manage this volume of data, we first group vulnerabilities by the software they affect, generating concise summaries for each group. These per-software summaries are then aggregated into a comprehensive, asset-wide summary. This structured summarization provides the LLM with a clearer and more relevant view of the weaknesses present in each asset. Subsequently, the LLM is prompted with the instruction: "`Analyze the given asset 'asset' and list techniques that could be used to achieve the tactic 'tactic' on the 'asset'.`". To ensure the generation of relevant and specific techniques, this prompt is enriched with essential contextual information, including:

- The asset description generated during the component analysis phase.
- Summaries of vulnerabilities and misconfigurations associated with the asset, allowing the LLM to identify techniques that exploit these specific weaknesses.
- A list of techniques corresponding to the specified tactic in the Threat Matrix for Kubernetes.
- A complete list of system assets, helping the LLM understand potential targets and the relationships between different components.

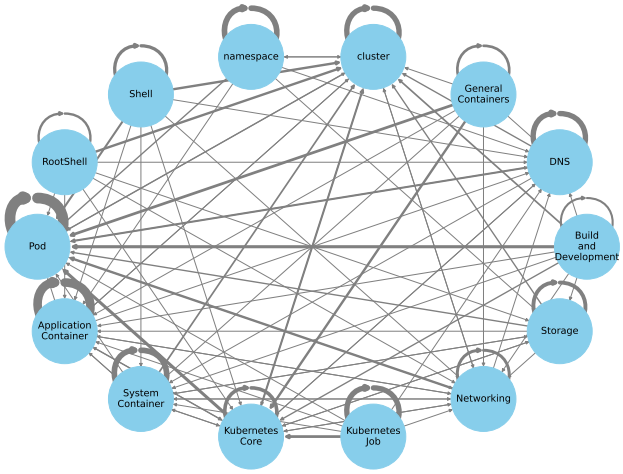

Figure 2: Threat model graph for Kubernetes Goat.

For every tactic identified per asset, the LLM generates a structured list of potential techniques. A key aspect of each proposed attack technique is the designation of a target asset, which may be either the asset from which the technique is initiated or a different asset within the system. Table 1 shows two example techniques. Each technique in this list is detailed by:

- Its name, aligning with the Threat Matrix for Kubernetes.
- A description of how the technique could be executed in the context of the specific asset and its vulnerabilities.
- The attack target, clearly identifying which asset the technique aims to compromise.
- Any prerequisite tactics that must be achieved before the successful execution of the technique, establishing a logical sequence of attack steps.

The resulting output is a structured set of threat modeling artifacts, from tactics to concrete techniques linked to asset vulnerabilities. Unlike prior work, which either produces unstructured outputs or relies on generic mappings, our method tightly integrates LLM reasoning with concrete system data and formal taxonomies—automating both the breadth and depth of threat modeling in Kubernetes environments.

4.3 Threat Model Composition

The output of the previous threat modeling steps is a list of techniques, each associated with a specific tactic and defined by a source asset (from which the technique is initiated) and a target asset (which the technique aims to compromise). Based on this list, a single unified threat model graph is constructed. This graph fulfills two main purposes: it provides a structured and visual representation of potential threats within the system, and it enables the simulation of attack paths for subsequent generation of attack graphs.

The threat model is formalized as a bi-directed graph, denoted as $TM = (N_{TM}, E_{TM})$, where:

- **Nodes** (N_{TM}) represent assets with associated security attributes.
- **Edges** (E_{TM}) define relationships between assets and attack techniques.

Table 1: Example techniques generated for the example application Kubernetes Goat.

Source	Target	Tactic	Technique	Required Tactic	Technique Description
Pod	Pod	Persistence	Writable hostPath mount	Privilege Escalation	Mount a writable hostPath volume to the Pod, allowing the attacker to write files to the host system and achieve persistence.
Application Container	Networking	Discovery	Network mapping	Initial Access	An attacker can use tools like 'nmap' or 'netcat' to map the network and identify active services, open ports, and potential targets within the cluster. This can be achieved by exploiting a container with system-level operations (7 containers) or by compromising a container with access to the Kubernetes API (18 containers).

A node is instantiated for each identified asset as described in section 4.2. Subsequently, each set of techniques from one source to one target asset are encoded as a directed edge between those two assets, reflecting the attack flow. Figure 2 illustrates an example threat model graph generated for an example Kubernetes application. This high-level representation facilitates the identification of feasible attack paths through graph traversal.

Conclusion. This approach automates threat modeling for Kubernetes environments, reducing the manual effort required to define security risks. By leveraging LLMs for structured threat rule generation, the framework enhances cybersecurity analysis and provides a scalable solution for security assessments.

The effectiveness of ThreatCompute's threat modeling stems from two core design decisions: domain restriction and task specialization. First, by constraining the LLM to operate within the predefined taxonomy of the Microsoft Threat Matrix for Kubernetes, we significantly reduce the space of possible outputs. This domain restriction ensures that the generated threats remain both relevant and realistic to Kubernetes environments, and reduces the risk of hallucination or irrelevant outputs—a known concern when using LLMs in open-ended tasks.

Second, we adopt a highly modular pipeline that breaks down the threat modeling process into tightly scoped subtasks. Each step in the pipeline, from component analysis to tactic and technique generation, isolates a specific reasoning challenge and provides targeted context based on the system model graph. This aligns with strengths demonstrated by LLMs in prior work: they are particularly well-suited to tasks involving summarization, pattern recognition, and context-aware mapping. In our framework, the LLM's role is not to generate free-form attack ideas, but to analyze structured system information and map it step-by-step into a curated threat space. This minimizes ambiguity and optimizes the quality and consistency of the generated threat model.

As a result, we believe that the generated threat models, and the subsequent attack graphs derived from them, are not only meaningful, but grounded in realistic adversarial behavior. The structure of the process lends itself to inductive trust: if each step remains within a constrained domain and performs reliably, then the composition of these steps can be expected to produce valuable, system-aware threat insights.

5 Time-To-Compromise as Quantitative Metric

To complement qualitative threat modeling with quantitative security risk assessment, we employ the Time-To-Compromise (TTC) metric, which estimates the expected time until an attacker compromises a system or a system component. Originally introduced by McQueen et al. [18], TTC aids organizations in prioritizing resources to secure high-risk assets. It enables data-driven system evaluation based on system-specific CTI and provides fast and intuitive information about the system's security state.

5.1 Foundational Model

McQueen's model computes TTC based on the number of system vulnerabilities and an attacker's skill level. It models the TTC as three main processes:

Process 1 A vulnerability is present in the system, and the attacker already has a suitable exploit, leading to a rapid potential compromise.

Process 2 A vulnerability exists, but the attacker lacks an exploit, requiring additional time for the attacker to develop or acquire the necessary exploit.

Process 3 No known vulnerabilities are currently exploitable, meaning that an attacker must wait for the discovery of a new (zero-day) vulnerability.

The model assigns discrete skill levels to attackers and assumes uniform exploit distributions. While this approach provides a general framework, it does not incorporate vulnerability-specific characteristics or environmental factors.

5.2 Refinements and Adaptations

Several works have extended the original TTC model to better represent reality. Zieger et al. [40] and Ling et al. [24] both incorporated vulnerability-specific characteristics, such as CVSS scores, into the TTC estimation. They also both update constant values in the model according to recent real-word datasets. The approach by Zieger et al. [40] relies on the complete set of possibly applicable vulnerabilities $\mathcal{W}$, making it impractical for large-scale systems like Kubernetes. Ling et al. [24] further refined the model for Industrial Control Systems by introducing exploitability-based scaling for TTC computations and setting fixed time estimates for different attacker skill levels. Ling's work is a very good example of how the original approach by McQueen et al. can be refined for one specific

attack environment, and we base our TTC computation on Ling's approach.

5.3 Time-to-Compromise for Kubernetes

The original TTC computation by McQueen et al. does not make use of the full system information available and uses constants based on CTI reports from 2006, which no longer reflect the current security landscape. To adapt the TTC concept for Kubernetes, we build upon McQueen et al.'s methodology while integrating improvements inspired by Zieger et al. [40] and Ling et al. [24]. These enhancements include incorporating CVSS scores into the three computational processes and updating constants based on current CTI, ensuring a more accurate and context-aware risk assessment.

Definition 5.1 (Kubernetes Time-To-Compromise). Let $S = \{novice, beginner, intermediate, expert\}$ represent a set of discrete skill levels. Let V denote the set of vulnerabilities in the component where each vulnerability i has a CVSS base score b_i and exploitability score x_i. For attacker skill level $s \in S$ the redefined time-to-compromise

$$\tau : V \times S \to \mathbb{R},$$

is defined as

$$\tau(V, s) = t_1 P_1 + t_2(1 - P_1)(1 - u) + t_3 u(1 - P_1).$$

5.3.1 Process 1. Like Ling et al. [24] we integrate CVSS scores into the computation of t_1. We align exploitability scores from CVSS v2 and v3.1 to a common scale by increasing the multiplication factor for v3.1 to 20. The worst-case exploitability score is used to determine:

$$t_1 = c_1 \frac{10}{x_{max}}, \quad x_{max} = \max(x_i \in V), \quad c_1 = 1 \text{ day}.$$

For process 1, t_1 incorporates CVSS exploitability scores, and we extend this by integrating vulnerability-specific values into P_1. As the CVSS *exploit code maturity* metric is often missing from public databases (e.g., NVD by NIST[4]), we approximate exploit availability based on the CVSS base score. Higher base scores typically indicate vulnerabilities that are easier to exploit and have greater impact on confidentiality, integrity, or availability.

Accordingly, we define P_1 as:

$$P_1 = 1 - e^{-\sum_{i=1}^{|V|} \frac{b_i}{b_{max}} \frac{m(s)}{k}}, \quad b_{max} = 10,$$

where b_i is the base score of vulnerability i, b_{max} is the maximum possible base score, and $m(s)$ is the number of exploits available to an attacker with skill level s. The product $\frac{b_i}{b_{max}} \cdot \frac{m(s)}{k}$ represents the conditional probability of successfully exploiting vulnerability i: first finding a suitable exploit ($\frac{m(s)}{k}$), then having the capability to use it ($\frac{b_i}{b_{max}}$). Following McQueen et al. [18], we assume exploits are uniformly distributed across all k vulnerabilities, making $\frac{m(s)}{k}$ independent of a specific i.

We set $k = 270139$, reflecting the number of vulnerabilities currently listed in the NIST National Vulnerability Database[5]. Table 2 presents the $m(s)$ values derived from the Metasploit exploit database, which classifies exploits across seven ranks[6]. Exploit counts per level were obtained by filtering the Metasploit dataset accordingly. Inspired by these rankings, we assign skill levels as follows: *novice*: excellent, *beginner*: excellent–good, *intermediate*: excellent–normal, and *expert*: excellent–manual. Vulnerabilities with a CVSS base score of 0 are excluded, as they represent no impact across all metrics and thus no exploitability.

Table 2: Number $m(s)$ of readily available exploits to an attacker of skill level s.

Skill level s	Number of exploits $m(s)$	$m(s)/k$
novice	2418	0.009
beginner	3220	0.012
intermediate	4956	0.018
expert	5800	0.021

5.3.2 Process 2. Since exploitability influences both t_1 and t_2, we redefine t_2 similarly:

$$t_2 = c_{2,s} \cdot \frac{10}{x_{max}},$$

Where $c_{2,s}$ is the mean time to develop an exploit for an attack of skill level s. [24] set these $c_{2,s}$ values as 37 days for a novice, 27 days for a beginner, 16 days for an intermediate, and 6 days for an expert attacker based on a threat intelligence report from 2017. According to the threat intelligence report on time to exploit trends by google [5] 12% of n-day vulnerabilities were exploited within 1 day and over half were exploited within 1 month (i.e. on average 30.4 days). As [24] we divide this range among the four attacker levels. Based on these values we update $c_{2,s}$ to: 30.4 days for a novice, 20.6 days for a beginner, 10.8 days for an intermediate, and 1 day for an expert attacker.

The probability u is refined by incorporating the fraction of exploitable vulnerabilities for each skill level:

$$u(V, s) = \min\left(u_{max}, \max\left(u_{min}, \left(1 - \frac{|\{x_i > x(s), x_i \in V\}|}{|V|}\right)^{|V|}\right)\right),$$

where $x(s)$ represents the maximum exploitability score an attacker of skill s can master [24], and $u_{min} = 0.05$, $u_{max} = 0.95$ ensure stability.

5.3.3 Process 3. The calculation of process 3 remains unchanged because the term $t_3 u(1 - P_1)$ already contains all the previously mentioned integrations of CVSS scores via u and P_1.

$$t_3 = \left(\frac{1}{f} - 0.5\right) c_3 + t_2, \quad \text{with } c_3 = 10.14 \text{ days}.$$

The constant c_3 represents the mean time between vulnerabilities and was set to 30.42 days by McQueen et al. [18] based on a threat intelligence report from 2004 (the following papers used the same constant). In 2023 Google's threat intelligence team tracked 36 zero-day vulnerabilities which targeted enterprise-focused technologies

[4] https://nvd.nist.gov/vuln-metrics/cvss [Accessed 18 Nov 2024]
[5] https://nvd.nist.gov/general/nvd-dashboard [Accessed 18 Nov 2024]
[6] https://docs.metasploit.com/docs/using-metasploit/intermediate/exploit-ranking.html [Accessed 18 Nov 2024]

[30]. Based on Google's review of zero-day vulnerabilities, we update the value to $c_3 = 36/365 = 10.14$ (on average, there was a new zero-day vulnerability targeting enterprise-focused technologies every 10.14 days in 2023).

Implementation and Conclusion. The proposed TTC computation is applied by determining the TTC for each asset instance within our cloud-native application, including containers, pods, and the cluster itself. For containers, this is directly derived from their associated vulnerabilities and misconfigurations. The TTC of a pod is derived by considering the vulnerabilities and misconfigurations of the container with the lowest TTC, combined with any pod-specific issues. Overall, the number of vulnerabilities and misconfigurations in a component has a much stronger impact on lowering the TTC for *novice* and *beginner* attackers than for *expert* attackers. This reflects the intended model behavior: less skilled attackers rely heavily on the presence of multiple vulnerabilities, whereas expert attackers can often succeed with fewer or more subtle weaknesses. This approach leverages system-specific CTI to provide a data-driven and objective estimation of the time required for an attacker to compromise each asset, moving beyond subjective expert assessments. The resulting quantitative TTC metric offers several key advantages. It enables organizations to prioritize security efforts by identifying high-risk assets, facilitates measurable evaluation of the system's security posture, and provides rapid, intuitive insights into its current state. Moreover, the computational model underpinning TTC allows for the automated and dynamic assessment of the relative efficacy of security interventions, such as vulnerability remediation, thereby informing strategic resource allocation to optimize the security of Kubernetes applications.

6 Attack Graph Generation

In the final step of the ThreatCompute pipeline, we use the system model, the LLM-generated threat model, and the component-specific TTC values to simulate adversarial behavior and generate an attack graph. A graph represents a structured view of feasible attack paths through the system, enabling both qualitative and quantitative risk analysis. While the threat model operates at the asset-technique level (e.g., "Application Container" → "Exploit Public-Facing Application"), the attack graph refines this to the instance-technique level, mapping abstract threats to concrete system components.

6.1 Graph Generation Algorithm

The graph generation algorithm simulates attacker behavior by performing guided random walks through the threat model. Each step in a walk is influenced by two key constraints: the *system architecture*, which enforces consistency with the structural relationships between components, and the TTC values, which bias the selection toward components that are easier to compromise (i.e., those with lower TTC). We adopt a weighted random walk approach rather than a shortest-path algorithm in order to model an attacker who lacks complete knowledge of the system and does not necessarily pursue specific high-value targets. Instead, the attacker explores the environment opportunistically from their initial access point. This approach follows the principles of MAL-based attack graph generation described by Widel et al. [35], in which attacker decision-making is represented as a probabilistic traversal of the threat model. The complete procedure is given in Algorithm 1, with notation summarized in Table 3. The resulting attack graph $AG = (N_{AG}, E_{AG})$ consists of:

- **Nodes** (N_{AG}): a subset of system model components ($N_{AG} \subseteq N_{SM}$).
- **Edges** (E_{AG}): attack techniques executed between instances, labeled with the applied technique $T_{AG}(i, j)$ and annotated by their TTC value τ_{ij}.

Each threat model node $n \in N_{TM}$ is mapped to one or more concrete instances in the system model $SM = (N_{SM}, E_{SM})$. During traversal, when a threat model edge is followed, the algorithm selects a corresponding instance pair in SM, weighted by TTC, and extends the walk with that step.

The process begins by computing all TTC values, initializing an empty attack graph, and identifying all valid starting steps IA, i.e., steps associated with techniques of type *Initial Access* (IsInitialAccess). Each walk then starts from a sampled step in IA and proceeds iteratively. For each step, the set of possible next moves S is determined from the threat model. Candidate steps S are then filtered by the procedure FilterFeasible, which ensures that:

Algorithm 1: Attack graph generation.

Input: Threat model $TM = (N_{TM}, E_{TM})$
System model $SM = (N_{SM}, E_{SM})$
Maximum walk length M
Attacker skill level *skill*
Output: Attack graph $AG = (N_{AG}, E_{AG})$
Compute $\tau(v, skill)$ for all $v \in N_{SM}$
Init $AG = (\emptyset, \emptyset)$
Init $IA = \{(t^1, i^1), \dots\}$ where IsInitialAccess(t^l)
for $k = 1$ **to** N **do**
  Sample start $(t_0, i_0) \in IA$ with weight $1/\tau(i^l)$
  Set walk $W = [(t_0, i_0)]$
  walk_successful ← false
  for $s = 1$ **to** M **do**
    $S \leftarrow \left\{(t_s^l, i_s^l) \mid (a(i_{s-1}), a(i_s^l)) \in E_{TM}, \; t_s^l \in T_{TM}(a(i_{s-1}), a(i_s))\right\}$
    $S \leftarrow$ FilterFeasible(S)
    if $S = \emptyset$ **then**
      break
    Sample $(t_s, i_s) \in S$ with weight $1/\tau(i_s^l)$
    Append (t_s, i_s) to W
    if IsImpact(t_s) **then**
      walk_successful ← true
      break
  if *walk_successful* **then**
    Add visited instances to N_{AG}
    Add edges $(i_{s-1} \rightarrow i_s)$ with technique t_s to E_{AG}
return AG

Table 3: Notation for attack graph generation.

Parameter	Definition
N_{TM}, E_{TM}	Nodes, Edges of threat model TM
N_{SM}, E_{SM}	Nodes, Edges of system model SM
N_{AG}, E_{AG}	Nodes, Edges of attack graph AG with $N_{AG} \subseteq N_{SM}$
$a(i) \in N_{TM}$	Asset type of system instance $i \in N_{SM}$
$I(i) = \{j \in N_{SM} : i \in N_{TM} \wedge a(j) = i\}$	Set of system instances belonging to asset $i \in N_{TM}$
$T_{TM}(i, j)$ for edge $(i, j) \in E_{TM}$	Set of techniques from asset i to j
$T_{AG}(i, j)$ for edge $(i, j) \in E_{AG}$	Set of techniques from instance i to j
$\tau_{ij} = \tau(j)$	TTC for edge $(i, j) \in E_{AG}$
(t_s, i_s) with $i_s \in N_{SM}$	Attack step s from instance i_{s-1} to i_s with technique $t_s \in T_{TM}(a(i_{s-1}), a(i_s))$
$W = ((t_0, i_0), (t_1, j_1), \cdots)$	Attack path, sequence of attack steps

(i) a valid path exists in the system model between the previously compromised instance i_{s-1} and the candidate instance i_s^l; (ii) the tactic required for t_s^l has already been satisfied by earlier steps in the walk; and (iii) the step does not correspond to the *Initial Access* tactic, which is only allowed at the beginning of the attack. A feasible next step is then sampled from the filtered set S according to TTC weights. Walks terminate when the maximum step length M is reached, when no feasible steps remain, or when a technique of the tactic *Impact* is achieved (IsImpact). Only successful walks—those that reach the *Impact* phase—are aggregated into the final attack graph.

Attack Graph–Based Security Assessment. The resulting attack graph and the set of successful attack paths serve as a foundation for both quantitative and qualitative security assessments. For *qualitative analysis*, visualizing the attack graph reveals frequently traversed components, with node sizes reflecting traversal frequency (e.g., Figure 3). This allows analysts to identify high-risk instances that exhibit low TTC values, appear across multiple attack chains, and connect diverse parts of the system. These nodes represent critical points of compromise and propagation, offering insight into attack origins and lateral movement potential. For *quantitative analysis*, we follow Wideł et al. [35] to compute a global TTC score for the system. This score is obtained by identifying the *minimal-cost attack path*-the successful path with the lowest cumulative TTC. This path reflects the most efficient route an attacker could exploit and serves as a lower bound for system compromise time. The global TTC metric can be used as a quantitative risk indicator to guide security prioritization and system hardening.

6.2 Data-Driven Validation Across the Pipeline

A key challenge in evaluating automated attack graph generation is the lack of a universally accepted ground truth. To address this, we designed the ThreatCompute framework to be data-driven and mutually validating at every level of the pipeline:

Threat Modeling The LLM-generated threat hypotheses are grounded in the complete system model and constrained by the Microsoft Threat Matrix for Kubernetes, ensuring that only realistic and domain-relevant tactics and techniques are considered.

Time-To-Compromise This quantitative metric is computed for each system component using vulnerability and misconfiguration data retrieved from Kubernetes security scanners. It provides a probabilistic estimate of exploitability that is grounded in component-specific, real-world security data.

Attack Graph Generation The final attack graph incorporates both the threat model and TTC values. Importantly, each simulated attack step is validated for feasibility within the system model, ensuring that no synthetic paths are introduced.

This multi-level integration not only ensures completeness and realism, but also allows for cross-checking between the layers. For example, if a component is targeted by many attack techniques in the threat model but has a very high TTC, it will rarely appear in the final attack graph—reflecting its low practical exploitability. Conversely, if a component has a low TTC but was underrepresented in the threat model, it may still emerge prominently in the graph due to its high attack attractiveness. In this way, TTC and the threat model act as checks on each other, and the attack graph synthesizes both to reflect likely attack paths. This mutual validation structure serves as a form of internal consistency, demonstrating that the generated attack graph is not only grounded in real data but also reflects systemic security dynamics—even in the absence of explicit ground truth. Importantly, our goal is not to produce a complete enumeration of all possible attack paths, but rather to generate a realistic and actionable sample of high-impact paths that reflect the most likely adversarial behavior given the current system state. A fully exhaustive graph would include many improbable paths, potentially overwhelming security teams with low-priority findings. By using risk-weighted sampling guided by TTC and constrained threat modeling, our approach focuses on meaningful attack trajectories that are grounded in system-specific vulnerabilities, privileges, and exposure. This targeted sampling approach makes the framework significantly more scalable, especially in large and dynamic Kubernetes environments where the number of possible paths grows rapidly with system complexity. Instead of attempting to model the full attack space, ThreatCompute concentrates computational and analytical effort on the most relevant threats—supporting efficient, high-value security assessments at scale.

7 Evaluation

We evaluate the ThreatCompute using two publicly available Kubernetes applications: *Kubernetes Goat* [7] and *Kubernetes Goof* [8]. Both applications are intentionally insecure and designed for security training and testing, making them ideal candidates for demonstrating the effectiveness of our automated threat modeling approach.

Both application were deployed locally using KIND (Kubernetes in Docker)[9]. We configured the clusters with the security scanners Trivy and Kubescape operators to extract vulnerability and misconfiguration data used throughout the analysis.

7.1 Threat Model Evaluation

For all threat modeling steps, we employed the open-source LLM `mistral-nemo` with a temperature of 0.3, following Mistral AI's recommendations[10]. We selected this model based on preparatory experiments in which it consistently produced relevant and detailed outputs. An additional advantage is its lack of moderation filters, which is crucial in cybersecurity contexts where accurate threat modeling requires unrestricted reasoning about adversarial behavior.

Qualitative Analysis. Evaluating automated threat modeling remains inherently challenging, as the process is typically performed by human experts and lacks a definitive ground truth [38]. Even for the same system, expert-generated threat models can differ significantly depending on decisions about asset granularity, technique categorization, and attacker assumptions. This variability complicates direct comparisons and highlights the need for data-driven, repeatable methodologies. ThreatCompute addresses this by offering a systematic, automated pipeline for generating threat models based on real system data and established threat taxonomies. Our framework aims to fill the gap in fully automated threat modeling for Kubernetes systems. However, the absence of comparable systems also means there is no comprehensive ground truth for direct validation. As a result, we focus our evaluation on transparency of the modeling process and plausibility of the output, using Kubernetes Goat documentation as a reference point.

The graph in Figure 2 illustrates the threat model generated for Kubernetes Goat, showing identified attack techniques between various asset classes. Core assets include *namespace*, *cluster*, *RootShell*, *Pod*, and several refined *Container* categories, such as *Application Container*, *Networking*, and *Storage*. The edge thickness encodes the number of techniques identified between asset pairs. In total, the threat model contains 98 edges and 449 techniques, with an average of 4.6 techniques per edge. Table 1 presents selected example techniques generated for Kubernetes Goat. These examples show that the identified techniques are context-specific and align with the known configuration and vulnerabilities of the analyzed system components, demonstrating both system-awareness and semantic relevance. Appendix A displays the threat model graph for Kubernetes Goof, which shows similar results.

[7] https://madhuakula.com/kubernetes-goat/
[8] https://github.com/snyk-labs/kubernetes-goof/
[9] https://kind.sigs.k8s.io/
[10] https://huggingface.co/mistralai/Mistral-Nemo-Base-2407

Comparison with Kubernetes Goat Ground Truth. To evaluate the quality and completeness of our generated threat model, we compared it to the scenarios provided by Kubernetes Goat. These scenarios are written in a tutorial format and mapped to techniques from Microsoft's Threat Matrix for Kubernetes, making them a practical benchmark for evaluating both technique discovery and relevance. To prepare the ground truth, we used ChatGPT-4 to summarize each attack scenario and generate concise descriptions of how each listed technique was applied. These outputs were then reviewed and validated by a security analyst to ensure accuracy and relevance. This hybrid approach enabled a consistent baseline for comparison while incorporating expert oversight to enhance the credibility of the evaluation. We assessed our model's performance on two dimensions:

- **Technique Discovery Rate:** The percentage of techniques correctly identified by our framework compared to those used in Kubernetes Goat scenarios (based on the Microsoft Threat Matrix for Kubernetes [19]).
- **Technique Description Quality:** A qualitative comparison of our generated technique descriptions with those derived from the official scenarios.

Our framework successfully identified 29 out of the 30 techniques used across the Kubernetes Goat scenarios, achieving a **93% discovery rate**. This demonstrates strong coverage of known adversarial behavior in Kubernetes environments.

To assess the quality of the technique descriptions, we applied the LLM-as-a-judge evaluation method, as proposed by Verga et al. [33]. Specifically, we used a Panel of LLM Evaluators (PoLL), with `mistral-nemo` and `llama3.3-70B` independently rating each technique description on a scale from 1 to 10, where 10 indicates perfect alignment with the expected behavior. The average rating across all evaluated descriptions was **6.62**. To contextualize this score, we conducted a small baseline evaluation using four ground truth descriptions. The original description evaluated against itself received an average score of **9.5**, a slightly rephrased version scored **7.13**, and a description of a completely different technique received a score of **3.0**. These results suggest that our model-generated descriptions are generally perceived as semantically aligned with the ground truth, though not perfectly identical. The baseline gap between rephrased (−2.75 points) and unrelated (−7.75 points) descriptions provides a reference scale for interpreting the 6.62 score.

While human evaluation would offer an additional layer of validation and remains an important direction for future work, prior studies have shown that panels of diverse LLMs can approximate human judgments with high consistency in text evaluation tasks [33]. Our use of multiple high-performing models helps reduce individual model bias and supports scalable, repeatable evaluation.

7.2 Attack Graph Evaluation

To evaluate the final stage of our framework, we generated attack graphs based on the threat models for the two applications: *Kubernetes Goat* and *Kubernetes Goof*. We focus on Kubernetes Goat here, while results for Goof—showing similar patterns—are provided in Appendix B.

7.2.1 Example Attack Graphs. The attack graph for Kubernetes Goat was generated assuming an attacker with *intermediate* skill

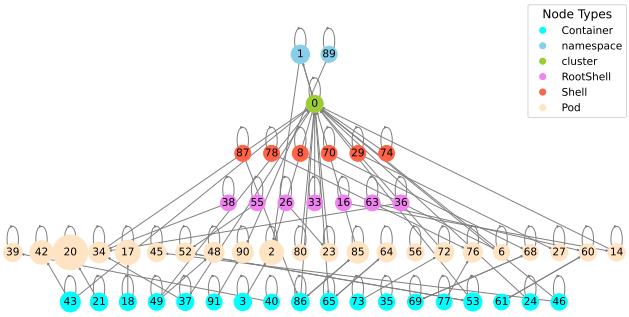

Figure 3: Attack graph for Kubernetes Goat based on 200 simulated walks through the threat model.

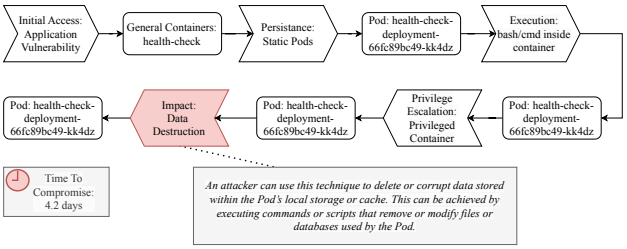

Figure 4: Shortest attack path leading to 'Data Destruction' in Kubernetes Goat.

level. We performed 200 simulated attack walks through the threat model (Figure 2), each limited to 15 steps. The number of walks was selected empirically, based on convergence in both the number of unique nodes/edges and the technique coverage, as shown in Figure 5 and Figure 7. Figure 3 shows the resulting attack graph, which aggregates the 122 walks (61%) that successfully reached an "Impact" tactic. These completed attack paths reveal a diverse set of outcomes: **39%** ended in *Data Destruction*, **32%** resulted in *Denial of Service*, and **29%** led to *Resource Hijacking*.

The graph contains a large number of nodes and edges, reflecting the system's complexity and potential attack surface. Node sizes indicate how frequently a component appears in successful attack paths. For example, Pod *health-check-deployment-66fc89bc49-kk4dz* (ID: 20) was targeted in 219 attack steps—a result of its low TTC of 1.06 days. This component also features prominently in the shortest-cost attack path, shown in Figure 4. This example path starts from a General Container (*health-check*) and concludes with the *Data Destruction* impact on the same Pod. The total TTC for the path is 4.2 days. These case-specific paths allow targeted analysis of system weaknesses and demonstrate how low-TTC nodes influence attack feasibility and path selection.

7.3 Attack Graph Analysis

To better understand the behavior of the attack graph generation process and the influence of TTC and attacker skill, we conducted a deeper analysis. Specifically, we examined (1) the convergence of the generated graph to the system model, (2) how attacker skill levels shape attack paths, and (3) the coverage of the ATT&CK Matrix as the number of simulations increases. We generated attack graphs for both *Kubernetes Goat* and *Kubernetes Goof* across

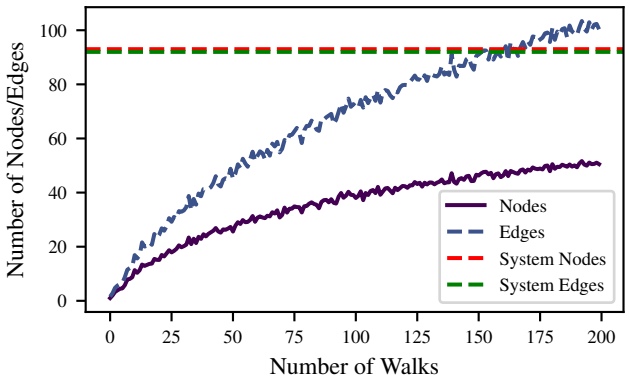

Figure 5: Number of nodes and edges in the attack graph as the number of walks increases.

four different attacker skill levels. For clarity and due to space constraints, we present here the results for an *intermediate* attacker in Kubernetes Goat. Each graph was generated by performing 1 to 200 walks through the threat model, and various statistics were collected across this range. Key results are shown in Figures 5, 6, and 7.

Convergence Toward the System Model. Figure 5 shows how the number of nodes and edges in the attack graph evolves as more walks are performed. The number of nodes steadily approaches that of the full system model, indicating that the attack graph becomes increasingly complete over time. In contrast, the number of edges grows more rapidly and eventually exceeds the edge count of the system model. This is due to attackers bypassing intermediate layers and connecting components in ways that reflect realistic attack behavior but are not explicitly defined in the system architecture. For instance, an attacker might establish a direct connection from a container to a cluster, creating an edge in the attack graph that is absent in the system model. In the system model, such a connection would typically traverse multiple levels: from container to pod, pod to namespace, and namespace to cluster. Thus, while the nodes in the attack graph form a subset of the nodes in the system model, the sets of edges exhibit partial overlap due to such attack-specific paths.

Influence of Attacker Skill Level. Attacker skill levels significantly influence how attack paths are formed. For lower-skilled attackers, the number of vulnerabilities and misconfigurations in a system component has a greater impact on its computed TTC. This leads to more pronounced differences between TTC values across components, resulting in a more concentrated attack pattern. In contrast, for higher-skilled attackers, the TTC values are more evenly distributed across components, which encourages broader exploration of the system. As shown in Figure 6, beginner attackers tend to repeatedly select a small number of highly vulnerable components as starting points, whereas intermediate and expert attackers distribute their choices more evenly. This behavior also influences the growth of the attack graph: lower-skilled attackers generate fewer nodes and edges due to their narrower focus. For intermediate attackers, the most frequently selected starting point was the *health-check* container, which had the lowest TTC in the system.

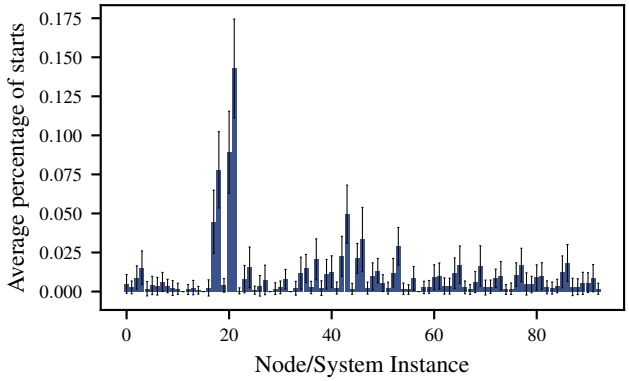

Figure 6: Average number of times each system component is selected as an initial access point.

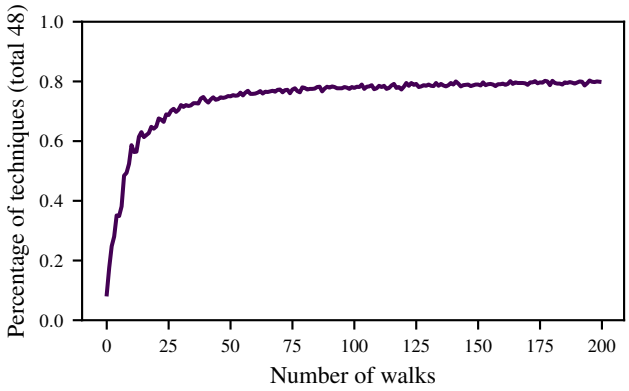

Figure 7: Coverage of ATT&CK techniques for increasing number of walks through the threat model.

Threat Matrix Technique Coverage. Figure 7 tracks the growth of technique coverage of the Threat Matrix for Kubernetes as the number of attack walks increases. As expected, more walks lead to broader coverage of the Threat Matrix for Kubernetes. However, some tactics such as *Collection* and *Exfiltration* remain underrepresented. This is primarily due to the limitations of the system model, which lacks explicit representations of sensitive data or data movement channels. Incorporating such domain knowledge could improve coverage in these areas, particularly for data-centric attack phases.

8 Discussion and Future Work

ThreatCompute demonstrates the feasibility of structured, LLM-driven threat modeling and attack graph generation for complex cloud-native systems. By guiding LLMs through a modular pipeline and grounding the threat modeling process in the Microsoft Threat Matrix for Kubernetes, we mitigate open-ended hallucinations and ensure relevance to the Kubernetes security domain.

Ensuring the completeness and correctness of LLM-generated threat models remains an open challenge. While we constrain model outputs using the Microsoft Threat Matrix for Kubernetes [19] to ensure consistency and alignment with known adversarial behaviors, this scope inherently limits the coverage of threats to those already captured in the matrix. As a result, novel or less-documented techniques may be missed. Furthermore, the reasoning capacity of general-purpose LLMs may fall short in identifying attacks that rely on implicit architectural flaws or emergent behaviors not visible in the static system model. Future work could explore fine-tuned LLMs trained on domain-specific CTI, or employ retrieval-augmented generation using structured knowledge bases or security reports to enrich threat hypotheses.

Our modular design follows the *divide and conquer* principle, enabling more precise prompts and improved reasoning within each step. This structure also facilitates extensibility: components of the pipeline can be independently upgraded or replaced with more specialized models or tools (e.g., for vulnerability scanning or risk quantification). Incorporating structured threat intelligence via knowledge graphs could further improve the accuracy and explainability of generated models.

Beyond static risk assessment, generated attack graphs can be leveraged for automated attack emulation and defense simulation [27]. Integrating ThreatCompute with red/blue team tools or cyber range platforms could enable dynamic scenario generation and support adversarial training. Moreover, modeling attacker incentives or costs may lead to more realistic threat prioritization in practice.

9 Conclusion

We introduced ThreatCompute, a fully automated threat modeling framework for Kubernetes-based systems. Traditional threat modeling is manual and time-consuming making it unsuited for the complexity and scale of cloud-native environments. ThreatCompute addresses this by using large language models to generate system-specific threat hypotheses based on real configuration and vulnerability data. Our approach connects high-level threat frameworks like MITRE ATT&CK with low-level system data through modular prompting and a structured graph representation of the system. This enables the LLM to generate realistic tactic-technique mappings and produce attack graphs that are both interpretable and relevant. This provides a significant advantage over traditional tools, which often struggle to contextualize vulnerabilities or suggest meaningful attack paths without extensive manual input. We further quantify risk using an adapted Time-to-Compromise model tailored for Kubernetes environments. This enables the prioritization of attack paths based on risk, helping security teams focus on the most critical threats. Unlike existing tools, ThreatCompute requires minimal manual input while maintaining coverage and contextual accuracy. Ultimately, this framework demonstrates that LLMs can streamline attack graph generation, reducing manual effort while maintaining high relevance. Future work can refine model selection, enhance data inputs, and expand applications to attack emulation and adversarial simulations.

Acknowledgments

This work was supported by the EU Horizon Europe programme, project SLICES-PP (10107977), and by the Bavarian Ministry of Economic Affairs, Regional Development and Energy, project 6G Future Lab Bavaria.

References

[1] Ehsan Aghaei, Xi Niu, Waseem Shadid, and Ehab Al-Shaer. 2022. SecureBERT: A Domain-Specific Language Model for Cybersecurity. arXiv:2204.02685 [cs.CL]

[2] Mohamed Ahmed, Sakshyam Panda, Christos Xenakis, and Emmanouil Panaousis. 2022. MITRE ATT&CK-driven cyber risk assessment. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 1–10.

[3] Md Tanvirul Alam, Dipkamal Bhusal, Youngja Park, and Nidhi Rastogi. 2023. Looking Beyond IoCs: Automatically Extracting Attack Patterns from External CTI. arXiv:2211.01753 [cs.CR]

[4] The Kubernetes Authors. 2023. Kubernetes Documentation: Concepts - Overview. https://kubernetes.io/docs/concepts/overview/. Accessed: 2024-08-25.

[5] Casey Charrier and Robert Weiner. 2024. *How Low Can You Go? An Analysis of 2023 Time-to-Exploit Trends*. Technical Report. Mandiant. https://cloud.google.com/blog/topics/threat-intelligence/time-to-exploit-trends-2023

[6] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/usenixsecurity24/presentation/deng

[7] Isra Elsharef, Zhen Zeng, and Zhongshu Gu. 2024. Facilitating Threat Modeling by Leveraging Large Language Models. (2024).

[8] Viktor Engström, Pontus Johnson, Robert Lagerström, Erik Ringdahl, and Max Wällstedt. 2023. Automated Security Assessments of Amazon Web Services Environments. *ACM Trans. Priv. Secur.* 26, 2, Article 20 (March 2023), 31 pages. doi:10.1145/3570903

[9] Viktor Engström and Robert Lagerström. 2022. Two decades of cyberattack simulations: A systematic literature review. *Computers & Security* 116 (2022), 102681.

[10] Scott Freitas, Jovan Kalajdjieski, Amir Gharib, and Robert McCann. 2025. AI-Driven Guided Response for Security Operation Centers with Microsoft Copilot for Security. In *Companion Proceedings of the ACM on Web Conference 2025* (Sydney NSW, Australia) *(WWW '25)*. Association for Computing Machinery, New York, NY, USA, 191–200. doi:10.1145/3701716.3715209

[11] Simon Hacks, Sotirios Katsikeas, Engla Ling, Robert Lagerström, and Mathias Ekstedt. 2020. powerLang: a probabilistic attack simulation language for the power domain. *Energy Informatics* 3 (2020), 1–17.

[12] Pontus Johnson, Robert Lagerström, and Mathias Ekstedt. 2018. A meta language for threat modeling and attack simulations. In *Proceedings of the 13th international conference on availability, reliability and security*. 1–8.

[13] Sotirios Katsikeas, Simon Hacks, Pontus Johnson, Mathias Ekstedt, Robert Lagerström, Joar Jacobsson, Max Wällstedt, and Per Eliasson. 2020. An Attack Simulation Language for the IT Domain. In *Graphical Models for Security*, Harley Eades III and Olga Gadyatskaya (Eds.). Springer International Publishing, Cham, 67–86.

[14] Sotirios Katsikeas, Pontus Johnson, Simon Hacks, and Robert Lagerström. 2019. Probabilistic Modeling and Simulation of Vehicular Cyber Attacks: An Application of the Meta Attack Language.. In *ICISSP*. 175–182.

[15] Nick Kirtley. 2022. PASTA Threat Modeling. https://threat-modeling.com/pasta-threat-modeling/. Accessed: 2024-08-25.

[16] Alyzia-Maria Konsta, Alberto Lluch Lafuente, Beatrice Spiga, and Nicola Dragoni. 2024. Survey: Automatic generation of attack trees and attack graphs. *Computers & Security* 137 (2024), 103602. doi:10.1016/j.cose.2023.103602

[17] Khang Mai, Jongmin Lee, Razvan Beuran, Ryosuke Hotchi, Sian En Ooi, Takayuki Kuroda, and Yasuo Tan. 2025. RAF-AG: Report analysis framework for attack path generation. *Computers & Security* 148 (2025), 104125. doi:10.1016/j.cose.2024.104125

[18] Miles A McQueen, Wayne F Boyer, Mark A Flynn, and George A Beitel. 2006. Time-to-compromise model for cyber risk reduction estimation. In *Quality of Protection: Security Measurements and Metrics*. Springer, 49–64.

[19] Microsoft. 2021. Threat Matrix for Kubernetes. https://microsoft.github.io/Threat-Matrix-for-Kubernetes/. Accessed: 2024-08-21.

[20] Microsoft Corporation. 2009. The STRIDE Threat Model. https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20). Accessed: 2024-08-25.

[21] Dmitry Namiot and Manfred sneps sneppe. 2014. On Micro-services Architecture. *Interenational Journal of Open Information Technologies* 2 (09 2014), 24–27.

[22] Ana Maria Pirca and Harjinder Singh Lallie. 2023. An empirical evaluation of the effectiveness of attack graphs and MITRE ATT&CK matrices in aiding cyber attack perception amongst decision-makers. *Computers & Security* 130 (2023), 103254. doi:10.1016/j.cose.2023.103254

[23] Renascence Tarafder Prapty, Ashish Kundu, and Arun Iyengar. 2024. Using Retriever Augmented Large Language Models for Attack Graph Generation. arXiv:2408.05855 [cs.CR] https://arxiv.org/abs/2408.05855

[24] Engla Rencelj Ling and Mathias Ekstedt. 2023. Estimating time-to-compromise for industrial control system attack techniques through vulnerability data. *SN Computer Science* 4, 3 (2023), 318.

[25] Dipayan Saha, Hasan Al Shaikh, Shams Tarek, and Farimah Farahmandi. 2025. ThreatLens: LLM-guided Threat Modeling and Test Plan Generation for Hardware Security Verification. Cryptology ePrint Archive, Paper 2025/561. https://eprint.iacr.org/2025/561

[26] Erik Schluntz and Barry Zhang. 2024. *Building Effective AI Agents*. https://www.anthropic.com/engineering/building-effective-agents Anthropic Engineering Blog, accessed June 24, 2025.

[27] Ömer Sen, Bozhidar Ivanov, Martin Henze, and Andreas Ulbig. 2023. Investigation of Multi-stage Attack and Defense Simulation for Data Synthesis. In *2023 International Conference on Smart Energy Systems and Technologies (SEST)*. IEEE, 1–6.

[28] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. 2020. Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. *2020 IEEE Secure Development (SecDev)* (2020), 58–64.

[29] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. 2002. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. 273–284. doi:10.1109/SECPRI.2002.1004377

[30] Maddie Stone and James Sadowski. 2024. A Year in Review of Zero-Days Exploited In-the-Wild in 2023. https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/Year_in_Review_of_ZeroDays.pdf Analyzes zero-day vulnerabilities actively exploited in 2023 and offers recommendations for ecosystem security..

[31] Blake E Strom, Andy Applebaum, Doug P Miller, Kathryn C Nickels, Adam G Pennington, and Cody B Thomas. 2018. Mitre att&ck: Design and philosophy. In *Technical report*. The MITRE Corporation.

[32] David Tayouri, Nick Baum, Asaf Shabtai, and Rami Puzis. 2022. A Survey of MulVAL Extensions and Their Attack Scenarios Coverage. arXiv:2208.05750 [cs.CR] https://arxiv.org/abs/2208.05750

[33] Pat Verga, Sebastian Hofstatter, Sophia Althammer, Yixuan Su, Aleksandra Piktus, Arkady Arkhangorodsky, Minjie Xu, Naomi White, and Patrick Lewis. 2024. Replacing Judges with Juries: Evaluating LLM Generations with a Panel of Diverse Models. arXiv:2404.18796 [cs.CL] https://arxiv.org/abs/2404.18796

[34] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. 2008. An attack graph-based probabilistic security metric. In *Data and Applications Security XXII: 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security London, UK, July 13-16, 2008 Proceedings 22*. Springer, 283–296.

[35] Wojciech Wideł, Simon Hacks, Mathias Ekstedt, Pontus Johnson, and Robert Lagerström. 2023. The meta attack language - a formal description. *Computers & Security* 130 (2023), 103284. doi:10.1016/j.cose.2023.103284

[36] Tingmin Wu, Shuiqiao Yang, Shigang Liu, David Nguyen, Seung Jang, and Alsharif Abuadbba. 2025. ThreatModeling-LLM: Automating Threat Modeling using Large Language Models for Banking System. arXiv:2411.17058 [cs.CR] https://arxiv.org/abs/2411.17058

[37] Wenjun Xiong, Simon Hacks, and Robert Lagerström. 2021. A method for assigning probability distributions in attack simulation languages. *Complex Systems Informatics and Modeling Quarterly* 26 (2021), 55–77.

[38] Wenjun Xiong and Robert Lagerström. 2019. Threat modeling–A systematic literature review. *Computers & security* 84 (2019), 53–69.

[39] Kengo Zenitani. 2023. Attack graph analysis: An explanatory guide. *Computers & Security* 126 (2023), 103081. doi:10.1016/j.cose.2022.103081

[40] Andrej Zieger, Felix Freiling, and Klaus-Peter Kossakowski. 2018. The β-Time-to-Compromise Metric for Practical Cyber Security Risk Estimation. In *2018 11th International Conference on IT Security Incident Management & IT Forensics (IMF)*. 115–133. doi:10.1109/IMF.2018.00017

A Kubernetes Goof Threat Model

Figure 8 shows the threat model graph for Kubernetes Goof.

B Kubernetes Goof Attack Graph

The attack graph for Kubernetes Goof is shown in Figure 9 and Table 4 shows the shortest attack path for the 'Resource Hijacking' technique in Kubernetes Goof. For Kubernetes Goat, 158 of the 200 walks ended with a successful attack path (i.e., reached the 'Impact' tactic). Among the successful walks, 3.8% ended with the 'Data Destruction' technique, 75.3% with 'Denial of Service', and 20.9% with 'Resource Hijacking'. With a maximum walk length of 15, the average number of steps per path was 6.1, and the average number of involved system instances per path was 1.78, with 80 paths involving only one instance. With 283 traversals, Pod *webadmin-58d6fb9cbd-gzxfp* (id: 2) was traversed the most often, which can be explained by its low TTC value of 1.06 days.

Table 4: Shortest attack path for Kubernetes Goof for 'Impact' technique 'Resource Hijacking'.

Source Node: Source Instance	Target Node: Target Instance	Technique	Description
Application Container: webadmin	Application Container: webadmin	Using cloud credentials	Attackers can exploit misconfigured or exposed cloud credentials, such as AWS access keys or Google Cloud service accounts, to gain initial access to the cluster. This can be achieved by scanning for exposed credentials, exploiting known vulnerabilities in cloud services, or using social engineering techniques to obtain valid credentials.
Application Container: webadmin	Application Container: webadmin	Privileged container	An attacker can exploit the fact that some containers in the application are privileged to escalate privileges. By gaining access to these containers, the attacker can execute commands with elevated privileges, potentially leading to unauthorized access to sensitive data or control of the application.
Application Container: webadmin	Application Container: webadmin	Application exploit (RCE)	An attacker can exploit a vulnerability in the application running within the container to remotely execute arbitrary commands, potentially leading to privilege escalation or data exfiltration.
Application Container: webadmin	Application Container: webadmin	Container service account	An attacker could create a service account with excessive privileges and use it to run a malicious container. This container could maintain persistence by modifying system resources or creating new containers.
Application Container: webadmin	Application Container: webadmin	Resource Hijacking	Attackers can exploit misconfigurations, such as running containers as root or not setting resource limits, to consume all available resources on the 'Application Container'. This could be done by running resource-intensive processes or by exploiting a vulnerability in one of the affected packages.

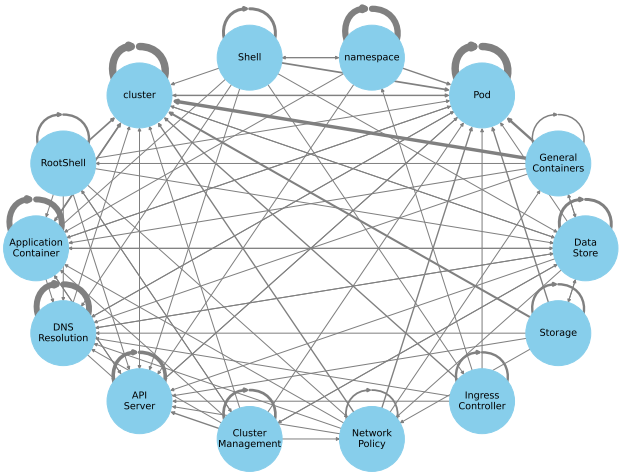

Figure 8: Threat model graph for Kubernetes Goof.

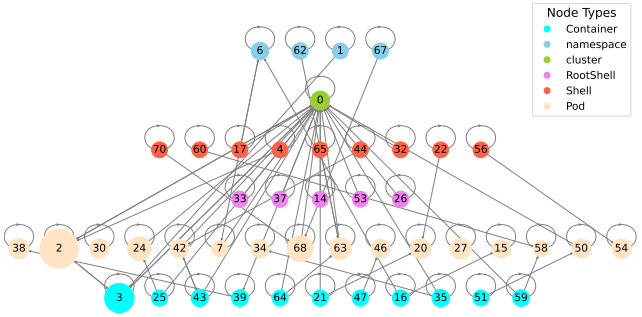

Figure 9: Attack graph for Kubernetes Goof based on 200 walks through the threat model.