TU WIEN Informatics

Can Coding Agents Build an IDS Testbed? A Benchmark Design and Empirical Evaluation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering

eingereicht von

Laurenz Kirchmayr, BSc.
Matrikelnummer 11940476

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc.
Mitwirkung: Dr. Dipl.-Ing. Max Landauer, BSc.
Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik

Wien, 30. Juni 2026

Laurenz Kirchmayr | Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Informatics

Can Coding Agents Build an IDS Testbed? A Benchmark Design and Empirical Evaluation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering

by

Laurenz Kirchmayr, BSc.
Registration Number 11940476

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc.
Assistance: Dr. Dipl.-Ing. Max Landauer, BSc.
Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Dr.rer.soc.oec. Florian Skopik

Vienna, June 30, 2026

Laurenz Kirchmayr | Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Laurenz Kirchmayr, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. Juni 2026

Laurenz Kirchmayr

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich bedanke mich herzlich bei Prof. Jürgen Cito, meinem Betreuer an der TU Wien. Danke für Ihre Zeit, Ihr Hineindenken in das Thema und Ihre richtungsweisenden Impulse in Zeiten der Unsicherheit.

Ein großer Dank geht an meine Betreuer vom AIT Austrian Institute of Technology, Max Landauer, Florian Skopik und Markus Wurzenberger. Danke für die Möglichkeit, mit euch in ein so spannendes Thema einzutauchen. Danke, dass ihr Woche um Woche mit mir an der Arbeit gefeilt habt. Ich hatte Spaß und habe viel von euch gelernt.

Ebenso vielen Dank an die Kolleginnen und Kollegen am AIT. Danke für eure Freundschaft, Hilfe bei Fragen, und euer Engagement bei der Validierung der Requirements.

Und danke an Mutti und Vati. Danke fürs Fördern meiner Neugierde und für das Fundament, das ihr mir gegeben habt. Ausführlicher Dank folgt persönlich.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I sincerely thank Prof. Jürgen Cito, my advisor at TU Wien. Thank you for your time, for your thoughtful engagement with the topic, and for your guidance in times of uncertainty.

Sincere thanks go to my advisors at the AIT Austrian Institute of Technology, Max Landauer, Florian Skopik, and Markus Wurzenberger. Thank you for the opportunity to work on such an exciting topic with you. Thank you for taking the time to refine this work with me week after week. I had fun and learned a lot from you.

I also thank my colleagues at the AIT. Thank you for your friendship, your help with my questions, and your effort in validating the requirements.

And thank you to Mum and Dad. Thank you for nurturing my curiosity and for the foundation you give me. A fuller thanks will follow in person.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Forschung an Intrusion-Detection-Systemen (IDS) benötigt kontrollierte Testumgebungen, sogenannte Testbeds. Sie simulieren unsichere Systeme, um Datensätze mit Angriffsspuren zu erzeugen. Das Erstellen und Anpassen solcher Testbeds ist allerdings aufwendig. Coding-Agenten versprechen, einen Teil dieser Arbeit zu übernehmen. Im allgemeinen Programmier-Benchmark SWE-bench Verified lösen sie bereits mehr als 75% der Aufgaben. Ob diese Fähigkeiten auf IDS-Testbeds übertragbar sind, lässt sich nicht mit bestehenden Benchmarks prüfen. Existierende IaC-Benchmarks erwarten sichere Konfigurationen, während Testbeds genau das Gegenteil benötigen. Existierende Penetration Testing Benchmarks evaluieren das interaktive Erforschen und Ausnutzen von Schwachstellen, während Testbeds reproduzierbare Angriffsskripte benötigen.

Deshalb entwickeln wir einen eigenen Benchmark, abgeleitet aus einem realen Testbed. Er deckt die Hauptaufgaben bei der Anpassung von IDS-Testbeds ab: das Konfigurieren unsicherer Infrastruktur und das Erstellen von Angriffsskripten. Mit diesem Benchmark testen wir zwei Coding-Agenten in 190 Durchläufen. Claude Code (Opus 4.6) erzeugte in 43% der Durchläufe eine korrekte Lösung, Mistral Vibe CLI (Devstral 2) in 14%. Das Konfigurieren von Services fiel beiden Agenten leichter als das Erstellen von Angriffsskripten. Fehler passierten meist punktuell und nicht aufgrund einer grundsätzlich falschen Herangehensweise. Coding-Agenten sind somit zwar nützliche Assistenten für die Entwicklung von IDS-Testbeds, erfordern jedoch weiterhin menschliche Kontrolle.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Intrusion detection system (IDS) research depends on controlled environments, so-called testbeds. These simulate vulnerable systems to produce labeled datasets containing traces of attacks. Such testbeds are resource-intensive to build and adapt. Coding agents could take on some of this work. They already resolve more than 75% of tasks on the general-purpose SWE-bench Verified benchmark. However, no existing benchmark can test whether this transfers to IDS testbed work. IaC benchmarks penalize insecure configurations, while testbeds need them. Offensive security benchmarks focus on interactive exploration and exploitation, while testbeds need deterministic, reproducible attack scripts.

We therefore reverse-engineer an expert-validated 19-task benchmark from a real-world testbed. It covers the main IDS testbed adaptation tasks: configuring vulnerable infrastructure and scripting reproducible attacks. We test two coding agents over 190 runs. Claude Code (Opus 4.6) generated a correct solution in 43% of runs, and Mistral Vibe CLI (Devstral 2) in 14%. Both agents did better at configuring services than at scripting attacks. Most failures are caused by localized mistakes rather than a fundamentally wrong approach. Coding agents are therefore useful assistants for IDS testbed development, but still require human oversight.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Questions . . . 2
1.3 Contributions . . . 2

2 Background **5**
2.1 IDS Testbeds and Infrastructure-as-Code . . . 5
2.2 LLM Coding Agents . . . 7
2.3 Benchmarks for IaC and Offensive Security . . . 9

3 Methodology **11**
3.1 Testbed Selection . . . 11
3.2 Anatomy of AttackBed . . . 12
3.3 Task Taxonomy . . . 14
3.4 Benchmark Construction . . . 15
3.5 Evaluation Strategy . . . 21
3.6 Agent Selection and Configuration . . . 23

4 Implementation **25**
4.1 Code Generation Pipeline . . . 26
4.2 Evaluation Pipeline . . . 27

5 Evaluation **31**
5.1 Experimental Setup . . . 31
5.2 Main Results . . . 32
5.3 Comparison to the Reference . . . 38
5.4 Discussion . . . 44

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6 Limitations and Future Work **47**
6.1 Threats to Validity 47
6.2 Future Work 49

7 Conclusion **51**

Overview of Generative AI Tools Used **53**

List of Figures **55**

List of Tables **57**

List of Algorithms **59**

Bibliography **61**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

Intrusion detection system (IDS) research depends on controlled simulation environments called testbeds [LSF+22, UHH+21]. Testbeds mirror production networks in terms of infrastructure, deployed services, and simulated traffic from legitimate and malicious users. Crucially, some of the services are intentionally deployed with vulnerabilities to allow for exploitation. These vulnerabilities are exploited as part of controlled experiments. The resulting monitoring data, such as network traffic and system logs, is then used to train and evaluate intrusion detectors. Without ongoing research and development of testbeds, IDS evaluation relies on stale datasets that do not reflect current infrastructure or attacker techniques [SSTG12].

Building a testbed requires expertise in system administration, network engineering, and offensive security. This makes testbed development resource-intensive [SSTG12, LSW+20]. Infrastructure-as-Code (IaC) tools such as Terraform, Ansible, or Crossplane have made testbeds more reusable. Still, IaC only shifts the effort from manual setup to scripting [LSW+20].

Recent progress in large language models (LLMs) has produced coding agents. These LLM-powered systems take natural-language prompts as input and then iteratively plan, write, and refine code. On SWE-bench Verified [JYW+23], a general-purpose programming benchmark, agents now resolve over 75% of tasks[1]. We hypothesize that coding agents could reduce the effort required for IDS testbed development. This thesis tests that hypothesis by constructing a domain-specific benchmark and evaluating how reliably state-of-the-art coding agents produce functionally correct testbed components.

[1] https://www.swebench.com/ (accessed: June 2026)

1.1 Problem Statement

IDS testbeds are rarely built from scratch. Once its architecture has been established, researchers adapt the testbed, e.g., by substituting attack tools, swapping services, or introducing new vulnerabilities. This still involves time-consuming and non-trivial work [LSW+20]. Adaptation effort concentrates on two task categories (see Section 3.3): configuring services or infrastructure to expose specific vulnerabilities, and writing attack sequences that reproducibly trigger them. No existing benchmark evaluates the tasks that IDS testbed adaptation requires.

General IaC benchmarks such as [ZPZ+25] include security compliance as an evaluation dimension. IDS testbeds require the opposite: infrastructure must contain a specific vulnerability. This cannot be evaluated via a benchmark that penalizes insecure configurations.

Pentesting and CTF benchmarks such as [ZPD+25], [ZKB+25] and [SLA+25] assess interactive exploitation, where agents autonomously exploit vulnerabilities in a live environment. Testbeds do not require autonomous discovery, but rather deterministic scripts that reproduce an exploit path across evaluation runs.

Therefore, no existing benchmark covers the combination of tasks that characterize IDS testbeds. Without such a benchmark, the question of whether coding agents can assist with IDS testbed development cannot be answered empirically. This thesis constructs such a benchmark and uses it to evaluate state-of-the-art coding agents.

1.2 Research Questions

The main question of this thesis is: "How well can coding agents assist in IDS testbed development?" This question is broken down into the following concrete research questions:

RQ1 What are the critical aspects that need to be simulated in an IDS testbed, and what tasks are commonly associated with them?

RQ2 To what extent can state-of-the-art coding agents generate functionally correct IaC for IDS testbeds?

RQ3 How do IDS testbeds generated by state-of-the-art coding agents compare to manually designed testbeds?

1.3 Contributions

This thesis makes the following contributions:

1. A method for extracting benchmark tasks from existing IDS testbeds.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

2. A benchmark dataset built from AIT AttackBed and reviewed by domain experts.
3. An open-source pipeline for generating IDS testbed components with coding agents and evaluating them.
4. Empirical baselines for two coding agents: Claude Code and Mistral Vibe CLI.
5. A comparative analysis of agent-generated and manually designed IDS testbed components.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

This chapter introduces the research areas this thesis builds on. Section 2.1 covers IDS testbeds and the Infrastructure-as-Code used to build them. Section 2.2 gives background on LLM coding agents. Section 2.3 reviews existing benchmarks for IaC and offensive security.

2.1 IDS Testbeds and Infrastructure-as-Code

An IDS testbed is a simulation environment designed to produce labeled observability data, often logs. Unlike in production environments, researchers have control over every component: which services are deployed, which vulnerabilities are exposed, how users behave, and how attacks are executed. This control enables pinpointing of normal and malicious log events and to create labeled datasets. These datasets are then used in the evaluation of IDS [LSF+22].

Early IDS testbeds were built physically, which could only be done by well-funded labs. Few datasets were published, and the ones that were dominated research. Two examples are KDD Cup 99 and NSL-KDD, which remained reference datasets for nearly two decades, even as infrastructure, applications, and attack methods changed substantially. This limitation motivated a shift toward on-demand testbeds, which were enabled by virtualization technologies. Researchers could now use cloud resources to build their testbeds. IaC tools sit on top of this, expressing the full testbed environment in a version-controlled, reproducible specification. Deployment of an IDS testbed now takes minutes instead of weeks of manual administration [LSF+22, SLG+18].

IaC tools in modern testbeds fall into two conventional categories, which we extend with a third IDS-testbed-specific category. The first is provisioning. Tools such as Terraform or Pulumi declare the infrastructure to create: virtual machines, networks, firewall rules, etc. The second is configuration management. Tools like Ansible or Puppet

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

specify the configuration of machines: installed packages, running services, filesystem state, etc. Beyond these two, we extend the IaC framing to a third category: attack orchestration. It applies the philosophy of IaC to attacker behavior. Tools like MITRE CALDERA [AMS+16] encode ATT&CK techniques as composable actions and automate their execution. This makes attacks reproducible alongside the rest of the testbed [LSF+22].

We use these categories to compare how real testbeds are built. We do so in the order they appeared, tracing the evolution of the field. This shows which capabilities a strong testbed needs. Table 2.1 summarizes this comparison across eight dimensions.

Algorizmi is the first open-source testbed built explicitly for IDS dataset generation. Its infrastructure is managed in Java: cloud-API calls to provision resources, and bash commands to configure them over a SSH connection. Attack orchestration via Metasploit was planned, but integration did not succeed fully, leaving attacks to be run manually. Benign user traffic comes from four Expect scripts that perform a single action, e.g., running a command or downloading a file. The network is limited to four VMs [Ali10][1].

KYPO is a cyber range for research and exercises. It uses custom provisioners that consume YAML based infrastructure definitions. Configuration of hosts happens via Ansible. Humans then play the roles of attackers and defenders, producing traffic that is captured by integrated monitoring. It lacks IDS-testbed-specific features like attack catalogs, automated dataset labeling, and reproducibility across runs [VOČ+17][2].

CIDDS emulates a small-business network consisting of mail, file, web, and backup hosts. VMs are provisioned manually on OpenStack and configured via Bash scripts. Simulated users and attackers interact with these hosts via randomly parametrized Python scripts, improving on Algorizmi's static templates. Timestamps and IPs of user and attacker actions are recorded to enable precise labeling of NetFlow traffic data. Attacks consist of randomly chained single actions including IP scanning, denial of service, and brute forcing [RWG+17][3].

SOCBED increases reproducibility over previous testbeds. Infrastructure is fully expressed in IaC and runs on VirtualBox VMs, thereby running on commodity laptops instead of clouds. Benign user activity is modeled via finite-state machines and performs tasks such as web browsing, email, and file activity. Attacks use real offensive tooling (such as Metasploit or sqlmap) and span across multiple MITRE ATT&CK techniques such as reconnaissance, initial access, and persistence. Attacks are packaged in 12 Python modules that can be chained in fixed and random ways. The default network emulates a small-business network of six hosts [UHH+21][4].

[1] https://sourceforge.net/projects/algorizmi/

[2] https://gitlab.ics.muni.cz/muni-kypo-crp, https://web.archive.org/web/20201124184122/https://docs.crp.kypo.muni.cz/

[3] https://github.com/markusring/CIDDS

[4] https://github.com/fkie-cad/socbed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

AttackBed is the testbed underlying the CAM-LDS dataset [LHB+26c]. Terraform is used to provision resources on OpenStack, and Ansible is used for configuring hosts. Attack orchestration is handled by AttackMate, a tool that chains and automates interactive tooling (such as vim or dmesg). This produces more realistic, human-like attack data [LHB+26a][5]. AttackBed ships with seven base attack scenarios spanning the full kill chain. The authors derive 34 distinct variants from these scenarios by swapping individual attack steps with functionally equivalent variants. Each scenario ships with a distinct network topology [LHB+26b][6]. We select AttackBed as the source of our benchmark because of its modular components, modern IaC stack, and complete automation. Section 3.1 gives the full reasoning.

Table 2.1 compares the five testbeds along the following dimensions:

- **Primary purpose**: The testbed's research goal.
- **IaC coverage**: The IaC tools used to automate testbed setup. Separated into the categories: provisioning, configuration, and attack orchestration.
- **User simulation**: The mechanism for simulating benign user behavior.
- **Attack diversity**: Number of distinct attacks scripted.
- **Max. Attack chain length**: Number of unique, chained steps in the longest attack.
- **Network diversity**: Number of distinct network topologies available.
- **Max. Network size**: Number of distinct, non-user hosts in the largest network.
- **Open source**: Whether the source code is publicly available.

2.2 LLM Coding Agents

Large language models (LLMs) take an instruction as input and produce a text response. They can answer questions, summarize documents, or implement functions [OWJ+22]. This makes LLMs useful for software development tasks. However, a text-generating model alone has no way to interact with its environment. It forces the developer to become its "hands", e.g., by pasting file content, running commands, or pasting command outputs back into the chat.

Tool use removes the developer from this loop. To act, the model generates structured output naming a function and its arguments. A harness parses the tool call, executes it, and feeds the outputs back to the model [SDYD+23]. This enables the use of any

[5] https://github.com/ait-testbed/attackmate
[6] https://github.com/ait-testbed/attackbed

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 2.1: Comparison of five representative IDS testbeds.

Dimension	**Algorizmi** [Ali10]	**KYPO** [VOČ+17]	**CIDDS** [RWG+17]	**SOCBED** [UHH+21]	**AttackBed** [LHB+26b]
Primary purpose	IDS dataset generation	Cyber range	IDS dataset generation	IDS dataset generation	IDS dataset generation
IaC: Resource provisioning	Cloud API calls via Java	Cloud API calls via YAML runner	Manual	VBoxManage calls via Python	Terraform
IaC: Service configuration	SSH commands via Java	Ansible	Bash scripts	Ansible	Ansible
IaC: Attack orchestration	Manual Metasploit operation	Manual	Python scripts	Python scripts	AttackMate playbooks
User simulation	Expect scripts (single action)	Manual	Python scripts (randomly chained single actions)	Modeled via state machine	-
Attack diversity	-	-	3 single step attacks	12 attack modules	34 end-to-end attacks
Max. attack chain length	-	-	2 steps	9 steps	16 steps
Network diversity	1 topology	5 topologies	1 topology	1 topology	7 topologies
Max. network size	4 hosts	3 hosts	4 hosts	7 hosts	8 hosts
Open source	yes	yes	yes	yes	yes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

text-based tool, such as reading a file, running a shell command, or searching the web. A coding agent wraps these tool calls in a loop. From a prompt, the agent plans a step, calls a tool, and observes the result. This sequence is repeated until the task is completed [YZY+22].

Therefore a coding agent consists of two distinct parts, the LLM and the harness. The model receives the prompt, reasons about the tasks, selects tool calls, and interprets their outputs. The harness is the model's runtime environment. It executes the loop, manages the model's context window, and exposes the tools. Model and harness are independent in principle. One model can run under different harnesses, and one harness can be used with different models. However, in practice a model's performance depends heavily on its harness [YJW+24]. For this reason, labs provide custom harnesses which are tailored to the capabilities and quirks of their models.

2.3 Benchmarks for IaC and Offensive Security

Benchmarks are the main tool for evaluating the capability of a coding agent. Most benchmarks are execution-based, so generated code is executed and scored with a test suite. The alternative is to judge the generated code directly, i.e., through a human review or an LLM judge. These alternatives are good at evaluating qualitative aspects but weaker for correctness. Execution based benchmarks score correctness automatically and reproducibly. For this reason it has become the standard for agent evaluation.

SWE-bench is the canonical example for an execution-based benchmark. It tasks an agent with a real GitHub issue and scores the generated code against the project's existing test suite [JYW+23]. Frontier coding agents now resolve more than 75% of SWE-bench Verified tasks[7]. While these are promising results, SWE-bench focuses on general purpose coding in Python. Strong performance here does not carry over directly to specialized domains. More focused benchmarks have emerged as a result, two of which are relevant for this thesis: IaC generation and offensive security.

IaC-Eval was the first benchmark for IaC generation. It contains 458 tasks written by human IaC experts. Generated code is scored statically against handwritten policies that assert required resources and attribute values. No resources are deployed. The results show that IaC generation is harder than general purpose coding. The strongest model cleared only 19.4% of IaC tasks, but 86.6% on a general purpose coding benchmark [KLQ+24].

DPIaC-Eval extends evaluation by adding a deployability test. The benchmark contains 153 tasks mined from real-world AWS CloudFormation scripts. Task prompts are written by human experts based on these scripts. Generated code is first checked syntactically and is then deployed to real infrastructure. This exposes a large gap: 42.7% of syntactically correct scripts failed to deploy. The benchmark also evaluates if generated scripts follow security best practices [ZPZ+25].

[7] `https://www.swebench.com/` (accessed: June 2026)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

SWE-InfraBench is the closest existing benchmark to this thesis. Like our benchmark, it constructs tasks systematically. Task-construction starts by masking a code block in a real-world IaC repository. An LLM then reverse-engineers a prompt and tests from the masked code. A second LLM then verifies that the prompt fully describes the masked code, that tests cover the prompt's requirements, and that tests generalize to alternative solutions. A human expert gives the final approval [TBSI+26].

Existing IaC benchmarks share the goal of secure infrastructure. They favor production-ready configuration that is free of vulnerabilities. IDS testbeds require the opposite: infrastructure must contain a specific vulnerability. This cannot be evaluated via a benchmark that penalizes insecure configurations.

The second group of benchmarks relevant to this thesis is offensive security. Cybench is a well-known Capture-the-Flag (CTF) benchmark. Each task provides a running target service in which the agent has to find and exploit a vulnerability. Evaluation showed that even the strongest models only solve 17.5% of tasks [ZPD+25].

CVE-Bench goes beyond CTFs and builds a benchmark from real-world CVEs. Each task reconstructs a CVE in a containerized sandbox and prompts an agent with the CVE's description. In this setup, the strongest agent framework exploits only 13% of tasks [ZKB+25].

MHBench resembles the attack side of our benchmark. In each task, the agent starts at the edge of an emulated enterprise network and must move deeper through chained exploits. Agents that write raw shell commands solve only 7.5% of tasks. The same agent, using a simplified, declarative attack interface, solves 92.5% [SLA+25].

In all these benchmarks, the agent works in a live environment to iteratively discover and exploit a vulnerability. IDS testbeds do not need discovery but rather a deterministic script that recreates the attack identically across runs. No existing benchmark covers this.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 3

Methodology

This chapter describes how we turn an existing IDS testbed into a benchmark and how we evaluate coding agents on it. Section 3.1 explains why we chose AttackBed as the source for the benchmark. Section 3.2 then describes AttackBed's structure and common tasks. Building on this, Section 3.3 defines a task taxonomy and selects the categories most relevant to the benchmark. Section 3.4 presents the workflow used to build the benchmark tasks. Section 3.5 defines how agents are graded and which metrics we report. Finally, Section 3.6 presents the agents we evaluate and how they are configured.

3.1 Testbed Selection

Section 2.1 compared five IDS testbeds. We build our benchmark from only one of them. This is because we grade generated artifacts by executing them, and each additional testbed would require a new evaluation runner. This comes at the cost of tying our results to the selected testbed's technologies (see Section 6.1).

Our experiment design implies the following filtering conditions. A testbed that violates any of them cannot be the source of our benchmark:

1. **Open source.** We reverse-engineer benchmark tasks from reference implementations. This requires access to the source code. We exclude code that is not publicly available to allow others to reproduce our results.
2. **Automated operation.** We grade tasks by deploying generated code and testing its behavior. To ensure our evaluation runner can do this, the testbed must deploy and run without manual steps.

These conditions filter out three of the five testbeds. Algorizmi and KYPO use human attackers and thereby fail condition 2 [Ali10, VOČ+17]. CIDDS provisions its hosts

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

manually and thereby fails condition 2 [RWG+17]. This leaves SOCBED and AttackBed, which pass both filters. We pick AttackBed for the following reasons.

First, AttackBed offers a larger and more diverse set of task candidates. It contains 34 end-to-end attacks of up to 16 steps across 7 topologies of up to 8 hosts, against SOCBED's 12 attack modules of up to 9 steps on a single 6 host network (Table 2.1). Having more task candidates lets us build a bigger benchmark, and gives us more freedom to choose diverse tasks.

Second, AttackBed is developed at AIT, which gives us direct access to its authors. This becomes useful during task construction where we reverse-engineer requirements from AttackBed's reference implementation. Author access lets us validate task prompts and tests more thoroughly.

3.2 Anatomy of AttackBed

AttackBed[1] consists of multiple scenarios. Each scenario contains a network of multiple, intentionally vulnerable hosts alongside multi-step attacks which exploit these vulnerabilities. These networks of hosts are built on top of the open-source cloud platform OpenStack. AttackBed is implemented across three layers of IaC:

1. **Configuration layer**: This first layer uses Packer to build VM images for each host. The process takes a VM base image (e.g., Ubuntu 24.04) and applies a collection of composable Ansible roles, which install packages, configure services, or start services. Packer then creates a snapshot of the machine state which will be deployed in the next stage.

   Each host in AttackBed has its own Packer image, the source code of which can be found in the `packer/<host>` subdirectories.

2. **Provisioning layer**: This second layer uses Terraform to deploy the virtual machines, networks, and security groups required for a scenario to OpenStack. It takes the VM images built in the previous stage and places them in the desired subnet of the network.

   Each scenario is defined across multiple Terraform modules. This includes one scenario-specific module which sets up the infrastructure uniquely required for a scenario. Two default modules, `bootstrap` and `attacker`, are part of every scenario and set up common infrastructure such as DNS, attacker, and management hosts. The source code can be found in the `terragrunt/<host>` subdirectories.

3. **Attack runtime layer**: This third layer uses Ansible to perform some minor attack specific configuration of hosts, uploads AttackMate playbooks to the attacker host and triggers the attack execution. Runtime playbooks can be found under `ansible/run/scenario<n>/`.

[1] Source code: `https://github.com/ait-testbed/attackbed`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.2.1 Vulnerable Infrastructure with Ansible Roles

Ansible roles are the core abstraction for configuring hosts. Roles fall into three categories:

1. Service roles are responsible for installing and configuring applications needed for the functionality of the testbed. An example would be `apache2`, which provides the webserver to expose other services over HTTP.
2. Vulnerability roles introduce a specific weakness. An example would be `zoneminder`, which installs ZoneMinder version 1.36.33 that contains a remote code execution vulnerability (CVE-2023-26035).
3. Instrumentation roles configure testbed specific monitoring tools. An example is `auditd` which configures auditd to automatically tag logs with MITRE ATT&CK technique IDs.

Ansible roles are composable, which we will illustrate by the example of the videoserver host. Its Packer playbook applies `apache2` which is a dependency for `zoneminder`. Zoneminder introduces the vulnerability of the host which will be exploited by an attacker to gain initial access to the server. Other roles are applied that configure further weaknesses, e.g., some misconfigurations which allow for privilege escalation or backdoor creation. Lastly, instrumentation roles like `auditd` are applied to enable capturing log data relevant for creating IDS datasets.

In most cases, roles can be replaced without modifying other roles. This modularity makes them the unit of work for infrastructure tasks in AttackBed.

3.2.2 Attack Orchestration with AttackMate

AttackMate executes attack playbooks against the deployed network. A playbook is a YAML-based sequence of commands executed on offensive tools like Metasploit, Sliver, or regular SSH connections. Outputs flow between steps, so later stages can access earlier results. Sessions also flow between steps, allowing human-like, multi-step interactions with interactive tools like vim or Metasploit.

Listing 3.1 shows an excerpt from an AttackMate playbook. It opens with network reconnaissance, then exploits ZoneMinder's vulnerability and stores the Metasploit session.

Listing 3.1: Excerpt from the AttackMate playbook `scenario_1_a_a.j2`.

```
commands:

  - type: shell
    cmd: dnsenum -f $DNS_LIST --dnsserver $DNS_SERVER $DOMAIN
    metadata:
      techniques: "T1590.002,T1591"
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
    tactics: "Reconnaissance"

- type: msf-module
  cmd: exploit/unix/webapp/zoneminder_snapshots
  creates_session: "foothold"
  options:
    RHOSTS: 192.42.0.254
  payload_options:
    LHOST: 192.42.1.174
  payload: cmd/unix/python/meterpreter/reverse_tcp
  metadata:
    techniques: "T1190,T1059.006"
    tactics: "Initial Access"
```

Within an AttackMate playbook, the unit of work is a sequence of steps that implements one attacker technique. For example, swapping out a privilege escalation technique requires rewriting only the relevant steps. Preceding reconnaissance or later persistence steps can remain unchanged.

3.3 Task Taxonomy

For the purposes of this benchmark, we define a task to be a unit of work a researcher would delegate when adapting a testbed. Each of AttackBed's three IaC layers expresses one such unit of work. The configuration layer uses Ansible roles to encapsulate services, vulnerabilities, or observability tools. The provisioning layer uses Terraform to encapsulate each scenario's network topology. Lastly, the attack layer organizes multi-step attacks as AttackMate playbooks.

These three categories can be observed across IDS testbeds more broadly. The comparison of testbeds in Chapter 2 shows that service configuration, infrastructure provisioning, and attack scripting appear across all examined testbeds. However, the comparison also reveals that normal user behavior simulation, present in most other testbeds, is absent from AttackBed and therefore falls outside this benchmark's scope [Ali10, VOČ+17, RWG+17, UHH+21, LSF+22].

Of the three categories of tasks AttackBed supports, we choose only configuration through Ansible roles, and attack scripting through AttackMate to form the basis of our benchmark. We do so for three reasons:

- **Size**: AttackBed contains 63 Ansible roles and 34 AttackMate playbooks, compared to only nine Terraform modules. A representative benchmark requires many unique tasks, which are not provided by the nine Terraform modules.
- **Research relevance**: Researchers primarily adapt testbeds to produce new log data for IDS evaluation. The most direct way to do so is by introducing new services, swapping vulnerabilities, or changing attack techniques. These types of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

changes primarily touch Ansible roles and AttackMate playbooks. In contrast, network topology is stable scaffolding that rarely changes between iterations.

- **Benchmark coverage**: AttackBed uses standard infrastructure provisioning in its Terraform modules. This type of code is already being evaluated by general purpose IaC benchmarks [KLQ+24]. On the other hand, the two remaining categories sit outside existing work for reasons outlined in Sections 2.2 and 2.3.

3.4 Benchmark Construction

Every benchmark task is built by hand. The following subsections describe the workflow in order.

3.4.1 Instance Selection

Contrary to other benchmarks, we cannot build tasks automatically from existing documentation (like GitHub issues) and test suites [JYW+23]. Tasks of our benchmark have to be built manually, which we do via the resource intensive process presented in this section. Building tasks from all 63 Ansible roles and 34 AttackMate playbooks exceeds the time budget of this thesis. Therefore, we select a subset that spans the main task categories encountered in AttackBed. Among task candidates we prioritize the most relevant ones for agent evaluation and leave out trivial ones.

Ansible roles. In Section 3.2.1 we classified the Ansible roles of AttackBed in three categories: (1) service roles which install regular applications, (2) vulnerability roles which introduce an exploitable weakness, and (3) instrumentation roles which configure testbed monitoring. We build tasks only from the first two categories. Instrumentation roles are excluded because they are stable scaffolding that does not need to be changed when a researcher introduces a new service or vulnerability.

Our selection is weighted towards vulnerability roles. This is because introducing an exploitable weakness is specific to IDS testbed development and is not covered by general IaC benchmarks. In contrast, service roles are a part of regular systems administration. Configuring a tool like `apache2` is common across organizations and will likely be covered in more depth by other benchmarks. Therefore we sample service roles lightly. Table 3.1 lists the selected Ansible roles.

AttackMate Tasks. We define a task as a sequence of AttackMate steps that serves one tactical objective, such as reconnaissance, initial access, or privilege escalation. We chose this level of granularity for two reasons: (1) It matches a common unit of work for researchers, who rewrite one phase of an attack while leaving surrounding stages unchanged (see Section 3.2.2). (2) It helps localize failures. If a task covers only one tactic, a failure points to a specific gap of agent capability. If a task covers the creation of an end-to-end playbook, failures only show that something broke along the way.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Type	Role	Purpose
Service	`apache2`	Webserver dependency for other roles
	`mariadb`	Database dependency for other roles
	`simulatedwebcam`	Simulates a webcam through an RTSP stream by chaining `ffmpeg` and MediaMTX
Vulnerability	`zoneminder`	Initial access through vulnerable ZoneMinder version
	`debiansnapshot`	Privilege escalation by pinning Polkit's vulnerable version
	`sudoweakness`	Privilege escalation through wrong sudoers rule
	`sshkeyprivesc`	Privilege escalation through wrong SSH key permissions
	`awffullprivesc`	Privilege escalation through a root cron script execution
	`logrotten`	Privilege escalation through a logrotate race condition

Table 3.1: Ansible roles selected as benchmark tasks.

In AttackBed's Scenario 1, all AttackMate playbooks are annotated with MITRE ATT&CK tactics. We select three tactics which form the core of a minimal end-to-end intrusion chain: initial access, privilege escalation, and persistence. We exclude the remaining tactics because they are thin wrappers around standard bash tools like `nmap` or `dnsenum`. Table 3.2 lists the selected AttackMate tasks.

3.4.2 Requirements Extraction

AttackBed does not contain issues, PR descriptions, and test suites which could be mined for tasks. Therefore, we must reverse-engineer prompts and oracles for each task from the reference implementation. Instead of writing prompts and oracles directly, we extract a list of requirements for each component first. Prompts and tests are then derived from these requirements. We do this to avoid drift between the two and to facilitate expert reviews (Section 3.4.3).

When collecting requirements from a reference implementation a difficulty is choosing a good level of abstraction. Prompts that spell out implementation details reduce the agent to a translator, translating from natural-language pseudocode to a programming language. This neither measures agent capability nor utility to researchers. On the other hand, prompts that are under-specified, e.g., by not specifying an interface variable, force the agent to invent details which might break interfaces with other components in the testbed. Tests inherit this tension. Tests modeled too closely on the reference implementation produce false negatives, rejecting correct alternative implementations. On the other hand,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Tactic	Task	Technique
Initial Access	`zoneminder_init...`	Exploit RCE vulnerability in ZoneMinder
Privilege Escalation	`pwnkit_privesc`	Exploit vulnerability in Polkit's `pkexec`
	`sudo_privesc`	Exploit a misconfigured sudoers rule
	`systemd_privesc`	Exploit a writable systemd unit executed as root
	`logrotten_privesc`	Exploit a race condition in `logrotate`
	`cron_privesc`	Exploit a writable script invoked by a root cron job
	`rootssh_privesc`	Exploit a readable root SSH private key
Persistence	`sshkey_persistence`	Add attacker key to `authorized_keys`
	`account_persistence`	Create an account with root shell access
	`pam_persistence`	Install a backdoored PAM module

Table 3.2: AttackMate tasks selected as benchmark tasks.

permissive tests lead to false positives by accepting trivial implementations.

Table 3.3 illustrates this over- vs. underspecification tension. It shows the same requirement of the `logrotten` Ansible role at three levels of abstraction. The over-specified version spells out implementation details making implementation trivial. The underspecified version forces the agent to fill specific gaps with safe defaults that risk breaking the intended vulnerability.

We aim to resolve this over- vs. underspecification tension by requiring each requirement to specify interfaces and externally observable behavior, but not internal implementation details.

We extract requirements by reading each task's reference implementation top to bottom. For every statement we decide whether it expresses an interface or behavior which must be preserved or an implementation choice that can be omitted. We aggregate preserved statements into a list of requirements which feeds into the expert validation step.

3.4.3 Expert Validation

The requirements extracted in the previous step are validated by experts. ISO/IEC/IEEE 29148 [ISO18] defines nine characteristics of a good requirement. Five of these character-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Level	Requirement
Under-specified	Configure logrotate so that logrotten can exploit it.
Over-specified	Configure logrotate by creating `/etc/logrotate.d/apache2-vhost` with a rotation rule for `{{logrotten_webdirectory}}/log/*.log` using the directives `daily`, `rotate 14`, `compress`, `delaycompress`, `notifempty`, `create 660 root www-data`, `sharedscripts`, and a `postrotate` hook that runs `invoke-rc.d apache2 reload`.
Balanced (shipped)	logrotate must be configured to rotate Apache2 virtual host log files. The configuration must maintain the directory permissions for newly created log files. This is the intentional vulnerability that logrotten exploits...

Table 3.3: Requirement R4 of the `logrotten` Ansible role at three levels of abstraction.

istics hold automatically.[2] The remaining four characteristics require human validation: Correct, Necessary, Appropriate, and Complete.

Three research engineers from AIT reviewed the requirements. They each have more than ten years of professional experience. Their familiarity with AttackBed varies: one is the main author, one is a regular collaborator, and one has extensive DevOps experience but limited exposure to AttackBed. This diversity in backgrounds is intended to spread reviewers' attention across the four characteristics. Insiders will catch domain-specific errors. Outsiders will catch implementation choices insiders have grown blind to.

Reviewers received surveys containing all the requirements of tasks featured in the benchmark. For each requirement, they rated three statements on a Likert scale from Strongly Disagree to Strongly Agree. Each statement maps to one of three characteristics:

- **Accuracy** (*Correct*): the requirement is technically correct.
- **Necessity** (*Necessary*): the requirement is essential, and omitting it causes a deficiency. In our setting this also includes requirements that are common knowledge to a skilled human software engineer.
- **Abstraction** (*Appropriate*): the requirement does not leak implementation details.

The standard's *Complete* characteristic asks whether anything is missing. This cannot be rated like the others since a missing requirement is one that is absent from the list.

[2] Feasible and Verifiable hold because each requirement is built from a reference implementation. Singular and Conforming hold because each requirement is written as a single statement to a template. Unambiguous holds because during the discussion rounds, each requirement is rephrased until all reviewers agree.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Therefore, each task-section of the survey closed with an open question asking reviewers to list missing requirements.

Figure 3.1 shows the survey results per reviewer, requirement, and dimension. Requirements are flagged for further discussion if any reviewer selected Disagree or Strongly Disagree on any of the three dimensions. Using this rule, 44 of 57 requirements were flagged. Disagreements concentrated on Abstraction (33) over Accuracy (19) and Necessity (15).

Figure 3.1: Survey results. Each cell is a reviewer's Likert rating of one requirement on one dimension. The top half covers Ansible role tasks, the bottom half covers AttackMate tasks.

Flagged requirements were resolved in discussion rounds with all reviewers present. The author presents the requirement, the dissenter(s) explain their reasoning, and other reviewers weigh in. The discussion ends with one of two outcomes: the dissenter is convinced by the agreeing reviewers, or the author rephrases the requirement until unanimous agreement is reached. Missing requirements surfaced through the survey are discussed in the same rounds. Each candidate is presented and is either added to the list if agreement is reached, or dropped otherwise.

Each flagged requirement yields one to three discussion items, one for each dimension

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

a reviewer disagreed on. Additionally, we add an item for each missing requirement candidate. This turned the 44 flagged requirements into 61 discussion items. After the discussion rounds 31 requirements remained unchanged, 19 were rephrased, 7 were removed, and 2 were added. The full audit trail from initial requirements, survey responses, and discussion notes is documented on GitHub.

3.4.4 Task Construction

Each task consists of three parts: a prompt for the agent, an initial repository state that the agent operates on, and tests to evaluate the generated code. All three are derived from the validated requirements of Section 3.4.3.

Prompt. Each task receives a single prompt. It is built by concatenating a short task introduction with the validated requirements. The task introduction is shared across task-instances. Listing 3.2 shows the prompt for the `simulatedwebcam` role.

Listing 3.2: Prompt for the `simulatedwebcam` task.

```
Create an Ansible role at `ansible/roles/simulatedwebcam` based on
    the following requirements. You will find a `defaults/main.yml` at
     the path which defines the role's interface. Use it and do not
    overwrite it.

**R1: RTSP Stream of a Looping Video**
The system must continuously broadcast a pre-recorded video file (`
    simulatedwebcam_video`) as a real-time RTSP stream to `
    simulatedwebcam_rtsp_url`.

**R2: Authenticated Access**
Publishing to the RTSP stream and reading from the stream each
    require separate credentials, both configurable (see `defaults/
    main.yml`).
```

Initial repository state. This defines the initial state of the repository the agent operates in. We are guided by three principles when constructing it: (1) remove the code the agent is expected to produce, (2) remove similar components the agent could copy the solution from, and (3) preserve interfaces to the rest of the testbed.

For Ansible tasks this means deleting the complete role directory (principle 1). However, we do keep `defaults/main.yml` which declares the role's input variables but implements none of its logic (principle 3). We also remove the role's entry from `requirements.yml` files so the agent will not find the reference's GitHub URL and attempt to install it (principle 2)[3].

For AttackMate tasks, we remove the relevant attack steps from the target playbook (`scenario_1_<n>_<n>.j2`), and, if required, we remove payload staging from

[3]Such installs would fail anyway because the sandbox the agent runs in blocks outbound GitHub traffic (see Section 3.6.2). Removing the `requirements.yml` entry avoids confusing the agent with a URL it cannot reach.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

`main.yml` (principle 1). We do not leave any `TODO` markers at the deletion locations. The agent must infer the insertion point from the surrounding code. Furthermore, we remove all other AttackMate playbooks to prevent the agent from copying a solution from a similar variant (principle 2). We only keep scenario 7's playbooks to serve as a reference for AttackMate's syntax. Scenario 7 differs enough from our task scenarios to not leak answers.

Tests. Each requirement is captured in one or more functional tests. Our test-runner deploys the generated component to real infrastructure and tests externally observable behavior. We do not check implementation choices hidden behind the component's interface to avoid false negatives on correct alternative implementations (see Section 3.4.2).

We use different test-runners for the two task types. Ansible tests are written using the testinfra-pytest library which asserts outputs of shell commands. AttackMate tests are written in a simple YAML DSL which pairs commands with assertions on their output. Listing 3.3 shows the test for R1 of the `simulatedwebcam` task.

Listing 3.3: Test for `simulatedwebcam`'s R1 (simplified).

```
def test_r1_stream_delivers_looping_video(host):
    cmd = host.run(
        f"ffmpeg -i 'rtsp://cam:cam@localhost:8554/mystream' -t 300 -
            f null -"
    )
    elapsed = ...          # parsed from ffmpeg progress output
    assert elapsed > 290   # provided 10s video looped for ~5 minutes
```

Section 3.5 motivates our choice of functional tests on real infrastructure over alternatives. Section 4.2 goes into more details on the implementation of the test-runner.

3.5 Evaluation Strategy

We grade each task by deploying generated code to real infrastructure and testing its behavior. We chose functional testing on live resources over three alternatives:

- **LLM-as-a-judge** evaluation misses subtle failures and produces false positives. Our early experiments revealed that LLMs miss subtle errors, such as a missing command flag or wrong file permissions.
- **Static analysis** risks overfitting to implementation choices of the reference solution and rejecting correct alternative implementations. This would create false negatives.
- **Expert review** does not scale to 190 generated solutions (2 agents, 19 tasks, 5 runs each).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

We chose to evaluate with functional tests on real infrastructure to avoid these problems. It makes results reproducible and objective. But most importantly, it evaluates actual utility, since a testbed component is useful if and only if it delivers the desired functionality. The trade-off is that the manual effort required for writing functional tests limits benchmark size.

3.5.1 Evaluation Runners

Both Ansible role and AttackMate task evaluation follows the same approach: deploy the artifact and execute assertions on externally observable behavior. However, the exact mechanics differ.

Ansible roles. We use Molecule[4] to provision a fresh OpenStack VM per task, and apply the generated Ansible role. If deployment succeeds we execute pytest-testinfra tests on the VM. These tests execute commands and assert exposed host behavior and state. Listing 3.3 shows an example of tests for the `simulatedwebcam` role.

AttackMate playbooks. First, the test runner deploys the vulnerable infrastructure for a target scenario from the known-good reference implementation. It then executes the generated AttackMate playbook against this infrastructure, and captures log files produced during execution. These log files contain the individual AttackMate commands executed alongside their outputs. The evaluation runner then parses these outputs, selects relevant commands, and asserts their outputs, e.g., if an output contains a substring, matches a regex, or has more than n lines.

3.5.2 Metrics

We use four metrics to measure agent performance:

Syntactic correctness. The fraction of syntactically valid solutions over five runs. Ansible roles are validated with Molecule's syntax checker and Ansible's `--syntax-check` flag. AttackMate playbooks are validated against AttackMate's playbook schema.

Deployment success. The fraction of runs that reach a testable state over five runs. For Ansible roles, this means the role applies without errors. For AttackMate playbooks, this means the playbook terminates.

Test pass rate. Per run we compute the relative pass rate of tests and average across the five runs. Runs that already failed at syntax or deployment count as failing all tests (i.e., 0).

pass@k. For each task, pass@k is the probability that at least one of k generated solutions passes all tests. It captures LLM sampling randomness, and shows if re-running the agent could help to find a correct solution. We report pass@1 and pass@5, computed over the five runs.

[4] `https://github.com/ansible/molecule/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.5.3 Handling Infrastructure Flakiness

Our tests run on real infrastructure which can be flaky. Upstream services could fail, or tasks could get scheduled in an order that eliminates a race condition. This can cause a correct artifact to fail tests.

We handle such failures by retrying any run that fails tests (except for syntax). A retry executes the full pipeline from scratch by provisioning a fresh VM or scenario-network, applying the artifacts and executing the tests. We retry up to three times and keep the best result. Three retries is enough to catch most flaky failures while staying within available compute time.

3.6 Agent Selection and Configuration

We selected coding agents based on strong performance on SWE-bench [JYW+23] and Terminal-Bench [Tea25]. From the leading models, we picked one open-weight and one proprietary agent. Each model runs inside its vendor's recommended agent-harness. This prevents limitations caused by a mismatch of model and agent-harness.

Following these criteria, we picked two agents. The first is Anthropic's Claude Code agent-harness using the Opus 4.6 model. At the time of writing, Opus 4.6 leads various public coding benchmarks like SWE-bench Verified (80.84%) and Terminal-Bench 2.0 (65.4%) [Ant26]. The second is Mistral's Vibe CLI agent-harness with the Devstral 2 model. Devstral 2 is an open-weight model which specializes in coding tasks. It reaches 72.2% on SWE-bench Verified [Mis25].

3.6.1 Configuration

We run both agents in their default configuration, and do not modify system prompt, model parameters, or available tools. Furthermore, we do not extend agents by adding MCP servers, skills, or custom subagents. We do this to avoid introducing configuration bias. Both agents ship with comparable default tools: file read, file edit, shell command execution, web search, code search, todo tracking, and subagent invocation.

3.6.2 Execution Context

Agents run inside a Docker container which provides a stable execution environment. The container ships with the full AttackBed toolchain installed (Ansible, Python, AttackMate, etc.). These tools can be used through the agent's bash tool to test generated code. The container is configured with OpenStack credentials, so in principle, agents could test their code on real infrastructure. However, no run attempted this, likely because the prompt does not explicitly ask for it.

The sandbox container blocks outbound traffic to GitHub to prevent agents from downloading the reference solution.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Implementation

The benchmark-runner[1] is a CLI with two subcommands: `run` triggers an agent to generate code and `eval` scores the generated artifacts. The two phases are decoupled to snapshot the outputs after the expensive code generation stage. This allows for repeated, independent evaluations of the generated artifacts. Figure 4.1 shows the pipeline architecture and how the two phases connect. Section 4.1 covers the `run` subcommand, Section 4.2 covers the `eval` subcommand.

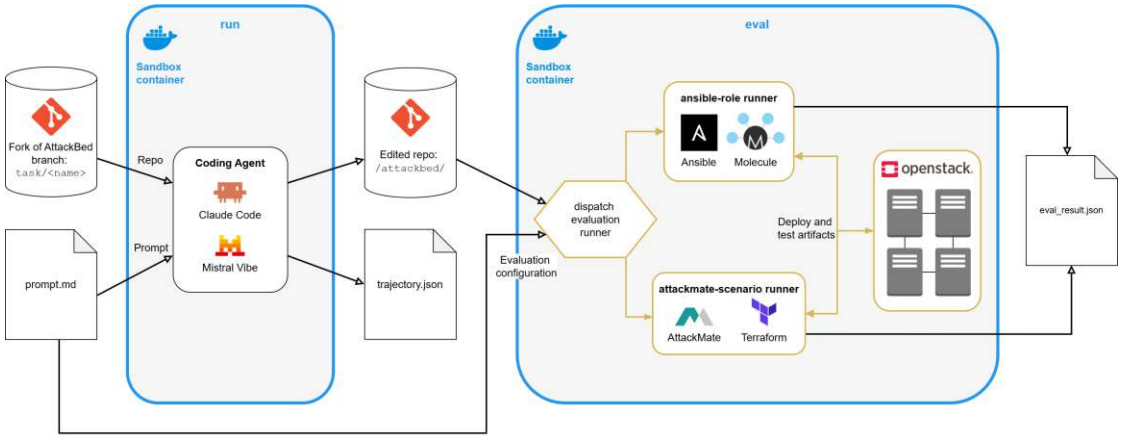

Figure 4.1: End-to-end pipeline. `run` sets up the task-repository, starts an agent that edits it, and saves the agent's trajectory. `eval` consumes the edited AttackBed and the evaluation-config, dispatches the configured evaluation runner which deploys artifacts on OpenStack and executes functional tests. Finally an evaluation result file is saved.

[1] Source code and task definitions on GitHub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.1 Code Generation Pipeline

The `run` command prepares the task-repository, spawns a sandbox container and starts an agent with the task's prompt. The outputs of this stage are the edits made to the task-repository and the agent's trajectory file.

Task Definition. Each task is defined in a single `prompt.md` file. The file contains the task-prompt as a markdown body and task-configuration as YAML frontmatter. The task-configuration specifies two things: (1) it pins a dedicated task-repository branch in which the reference solution has been removed as described in Section 3.4.4, and (2) it defines the evaluation-runner and its arguments to evaluate generated code.

Run Pipeline. The runner clones the task-repository, which is a specific branch of an AttackBed fork. Then it wipes the `.git` directory and re-initializes the repository, so the agent cannot recover the deleted reference solution from git objects.

Next, the runner creates a Docker container and mounts three directories: (1) an artifacts directory containing the task's prompt file, (2) the task-repository from the previous step, and (3) the agent directory containing the `run.sh` entrypoint.

The interface between benchmark-runner and agent is defined through three fixed paths. The benchmark-runner invokes the agent through its entrypoint at `agents/run.sh`. In turn the agent reads its task-prompt from `/artifacts/prompt.txt` and edits the task-repository at `/artifacts/attackbed/`. The benchmark-runner itself contains no agent-specific code. Adding a new agent only requires defining a new agent entrypoint under `agents/<name>/run.sh`.

When the agent exits, the benchmark-runner collects two artifacts: the task-repository with the agent's edits, and a trajectory file containing the agent's tool-call trace. To speed up this stage, code-generation runs concurrently.

4.1.1 Sandbox Image

A Docker image pins the entire AttackBed toolchain at required versions. Both the `run` and `eval` phases instantiate fresh containers from this image.

A fresh container per `run` guarantees that each agent starts from an identical state. This prevents leakage of agent installed packages or config changes from one run into the next.

Reusing the same image across `run` and `eval` phases ensures agents can test their work during development using the same toolchain version as the evaluator. This eliminates failures in which the agent installs a required tool at a version that is incompatible with the evaluation-runner's version.

The container is configured to block outbound traffic to GitHub. This prevents the agent from fetching the reference solution from the upstream repository.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

4.2 Evaluation Pipeline

The `eval` command creates a fresh Docker container with the task-repository from `run` mounted. It then reads the task-configuration from `prompt.md`'s frontmatter and dispatches the specified evaluation-runner. Each evaluation-runner executes three stages: (1) it performs syntax checks, (2) deploys generated artifacts to OpenStack, and (3) executes functional tests. These three stages enable attribution of failures and map directly to the metrics defined in Section 3.5. Each task's evaluation results are written to an `eval_result.json` file.

The benchmark defines two task types: Ansible roles and AttackMate playbooks. The evaluation-runner contains a handler for each.

4.2.1 Ansible Role Handler

Molecule is the standard framework for testing Ansible roles[2]. It provides drivers and commands to provision target hosts, apply Ansible roles, and run tests against these hosts.

The Ansible role evaluation handler wraps Molecule. It generates task specific Molecule configuration, and maps the three evaluation stages to Molecule commands. The syntax stage runs `molecule syntax`. The deployment stage runs `molecule converge` to provision a fresh OpenStack VM and apply the role. The behavior stage runs `molecule verify` to execute tests against the live VM. Each task's tests use the testinfra-pytest library. A task's test file is specified in `prompt.md`'s configuration block.

The Ansible role evaluation handler records each stage's result in `eval_result.json`. Listing 4.1 shows the result for the `simulatedwebcam` task. The syntax and deployment stages pass. The behavior stage fails one test, `test_r1_stream_delivers_looping_video` (see Listing 3.3). The `output` field contains the raw `molecule verify` output.

Listing 4.1: `eval_result.json` for the `simulatedwebcam` task (simplified).

```
{
  "status": "fail",
  "stages": [
    { "name": "syntax",     "status": "pass" },
    { "name": "deployment", "status": "pass" },
    { "name": "behaviour",  "status": "fail",
      "tests_passed": 2,
      "tests_total": 3,
      "output": "tests/test_default.py F..                    [100%]
        ======================== FAILURES ===========================
        > assert elapsed > 290
        E AssertionError: assert 10.0 > 290
        1 failed, 2 passed in 11.4s"
    } ] }
```

[2] https://github.com/ansible/molecule

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.2.2 AttackMate Playbook Handler

Before evaluation begins, the handler deploys a fresh network from the scenario's reference implementation. In the later stages this network will be attacked by the generated AttackMate playbook.

AttackMate has no standalone syntax checker. Syntax errors surface as runtime exceptions before any other command gets executed. The syntax stage therefore infers syntactic correctness from AttackMate's `output.log` after deployment. An empty log indicates that the scenario aborted before its first command, which means a syntax error occurred.

The deployment stage executes the generated AttackMate playbook. If the process times out, it is classified as a failed deployment. Common causes are logically incorrect commands, such as a Metasploit handler waiting for a reverse shell that never arrives. If the process exits normally it is classified as a successful deployment. During execution AttackMate produces three logfiles, which are collected by the handler.

The behavior stage parses the AttackMate log files and executes assertions on their contents. The most relevant artifact is `output.log`, which records every executed command alongside its output. Assertions over this log are specified per task in `test.yml`. It names the commands of interest and defines assertions that must hold on its output. The handler supports five assertion types: `command_ran`, `contains`, `not_contains`, `regex`, and `line_count_min`.

This is illustrated in Listing 4.2. It shows a test from the `sudo_privesc` task, where the agent must use a misconfigured sudo rule to elevate privileges. The test targets a later persistence command that writes to the `/root` directory. This only succeeds if the privilege escalation worked. `command_ran` asserts the command executed, and `not_contains` asserts the output does not contain the text `Permission denied`.

Listing 4.2: An AttackMate behavior test from the `sudo_privesc` task.

```
- name: "Privilege escalation successful: Can write to /root"
  command:
    type: msf-session
    cmd: "cd /root; wget -O README.txt http://192.42.1.74/README.txt"
  assertions:
    - type: command_ran
    - type: not_contains
      pattern: "Permission denied"
```

Listing 4.3 shows the `output.log` block which is matched by the test. In this particular run the agent's privilege escalation failed, which causes `cd /root` to fail. The test's `command_ran` assertion passes because the command appears in the trace. However, the `not_contains` assertion fails because the output contains `Permission denied`.

Listing 4.3: `output.log` matched by the test in Listing 4.2. In this run the privilege escalation failed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
Command: cd /root; wget -O README.txt http://192.42.1.74/README.txt
cd /root; wget -O README.txt http://192.42.1.74/README.txt
bash: cd: /root: Permission denied
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Evaluation

This chapter reports the results of evaluating two coding agents on the benchmark. Section 5.1 sets up the experiment. Section 5.2 presents the results, moving from a high-level view to per-task and per-task-category breakdowns. Section 5.3 compares a subset of agent solutions against the human-authored reference. Section 5.4 distills the findings into recommendations for researchers who want to use agents to build IDS testbeds.

5.1 Experimental Setup

This section describes the experiment's scale, its environment, and how results are reported. Metric definitions and agent configurations can be found in Sections 3.5 and 3.6.

5.1.1 Experiment Scale

We executed two agents on 19 tasks, five times each. This yields a total of 190 generated solutions. We chose five runs per agent and task to capture the run-to-run randomness of LLMs. Notably, these five task runs are distinct from the up-to-three evaluation retries, which are designed to absorb transient infrastructure failures (see Section 3.5).

The experiment's code generation phase ran between May 3rd, 2026 and May 8th, 2026. The total Claude API cost was 148.12 USD, and the Devstral cost was below 5 USD. The experiment's evaluation phase ran between May 8th, 2026 and May 14th, 2026.

5.1.2 Pinned Versions

We pin AttackBed at commit 8989c41. From this version, we build all VM images used during AttackMate evaluation, and derive the task-branches which define each task's initial repository state. Table 5.1 lists the versions of tools and agents used for all runs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Category	Component	Version
Tool	Ansible	2.17.14
	Molecule	25.12.0
	AttackMate	0.6.0
	OpenTofu	1.8.1
	Terragrunt	0.94.0
Claude	Claude Code	2.0.76
	Opus model	`claude-opus-4-6`
Mistral	Vibe CLI	2.8.1
	Devstral model	`devstral-2`

Table 5.1: Pinned versions of tools and agents.

Reproducing these experiments requires:

- An OpenStack project with a minimum quota of 8 Compute Instances, 12 vCPUs, 40GB of RAM, 250GB of disk storage, 1 Floating IP, and 6 Networks.
- API access to Anthropic.
- API access to Devstral 2.

5.1.3 Experimental Integrity

Each task's prompt and tests were finalized in the expert validation phase (see Section 3.4.3), before any benchmark runs. Prompts and tests were not adjusted in response to observed agent outputs. In none of the 190 runs did an agent refuse to implement offensive security code, so tuning of prompt headers was also not necessary.

Raw results for all 190 runs are published on GitHub. It contains generated solutions (`attackbed.diff`), agent trajectories (`trajectory.json`), and `eval_result.json` files. Additionally, AttackMate tasks contain the log files used for scoring.

5.2 Main Results

This section presents the experiment's results. It starts with an aggregate view to understand whether agents can build IDS testbeds at all. It then breaks results down per task and task-category to show where agents succeed and where they fail. Finally, it examines the effects of repeated sampling to understand the impact of LLM's run-to-run randomness. The metrics used in this chapter are defined in Section 3.5.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.2.1 Aggregate and Per-task Results

Figure 5.1 shows the results of all runs per agent, bucketed by the stage at which each run fails. Evaluation moves through three stages: syntax, deployment, and behavior. A run that fails one stage is excluded from the next. This means each run lands in exactly one of the following buckets: fails syntax, fails deployment, fails tests, or all tests pass.

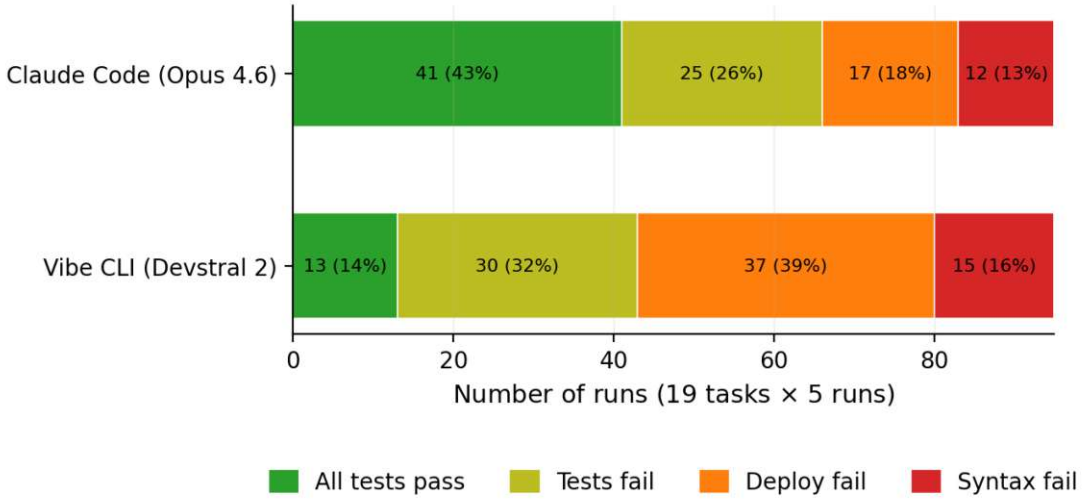

Figure 5.1: Results across all 95 runs per agent, bucketed by the stage at which each run fails. Each run lands in exactly one bucket.

Claude Code generates a solution which passes all tests in 41 of 95 runs. Vibe CLI does so in 13 of 95. Both agents produce a comparable amount of syntactic errors. The gap between the agents first appears during deployment, where Vibe CLI loses 20 more runs than Claude. This gap widens during testing. Of the runs that do deploy, Claude Code passes all tests in 62% of runs, compared to Vibe CLI's 30%.

Figure 5.2 breaks down the same 190 runs by task.

Several observations stand out:

Always solved. Both agents pass all tests of `6_sudoweakness` and `8_awffullprivesc` in every run.

Never solved. Neither agent passes all tests of the following tasks: `4_zoneminder`, `5_debiansnapshot`, `7_logrotten`, `13_systemd_privesc`, `14_logrotten_privesc`, `15_cron_privesc`, `16_rootssh_privesc`, and `18_account_persistence`. Concretely, tasks 4, 5, 14, and 15 fail already at the syntax or deployment stage. Tasks 7 and 16 always deploy but fail to pass all tests. Tasks 13 and 18 fail for various reasons.

Claude beats Vibe CLI. On 17 tasks Claude's best run reaches at least the same stage as Vibe CLI's best run. The opposite is true only in two tasks. In task 14, all of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

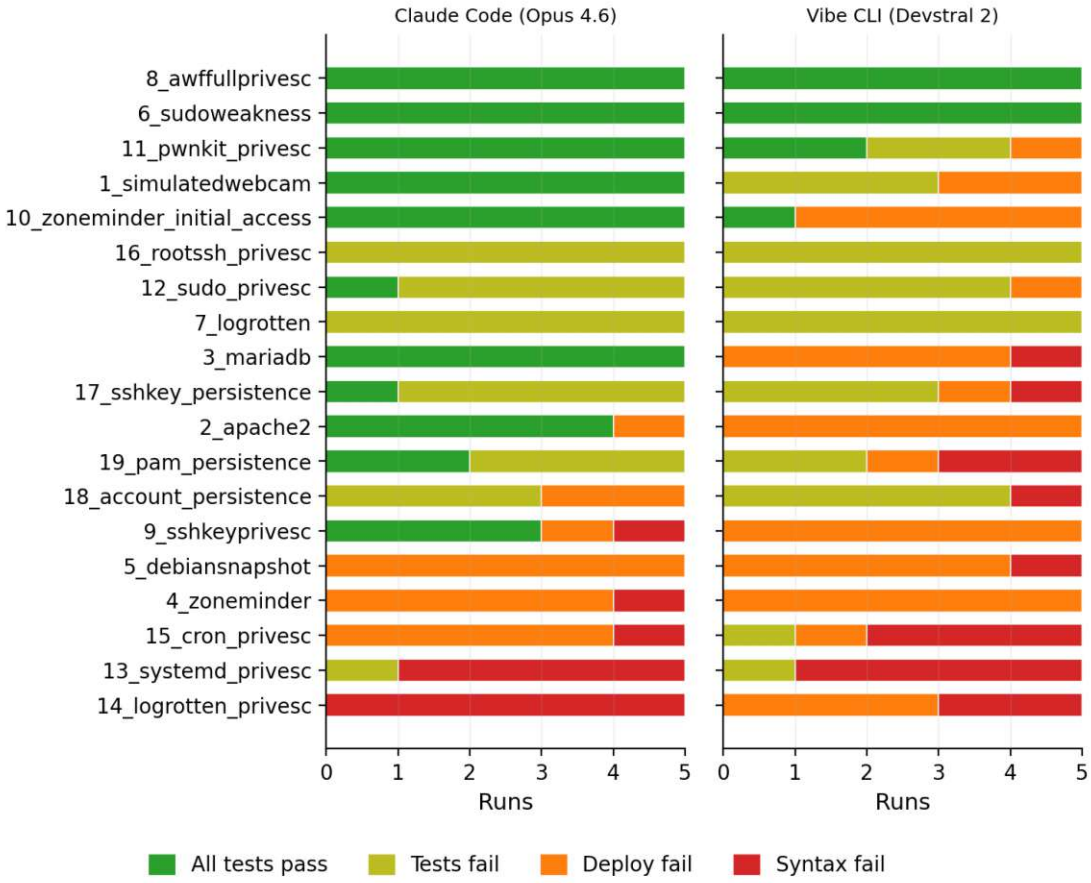

Figure 5.2: Results per task per agent, bucketed by the stage at which each run fails. Each run lands in exactly one bucket. Tasks are ordered by outcome.

Claude's runs fail syntax, while Vibe CLI clears three runs. In task 15, one of Vibe's runs deploys successfully while none of Claude's do.

Ansible is easier than AttackMate. Both agents pass all tests of Ansible tasks more often than AttackMate tasks. Claude fully passes 60% of Ansible runs but only 28% of AttackMate runs. Vibe CLI fully passes 22% of Ansible runs but only 6% of AttackMate runs.

Table 5.2 shows the per-task results underlying Figures 5.1 and 5.2.

5.2.2 Results by Task Category

Figure 5.3 aggregates the results by task category, using the categories introduced in Section 3.4.1. Each task's category assignments is shown in Table 5.2. Notably, the initial access category contains only one task and should only be considered anecdotal. It is not discussed below.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Task	Category	Agent	Syntax	Deploy	Tests pass	All tests pass
1_simulatedwebcam	Service role	Claude	5/5	5/5	100%	5/5
		Devstral	5/5	3/5	40%	0/5
2_apache2	Service role	Claude	5/5	4/5	80%	4/5
		Devstral	5/5	0/5	0%	0/5
3_mariadb	Service role	Claude	5/5	5/5	100%	5/5
		Devstral	4/5	0/5	0%	0/5
4_zoneminder	Vulnerability role	Claude	4/5	0/5	0%	0/5
		Devstral	5/5	0/5	0%	0/5
5_debiansnapshot	Vulnerability role	Claude	5/5	0/5	0%	0/5
		Devstral	4/5	0/5	0%	0/5
6_sudoweakness	Vulnerability role	Claude	5/5	5/5	100%	5/5
		Devstral	5/5	5/5	100%	5/5
7_logrotten	Vulnerability role	Claude	5/5	5/5	88%	0/5
		Devstral	5/5	5/5	90%	0/5
8_awffullprivesc	Vulnerability role	Claude	5/5	5/5	100%	5/5
		Devstral	5/5	5/5	100%	5/5
9_sshkeyprivesc	Vulnerability role	Claude	4/5	3/5	60%	3/5
		Devstral	5/5	0/5	0%	0/5
10_zoneminder_in...	Initial access	Claude	5/5	5/5	100%	5/5
		Devstral	5/5	1/5	20%	1/5
11_pwnkit_privesc	Privilege escalation	Claude	5/5	5/5	100%	5/5
		Devstral	5/5	4/5	60%	2/5
12_sudo_privesc	Privilege escalation	Claude	5/5	5/5	20%	1/5
		Devstral	5/5	4/5	0%	0/5
13_systemd_privesc	Privilege escalation	Claude	1/5	1/5	0%	0/5
		Devstral	1/5	1/5	0%	0/5
14_logrotten_privesc	Privilege escalation	Claude	0/5	0/5	0%	0/5
		Devstral	3/5	0/5	0%	0/5
15_cron_privesc	Privilege escalation	Claude	4/5	0/5	0%	0/5
		Devstral	2/5	1/5	0%	0/5
16_rootssh_privesc	Privilege escalation	Claude	5/5	5/5	0%	0/5
		Devstral	5/5	5/5	0%	0/5
17_sshkey_persistence	Persistence	Claude	5/5	5/5	20%	1/5
		Devstral	4/5	3/5	0%	0/5
18_account_persistence	Persistence	Claude	5/5	3/5	0%	0/5
		Devstral	4/5	4/5	0%	0/5
19_pam_persistence	Persistence	Claude	5/5	5/5	40%	2/5
		Devstral	3/5	2/5	0%	0/5

Table 5.2: Results over five runs per task, per agent. Metrics definitions in Section 3.5.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Each heatmap-cell shows the pass rate of a task category per evaluation dimension. A cell's value is calculated as follows: The denominator is always the total number of runs per agent and task-category, including runs that already failed at an earlier stage. The numerator of the *Syntax*, *Deploy*, and *All tests pass* columns is the count of runs which passed that stage. For the *Tests* column, the numerator is the sum of the fraction of the passing tests per run. Runs that never reached the behavior stage count as zero.

Reading a row from left to right shows at which stage a task category fails. Reading a column from top to bottom compares the inherent difficulties of each task category.

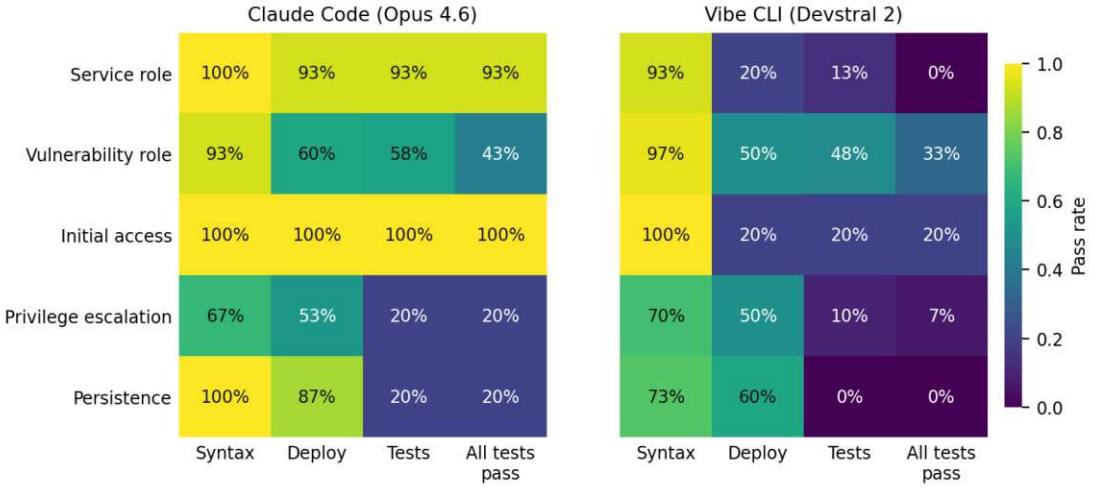

Figure 5.3: Pass rate by task category.

Several observations stand out:

Drop-off points are category-specific. The evaluation-stage in which a category loses the most runs is the same for Claude and Vibe CLI. Service roles and Vulnerability roles (Ansible tasks) see the sharpest drop-off at deployment. Privilege escalation and Persistence tasks (AttackMate tasks) see the sharpest drop-off at the testing stage.

Syntax is solved for Ansible roles, but not for AttackMate. The syntax pass rate across all Ansible role runs is 96%, with every cell above 93%. For AttackMate tasks the average syntax pass rate drops to 77%, with every cell above 67%.

Service roles are reliably solvable for Claude. Claude passes all tests on 93% of service role runs. The single miss is a deployment failure on a `2_apache2` run. For Vibe CLI, service roles are one of the hardest task categories. It fully passes none of the 15 service-role runs.

AttackMate tasks are rarely solved. None of the AttackMate task categories exceeds 20% pass rate on all tests, excluding the non-representative initial access category. Across all of Claude's attempts on AttackMate tasks, it passed all tests in only 28% of runs

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

(including initial access). Vibe CLI fares even worse, with only 6% pass rate across all AttackMate task categories.

5.2.3 Effects of Repeated Sampling

Figure 5.4 compares pass@1 and pass@5 per task. Pass@1 is the probability that a single run passes all tests. It is computed as the number of runs which pass all tests, divided by five runs. Pass@5 is the probability that at least one of five runs passes all tests. The difference between the two shows how much the LLM's run-to-run variance affects solutions.

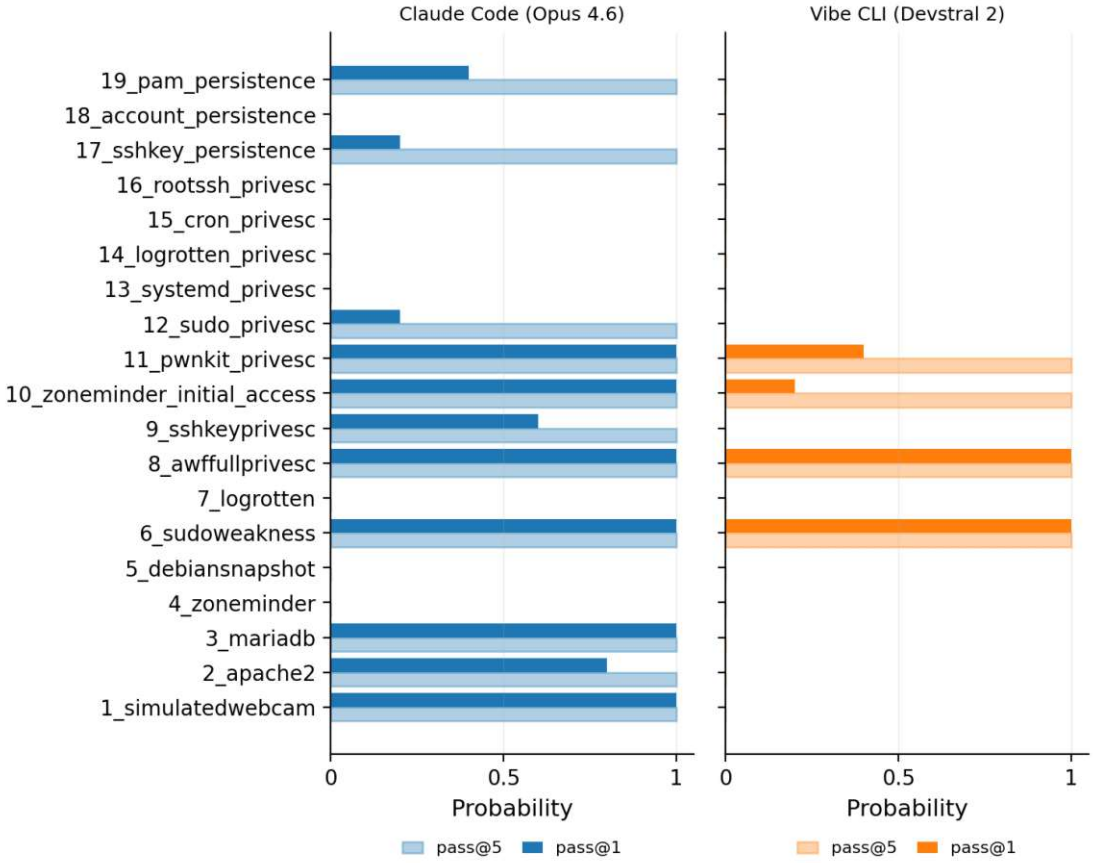

Figure 5.4: Pass@1 and pass@5 of each task.

Two observations stand out:

Pass@1 is usually 0 or 1. Most tasks get either all attempts correct (pass@1 = 1) or none (pass@1 = 0). For Claude, 6 tasks get all attempts correct, 8 tasks get no attempt correct, and only 5 get some attempts correct. For Vibe CLI, 2 tasks get all attempts correct, 15 tasks get no attempt correct, and only 2 tasks get some attempts correct. Across all runs, repeated sampling improved outcomes in only 7 agent-task pairs (18%).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Claude benefits more from retries. Claude's average pass rate moves from 43% (pass@1) to 58% (pass@5). Vibe CLI moves from 14% (pass@1) to 21% (pass@5).

5.3 Comparison to the Reference

Section 5.2.1 analyzed where tasks fail in the evaluation pipeline. It does not show how a task failed, i.e., whether an agent chose a fundamentally wrong approach or simply made a small mistake. To see the difference, this section compares agents' solutions to the human-authored reference.

We cannot manually compare all 190 runs with the human-authored reference. Therefore we group tasks by where they fail in the evaluation pipeline, and pick one representative task per group for illustration.

We define the following failure groups: A run is *solved* if every test passes, *near-miss* if at least one test fails but deployment succeeded, *deploy-failed* if deployment failed but the syntax checks succeeded, and *syntax-failed* if syntax fails. Each pair of agent and task is assigned to the group its runs land in most often. If there is a tie, the task is assigned to the more advanced stage. Table 5.3 shows the group-assignments.

Group	Claude Code tasks	Vibe CLI tasks
Syntax-fail	13, 14	13, 15
Deploy-fail	4, 5, 15	2, 3, 4, 5, 9, 10, 14
Near-miss	7, 12, 16, 17, 18, 19	1, 7, 12, 16, 17, 18, 19
Solved	1, 2, 3, 6, 8, 9, 10, 11	6, 8, 11

Table 5.3: Group assignment per task and agent.

5.3.1 Walkthroughs

We randomly pick one task per failure group from Table 5.3, drawing only from tasks where both agents landed in the same group. This makes comparison more compact but also isolates task difficulty from agent specific strengths and weaknesses. From the task's five runs we select one which matches its group's failure.

Syntax-fail: `13_systemd_privesc`

This task's goal is to escalate privileges via a world-writable systemd unit. The agent receives an unfinished AttackMate playbook, which already establishes a meterpreter shell on the videoserver, schedules a reboot which executes the systemd unit as root, and sets up a Metasploit handler to catch the root reverse shell. The agent has to fill in the rest: stage a reverse shell payload, and edit a systemd unit which executes the payload on reboot. Figure 5.5 compares the three approaches step-by-step.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

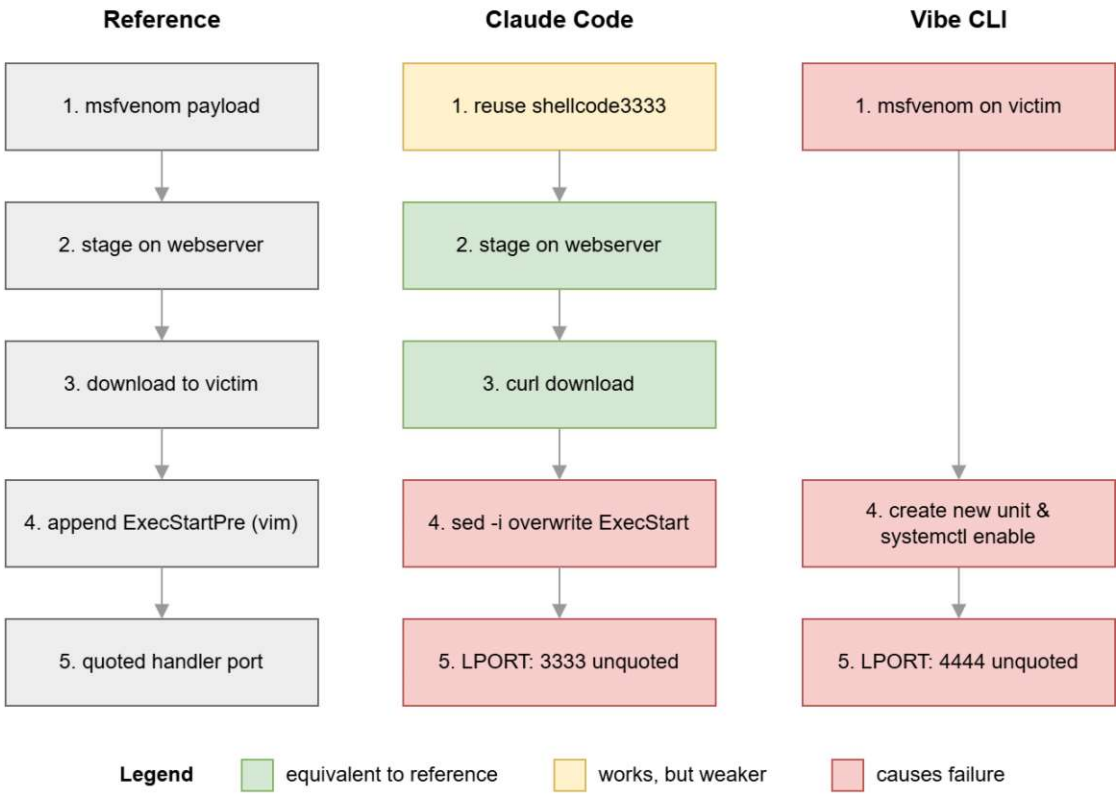

Figure 5.5: Step-by-step comparison of the three approaches to task `13_systemd_privesc`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake.

Reference. The scenario's Ansible setup uses `msfvenom` to create a reverse-shell payload, and stages it on a webserver. The AttackMate playbook downloads the payload onto the compromised videoserver. `vim` is used to wire up the payload with a systemd unit, by appending an `ExecStartPre` which calls the payload script on reboot.

Claude Code. In the scenario's Ansible setup, the agent creates a reverse shell payload by reusing the existing `shellcode3333` variable, and stages it on a webserver. In the AttackMate playbook it uses `curl` to download the payload onto the videoserver. Then it uses `sed -i` to overwrite a systemd unit's `ExecStart` line, to launch the payload on reboot. Lastly, it configures the reverse-shell handler to listen on port `LPORT: 3333`. This last step is what caused the syntax error: AttackMate expected a quoted string argument, not an integer.

Claude's approach is similar to the reference but differs in three places: (1) It reuses the existing `shellcode3333` variable instead of regenerating the payload. This works but is arguably less maintainable. (2) It overwrites `ExecStart` instead of appending

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

`ExecStartPre`. This works but means the original systemd unit's binary never starts. This makes the privilege escalation more detectable. (3) It uses `sed -i` to edit the systemd unit. This would fail because `sed -i` does not just require write permissions on the systemd unit file but also on its parent directory, which is not the case.

The full diff for this run is published on GitHub.

Vibe CLI. The agent attempts to use the meterpreter shell on the compromised videoserver to create a payload by calling `msfvenom`. Then it writes a new systemd unit whose `ExecStart` executes the payload. It tries to enable the unit by calling `systemctl enable`. Lastly, it changes the configuration of the existing reverse-shell handler to listen on port `LPORT: 4444`. This last step is what caused the syntax error: AttackMate expected a quoted string argument, not an integer.

Vibe CLI's approach is fundamentally wrong: (1) It tries to run `msfvenom` from the meterpreter shell. The tool is not available on the victim host. (2) It tries to create a new systemd unit. This fails because the compromised user does not have write permissions on the directory. (3) It tries to call `systemctl enable` to enable the new systemd unit. This requires root permissions.

The full diff for this run is published on GitHub.

Deploy-fail: `5_debiansnapshot`

This task's goal is to configure APT to only install packages from a historical snapshot. The agent receives the Debian release codename and the snapshot's timestamp, and has to implement the rest. Figure 5.6 compares the three approaches step-by-step.

Reference. The Ansible role configures APT with a single repository at `https://snapshot.debian.org/archive/debian/<timestamp>/` and empties `sources.list` so no other APT repositories remain active. Then, it configures APT to accept expired `Release` files by writing `Acquire::Check-Valid-Until "false"` into `apt.conf.d`. Finally, it refreshes the APT cache.

Claude Code. The agent approaches the task similar to the reference. It writes a new `sources.list` file containing two repositories, the main `debian` archive and `debian-security`. It empties `sources.list.d` so no other APT repositories remain active. It also configures APT to accept expired `Release` files. Claude's deployment failure is caused by the added `debian-security` URL. It assumes that `debian-security` provides a snapshot at the same timestamp as the main archive, which it does not.

Claude's approach is almost correct, and would have worked if: (1) it did not provide the `debian-security` snapshot URL at all, or (2) it provided a valid timestamp at which the `debian-security` snapshot exists.

The full diff for this run is published on GitHub.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

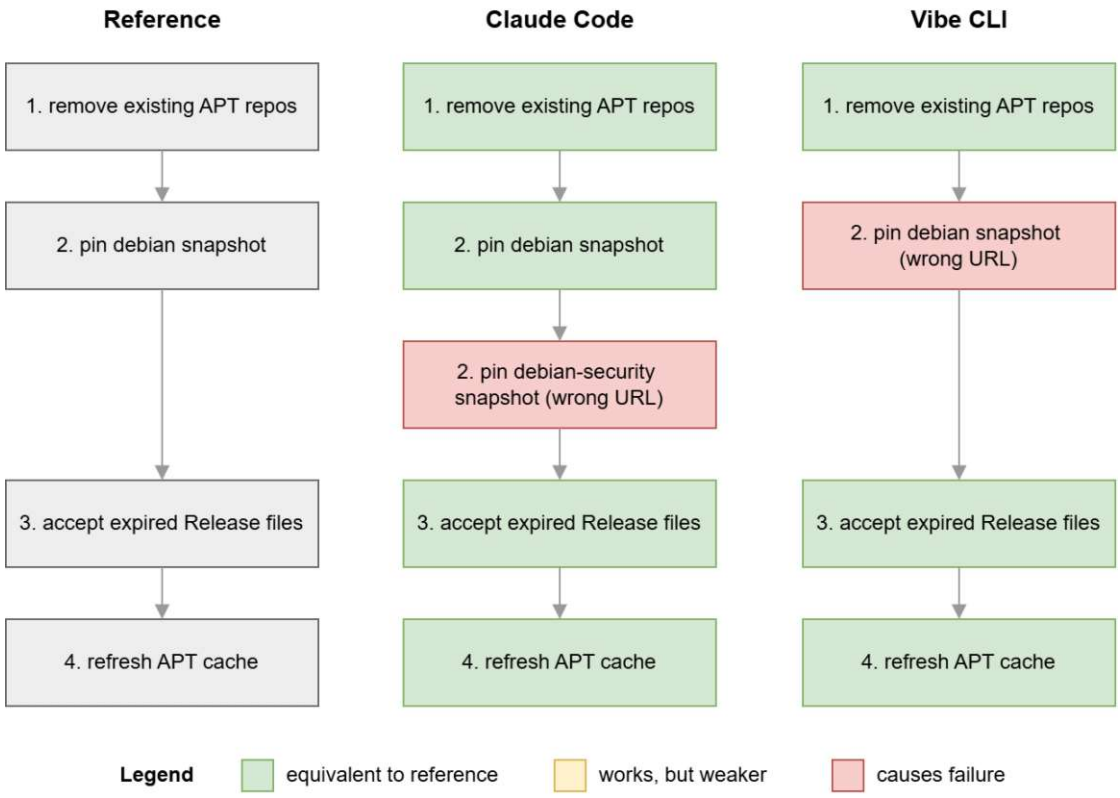

Figure 5.6: Step-by-step comparison of the three approaches to task `5_debiansnapshot`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake.

Vibe CLI. Vibe approaches the task similar to the reference. It starts by removing existing sources, and registers the new snapshot repository via the Ansible `apt_repository` module. Then it adds an `apt.conf.d` drop-in which configures APT to accept expired `Release` files. Vibe's deployment failure is caused by a wrong snapshot URL path, `/file/<timestamp>/<release>` instead of `/archive/debian/<timestamp>/`.

Vibe's approach is almost correct and would have worked with the correct URL path.

The full diff for this run is published on GitHub.

Near-miss: `18_account_persistence`

This task's goal is to establish persistence by creating a backdoor user. The agent receives an unfinished AttackMate playbook, which already established a meterpreter shell on the videoserver and escalated privileges to root. The agent must fill in the rest: Stage a payload which creates a backdoor account with root-access, and execute it with a proxy

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

command (MITRE T1218) to evade detection. Figure 5.7 compares the three approaches step-by-step.

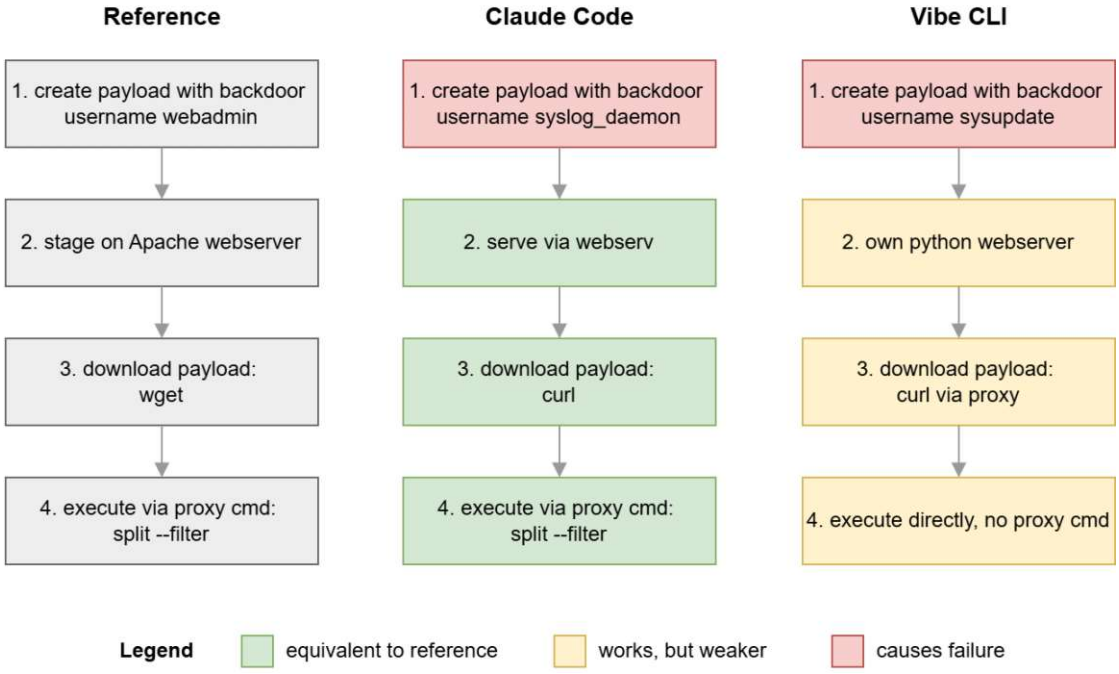

Figure 5.7: Step-by-step comparison of the three approaches to task `18_account_persistence`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake.

Reference. The scenario's Ansible setup creates the payload and stages it on the attacker's Apache server. The payload is a shell script that creates a user `webadmin`, gives it passwordless sudo permissions, and adds the attacker's key to its `authorized_keys`. In the AttackMate playbook the payload is downloaded and executed via the proxy command `split --filter="cat $FILE | xargs bash -c "`.

Claude Code. The agent writes a payload that creates a new user `syslog_daemon`, gives it uid 0 for root level access, and adds the attacker's public key to the user's `authorized_keys`. The payload is served via AttackMate's `webserv` command, downloaded to the victim host, and executed via `split --filter="bash" -l 9999 /dev/stdin`. Claude's solution deploys but fails its tests. This is caused by naming the user `syslog_daemon`, and not `webadmin`. Naming the user `webadmin` is an implicit requirement specified by the SSH verification step already present in the AttackMate script.

Claude's solution is almost correct and would have worked, had it named the user `webadmin`. Arguably, this error could be attributed to ambiguity in the prompt. The prompt only specified that an inconspicuous username should be chosen. It did not specify

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the exact name `webadmin`, but expected the agent to infer this implicit requirement from the SSH verification step already present in the AttackMate script.

The full diff for this run is published on GitHub.

Vibe CLI. The agent writes a payload that creates a new user `sysupdate`, gives it passwordless sudo permissions, and adds the attacker's key to its `authorized_keys`. It stages the payload on a Python webserver. Vibe CLI misinterprets the prompt's specification *"use a proxy command"*. It downloads the payload through an HTTP proxy and executes the payload directly. It does not use the intended proxy command (MITRE T1218) used by Claude and the reference. The solution's tests fail for the same reason as Claude's. Downstream SSH verification fails because the chosen username was not `webadmin`.

Vibe CLI's approach is similar to the reference's but differs in three places: (1) It creates its own webserver and does not reuse the Apache webserver. This works but is less clean. (2) It executes the payload directly and does not use a proxy command. This works but is more detectable than the reference. (3) It chooses a username which breaks a downstream task.

The full diff for this run is published on GitHub.

Solved: `6_sudoweakness`

The task's goal is to enable a privilege escalation via a misconfigured sudo rule. The agent must write a sudoers rule that grants passwordless sudo to a specified user on a specified binary.

Reference. The Ansible role installs the `sudo` package. Then it creates a drop-in at `/etc/sudoers.d/` which grants passwordless sudo to the specified user on the specified binary.

Claude Code. The agent installs `sudo`, and writes the sudoers rule to a drop-in at `/etc/sudoers.d/`. Claude's solution passes all tests.

Claude's approach is functionally equivalent to the reference but differs in two ways: (1) It gives the sudoers drop-in permissions `0440` instead of the reference's `0644`. (2) It validates the contents of the sudoers file via `validate: visudo -cf %s`. This is omitted in the reference.

The full diff for this run is published on GitHub.

Vibe CLI. The agent uses the exact same approach as Claude, and passes all tests. The only functional difference is that it guards `sudo` install with a `when: ansible_os_family == 'Debian'` condition.

The full diff for this run is published on GitHub.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.3.2 Takeaways

We notice three patterns across the walkthroughs.

Most failures are local, not fundamental. In seven of eight walkthroughs agents failed by making a local mistake, not because of a fundamentally wrong approach. Often these mistakes are limited to single lines of code, for instance choosing the wrong tool to edit a file (`sed -i` in task 13), picking a wrong value (snapshot URL in task 5, username in task 18), or making a formatting mistake (not quoting port number in task 13). This indicates the agents generally understood their tasks.

The biggest exception is Vibe CLI's task 13. It made several logical mistakes: it assumed offensive tools like `msfvenom` were preinstalled on the victim, and tried to invoke a root-only command `systemctl enable` from an unprivileged user. This indicates a fundamentally wrong understanding of the agent's environment.

Tasks fail the same way across agents. Tasks where both agents fail often fail for the same reason. In task 13, both agents provide an unquoted integer where AttackMate expects a string. In task 18, neither agent uses the expected backdoor username `webadmin`. In task 5, both agents provide a malformed archive snapshot URL.

Section 5.3 revealed that the evaluation-stage where a task loses most of its runs is shared between the agents. The walkthroughs show that this pattern runs even deeper and that even the exact failure causes are shared in many cases.

Correct solutions can still be better or worse than the reference. In some cases agent-generated solutions improve on the reference's solution. For instance, in task 6 Claude validates sudo rules with `visudo` and chooses better permissions for the sudoers file (`0440`). However, in many cases the agent produces a correct, but weaker solution than the reference. For instance, in task 13 Claude overwrites `ExecStart` instead of adding an `ExecStartPre` which leads to the original service not starting. Or, in task 18 Vibe CLI executes a payload directly instead of running it through a proxy command. Both attacks are functionally correct, but more detectable than the reference. Therefore, IDS datasets derived from them may be less sophisticated than those built from the reference solutions.

5.4 Discussion

The previous two sections presented the experiment's results and explained where and why tasks fail. This discussion combines the observations into practical recommendations for researchers who want to automate IDS testbed tasks with coding agents.

Ansible service roles can be delegated to agents. A service role is an Ansible role that installs and configures regular applications, for instance the `apache2` webserver. These tasks are the strongest candidate for automation among the task categories we tested. The strongest evaluated agent solves these tasks reliably, with Claude Code generating fully working solutions in 93% of runs. However, agent choice still matters,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

with Vibe CLI passing none of the 15 service-role runs. The benefit of choosing a state-of-the-art agent is large for this task category. Still, this difference cannot be generalized because only two agents were evaluated.

Keep a human in the loop for security tasks. Agents struggle with security-related tasks, specifically configuring vulnerable Ansible roles and writing AttackMate playbooks. Across 160 runs on these categories, only 40 passed all tests. This is a large drop from Ansible service role tasks. One reason for this drop is the agent's familiarity with tools. Ansible is widely used, so agents have seen it far more during training than a niche tool like AttackMate.

While agents' pass rates on security tasks sound low, this type of task is still a strong candidate for being delegated to agents. This is because the walkthroughs (see Section 5.3.1) revealed that in many cases errors were caused by local mistakes and not a fundamentally wrong approach. We recommend treating agent generated solutions as a first draft. Human experts should review these drafts, placing particular focus on interface details such as names, ports, or permissions. Agents' general approaches require less critical reviews, especially when using a state-of-the-art coding agent.

Hard tasks require human review, not more attempts. For most tasks pass@1 is either 0 or 1 (see Section 5.2.3). Retrying code generation five times improved results only for 18% of agent-task pairs. This indicates that LLM randomness is not a limiting factor in finding correct solutions. A budget of five attempts is cheap at the prices reported in Section 5.1.

We recommend retrying code generation a few times to capture some of the run-to-run variance of LLMs. But repeated sampling offers diminishing returns. Beyond a few attempts, human expert review converges on a working solution faster than further sampling. This is because agent mistakes tend to be localized, which a human reviewer can spot and fix efficiently.

Review solutions for realism, not just correctness. A functionally correct solution can still be an insufficient testbed component. In our experiment, agents sometimes produced correct but more detectable attacks. For instance, in task 13 the agent executed its attack but broke a legitimate service. More detectable attacks produce different observability data. Humans must judge whether generated attacks are realistic enough for a dataset's purpose.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Limitations and Future Work

This chapter looks back critically at the thesis. Section 6.1 examines how far the results can be trusted. Section 6.2 presents directions for further research.

6.1 Threats to Validity

The previous chapter reported the agents' results. This section examines how far those results can be trusted by following the steps that produced them.

6.1.1 From Reference to Task

A single author reverse-engineered tasks from the reference implementations. Experts later reviewed tasks to mitigate the risk of extraction errors. Still, this review is not fully independent of the original extraction. The risk of misinterpretation remains and applies equally to a task's prompt and tests.

Prompts can err in two ways. An over-specified prompt leaks parts of the solution, which makes the task easier for agents and produces false positives. An under-specified prompt withholds a requirement, which can cause capable agents to fail and produces false negatives. This tension is described in more detail in Section 3.4.2. The walkthrough of task `18_account_persistence` illustrates a failure which was potentially caused by under-specification (see Section 5.3.1). The prompt asks for a backdoor user to be created, but does not specify how to name this user. The desired username, `webadmin`, is defined implicitly in a later SSH step in the provided AttackMate playbook. If the agent does not create this user, the playbook fails. Both agents pick a different name and fail, even though their approaches are otherwise correct. Whether this is because of an under-specified prompt or an agent that failed to infer `webadmin` from the visible SSH step cannot be determined. The threat to validity remains either way.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Tests carry the same risk. Under-constrained tests accept wrong solutions and produce false positives. Over-constrained tests are modeled too closely to the reference, thereby rejecting valid alternatives and producing false negatives. To limit this, we aim to only test externally observable behavior and not internal implementation details. The walkthroughs in Section 5.3.1 indicate that this worked well, since none showed a misclassification. However, the walkthroughs only inspected eight runs and misclassification cannot be ruled out for the rest. The threat remains.

6.1.2 From Task to Score

Suppose we managed to build benchmark tasks perfectly, where every prompt and test strikes the right balance between over- and under-specification. Other factors beyond agent capability still influence final results.

A big threat is data leakage from AttackBed's public GitHub repository, which can happen in two ways. First, an agent could simply download the reference solution at runtime. We eliminate this threat by blocking outbound traffic to GitHub during agent runs. Second, and harder to rule out, is that the LLM may have been trained on AttackBed's source code. The model could have memorized the reference solutions and reproduced them during evaluation. Then the evaluation result would not reflect real agent capability. This threat cannot be ruled out, but two observations argue against it. First, memorization would manifest in generated artifacts that are near-identical to the reference solution. Yet every solution we inspected in the walkthroughs (Section 5.3.1) differed from the reference and showed no signs of reproduction. Second, AttackMate code appears only in AttackBed while Ansible is more generic and widespread. Therefore, AttackMate tasks are more likely to be memorized. However, agents perform worse on AttackMate tasks than on Ansible tasks. This indicates that memorization did not affect results.

Even with a perfect benchmark and no reference leakage, three design decisions still affect the results. (1) Generated solutions are tested on real infrastructure which can fail transiently. By re-running the evaluation of failing tasks up to three times, we make false negatives caused by flakiness negligible (below 0.1% even at a pessimistic 10% per run failure rate). (2) We sample five solutions per agent on each task. This sits within a standard range found in the literature ([JYW+23] samples only once, [KLQ+24] samples 20 times). More repetitions would capture more of the LLM's randomness. (3) We run both agents in their default configurations. This avoids tuning bias but might underestimate results possible with an optimized setup.

6.1.3 From Score to Claim

Our results are tied to AttackBed alone. This restriction is deliberate but limits how far the results generalize. Some findings should still transfer to other testbeds. This is because the configuration IaC used in AttackBed, Ansible, is used in other testbeds as well (see Section 2.1). Conclusions about Ansible tasks likely carry over to other testbeds.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

In contrast, AttackMate is used only in AttackBed. Its results are less likely to transfer to other attack-orchestration tools.

Our benchmark does not sample testbed tasks evenly. Tasks are drawn from a single scenario and are biased toward hard IDS-specific tasks. They exclude or underweight easier tasks, e.g., those already covered by general-purpose benchmarks, or one-time scaffolding tasks that do not change in day-to-day adaptation. We therefore expect that reported pass rates are likely conservative, and that an agent working on the full testbed could score higher.

Both the benchmark and the evaluation are small. The benchmark contains 19 tasks and is evaluated on two coding agents. Therefore, the measured results have limited statistical power and may not hold on a larger benchmark or other agents. Furthermore, coding agents evolve quickly, which is why the measured numbers represent a snapshot from May 2026. While the small size threatens absolute pass rates we expect the comparative findings to hold up better. The proprietary agent beats the open-weight one on 17 of 19 tasks, and both agents solve Ansible tasks more often than AttackMate tasks. We trust that ranking even if the exact percentages change on a larger benchmark or newer agents.

6.2 Future Work

The limitations above, together with gaps the thesis identified along the way, point to several directions for future work. We list the most promising ones below.

Expand the task coverage. The benchmark selects its 19 tasks from a single AttackBed scenario. Future work should add tasks from the other six AttackBed scenarios. A benchmark with more tasks would increase the results' statistical power. It would also let the benchmark sample tasks more evenly. The current selection is biased toward hard, IDS-specific tasks.

Port the construction method to other testbeds. We constructed the benchmark from tasks in AttackBed. Future work should apply the task construction method to other testbeds, such as SOCBED or CIDDS. This would test whether the results we observed hold in other testbed architectures and stacks. Additionally, it would increase benchmark size and improve the results' statistical power.

Evaluate more agents. Evaluating two agents in their default configurations only gives a snapshot. A larger pool of agents would highlight systematic differences better, e.g., how well proprietary agents fare against open-weight ones. A regularly refreshed pool of agents would track how fast capability improves. Additionally, going beyond the agent's default configuration would show how far custom prompts, skills, or tools can improve results.

Give agents a feedback loop. Agents generated solutions blindly. They never executed the generated artifacts by deploying them. The walkthroughs showed that many failures are caused by local mistakes, such as a malformed snapshot URL in task

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

`5_debiansnapshot` or an unquoted port in task `13_systemd_privesc`. Giving agents skills and tools to deploy and test generated artifacts could surface such errors and let the agent correct them. Future work should evaluate how much results can be improved by providing infrastructure access and actively encouraging the agent to test the generated code.

Evaluate dataset usefulness. Our tests accept attacks that are functionally correct but more detectable than the reference. Future work should evaluate the realism of generated components, e.g., by evaluating the monitoring data produced during attacks. This would extend functional-correctness evaluation with an assessment of whether the generated component produces a high-quality dataset for IDS research.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Conclusion

This thesis asked how well coding agents can assist in IDS testbed development. Answering this empirically required a benchmark, but none existed that contained the tasks commonly found in testbeds: configuring vulnerable infrastructure and writing attack sequences that reproducibly exploit them. Therefore, we built such a benchmark. In the process we contributed: a method for reverse-engineering benchmark tasks from existing testbeds, a 19-task benchmark derived from AIT AttackBed, an open-source pipeline for evaluating coding agents on the benchmark, and an empirical evaluation of Claude Code (Opus 4.6) and Mistral Vibe CLI (Devstral 2) across 190 runs.

Our comparison identifies four aspects recurring across IDS testbeds: infrastructure, services/vulnerabilities, benign user activity, and attacks. These aspects map directly to the main testbed development tasks. Simulation of benign user activity is excluded from the benchmark because it is not modeled in AttackBed. Of the remaining three aspects, infrastructure setup is comparatively stable and effort concentrates on the other two: configuring new services and vulnerabilities (via Ansible roles) and scripting attacks that exploit them (via AttackMate playbooks). Our benchmark evaluates these two task families.

Evaluation showed that coding agents generate functionally correct IaC, but results vary by agent and task category. Claude Code generated functionally correct solutions in 43% of its runs, while Vibe CLI managed only 14%. The gap is task-dependent: Claude correctly configured regular services via Ansible on 93% of runs, but configuring vulnerabilities or exploiting them remained challenging for both agents.

Comparison of agent-generated components to the manually designed reference showed how agents fail. Failures are rarely fundamental. In most of the analyzed cases, agents chose a correct approach but made localized mistakes, e.g., by picking a wrong URL or username. Artifacts sometimes improve on the reference, e.g., by tightening permissions or adding validation. But agents also produce artifacts that are functionally correct but

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

less professional, e.g., by scripting an attack that is more detectable. Therefore, coding agents should not be viewed as autonomous IDS testbed authors but rather as useful assistants. Tasks that configure regular services can be delegated and require only spot checks. Tasks that configure vulnerable infrastructure or script attacks need a human in the loop who reviews generated code for functional correctness and whether the generated artifacts would yield good IDS datasets.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Generative AI Tools Used

I used generative AI in the following ways. All output was reviewed and verified, and I take full responsibility for this thesis' content.

- **Learning and Brainstorming**: I used Google Gemini (2.5 Flash, 3.1 Pro, 3.5 Pro) chat to brainstorm and build intuition about new concepts.
- **Development**: I used Anthropic's Claude Code with Sonnet and Opus models to assist development. This focused on one-time use cases, such as building tooling to inspect data and check hypotheses. For core parts of the thesis I took a more hands-on approach to architecture and core components, but got assistance when implementing certain functions.
- **Writing**: I wrote the text myself but used Anthropic Claude to smooth out my writing, simplify language, and enhance readability.
- **Proofreading**: I used Anthropic Claude to check the finished text for spelling, grammar, and punctuation mistakes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

3.1 Survey results. Each cell is a reviewer's Likert rating of one requirement on one dimension. The top half covers Ansible role tasks, the bottom half covers AttackMate tasks. 19

4.1 End-to-end pipeline. `run` sets up the task-repository, starts an agent that edits it, and saves the agent's trajectory. `eval` consumes the edited AttackBed and the evaluation-config, dispatches the configured evaluation runner which deploys artifacts on OpenStack and executes functional tests. Finally an evaluation result file is saved. 25

5.1 Results across all 95 runs per agent, bucketed by the stage at which each run fails. Each run lands in exactly one bucket. 33
5.2 Results per task per agent, bucketed by the stage at which each run fails. Each run lands in exactly one bucket. Tasks are ordered by outcome. . . 34
5.3 Pass rate by task category. 36
5.4 Pass@1 and pass@5 of each task. 37
5.5 Step-by-step comparison of the three approaches to task `13_systemd_privesc`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake. 39
5.6 Step-by-step comparison of the three approaches to task `5_debiansnapshot`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake. 41
5.7 Step-by-step comparison of the three approaches to task `18_account_persistence`. Colors show divergence from the reference: green indicates a solution equivalent to the reference; yellow indicates a solution that is functionally correct but less professional than the reference; red indicates a mistake. 42

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

2.1 Comparison of five representative IDS testbeds. 8

3.1 Ansible roles selected as benchmark tasks. 16
3.2 AttackMate tasks selected as benchmark tasks. 17
3.3 Requirement R4 of the `logrotten` Ansible role at three levels of abstraction. 18

5.1 Pinned versions of tools and agents. 32
5.2 Results over five runs per task, per agent. Metrics definitions in Section 3.5.2. 35
5.3 Group assignment per task and agent. 38

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[Ali10] Karim Ali. Algorizmi: A configurable virtual testbed to generate datasets for offline evaluation of intrusion detection systems. 2010.

[AMS+16] Andy Applebaum, Doug Miller, Blake Strom, Chris Korban, and Ross Wolf. Intelligent, automated red team emulation. In *Proceedings of the 32nd annual conference on computer security applications*, pages 363–373, 2016.

[Ant26] Anthropic. Claude opus 4.6 system card, Feb 2026. Accessed: 2026-05-19.

[ISO18] ISO/IEC/IEEE. ISO/IEC/IEEE 29148:2018 – systems and software engineering – life cycle processes – requirements engineering. Standard ISO/IEC/IEEE 29148:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[JYW+23] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[KLQ+24] Patrick T Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M Park, George S Elengikal, Yuxin Kang, et al. Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs. *Advances in Neural Information Processing Systems*, 37:134488–134506, 2024.

[LHB+26a] Max Landauer, Wolfgang Hotwagner, Thorina Boenke, Florian Skopik, and Markus Wurzenberger. Attackmate: Realistic emulation and automation of cyber attack scenarios across the kill chain. *arXiv preprint arXiv:2601.14108*, 2026.

[LHB+26b] Max Landauer, Wolfgang Hotwagner, Thorina Boenke, Florian Skopik, and Markus Wurzenberger. Cam-lds: Cyber attack manifestations for automatic interpretation of system logs and security alerts. *arXiv preprint arXiv:2603.04186*, 2026.

[LHB+26c] Max Landauer, Wolfgang Hotwagner, Thorina Boenke, Florian Skopik, and Markus Wurzenberger. Cyber attack manifestations - log data set (CAM-LDS), 2026.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[LSF+22] Max Landauer, Florian Skopik, Maximilian Frank, Wolfgang Hotwagner, Markus Wurzenberger, and Andreas Rauber. Maintainable log datasets for evaluation of intrusion detection systems. *IEEE Transactions on Dependable and Secure Computing*, 20(4):3466–3482, 2022.

[LSW+20] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. Have it your way: Generating customized log datasets with a model-driven simulation testbed. *IEEE Transactions on Reliability*, 70(1):402–415, 2020.

[Mis25] Mistral AI. Introducing devstral 2 and mistral vibe cli, Dec 2025. Accessed: 2026-05-19.

[OWJ+22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[RWG+17] Markus Ring, Sarah Wunderlich, Dominik Grüdl, Dieter Landes, and Andreas Hotho. Flow-based benchmark data sets for intrusion detection. In *Proceedings of the 16th European Conference on Cyber Warfare and Security (ECCWS)*, pages 361–369. ACPI South Oxfordshire, UK, 2017.

[SDYD+23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

[SLA+25] Brian Singer, Keane Lucas, Lakshmi Adiga, Meghna Jain, Lujo Bauer, and Vyas Sekar. Incalmo: An autonomous llm-assisted system for red teaming multi-host networks. *arXiv preprint arXiv:2501.16466*, 2025.

[SLG+18] Iman Sharafaldin, Arash Habibi Lashkari, Ali A Ghorbani, et al. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1(2018):108–116, 2018.

[SSTG12] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374, 2012.

[TBSI+26] Natalia Tarasova, Enrique Balp-Straffon, Aleksei Iancheruk, Yevhenii Sielskyi, Nikita Kozodoi, Liam H Byrne, Jack Butler, Dayuan Jiang, Marcin Czelej, Andrew Ang, et al. Swe-infrabench: Evaluating language models on cloud infrastructure code. *arXiv preprint arXiv:2606.05249*, 2026.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[Tea25] The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025.

[UHH+21] Rafael Uetz, Christian Hemminghaus, Louis Hackländer, Philipp Schlipper, and Martin Henze. Reproducible and adaptable log data generation for sound cybersecurity experiments. In *Proceedings of the 37th annual computer security applications conference*, pages 690–705, 2021.

[VOČ+17] Jan Vykopal, Radek Ošlejšek, Pavel Čeleda, Martin Vizvary, and Daniel Tovarňák. Kypo cyber range: Design and use cases. 2017.

[YJW+24] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[YZY+22] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[ZKB+25] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, et al. Cve-bench: a benchmark for ai agents' ability to exploit real-world web application vulnerabilities. *arXiv preprint arXiv:2503.17332*, 2025.

[ZPD+25] Andy K Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, et al. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *International Conference on Learning Representations*, volume 2025, pages 25094–25243, 2025.

[ZPZ+25] Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. Deployability-centric infrastructure-as-code generation: An llm-based iterative framework. *arXiv e-prints*, pages arXiv–2506, 2025.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN