Degree Project in Cybersecurity

Second cycle, 30 credits

Automated Validation and Remediation of Security Misconfigurations in Kubernetes Deployments

Development and Comparative Cross-Model Evaluation of an Agentic LLM-Driven Static Analysis and Remediation Pipeline for Helm Charts

ALEX SHARIAT

Automated Validation and Remediation of Security Misconfigurations in Kubernetes Deployments

Development and Comparative Cross-Model Evaluation of an Agentic LLM-Driven Static Analysis and Remediation Pipeline for Helm Charts

ALEX SHARIAT

Date: June 24, 2026

Supervisor: Yufei Liu
Examiner: Hamed Nemati

Host company: Finansinspektionen
Swedish title: Automatiserad validering och åtgärdande av inkorrekta säkerhetskonfigurationer i Kubernetes-driftsättningar
Swedish subtitle: Utveckling och utvärdering av modeller i en agentbaserad LLM pipeline för statisk analys och åtgärdande av Helm-charts

© 2026 Alex Shariat

Abstract

This thesis investigates whether large language models, operating as an autonomous agent with access to file modification tools, can reliably remediate security findings in Helm charts, and under what conditions, with what limitations, and at what cost. Following a Design Science Research methodology, the work designs, implements, and empirically evaluates an agentic validation and remediation pipeline that uses static analysis from Trivy and Checkov with iterative LLM-driven remediation, deterministic post-remediation verification and schema validation, and structured behavioral-change classification. A key finding emerged during development, constraining the agent to read- and write-only operations, while front loading manifests, scanner-derived fix hints, and a structured task list into the initial context, produced superior results with higher fix rates and improved usage of tokens than granting unrestricted agentic tool access.

The pipeline was evaluated across twenty Helm charts of varying structural complexity using six models from the GPT-5 family, totaling 120 runs and yielding a median instance-level fix rate of 95.7 percent on finding-ID to target-file pairs. Outcomes were distributed bimodally, most runs resolved nearly all findings, while failures clustered at the pipeline level rather than exhibiting gradual quality degradation, and were attributable to failure modes spanning structural YAML corruption, template syntax violations, context-window saturation, and values-schema breakage. GPT-5 and GPT-5.4 were the most consistent performers across charts in both accuracy and reliability, whereas smaller tier variants exhibited markedly higher variance. Agent self-reporting was systematically conservative, with 61.5 percent of runs underclaiming relative to scanner verified ground truth. The thesis contributes a working security validation and remediation pipeline, an empirically grounded design pattern in which constrained agency outperforms full agency, and empirical evidence on the accuracy, reliability, self-report accuracy, and cost boundaries of LLM-driven infrastructure-as-code remediation across OpenAI API models.

Keywords

Kubernetes security, Helm charts, Container security, DevSecOps, Static analysis, Security validation, Automated remediation, Large language models, Agentic AI, LLM agents, Infrastructure as code, OpenAI API

Sammanfattning

Denna avhandling undersöker om stora språkmodeller som agerar som autonoma agenter med tillgång till filmodifieringsverktyg, tillförlitligt kan åtgärda säkerhetsbrister i Helm charts, och under vilka förutsättningar, med vilka begränsningar och kostnaden. Med utgångspunkt i en Design Science Research-metodik designar, implementerar och utvärderar arbetet empiriskt en agentbaserad validerings- och åtgärdspipeline som kombinerar statisk analys från Trivy och Checkov med iterativ LLM-driven åtgärd, deterministisk verifiering och schemavalidering, samt strukturerad klassificering av beteendeförändringar. Ett resultat framkom under utvecklingen, att begränsa agenten med huvudsakligen läs- och skriv-operationer tillsammans med åtgärdsförslag och en strukturerad instruktionslista i den initiala kontexten gav bättre resultat med högre åtgärdsfrekvens och förbättrad tokenförbrukning än att ge obegränsad agentbaserad verktygsåtkomst.

Pipelinen utvärderades på tjugo Helm charts av varierande strukturell komplexitet med sex modeller från GPT-5-familjen, totalt 120 körningar, och uppnådde en median instansbaserad åtgärdsfrekvens på 95,7 procent över par av ID och målfil. Utfallen var bimodalt fördelade, de flesta körningar löste nästan alla fynd, medan misslyckande körningar klustrade på pipelinenivå snarare än att uppvisa gradvis kvalitetsförsämring, och felorsaker kunde tillskrivas som strukturell YAML-korruption, templatesyntax förstörelse, kontextfönstermättnad, och förstörelse av values-schemat. GPT-5 och GPT-5.4 var de mest konsekventa modellerna över olika charts i både precision och tillförlitlighet, medan mindre varianter uppvisade markant högre varians. Agentens självrapportering var systematiskt konservativ, där 61,5 procent av körningarna underrapporterade relativt skannerverifierad sanning. Avhandlingen bidrar med en fungerande pipeline för säkerhetsvalidering och åtgärd, ett empiriskt grundat designmönster där begränsad agentförmåga överträffar full agentförmåga, samt empirisk evidens om träffsäkerhet, tillförlitlighet, självrapporteringsnoggrannhet och kostnadsgränser för LLM-driven åtgärd av infrastructure-as-code över OpenAI API-modeller.

Nyckelord

Kubernetessäkerhet, Helm charts, Containersäkerhet, DevSecOps, Statisk analys, Säkerhetsvalidering, Automatiserad åtgärdning, Stora språkmodeller, Agentisk AI, Språkmodellsagenter, Infrastruktur som kod, OpenAI API

Acknowledgments

This thesis would not have come together without the support and guidance of a number of people, and it is a pleasure to thank them.

I owe my deepest thanks to my KTH supervisor, Yufei Liu whose guidance throughout this project has been exceptionally valuable. Your feedback helped shape the direction of the research and the quality of project, ranging from technical detail to the higher-level framing of the thesis, and made an enormous difference. I am very grateful for the time and interest you brought to the discussions.

I would also like to thank my examiner, Hamed Nemati, for your oversight and for the constructive feedback provided throughout the project.

This work was carried at Finansinspektionen, the Swedish Financial Supervisory Authority, and I am deeply grateful to my industry supervisor at FI, Kent Magnusson, for the autonomy granted throughout the project and the consistent support in navigating the practical and organisational dimensions of the work. It has been a pleasure to have the opportunity to develop and evaluate this pipeline in a real operational context.

My thanks also go to my colleagues at FI, Rodrigo Carrasco, Susanne Lindström, and Sara Jalilian, who took the time to discuss ideas, technical questions, and offer feedback. The everyday conversations and guidance of the team made the work more enjoyable.

Stockholm, June 2026
Alex Shariat

Contents

1 Introduction **1**
1.1 Background . . . 2
1.2 Problem . . . 4
1.3 Purpose . . . 5
1.4 Goals . . . 5
1.5 Research Methodology . . . 6
1.6 Delimitations . . . 7
1.7 Ethics and Sustainability . . . 7
1.8 Structure of the Thesis . . . 7

2 Background **9**
2.1 Kubernetes and Container Orchestration . . . 9
2.1.1 Container Fundamentals . . . 9
2.1.1.1 Process Isolation and Namespaces . . . 9
2.1.1.2 Container Images and Registries . . . 10
2.1.1.3 Why Containers Over Virtual Machines: Portability and Consistency . . . 11
2.1.2 Kubernetes Cluster Architecture . . . 11
2.1.2.1 Control Plane Components . . . 12
2.1.2.2 Worker Node Components . . . 13
2.1.2.3 Core Resources: Pods, Deployments, Services, ConfigMaps, and Secrets . . . 13
2.1.3 Helm Charts and Package Management . . . 14
2.1.3.1 Helm in Kubernetes Ecosystems . . . 14
2.1.3.2 Chart Structure . . . 14
2.1.3.3 Helm Templating Engine . . . 15
2.1.3.4 Values and Overrides . . . 15
2.1.3.5 Helm Chart Security Considerations . . . 16
2.2 Kubernetes Security . . . 16

2.2.1 Security Model and Threat Landscape 16
2.2.1.1 The 4Cs Model: Code, Container, Cluster, Cloud 16
2.2.1.2 Common Attack Vectors 17
2.2.2 Workload Security Configuration 18
2.2.2.1 Security Contexts 18
2.2.2.2 Resource Management 19
2.2.3 Access Control and Network Segmentation 20
2.2.3.1 Role-Based Access Control (RBAC) 20
2.2.3.2 Network Policies 21
2.2.4 Common Misconfigurations 22
2.2.4.1 Empirical Studies on Misconfigurations in Kubernetes Environments 22
2.2.4.2 Systematisation of Security Practices 23
2.3 Security Validation Approaches 24
2.3.1 Static Analysis Fundamentals 24
2.3.1.1 Static vs. Dynamic Analysis and Limitations 25
2.3.2 Kubernetes Static Analysis Tools 26
2.3.2.1 Public Helm Chart Repositories 26
2.3.3 Validation Methods 26
2.3.3.1 Traditional Rule-Based Checking 26
2.3.3.2 Requirements-Based Validation 27
2.3.4 From Detection to Remediation 27
2.4 Large Language Models for Automated Code Remediation . . 28
2.4.1 Agentic AI Pipelines and Tool Use 29
2.4.1.1 Automated Remediation of Infrastructure as Code Misconfigurations 30
2.4.2 Prompt Engineering for Remediation Tasks 30
2.4.3 Known Failure Modes: Hallucination, Drift, and Self-Report Gaps 31
2.4.4 Single-Agent vs. Multi-Agent Architectures 32
2.4.5 Model Selection: API-Hosted vs. Locally Deployed Models 33
2.4.6 Verification and Ground Truth in LLM-Based Remediation 33
2.5 Related Work 34
2.5.1 Previous Studies on Kubernetes Security 34
2.5.2 Static Analysis for Container Configurations 34
2.5.3 Automated Analysis and Repair of Helm Charts 35

2.5.4 LLM-Based Automated Program Repair 35
2.5.5 LLM-Based Kubernetes Misconfiguration Detection and Remediation 36

3 Method **39**
3.1 Research Process 40
3.2 Research Paradigm 41
3.3 Data Collection 41
3.3.1 Sampling 41
3.3.2 Sample Size 42
3.4 Experimental Design and Planned Measurements 42
3.4.1 Test Environment 42
3.4.2 Hardware and Software 42
3.5 Pipeline Architecture 43
3.5.1 Initialisation and Environment Setup 43
3.5.2 Tool Definitions 44
3.5.3 Pre-Agent Scanning and Analysis 46
3.5.4 Agent Prompt Engineering 50
3.5.5 Agentic Remediation Loop 51
3.5.5.1 Agentic scope restriction 55
3.5.6 Post Remediation Phase 57
3.5.6.1 Second-Pass Remediation 58
3.5.6.2 Adversarial Review and Remediation 59
3.5.7 Deterministic Verification and Report Generation . . . 63
3.6 Assessing reliability and validity of the data collected 66
3.6.1 Validity of method 66
3.6.2 Reliability of method 67
3.6.3 Data validity 68
3.6.4 Reliability of data 68
3.7 Evaluation framework 68
3.7.1 API Models Used in Comparison 69

4 Results and Analysis **71**
4.1 Pipeline Configurations 71
4.2 Remediation Pipeline Results 73
4.2.1 Fix Rates Across Chart–Model Pairs 74
4.2.2 Instance-Level vs. Unique-ID-Level Fix Rates 75
4.2.3 Model Comparison 75
4.2.3.1 Fix Rate Distribution 76

4.2.3.2 Model Consistency Across Charts 77
4.2.3.3 Cost per Fixed Instance 78
4.2.3.4 Token Consumption, Cost, and Fix Rate . . 79
4.2.4 Failure Modes 80
4.2.4.1 The Mediocre Runs of `snappass` 81
4.2.5 Self-Report Accuracy and Discrepancies 83
4.2.6 Validation Gate Analysis 85
4.2.6.1 Functional Equivalence and Behavioural Change 87
4.2.6.2 Validation Pass Rates by Model 88
4.2.7 Behavioural Classification of Changes 89
4.3 Retrying Failed Runs with Higher Reasoning Effort 90
4.3.1 Implications for Pipeline Deployment 94
4.4 Reliability Analysis 94
4.5 Validity Analysis 95

5 Discussion **97**
5.1 Restricted Agency Over Full Tool Access 97
5.2 Pipeline Capabilities and Boundaries 98
5.2.1 Addressing RQ1 99
5.3 Cross–Model Comparison and the Cost–Performance Factor . 100
5.3.1 Addressing RQ2 101
5.4 The Self-Report Gap 102
5.5 Behavioural Change and the Functional Equivalence 103
5.6 Threats to validity 103

6 Conclusions and Future work **105**
6.1 Conclusions 105
6.2 Limitations 106
6.3 Future Work 107

References **109**

List of Figures

1.1 High-level overview of the agentic LLM remediation pipeline. A Helm chart is scanned by Trivy and Checkov, producing a classified task list of findings. An LLM agent then iteratively applies fixes through a constrained write_file interface, with deterministic checks running after every modification. The remediated chart is verified through deterministic scanning, adversarial red-team review, and a structured verification report that classifies each change by risk tier and produces a deployment readiness verdict. 4

2.1 Kubernetes cluster components. 12

3.1 Initialisation phase of the pipeline. Three input sources are resolved. The environment file supplies the OpenAI API key and model parameters to the API client, the requirements file is parsed into a not-applicable list and a requires-manual-review list encoding deployment-context policy rules, and the chart source path is used to establish an isolated sandbox working directory. The outputs of all three paths are consolidated into a single execution state object carrying the client handle, policy requirements, audit log reference, working directory path, and runtime constants being model identifier, token budget, iteration caps for the pipeline run. . . 44

3.2 Tool definitions and sandbox boundary. The LLM agent issues function calls through a JSON tool dispatcher, invocations are routed through a sandboxed execution layer including path validation, extension filtering, size limiting, and audit logging. Within the sandbox, six tools are available, `helm_template`, `run_trivy`, `run_checkov`, `read_file`, `write_file`, and `list_files`. 45

3.3 Pre-agent scanning and task list construction. The original Helm chart is rendered with `helm-template` and scanned in parallel by Trivy and Checkov, producing two sets of vulnerability findings. The combined findings are classified against the organisational policy rules into three categories, agent-fixable findings that proceed to remediation, not-applicable findings that are excluded, and findings flagged for manual review that are passed to the final report. Agent-fixable findings enter the task list builder, which groups them by source file, inspects the template structure to determine the appropriate fix pattern, and generates concrete fix hints. The structured task list is embedded in the agent's system prompt as the primary remediation brief. 49

3.4 Main agentic remediation loop. The agent receives an initial context containing file contents and the structured task list, then enters an iterative loop. On each turn, the agent either issues tool calls or produces a text response. If the token budget is exceeded, the loop terminates immediately. Turns that include at least one write operation trigger the post-write validation stage, which runs the violation checker and missing-fix checker, auto-restores files where violations are detected, and performs an exhaustive integrity check. The combined checker output is returned to the agent as a feedback message, and the loop continues. If the write-round ceiling is reached, the loop terminates. When the agent produces a text response, the pipeline checks whether it contains the required JSON summary, if not, the agent is prompted again. Once a valid summary is extracted, reporting findings fixed, unfixable, files modified, and a GO/NO-GO deployment decision, the loop exits and the summary is retained for comparison against ground-truth scanner results. 53

3.5 Constrained agentic remediation loop with read-write file only tool access. The loop structure is identical to Figure 3.4, but 4 of the six tools are disabled. This restriction prevents the agent from consuming iteration and token budget on re-rendering or re-scanning files whose contents were already supplied in the initial context. Testing showed this constrained configuration was substantially better than the fully agentic alternative. . . . 57

3.6 Second remediation pass. If more than two fixable findings remain, a second remediation pass is triggered, otherwise the pipeline proceeds directly to adversarial review. The second pass receives a narrowed context containing only the affected files and a focused task list restricted to outstanding findings, reducing token consumption and preventing the agent from revisiting already-corrected files. The agentic loop runs under the same constraints as the first pass, and its metrics are merged with the first-pass results before the combined output is forwarded to the adversarial review stage. 59
3.7 Adversarial review and third remediation pass. LLM agent operating under a red-team reviewer persona assesses each change across five dimensions. Changes classified as REVERT, ADJUST, or ACCEPT. If any REVERT or ADJUST items are identified, a third and final remediation pass is triggered, in which a scope-limited agentic loop applies the reviewer's specific instructions while preserving all security improvements. If no actionable items are found, or once the third pass completes, the combined results are forwarded to the deterministic verification and report generation stage. 62
3.8 Deterministic verification and report generation. The final chart is subjected to integrity checks. Final Trivy and Checkov scans and their results compared against the pre-remediation baseline, tracking fixed, partially fixed, and remaining findings at both unique-ID and instance level. An LLM auditor review traces each fix to its corresponding diff, assigns risk tiers, and classifies behavioural changes, a deterministic diff coverage check verifies that every modified file was addressed. Functional validation, kubeconform schema validation, and functional equivalence comparison is done as well. The pipeline concludes by producing a structured JSON report and a Markdown report for human review. 65

4.1 Instance level fix-rate per chart for `GPT-5` and `GPT-5.4` under the constrained (restricted tool use) and fully agentic (all tools) configurations. 72

4.2 Per-chart performance of the constrained (green dots) and fully agentic (red dots) pipelines on `GPT-5` and `GPT-5.4`, showing instance level fix-rate and total token consumption. Each dot represents one chart. 73
4.3 Fix rates at instance level (left) and unique-ID level (right) for all 20 chart–model combinations across six models and 120 runs. Green indicates high fix rates, orange indicates medium fix rates, and red indicates low or zero. 74
4.4 Instance-level fix rate per chart run for each model. Each dot represents one chart–model combination. The horizontal bar denotes the mean. 76
4.5 Consistency of fix rates across charts per model, measured as standard deviation (σ) and coefficient of variation (CV). Lower values indicate more predictable performance. 77
4.6 Cost per fixed instance (USD) for each model, annotated with total tokens consumed per fixed instance. 78
4.7 Average tokens per run (blue bars, left axis), average cost per run in USD (orange bars, left axis), and average fix rate (green line) for each model. 79
4.8 Distribution of self-report error (claimed fixes - ground-truth fixes) across runs. Negative values indicate underclaiming, positive values indicate overclaiming. 83
4.9 Agent self-reported fix count versus ground-truth fix count from deterministic re-scanning, per run. Points on the dashed diagonal indicate perfect self-report accuracy. Points below the diagonal represent underclaiming and points above represent overclaiming. 85
4.10 Validation gate pass rates across all 119 pipeline runs. 86
4.11 Validation gate pass rates by model. 88
4.12 Behavioural change classification per chart–model combination. Each bar shows the proportion of applied changes classified as additive-only (A, green), behavioural modification (B, blue), or values override (C, red). Five runs produced no classifiable changes and appear as empty rows. 89
4.13 Instance-level fix rates for the 12 chart–model pairs that scored 0% in the primary evaluation, re-run with `reasoning_effort=medium`. 91
4.14 Per-model mean fix rate, run 1 only versus run 1 with the run-2 retries overlaid. 92
4.15 Token usage and duration per pair, run-1 versus run-2. 93

List of Tables

4.1 Worst overclaim cases. 84
4.2 Validation gate pass rates across all pipeline runs. 86
4.3 Per-model validation gate pass rates. 88
4.4 Per-model mean instance fix rate before and after the run-2 overlay. 92
4.5 Mean instance-level fix rate per model with 95% confidence intervals. Run-1 figures use a single run per chart at default reasoning effort. Post-retry figures overlay the run-2 results, in which the run-1 zero-rate cells were re-executed at `reasoning_effort=medium`. 93

Chapter 1

Introduction

Kubernetes has become the dominant platform for deploying containerised applications, and Helm charts have emerged as the standard packaging format for defining, installing, and managing Kubernetes workloads. In regulated industries, every deployment must satisfy security requirements derived from established hardening benchmark, such as containers should run as non-root users, file systems should usually be read-only, resource limits must be set, and privileges must be minimised. Static analysis tools can scan Helm charts and identify misconfigurations that violate these requirements, often producing dozens if not hundreds of findings per chart. However, remediation today is performed manually by engineers who must understand the chart's template structure, determine where each fix should be applied, and verify that the change does not break the application. This manual process is time-consuming, error-prone, and does not scale as the number of charts and findings grows.

The immediate, industry-specific problem that motivates this thesis comes from Finansinspektionen (FI), Sweden's financial supervisory authority. FI maintains a Kubernetes infrastructure where security is paramount for deployments. The volume of scanner findings across charts creates a remediation bottleneck: engineers spend significant time interpreting findings, applying fixes, and verifying that changes do not alter the application's intended behaviour. FI's interest is in a tool that can automate as much of this remediation as possible while clearly marking what still requires human judgement.

However, framing this thesis as building a tool for one organisation would limit its contribution to engineering practice. The broader question investigated here is whether large language models (LLMs), operating as

autonomous *agents* with access to file modification tools, can reliably remediate security findings in infrastructure-as-code configurations — and if so, under what conditions, with what limitations, and at what cost. This question is relevant beyond FI's specific context because every organisation that uses Kubernetes faces the same gap between detection and remediation, and the rapid development of LLM capabilities makes automated remediation an increasingly realistic prospect. By constructing a pipeline that uses LLMs as the reasoning engine and empirically evaluating it across multiple models and charts, this thesis produces findings about the effectiveness, reliability, and safety boundaries of agentic LLM-driven remediation — findings that inform practitioners considering adoption and researchers studying LLM capabilities in structured code modification tasks.

1.1 Background

Kubernetes orchestrates containerised workloads through declarative configuration files that specify how applications should be deployed, scaled, and secured. Helm charts package these configurations into reusable, parameterised templates, allowing organisations to deploy complex applications by customising a values file rather than writing raw Kubernetes manifests from scratch. This is powerful but introduces complexity: security-relevant settings are scattered across multiple template files, conditionally gated by Helm control structures, and parameterised through layered value references. A single security finding may require changes in multiple files depending on the chart structure and template logic.

Static analysis tools have matured to the point where they can reliably detect most common Kubernetes misconfigurations, covering a wide range of security findings from missing resource limits and overly permissive security contexts to absent network policies and unpinned image tags. While coverage gaps remain, particularly for semantic and runtime-dependent issues, the detection side of the problem is comparatively well-supported.

The remediation side is not. Applying a fix requires understanding not only what the scanner expects but also how the chart is structured.

Helm charts use three distinct patterns for security configuration. Some charts fully render a values block into the template through expressions such as *toYaml* or *nindent*, requiring fixes only in values.yaml (**Pattern A**). Others use individual `.Values` field references, requiring a merge of existing and new fields (**Pattern B**). A third group contains no values references at all, requiring direct template modification (**Pattern C**). Applying the wrong pattern, for instance, hardcoding a value in a template that already references `.Values` creates a conflict that either breaks rendering or silently overrides user configuration.

This structural complexity is what makes automated remediation difficult and what makes it interesting as an engineering research problem.

Advances in large language models, particularly in function calling and tool use have created new possibilities, an LLM agent that reads chart files, understands the template patterns, applies appropriate fixes through file writes, and receives feedback on the correctness of its modifications. Unlike traditional code generation, this is an *agentic* task — the model must plan a sequence of actions, handle multi-file dependencies, and adapt to feedback from deterministic validation checks. This differs from existing rule-based auto-remediation tools, which apply predefined patches to known finding patterns but cannot reason about the chart's templating structure or choose between alternative fix strategies. The agentic approach would handle charts whose structure was not anticipated at rule-authoring time, at the cost of introducing the new validity concerns examined in this thesis.

The question is not whether LLMs can generate plausible-looking YAML, but whether an agentic pipeline built around them can produce fixes that are verifiably correct, functionally equivalent, and safe to deploy. Figure 1.1 shows the overall high-level structure of the pipeline developed in this thesis to investigate this question.

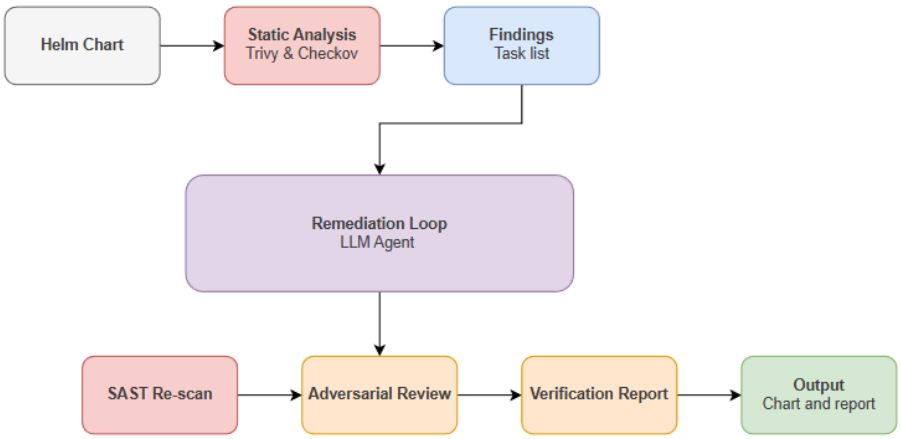

Figure 1.1: High-level overview of the agentic LLM remediation pipeline. A Helm chart is scanned by Trivy and Checkov, producing a classified task list of findings. An LLM agent then iteratively applies fixes through a constrained write_file interface, with deterministic checks running after every modification. The remediated chart is verified through deterministic scanning, adversarial red-team review, and a structured verification report that classifies each change by risk tier and produces a deployment readiness verdict.

1.2 Problem

The original problem, as defined by FI, is operational: given a Helm chart and a set of security findings from static analysis tools, produce a remediated chart that resolves as many findings as possible without altering the application's intended behaviour. The solution should work across diverse chart architectures, handle the structural complexity of Helm templating, and clearly separate what has been automated from what still requires human review. The pipeline itself demonstrates how an agentic approach can bridge detection and remediation, and the empirical evaluation reveals how effective the pipeline is in agentic security workflows and where autonomous remediation breaks down. To investigate the broader engineering research question motivated, two research questions guide this work:

RQ1: To what extent can an agentic LLM pipeline autonomously remediate security findings in Helm chart configurations, and what are the reliability and safety boundaries of that remediation?

RQ2: How do different API models compare in remediation accuracy,

consistency, coverage, and self-report accuracy when used as the reasoning engine within the same agentic security validation pipeline?

1.3 Purpose

The purpose of this thesis is to design, develop, and empirically evaluate an agentic pipeline that uses large language models to autonomously remediate security findings in Kubernetes Helm charts. The pipeline developed here is the central research artefact of the thesis.

The novelty of the approach lies in combining an agentic LLM remediator with a multi-layer verification architecture that integrates deterministic ground-truth re-scanning, schema validation, functional equivalence diffing, an adversarial red-team review, and a structured verification report producing risk-tiered fix tracing and deployment readiness verdicts.

This combination is intended to bridge the gap between detection and remediation while maintaining the safety guarantees that a regulated environment demands, producing a defence-in-depth strategy in which the deterministic layers provide evidence and the LLM-based layers provide complementary judgement.

The evaluation aims to produce findings about the capabilities and limitations of LLM-driven remediation that are useful to both practitioners considering adoption and researchers studying LLM capabilities in structured code modification tasks.

1.4 Goals

The concrete goals of the thesis are fourfold. First, to construct a working agentic remediation pipeline that accepts a Helm chart and scanner findings as input, applies fixes autonomously through a tool interface, and validates the results through multiple quality gates including deterministic re-scanning, functional equivalence checking, and adversarial review. Second, to evaluate the pipeline's effectiveness by measuring fix rates across a corpus of real Helm charts of varying complexity, and to characterise where autonomous remediation breaks down by looking at which finding types resist automation, which chart structures defeat the agent, and how often the agent's self-assessment diverges from ground truth. Third, to compare the performance of OpenAI models within the same pipeline architecture, producing empirical data on accuracy, consistency, token efficiency, and self-report accuracy

rate across model tiers and isolating differences attributable to the model itself rather than to pipeline-level confounds. Fourth, to frame the limits of autonomous remediation, looking at the finding types, chart structures, and configuration patterns that resist automation as empirical findings about LLM-based remediation rather than as implementation defects to be fixed.

1.5 Research Methodology

This project follows a Design Science Research (DSR) methodology, applicable for studies that both construct a practical artefact and empirically evaluate it. DSR emphasises iterative cycles of design, implementation, and evaluation, producing not only a working tool but also generalisable knowledge about its effectiveness. The research proceeds through three phases: requirements analysis, design and implementation, and empirical evaluation.

In the requirements analysis phase, a systematic literature review was conducted alongside an analysis of existing static analysis tools and organisational requirements to establish the security requirements guiding the framework's design.

The design and implementation phase followed an iterative development process with regular supervisor feedback. The resulting pipeline uses an agentic architecture: deterministic scanners identify security findings, an LLM agent applies remediations via tool use, and deterministic re-scanning establishes ground truth by verifying fixes with the same tools that produced the original findings. A structured verification report, functional equivalence validation through `helm template` diffing, and an adversarial red-team review pass provide additional layers of verification.

The empirical evaluation phase applied this pipeline across Helm charts of varying complexity sourced from public repositories, organised into smaller and larger categories to assess how chart complexity affects remediation performance. Multiple OpenAI API models were evaluated to enable cross-model comparison, and statistical analysis was conducted across models and charts. This evaluation design enables quantitative measurement of fix effectiveness, efficiency, reliability — including self-report accuracy rates derived from discrepancies between agent self-reporting and ground-truth scanner results — and safety across models and chart complexities.

1.6 Delimitations

The scope is limited to Helm charts since they are the dominant packaging format for Kubernetes applications, other IaC formats are excluded. Only OpenAI API-hosted models are evaluated, a cross-vendor comparison is left to future work. Security validation covers only misconfigurations detectable by Trivy and Checkov; runtime vulnerabilities and application-level code security are out of scope. Functional equivalence is assessed via `helm template` output diffing, not actual cluster deployment. The pipeline operates on chart files only and cannot validate runtime cluster state. Results are tied to the specific model versions and prompt designs used and may not generalise to all models. The pipeline does not use retrieval-augmented generation (RAG) or other context-augmentation approaches that dynamically retrieve external information during execution. All context is provided to the model in the initial prompt.

1.7 Ethics and Sustainability

The project does not involve human subjects research, personal data collection, or ethically sensitive applications. All development and testing is carried out in isolated environments using publicly available test cases. An ethical consideration is the risk of over-reliance on LLM-generated fixes in a security-critical setting with the use of autonomous agents that relies on less scrutiny than manual fixes would receive. From a sustainability standpoint, improved security automation reduces resource expenditure on manual review processes and can decrease the risk of security incidents that could have broader societal impact.

1.8 Structure of the Thesis

Chapter 2 establishes the technical and research context, Kubernetes security, Helm chart architecture, static analysis tooling, and the state of LLM-based automated remediation. Chapter 3 describes the pipeline design and the evaluation framework, providing the methodological foundation for interpreting the results. Chapter 4 presents the empirical results across all chart–model combinations. Chapter 5 interprets those results, discussing what the failure modes, cost structure, and self-report gap imply for practitioners considering adoption. Chapter 6 concludes with the key findings, limitations,

and directions for future work.

Chapter 2

Background

2.1 Kubernetes and Container Orchestration

2.1.1 Container Fundamentals

2.1.1.1 Process Isolation and Namespaces

Container isolation, the foundation enabling Kubernetes to safely run multiple workloads on shared infrastructure is implemented through Linux kernel namespaces, which create separate views of system resources for each container [1]. Kubernetes Pods leverage namespace isolation to provide security boundaries between applications. The *Network Namespace* is particularly critical: it isolates the network stack, including interfaces, routing tables, and sockets, giving each Pod its own isolated network environment [1]. Kubernetes assigns each Pod a unique IP address and enables Pod-to-Pod communication across the cluster [2]. The *Mount Namespace* isolates filesystem mount points, allowing each container to have an independent root filesystem and mounted volumes. The *PID Namespace* isolates process IDs, enabling containers to run init processes as PID 1 while remaining isolated from host processes. The *User Namespace* maps container user IDs to host user IDs, which is critical for Kubernetes security contexts where containers run as non-root users without requiring elevated privileges on the host.

Complementing namespace isolation, control groups (cgroups) enforce resource limits, preventing individual containers from monopolising cluster resources [3]. In Kubernetes clusters, cgroups are essential for enforcing the `resources.requests` and `resources.limits` specified in Pod

specifications. Without cgroup enforcement, a misconfigured container could consume all CPU or memory on a node, causing denial of service to other Pods and potentially crashing the kubelet. The *Cgroup Namespace* virtualises the cgroup hierarchy, isolating containers from viewing other containers' resource usage and preventing information leakage.

For Kubernetes deployments, namespace and cgroup isolation together create the fundamental security boundary. Namespaces prevent processes from seeing host resources or other containers, while cgroups prevent resource exhaustion attacks. However, neither mechanism provides complete isolation: a compromised container could still exploit kernel vulnerabilities to escape the container and access the host.

2.1.1.2 Container Images and Registries

In Kubernetes deployments, container images are the units of application distribution. Each Pod specification references a container image by name and tag — for example, `nginx:1.25` — which the kubelet fetches from a container registry at Pod creation time [4]. Container images are standardised, layered packages containing application code, dependencies, and runtime configuration. Most Kubernetes container runtimes (Docker, containerd, CRI-O) conform to the Open Container Initiative (OCI) standards, ensuring image portability and the ability to build images locally and deploy them identically in any Kubernetes cluster.

Container registries are centralised image storage services that the kubelet queries when starting Pods on a node [5]. When a developer pushes a container image to a registry, the registry stores individual layers alongside an image manifest describing those layers, their size, and associated metadata [6].

For Kubernetes security, image registry practices are critical. Vulnerability scanning at push time identifies weaknesses in base images or dependencies before they reach clusters [4]. Pulling by digest (e.g., `myapp@sha256:abc123`) rather than by tag ensures exact image specification regardless of tag mutations. Private registries isolate proprietary applications, while image signing and verification detects tampering, and authenticated push/pull prevents unauthorised image distribution. A container image with unpatched vulnerabilities or embedded secrets represents a foundational security risk, as it propagates to every Pod deployed from that image across the cluster.

2.1.1.3 Why Containers Over Virtual Machines: Portability and Consistency

Kubernetes was designed around containerised applications because containers provide the portability and consistency essential for cloud-native deployments [7, 8]. Unlike virtual machines, containers share the host OS kernel, reducing resource usage and enabling faster startup times. This efficiency is critical at Kubernetes scale: clusters with thousands of Pods would be prohibitively expensive if each Pod required its own operating system.

However, this architecture introduces a security trade-off. Because containers share the host OS kernel, a successful kernel exploit could compromise all containers on that host, whereas VMs provide stronger isolation since each runs its own kernel. This is why Kubernetes security emphasises defence-in-depth: while container isolation prevents one container from accessing another's filesystem or processes, sophisticated attackers may exploit kernel vulnerabilities to escape the container entirely [9, 10]. Kubernetes therefore implements additional security layers beyond container isolation — including RBAC, NetworkPolicies, security contexts, and admission control.

2.1.2 Kubernetes Cluster Architecture

A Kubernetes cluster, as shown in Figure 2.1, consists of two distinct types of nodes: control plane nodes and worker nodes. The control plane manages the overall cluster state, deciding which applications run, where they run, and ensuring the actual state matches the desired state. Worker nodes execute the actual application workloads as containers within Pods [11].

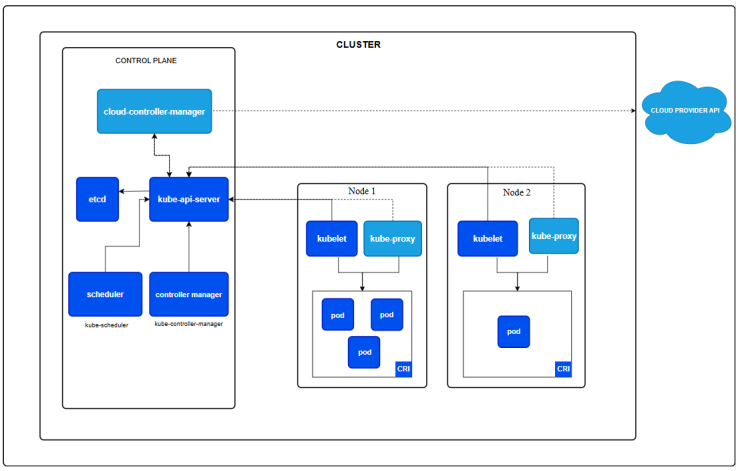

Figure 2.1: Kubernetes cluster components.

2.1.2.1 Control Plane Components

Kubernetes employs a hub-and-spoke architecture where all cluster management traffic terminates at the *API Server* (`kube-apiserver`), the central control plane component [11]. The API Server handles authentication, authorisation, and REST API requests from `kubectl`, other tools, and internal components. Communication between workers and the control plane is secured by default using TLS certificates and client authentication, ensuring cluster security even across untrusted networks.

The *etcd* database is a distributed key-value store serving as the backing store for all Kubernetes cluster data [11, 12]. Every Kubernetes object – Pods, Deployments, Services, ConfigMaps, Secrets, RBAC resources, and NetworkPolicies – is stored in etcd, making it the central coordination point of the entire cluster. etcd uses the Raft consensus algorithm to maintain strong consistency across all cluster nodes, ensuring all instances agree on the same data even during network partitions or node failures [13]. This consistency guarantee is critical for preventing split-brain scenarios, such as two nodes simultaneously attempting to attach the same persistent volume or schedule the same Pod. When a change is committed to etcd, it is replicated across all etcd nodes before being acknowledged, ensuring data durability and the ability to recover cluster state.

The *kube-scheduler* continuously monitors newly created Pods that have not yet been assigned to a node and selects an appropriate node for execution [11]. The scheduler evaluates resource requirements, hardware and software constraints, and inter-workload interference before making

placement decisions.

The *kube-controller-manager* runs a set of controller processes — control loops that watch cluster state through the API Server and make changes to drive the current state towards the desired state defined in etcd [14]. For example, the node controller monitors node health and evicts Pods from failed nodes, while the replication controller ensures the desired number of Pod replicas are running for Deployments and ReplicaSets.

The *cloud-controller-manager* embeds cloud-specific control logic, linking the cluster to a cloud provider's API and separating cloud-platform interactions from cluster-internal components [11]. It only runs controllers specific to the configured cloud provider and is absent in on-premises deployments.

2.1.2.2 Worker Node Components

On worker nodes, the *kubelet* is an agent that runs on each node and ensures that the containers described in assigned PodSpecs are running and healthy [11, 15]. The kubelet continuously monitors running containers, restarts those that fail health checks, and reports node status back to the control plane. The *kube-proxy* maintains network rules on each node that enable traffic routing to the appropriate Pods based on Service definitions [16].

2.1.2.3 Core Resources: Pods, Deployments, Services, ConfigMaps, and Secrets

Pods are the smallest deployable units in Kubernetes, encapsulating one or more containers with shared storage and network resources [17]. Every container in a Pod shares the same network namespace, including the IP address and port space. Containers within a Pod communicate with one another via `localhost` and must coordinate their use of shared network resources such as ports when communicating with external entities.

Deployments are higher-level objects that manage a set of Pods via ReplicaSets, defining a desired state that the Deployment Controller continuously reconciles against the actual cluster state [18]. Common use cases include rolling out a new ReplicaSet where Pods are created in the background or updating the PodTemplateSpec to trigger a controlled rollout in which the new ReplicaSet is gradually scaled up while the old one is scaled down.

Services provide stable network endpoints and load balancing for accessing Pods, acting as an abstraction layer that exposes applications running

in Pods to other services or external clients, enabling service discovery and consistent accessibility regardless of Pod restarts or rescheduling [19, 20].

ConfigMaps store non-confidential configuration data as key-value pairs, enabling configuration to be kept separate from application code and supporting multiple environment configurations [21]. Unlike most Kubernetes objects, a ConfigMap has `data` and `binaryData` fields rather than a `spec`. A Pod references a ConfigMap in its specification to configure containers based on that data, provided the Pod and ConfigMap reside in the same namespace.

Secrets are similar to ConfigMaps but are intended for sensitive data such as API keys, credentials, and tokens, preventing confidential information from being embedded directly in application code or container images [22].

2.1.3 Helm Charts and Package Management

2.1.3.1 Helm in Kubernetes Ecosystems

Managing complex Kubernetes applications without tooling requires coordinating multiple YAML manifests, handling dependencies manually, and duplicating configuration across environments. Helm addresses this challenge as a package manager for Kubernetes [23]. Its packaging format, the *Helm chart*, bundles all Kubernetes manifests for an application into a single, versioned, templated unit. Helm supports the full application lifecycle: installation, upgrades, rollbacks, and configuration overrides, enabling teams to apply the same chart definition across multiple environments using different value sets [24, 25, 26].

Kubernetes manifest files define individual resources using a `kind` field that specifies the resource type — such as `Deployment`, `Service`, or `ConfigMap`. In a non-Helm workflow, each of these manifests must be authored, versioned, and applied independently. Helm charts collect these manifests into templates, rendering them dynamically from a shared set of values, which eliminates repetition and reduces the risk of configuration drift between environments.

2.1.3.2 Chart Structure

A Helm chart is a directory with a defined layout. The `Chart.yaml` file contains chart metadata: the chart name, version, description, dependencies, and maintainer information. The `values.yaml` file defines default configuration values that users can override at installation time, specifying

defaults for replica counts, image versions, resource limits, and other parameterisable settings. The `templates/` directory contains Kubernetes manifest templates that are rendered against the values to produce final manifests. The `charts/` subdirectory stores packaged chart dependencies declared in `Chart.yaml`.

2.1.3.3 Helm Templating Engine

Helm uses the Go templating language extended by the Sprig function library, having over 60 functions for dynamic manifest generation [27, 28, 29]. Templates use double-brace syntax — `{{ }}` — to reference values from `values.yaml`. For example, `{{ .Values.replicaCount }}` renders to whatever value is set for `replicaCount` in the values file. Sprig extends this with string manipulation, type conversion, arithmetic, and dictionary operations [30].

Helm's control structures enable conditional resource creation and iterative generation. The following conditional block creates a ServiceAccount only when the corresponding value is set to true [30]:

```
{{- if .Values.serviceAccount.create }}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ .Values.serviceAccountName }}
{{- end }}
```

Loops enable generating repeated structures from a list value:

```
{{- range .Values.additionalLabels }}
{{ . }}: "true"
{{- end }}
```

2.1.3.4 Values and Overrides

The power of the Helm templating system comes from value overrides. While `values.yaml` defines defaults, users can override any value at installation time using `-set` parameters or by providing a custom values file with `-f custom-values.yaml`. This enables a single chart definition to serve multiple environments with different configurations [31]. For example, a development environment might use one replica, no resource limits, and debug-level logging, while the production environment uses three replicas,

CPU and memory limits, and warn-level logging — both deployed from the same chart.

2.1.3.5 Helm Chart Security Considerations

Helm charts introduce security risks that arise from how templates, values, and defaults are designed and used [32]. An issue is storing sensitive information such as passwords, tokens, or API keys directly in `values.yaml` or templates, which exposes secrets when charts are committed to version control or shared externally. Charts frequently ship with permissive defaults intended for quick setup, such as default credentials, overly broad security contexts, and missing resource limits, if these defaults are left unchanged in production, they produce vulnerable workloads that violate organisational security requirements.

Helm's Go templating allows dynamic logic, and complex template logic increases the likelihood of subtle misconfigurations and, in some cases, injection risks when user-controlled values are rendered without sanitation. Charts that define RBAC resources and service accounts may inadvertently grant unnecessary cluster-admin roles or overly permissive service accounts, leading to unintended access to the Kubernetes API and sensitive resources. Finally, reusing legacy templates without reassessing them for a new environment propagates outdated or insecure patterns, introducing vulnerable configurations into clusters.

2.2 Kubernetes Security

2.2.1 Security Model and Threat Landscape

2.2.1.1 The 4Cs Model: Code, Container, Cluster, Cloud

Security in cloud-native systems is layered across four dimensions [33, 34]. *Code* is the innermost layer and a key attack surface: application logic vulnerabilities can enable data exfiltration, tampering, or remote code execution. Secure coding practices, dependency scanning, and static or dynamic code review are used to detect flaws before deployment.

Container security focuses on the build and packaging stage: scanning images for known vulnerabilities, enforcing image signing and provenance, and running containers with least privilege using non-root users and minimal Linux capabilities.

Cluster security covers Kubernetes itself — securing the control plane, enforcing RBAC and strong authentication, applying NetworkPolicies, enforcing Pod Security Standards, encrypting Secrets at rest, and hardening cluster components. Cluster misconfigurations are consistently identified as a key challenge, making it necessary to secure both the configurable control plane components and the workloads running on the cluster.

Cloud is the outermost layer and encompasses the underlying infrastructure and managed services. Cloud security relies on a shared responsibility model: the provider secures the platform, and the customer secures workloads, identities, and configuration.

Each layer depends on the security of the one surrounding it, code-level security cannot compensate for weak cluster-level access controls, and container security cannot overcome cloud infrastructure compromise. This hierarchical model emphasises that effective security requires defence at every layer.

Threat modelling frameworks systematically map the attack surface and adversary techniques relevant to Kubernetes environments. Microsoft's Threat Matrix for Kubernetes maps common attacker goals and specific techniques targeting Kubernetes, building on the MITRE ATT&CK framework for Containers [35], which catalogues the attack techniques and sub-techniques that threat actors actively leverage to compromise containers and clusters [36, 37]. These frameworks guide organisations in prioritising defensive efforts on the weaknesses that attackers most frequently exploit.

2.2.1.2 Common Attack Vectors

Several recurring attack vectors appear in the literature, many stemming from misconfigurations and weak access controls.

Weakly Protected Control Plane Endpoints. Attackers can scan for exposed Kubernetes API servers, dashboards, and kubelet APIs with weak or missing authentication, then use these endpoints for initial access and cluster takeover [38]. Misconfigured API access is consistently cited as one of the most dangerous entry points because it frequently grants direct control over cluster resources [39, 40].

Misconfigured RBAC and Privilege Escalation. Overly permissive Roles and ClusterRoles pose significant security risks. Once an attacker gains a foothold in a Pod or obtains stolen credentials, they can harvest service account tokens and exploit excessive RBAC privileges to move laterally or escalate to cluster-admin [41]. A compromised application Pod whose token

is bound to a permissive role could, for example, allow an attacker to create new Pods and read cluster Secrets.

Vulnerable Container Images. Images built from unpatched base layers, containing outdated libraries, or pulled from untrusted registries expose clusters to known CVEs and malware. A Sysdig report found that 85 percent of images running in production contain at least one patchable vulnerability, and 75 percent contain vulnerabilities of high or critical severity [42], making image scanning a critical control.

Privileged Containers. Containers running with `privileged: true` can be abused to access the underlying host [43]. Combined with kernel vulnerabilities such as CVE-2022-0185, attackers can break out of containers, gain control of the node, and pivot to other workloads [44].

Network Misconfigurations. By default, all Pods within a cluster can reach all other Pods without restriction. If an attacker compromises one Pod, this default-allow model enables straightforward lateral movement to other Pods and services [45]. Misconfigured NetworkPolicies and open ingress and egress configurations therefore create an exploitable attack surface.

Poor Credential and Secret Management. Kubernetes provides a built-in Secrets object, but incorrect usage leads to leakage [46]. A common mistake is embedding hard-coded secrets in container images or configuration files, or granting overly broad access to Secrets within the cluster. Common mitigations include enabling encryption at rest for Secrets and restricting which Pods can access each Secret.

2.2.2 Workload Security Configuration

2.2.2.1 Security Contexts

Security contexts define permission and access control settings at the Pod and container level [47, 48]. The most security-relevant fields are as follows.

`runAsNonRoot` Boolean enforcing that containers cannot run as the root user, preventing container escapes from granting root access to the host.

`runAsUser` Specifies the numeric user ID (UID) under which container processes execute. Common practice is to run containers as UID 1000 or higher, preventing root privileges on the host even if containerisation is bypassed.

`runAsGroup` Specifies the primary group ID for container processes.

`allowPrivilegeEscalation` Boolean controlling whether child processes can gain more privileges than their parent. Setting this to `false` prevents privilege escalation vulnerabilities that could be exploited to gain unauthorised access to the host and potentially escalate to full cluster compromise.

`readOnlyRootFilesystem` Boolean mounting the container's root filesystem as read-only, preventing modification of system files except via explicitly mounted volumes. This limits an attacker's ability to modify system binaries, install malware or backdoors, or support lateral movement [49].

`capabilities` Linux capability set controlling fine-grained kernel privileges. Best practice is to set `drop: ["ALL"]` to remove all capabilities, then explicitly add only those strictly required.

`fsGroup` Group ID applied to all files in mounted volumes, enabling group-based permission management.

Security contexts operate at two scopes: Pod-level contexts apply to all containers in the Pod unless explicitly overridden, and container-level contexts override Pod-level settings for specific containers [47]. Security contexts are limited to Pod and container-level enforcement and cannot regulate privileges at the cluster or node level, necessitating complementary controls such as RBAC and NetworkPolicies.

2.2.2.2 Resource Management

Resource requests specify the minimum guaranteed resources a Pod requires; the Kubernetes scheduler uses these values to determine node placement and refuses to schedule Pods if insufficient resources are available [50]. Resource limits specify the maximum resources a Pod may consume; if a container exceeds its memory limit it is terminated, and if it exceeds its CPU limit it is throttled [51].

LimitRange objects enforce constraints on individual Pods and containers within a namespace [51]. They can enforce minimum and maximum compute resources per Pod or container, minimum and maximum storage per PersistentVolumeClaim, ratios between requests and limits, and default request and limit values automatically injected into Pods that lack explicit resource specifications.

ResourceQuota objects limit total resource consumption across an entire namespace [52], capping the total CPU and memory consumed by all Pods or the number of objects that can be created.

Without resource limits, a single misconfigured container can monopolise cluster resources, causing denial-of-service conditions for other workloads. Resource limits prevent such abuse and ensure fair allocation in multi-tenant clusters.

2.2.3 Access Control and Network Segmentation

2.2.3.1 Role-Based Access Control (RBAC)

RBAC is Kubernetes' mechanism for regulating access to cluster resources based on the roles assigned to users, groups, and service accounts [53, 54, 55, 56, 57]. The goal is least privilege: each identity receives only the minimum permissions necessary to perform its function. RBAC is implemented through four resource types.

Roles are namespace-scoped objects defining sets of permissions — specifying which API groups, resources, and verbs a subject may use. *ClusterRoles* are cluster-wide equivalents, applying across all namespaces. *RoleBindings* are namespace-scoped bindings that associate a Role with one or more subjects (users, groups, or service accounts). *ClusterRoleBindings* are cluster-wide bindings that associate a ClusterRole with subjects.

Each Role or ClusterRole specifies rules composed of three elements: `apiGroups` (the API groups containing the target resources), `resources` (the resource types, such as `pods`, `deployments`, or `services`), and `verbs` (the permitted operations, such as `get`, `list`, `create`, `update`, or `delete`). The following Role permits subjects to view and list Pods but not create, update, or delete them:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

Common RBAC misconfigurations include using wildcard verbs or resources (`"*"`) in Roles, assigning ClusterRoles when namespace-scoped Roles would suffice, and binding roles to the default service account shared by all Pods in a namespace, which inadvertently grants identical permissions to every workload.

Privilege escalation through RBAC arises when subjects hold permissions beyond what their role requires [58]. The `pods/exec` permission allows users to execute commands inside other Pods and access their environment, Secrets, and running processes. Permissions to create Roles or ClusterRoles allow a subject to define and assign arbitrary privileges. Once an attacker gains access to a Pod with elevated permissions, these vectors can enable lateral movement and eventual escalation to cluster-admin.

2.2.3.2 Network Policies

NetworkPolicies act as firewalls at the network layer, controlling traffic between Pods and between Pods and external endpoints [59]. By default, Kubernetes permits all network traffic, every Pod can reach every other Pod and service without restriction, a default-allow posture that enables unrestricted lateral movement if any Pod is compromised, unless a *NetworkPolicy* explicitly restricts it. NetworkPolicies are enforced by the Container Network Interface (CNI) plugin [60], provided the plugin supports policy enforcement, such as Cilium or Calico, and support both ingress rules, which specify allowed inbound traffic to a Pod, and egress rules, which specify allowed outbound traffic from a Pod [61].

Without NetworkPolicies, even correctly configured RBAC and security contexts cannot prevent an attacker with access to one Pod from scanning the network, discovering services, extracting data, and pivoting to other workloads. Common practice is to apply a default-deny NetworkPolicy blocking all ingress and egress, then explicitly open only the required communication paths using label selectors, namespace selectors, and port specifications. Restricting who can create or modify NetworkPolicies via RBAC is also recommended to prevent unauthorised policy changes.

2.2.4 Common Misconfigurations

2.2.4.1 Empirical Studies on Misconfigurations in Kubernetes Environments

Rahman et al. analysed 2039 Kubernetes manifests mined from 92 open-source software repositories, identifying security misconfigurations and developing a static analysis tool called Security Linter for Kubernetes Manifests (SLI-KUBE) [62]. The study catalogued 1051 misconfigurations across 11 categories, many of which enable severe attacks:

1. **Absent Resource Limit.** Pods lacking resource requests or limits, enabling resource exhaustion and denial-of-service attacks.
2. **Absent securityContext.** Pods with no security context specified, defaulting to permissive settings that can be exploited to gain access to the cluster.
3. **Activation of hostIPC.** Setting `hostIPC: true` shares the host's inter-process communication namespace with the container, allowing processes within the container to observe all IPC communications on the host.
4. **Activation of hostNetwork.** Setting `hostNetwork: true` places the Pod directly in the host's network namespace, exposing all host network interfaces and enabling packet sniffing against other workloads.
5. **Activation of hostPID.** Setting `hostPID: true` shares the host's process ID namespace with the container, giving the Pod visibility into all processes running on the host.
6. **Capability Misuse.** Activating dangerous Linux capabilities — such as `CAP_SYS_ADMIN` or `CAP_SYS_MODULE` — grants broad privileged access that malicious actors can use to obtain root-level control of the cluster.
7. **Docker Socket Mounting.** Mounting the Docker socket without correct permissions allows the container to spin up new containers, create images, or shut down existing ones, leaking control of the container runtime.
8. **Escalated Privileges for Child Processes.** Setting `allowPrivilegeEscalation: true` allows a child process to acquire more privileges than its parent, enabling unauthorised access escalation within the cluster.

9. **Hard-coded Secrets.** Credentials such as usernames, passwords, and tokens stored in plaintext in manifests or environment variables rather than managed as encrypted Secrets.

10. **Insecure HTTP.** Services transmitting traffic without SSL/TLS, leaving data vulnerable to man-in-the-middle attacks.

11. **Privileged securityContext.** Containers running in privileged mode, acquiring full kernel capabilities and effectively unrestricted host access.

The prevalence of such misconfigurations in practice is corroborated by industry data. The Red Hat 2024 State of Kubernetes Security report found that 40 percent of organisations detected misconfigurations in their container or Kubernetes environments, and 26 percent reported failing a security audit [63]. These figures illustrate the complexity of the Kubernetes security model and the ease with which configuration pathways can inadvertently violate security principles.

2.2.4.2 Systematisation of Security Practices

Shamim et al. conducted a grey literature review of 104 practitioner sources to systematise Kubernetes security, identifying 11 critical practices [64]:

1. **Authentication and authorisation.** Apply strong authentication for API requests and enforce least privilege through robust authorisation. Recommended measures include disabling anonymous API server access, disabling insecure default authorisation modes, enabling admission controllers, and avoiding insecure defaults.

2. **Kubernetes-specific security policies.** Define and enforce policies at network, Pod, and cluster levels, including NetworkPolicies to restrict Pod-to-Pod communication, security context policies enforcing non-root execution and read-only filesystems, and generic hardening such as closing unnecessary TCP ports and configuring API server audit policies.

3. **Vulnerability scanning.** Continuously scan Kubernetes components, container images, and CI/CD artefacts for known vulnerabilities and malware, pulling only from trusted private registries and validating code and images before deployment.

4. **Logging.** Enable and centrally collect logs for applications, containers, and cluster components, monitoring them regularly and triggering alerts when metrics deviate significantly from established baselines.

5. **Namespace separation.** Use dedicated namespaces to logically isolate resources, teams, and environments rather than relying on the default namespace, limiting the blast radius of a compromise to a single namespace.

6. **Encrypt and restrict access to etcd.** Restrict etcd access to the API server only and encrypt Secret data stored in etcd. Given that encryption keys may reside on the master node, external secret management tools are also recommended for additional protection.

7. **Continuous updates.** Apply security patches to Kubernetes and workloads promptly. The study highlights CVE-2019-16276 and CVE-2019-11253 as examples where unpatched clusters remain vulnerable to denial-of-service attacks.

8. **Limit CPU and memory quotas.** Configure resource requests and limits per Pod or namespace to prevent resource exhaustion. Without limits, a malicious Pod can consume all node resources, causing starvation and outages for co-located workloads.

9. **Enable SSL/TLS.** Encrypt communication between all cluster components including the API server, etcd, kubelet, and kubectl to prevent eavesdropping and tampering.

10. **Separate sensitive workloads.** Run sensitive applications on dedicated nodes to reduce the attack surface if a node is compromised, using Kubernetes taints and tolerations to control Pod placement.

11. **Secure metadata access.** Protect sensitive cluster metadata, as metadata APIs can expose kubelet admin credentials and other sensitive information if left unprotected.

2.3 Security Validation Approaches

2.3.1 Static Analysis Fundamentals

Static analysis examines source code, configurations, or Infrastructure as Code files without executing them, detecting potential vulnerabilities, policy

violations, and best-practice deviations at development time [65]. Unlike dynamic analysis, which requires a running system, static analysis enables fast feedback during code review and build phases where remediation is least costly. In the context of Kubernetes, static analysis examines YAML manifests and Helm charts to detect missing or misconfigured security contexts, absent resource limits, missing or overly permissive RBAC bindings, hard-coded secrets in environment variables, missing NetworkPolicies, and similar misconfigurations.

2.3.1.1 Static vs. Dynamic Analysis and Limitations

Static analysis offers several practical advantages: scans execute quickly without a runtime environment, analysis can be applied early in the development lifecycle, and the entire codebase can be covered in a single pass. These properties make it well suited to detecting security misconfigurations before deployment.

Static analysis is nonetheless limited in scope. It cannot detect runtime anomalies or behavioural issues, cannot identify zero-day exploits that have no known signature, may produce false positives on legitimate configurations, and lacks visibility into the full execution context of an application [66, 67]. Dynamic analysis complements static analysis by detecting actual malicious behaviour, unauthorised access attempts, and policy violations as they occur at runtime.

Static analysis tools face limitations in organisations. *Context blindness* is the most operationally significant limitation, tools implement generic industry best practices and cannot understand organisational requirements, leading them to over-alert on acceptable configurations or miss checks that matter specifically to a given policy. Without understanding organisational context, static tools cannot distinguish between genuine risks and accepted trade-offs.

Static code analysis operates primarily through pattern matching and data-flow analysis [68]. Since Kubernetes manifests define configuration rather than executable logic, pattern matching is the relevant technique. A core limitation is that detection is bounded by the tool's rule set, vulnerabilities for which no rule exists go undetected, and rule databases must be continuously updated. Static analyzers also cannot guarantee correctness and are susceptible to false positives, which directly affects developer trust and tool adoption. Trivy and Checkov, the scanners used in this pipeline, share these properties and their findings reflect known misconfiguration patterns, and their false-positive behaviour is an empirical concern addressed in the

evaluation.

2.3.2 Kubernetes Static Analysis Tools

Checkov is a broad IaC scanner designed to prevent cloud misconfigurations, with support for multiple IaC formats including Kubernetes and Helm [69]. Checkov ships with a large library of built-in policies and provides multi-framework coverage, making it a realistic baseline representative of tools used in practice. The corresponding trade-off is that breadth often increases configuration effort and produces a higher volume of findings compared to Kubernetes-specific linters. Checkov outputs findings as structured JSON, with each finding carrying a check identifier, a severity level, a human-readable description, and the file path and line range of the offending configuration.

Trivy is a comprehensive security scanner developed by Aqua Security that covers vulnerability scanning, misconfiguration detection, and secret scanning across container images, filesystems, and IaC files including Kubernetes manifests and Helm charts [70]. Trivy scans the chart, producing JSON output in which each finding is identified by an identifier, a title, a severity rating, and a description. Both tools were selected for this thesis since they are used in security workflows at FI.

2.3.2.1 Public Helm Chart Repositories

The Helm charts used in the empirical evaluation were sourced from Artifact Hub [71], a centralised repository for publicly available Helm charts covering a broad range of application categories. Charts were selected to achieve diversity in size and structural complexity.

2.3.3 Validation Methods

2.3.3.1 Traditional Rule-Based Checking

Rule-based validation uses explicit, deterministic checks matched against known patterns and established best practices. Typical examples include: absent resource limits flagged as a violation; `runAsNonRoot` not set flagged as a violation; privileged mode enabled flagged as a violation. Rule-based systems are straightforward to implement, fast to execute, and easy to audit, but they lack the flexibility to accommodate organisational context or to distinguish between genuinely risky configurations and deliberate, acceptable trade-offs.

2.3.3.2 Requirements-Based Validation

Requirements-based validation translates high-level security requirements into automatable checks while acknowledging that some validations require human judgement. Consider the following security requirement: all production Pods must run as non-root, have resource limits set, and be subject to a NetworkPolicy. This decomposes into checks of varying automation fidelity:

1. Verify that `runAsNonRoot: true` is set — fully automatable and deterministic.
2. Verify that `resources.requests` and `resources.limits` are specified — automatable and deterministic.
3. Verify that a NetworkPolicy exists restricting the Pod's traffic — automatable.
4. Verify that the NetworkPolicy restrictions match organisational intent and architectural requirements, and resource limits are enough for intended use — *Human judgement required for 2 and 4*.

The OWASP Application Security Verification Standard exemplifies requirements-based frameworks, providing a structured catalogue of security requirements and controls used both as compliance metrics and as guidance for control developers [72]. ASVS explicitly recognises that comprehensive security encompasses both deterministic violations and judgement-based decisions, a distinction that is central to how the pipeline in this thesis classifies its findings.

2.3.4 From Detection to Remediation

Static analysis tools excel at identifying misconfigurations but stop at detection, they report what is wrong without correcting it. This leaves a significant operational gap. Security teams receive lists of findings that must be manually triaged, contextualised against organisational requirements, and remediated one by one. Some decisions genuinely require human judgement, for instance, whether a service account truly needs cluster-admin permissions, or whether traffic between two specific Pods is architecturally necessary.

However, a large proportion of flagged misconfigurations have well understood, deterministic fixes, such as adding `runAsNonRoot: true`, dropping unnecessary Linux capabilities, or disabling privilege escalation.

For these categories, remediation is mechanical and repetitive. **The question is not whether these fixes are known, but whether an automated system can apply them reliably enough to be trusted.**

This observation motivates the exploration of automated remediation approaches that can apply known-good fixes at scale while preserving functional correctness, reducing the manual burden on security teams and accelerating the path from detection to a hardened deployment.

2.4 Large Language Models for Automated Code Remediation

Large Language Models (LLMs) are deep neural networks trained on large corpora of text and code, enabling them to generate, complete, and transform both natural language and programming languages. Frontier LLMs, including those from OpenAI used in this thesis, have demonstrated strong performance on code-related tasks including code completion, bug detection, code translation, and documentation generation. The effectiveness stems from pre-training on repositories of open-source code and documentation, allowing them to internalise patterns and relationships across languages and configuration formats [73].

For structured configuration languages such as YAML, the use of LLMs offers advantages. Kubernetes manifests use standardised field names, enumerated values, and documented schemas, which in principle constrain the output space compared to general programming languages. In practice, however, models can still produce structurally or semantically invalid YAML, a failure mode characterised in the evaluation. LLMs can also reason about relationships across files in a Helm chart, for example, understanding that a change to `securityContext.runAsNonRoot` in `values.yaml` propagates through `{{ .Values.securityContext.runAsNonRoot }}` references in deployment templates. This capacity to trace value propagation is essential for Helm chart remediation, where fixing a security misconfiguration often requires modifying the correct layer of abstraction: changing defaults in `values.yaml` rather than hardcoding values directly in templates [74].

However, LLM code generation does not operate deterministically. The same prompt may produce different outputs across invocations, and models can generate syntactically valid but semantically incorrect configurations, for example, producing valid YAML that passes schema validation but

introduces a functional regression by changing a container port or removing a volume mount. This probabilistic nature necessitates deterministic verification mechanisms when LLMs are used in security-sensitive remediation contexts.

2.4.1 Agentic AI Pipelines and Tool Use

Traditional LLM usage follows a single-turn pattern where the user provides a prompt, the model generates a response, and the interaction ends. **Agentic** AI pipelines extend this paradigm by placing the LLM within an iterative loop where it can plan actions, execute them through external tools, observe results, and refine its approach across multiple rounds.

LLMs have a function calling capability that allows them to invoke external functions during generation. Instead of producing only text, the model outputs structured function calls that an orchestrating system executes, returning results back to the model for further reasoning [75]. In remediation contexts, relevant tools include file reading and writing operations, command execution, and structured output parsing. The model reasons about which tool to invoke, with what parameters, and how to interpret the results, effectively acting as a controller within a larger automated system.

A key architectural pattern in agentic pipelines is the **feedback loop**. The agent applies a fix, a deterministic validator checks whether the fix resolved the targeted issue without introducing new problems, and the validation result is returned to the agent for further iteration if needed. This loop continues until either all issues are resolved or a maximum iteration bound is reached. The deterministic validator serves as a ground-truth oracle, preventing the agent from self-certifying fixes that do not actually resolve the underlying misconfiguration. Without such external verification, the agent would rely solely on its own assessment of fix correctness.

Agentic pipelines introduce additional *complexity*. Each iteration consumes tokens and incurs latency and cost. Multi-round conversations risk context window exhaustion or attention degradation in long sequences and the agent may enter unproductive loops where it repeatedly attempts and reverts the same fix. Designing effective termination conditions and detecting stale loops are practical engineering challenges that directly affect pipeline reliability and cost.

Model selection also introduces a capability and cost trade-off within agentic pipelines. Model families typically span multiple tiers: full-size models with stronger reasoning and larger context windows, and smaller, faster variants optimised for throughput and lower cost per token. Reasoning-

oriented models may produce more thorough analysis of chart architecture and more accurate fixes, but consume significantly more tokens per round due to longer internal reasoning chains, multiplying costs in pipelines that iterate over many findings across many files. Smaller models may handle straightforward single-file fixes adequately while struggling with complex charts that require cross-file reasoning. The choice of model tier therefore determines not only fix quality but also the practical cost ceiling of the pipeline.

2.4.1.1 Automated Remediation of Infrastructure as Code Misconfigurations

The gap between detecting and remediating security misconfigurations in Infrastructure as Code remains a significant operational challenge. Static analysis tools can identify misconfigurations in Kubernetes manifests and Helm charts with high accuracy and speed, but applying fixes to configuration files remains a manual process. Scanner reports listing dozens or hundreds of findings per chart must be triaged, understood, and addressed individually, a process that is slow, error-prone, and does not scale across large deployments with many charts and frequent updates.

LLM-based remediation aims to close the gap by bringing natural language reasoning to the repair process [76]. Rather than matching against fix templates, an LLM can read a scanner finding, examine the affected files, understand the chart's architecture, and generate a contextually appropriate fix. This flexibility comes at the cost of the failure modes described in the prior section, making the trade-off between deterministic reliability and contextual flexibility a central question for automated IaC remediation.

2.4.2 Prompt Engineering for Remediation Tasks

Prompt engineering is the design of instructions, context, and constraints provided to an LLM. It is central to the quality and reliability of generated outputs. For code remediation tasks, prompt design must balance three competing requirements: providing enough context for the model to understand the codebase and the security issue, constraining the model to avoid unintended modifications, and structuring the expected output format to enable reliable downstream parsing.

Several prompt engineering principles are particularly relevant for configuration remediation. *Explicit constraint specification* involves enumerating what the model must not do, for instance, instructing the model never to delete comments, never to change resource kinds, never to modify container ports,

and never to alter RBAC resources [77]. Without these constraints, LLMs tend to make unnecessary changes beyond the scope of the identified issue, which risks introducing functional regressions.

Providing the model with scanner findings including the rule identifier, severity, affected file, and a description of the expected fix, focuses the model's attention and reduces the likelihood of hallucinated or misdirected changes. The distinction between fixing `values.yaml` versus fixing templates is a critical consideration for Helm charts specifically. When a template references a value (e.g., `{{ .Values.securityContext.runAsNonRoot }}`), the correct remediation is to change the default in `values.yaml` rather than hardcoding the value in the template. Hardcoding a value alongside an existing `.Values` reference creates a contradictory configuration in which the template ignores user-provided overrides. Prompt instructions must explicitly encode this values-first principle, since LLMs without this guidance frequently default to editing the template file where the misconfiguration is syntactically visible.

2.4.3 Known Failure Modes: Hallucination, Drift, and Self-Report Gaps

LLMs exhibit several failure modes that are particularly consequential in automated remediation contexts, where incorrect outputs can degrade security posture or break application functionality.

Hallucination refers to the model generating outputs that are fluent and confident but factually incorrect or fabricated [78, 79]. In code remediation, hallucination manifests as the model claiming to have fixed an issue when the generated code does not actually resolve the underlying misconfiguration, referencing file paths or configuration keys that do not exist in the target chart, or inventing Kubernetes API fields absent from any known schema. Hallucinated fixes are particularly dangerous because they may pass superficial review while failing deterministic re-scanning. The gap between what the model reports as fixed and what deterministic scanners confirm as actually fixed provides a quantifiable proxy for hallucination rate.

Context drift describes the degradation of model attention and coherence over long conversations or multi-round interactions [80]. As an agentic pipeline iterates through multiple fix cycles, the conversation history grows and earlier instructions or constraints may receive diminishing attention. This can cause the model to disregard modification rules specified at the start of the conversation, revert previously applied fixes when addressing a different issue,

or lose track of which files it has already modified. Context drift is particularly problematic for large Helm charts with many files and many findings, where remediation may span dozens of interaction rounds. The severity of context drift is bounded by the model's context window, once the conversation exceeds this limit, the oldest content is truncated, potentially removing the initial system prompt containing critical constraints.

Over-fixing is a related failure mode in which the model makes changes beyond the scope of the identified issue. Examples include restructuring YAML formatting, removing comments, adding unrequested security hardening that breaks functionality, or deleting configuration blocks the model judges unnecessary. Over-fixing is particularly problematic because individual changes may each appear defensible in isolation, while their cumulative effect alters the functional behaviour of the deployed application. Violation checking mechanisms must therefore be implemented to detect unintended modifications.

2.4.4 Single-Agent vs. Multi-Agent Architectures

LLM-based remediation systems can have varying levels of agent complexity. In a **single-agent** architecture, one LLM instance handles the entire remediation workflow, reading scanner findings, analysing chart structure, generating fixes, and interpreting validation results within one continuous conversation. This approach is simpler to implement and debug, maintains full conversational context across the remediation process, and avoids inter-agent coordination overhead. However, a single agent must simultaneously balance security reasoning, YAML generation, architectural understanding, and self-correction within a single context window, which can lead to attention degradation as task complexity grows.

Multi-agent architectures decompose the remediation task across specialised agents. For example, a planning agent that triages findings and determines fix strategies, a coding agent that generates YAML modifications, a review agent that checks fixes for functional correctness, and an adversarial agent that attempts to identify flaws in the proposed changes [81]. This separation of concerns can improve output quality by allowing each agent to focus on a narrower task with a tailored prompt, and adversarial review agents can catch issues that the fixing agent's affirmative bias would miss. The trade-off is increased system complexity, possible higher token consumption from duplicated context across agents, and coordination challenges when agents produce contradictory fixes.

2.4.5 Model Selection: API-Hosted vs. Locally Deployed Models

The choice of model deployment strategy has practical implications for remediation pipeline design. API-hosted models from providers such as OpenAI offer access to frontier capabilities and require no local hardware. The disadvantages are dependency on external infrastructure, per-token costs that scale with pipeline complexity and chart size, data privacy concerns when transmitting infrastructure configuration details to external APIs, and rate limits.

Locally deployed open-source models eliminate data privacy concerns and external API dependency by running inference on-premise. However, many locally deployed models are currently behind frontier API models in instruction following, code generation quality, and context window size, all of which directly affect remediation accuracy. Local deployment also requires significant GPU infrastructure, introduces operational overhead for model serving and updates, and limits the ability to rapidly evaluate new model releases. The performance gap between local and API-hosted models is widely reported to be narrowing, but it remains meaningful for complex multi-file reasoning tasks such as Helm chart remediation, where the model must simultaneously reason about chart architecture, Kubernetes security semantics, and YAML syntax.

2.4.6 Verification and Ground Truth in LLM-Based Remediation

A central challenge in LLM-based IaC remediation is the nature of correctness verification. Fix correctness must be assessed through multiple complementary mechanisms, such as deterministic re-scanning with the same static analysis tools that identified the original findings, schema validation to ensure generated YAML conforms to Kubernetes API specifications, functional equivalence checking to verify that fixes do not alter the application's deployment behaviour, and adversarial review to detect subtle regressions that pass the other checks.

Deterministic re-scanning is particularly valuable because it provides an objective, reproducible ground-truth measurement. Either the scanner finding is resolved in the fixed chart or it is not. This binary signal is independent of the model's own assessment of fix correctness, enabling direct measurement of the self-report gap and serving as the primary success metric in the evaluation

presented in this thesis.

2.5 Related Work

2.5.1 Previous Studies on Kubernetes Security

Rahman et al. conducted a comprehensive empirical analysis of security misconfigurations in open-source Kubernetes manifests, taking 2039 manifests from 92 repositories and identifying 1051 instances across 11 misconfiguration categories [62]. Their categories span missing resource limits, absent or overly permissive security contexts, host namespace sharing (`hostPID`, `hostIPC`, `hostNetwork`), capability misuse, Docker socket mounting, privilege escalation flags, hard-coded Secrets, insecure HTTP, and privileged containers, summarised in Section 2.2.4.1. Building on these findings, they prototype **SLI-KUBE**, a static analysis linter encoding these 11 patterns, and show that more than half of the studied manifests contain at least one such issue. This work establishes both the prevalence of Kubernetes misconfigurations and the feasibility of static detection, and directly informs the misconfiguration taxonomy used in this thesis.

Shamim et al. conducted a grey-literature review of 104 practitioner sources to synthesise reported Kubernetes security practices [64]. Through qualitative coding they derive eleven practices covering API authentication and authorisation, Kubernetes-specific policies at the network, Pod, and cluster levels, vulnerability scanning, logging, namespace separation, protection of etcd, continuous updating, resource quotas, TLS for control-plane traffic, separation of sensitive workloads, and secure metadata access, described in detail in Section 2.2.4.2. The contribution of this work is a baseline that reflects what the security community currently recommends, spanning multiple layers and components of a Kubernetes deployment.

2.5.2 Static Analysis for Container Configurations

Müller presented a systematic attempt to design and evaluate a static analysis tool specifically targeting Kubernetes manifests and Helm charts [82]. The configuration best practices are grouped into security, resource management, availability, and structural categories, and the tool implements checks for 28 of them. The tool parses Kubernetes manifests, detects and locally renders Helm charts, and performs pattern-based checks — covering `runAsNonRoot`, `readOnlyRootFilesystem`, privileged mode, host namespaces, RBAC,

NetworkPolicies, resource requests and limits, and use of Secrets — to flag violations before deployment.

In an empirical evaluation, the tool reports an average of 31.87 findings per 1,000 lines of configuration, with security-related best practices violated most frequently, and an adjusted false-positive rate of 2.86 % once project-specific circumstances and obsolete checks are excluded. Two observations are particularly relevant to this thesis, static pattern-based analysis can surface substantial numbers of misconfigurations in Kubernetes and Helm artefacts, and scoping checks to project context is essential to keep false positives at an acceptable level. The pipeline in this thesis builds on these findings while extending the scope from detection to automated remediation.

2.5.3 Automated Analysis and Repair of Helm Charts

Minna et al. propose a multi-step pipeline for automated analysis of security policy violations in Helm charts [83]. Their approach runs a static analyzer to identify policy violations, applies default mitigations to satisfy reported policies, reruns the analyzer as a control step, and then generates *functionality profiles* that re-introduce only the privileges required for the workload to operate correctly. A final analyzer pass exposes surviving misconfigurations and tool discrepancies. The explicit goal is configurations that adhere to the principle of least privilege while remaining deployable.

For evaluation, they retrieve the top 60 Artifact Hub charts, successfully deploy 53, and analyse 52 with seven open-source analysers [84]. Overly permissive ClusterRoles emerge as the most common misconfiguration category, while many frequent issues are environment-specific or intentionally managed by external components. The authors conclude that securing Helm-based deployments still requires substantial manual intervention and call for standardisation of misconfiguration definitions and reporting formats. Their finding that certain hardening actions, changes affecting user IDs, root usage, and filesystem writability, are particularly likely to break workloads is directly relevant to the safety mechanisms such as multi-layer verification steps in this thesis's pipeline.

2.5.4 LLM-Based Automated Program Repair

Automated Program Repair has been substantially advanced by the introduction of LLMs into the repair pipeline. Xia and Zhang (2024) propose **ChatRepair**, a conversation-driven approach that uses ChatGPT

and interleaves patch generation with test execution feedback across multiple conversational rounds [85]. Rather than generating patches in a single shot, ChatRepair feeds failing test information back to the model, allowing it to learn from both successful and failed earlier attempts within the same conversation. The conversational feedback loop where validation results guide subsequent generations is structurally similar to the agentic remediation pattern used in this thesis, where scanner re-validation results are returned to the agent after each fix attempt.

Bouzenia et al. (2024) present **RepairAgent**, an autonomous LLM-based agent for program repair that moves beyond fixed feedback loops [86]. RepairAgent treats the LLM as an autonomous controller that plans and executes actions by invoking tools from a set of 14 repair-specific functions such as reading code, searching the codebase for similar API calls, applying patches, and running tests. Rather than following a scripted sequence, the agent freely decides which tool to invoke at each step based on previously gathered information and feedback from earlier attempts. RepairAgent outperforms procedural baselines on multi-line bugs due to the agent's ability to autonomously retrieve repair ingredients from the codebase rather than relying solely on the initial prompt context. The work demonstrates that selective autonomy has advantages in repair effectiveness. However, the agent occasionally produces overly complex fixes for simple bugs, and multi-file repairs remain difficult where it only edits a subset of required files.

2.5.5 LLM-Based Kubernetes Misconfiguration Detection and Remediation

Malul et al. (2024) propose **GenKubeSec**, an LLM-based system for Kubernetes configuration misconfiguration detection, localisation, reasoning, and remediation [87]. The system combines a fine-tuned detection model, *GenKubeDetect*, with a locally deployed model, *GenKubeResolve*, adapted via prompt engineering and few-shot learning to provide explanations and suggested fixes. A unified misconfiguration index standardises labels across three rule-based tools, directly addressing the tool-discrepancy problem where different scanners label semantically equivalent issues with different identifiers. Evaluated on over 276000 Kubernetes configuration files, *GenKubeDetect* achieves precision of 0.990 and recall of 0.999 relative to the individual rule-based tools, and a Kubernetes security expert confirmed the correctness of all remediation outputs in a validated sample.

An aspects of GenKubeSec are particularly relevant to this thesis is

the authors use of a unified policy index to reconcile tool-specific findings foreshadows the dual-scanner approach used here, where Trivy and Checkov findings are classified and deduplicated before being passed to the agent. In contrast, this thesis evaluates API-hosted models, focuses on the structural complexity of Helm charts, and combines remediation with multi-layer verification including adversarial review, schema validation, and functional-equivalence diffing.

Chapter 3

Method

This chapter describes the research methodology, experimental design, and analytical procedures used in this thesis. The study follows a **Design Science Research (DSR)** methodology [88], which organises research around iterative cycles of design, implementation, and evaluation, producing a working artefact and generalisable knowledge about its effectiveness. The artefact is an agentic LLM-based pipeline that autonomously remediates security findings in Kubernetes Helm charts. The empirical contribution is a controlled evaluation of that pipeline across multiple charts and models on several performance metrics.

DSR addresses two interrelated problems simultaneously: a *design problem*, how to construct the artefact, and an *empirical problem*, how well the artefact performs. The methodology maps onto this thesis through the following components.

Design as an artefact. The primary artefact is the agentic remediation pipeline itself. It accepts a Helm chart as input, identifies security misconfigurations through deterministic scanning, autonomously applies fixes via LLM-driven tool use, and validates results through multiple verification layers. The pipeline is implemented as executable software.

Problem relevance. The problem addressed is operationally significant. Kubernetes Helm charts in production environments frequently contain security misconfigurations that static analysis tools can detect but cannot remediate. Manual remediation is time-consuming and creates a bottleneck between security findings and deployment readiness. The pipeline targets the gap between detection and remediation by automating the fix process while preserving functional equivalence.

Design evaluation. The artefact is evaluated through an empirical

study. Ground truth is established by deterministic re-scanning with the same tools that identified the original findings, ensuring objective verification independent of LLM self-assessment. Functional equivalence and schema correctness is validated, and an adversarial red-team review provides an additional verification pass. The evaluation spans multiple charts of varying complexity and different OpenAI API models, enabling performance measurement and cross-model comparison.

Research contributions. The artefact is a pipeline architecture integrating deterministic scanners, agentic LLM remediation, violation-checking guardrails, and adversarial review into a multi-pass remediation workflow. The empirical contribution consists of quantified measurements of fix rate, token efficiency, self-report accuracy rates, consistency, and failure mode categories across model tiers and chart complexities.

Research rigour. Construction rigour is achieved through iterative development with systematic testing against real charts, where design decisions such as prompt engineering strategies, violation checker guardrails, stale loop detection, and tool configuration were tested and validated empirically.

Design as a search process. The pipeline evolved substantially during development. Early iterations explored more complex configurations including patch-file tools, elaborate delegation detection, and extensive missing-fix feedback loops. These were progressively discarded in favour of simpler mechanisms that empirically produced better results.

Communication of results. The pipeline architecture is documented and the evaluation framework maps research questions to specific measurable metrics.

3.1 Research Process

Requirements analysis. A systematic literature review was conducted alongside an analysis of existing static analysis tools and organisational requirements gathered from FI. This phase established the security requirements guiding the pipeline's design and output, including which scanner findings are considered not applicable in the organisational context and which require manual human review rather than automated remediation.

Design and implementation. The pipeline was developed iteratively with regular feedback from both academic and industry supervisors. The design evolved substantially over the course of the project. Key design decisions emerged from empirical testing against real Helm charts, with each iteration

producing measurable changes in primarily remediation quality, as well as fix rate, token consumption, and reliability that informed subsequent design choices.

Empirical evaluation. The completed pipeline was executed across Helm charts of varying complexity, sourced from public repositories, and evaluated against multiple OpenAI API models. Results were collected as structured JSON output and aggregated for statistical analysis.

3.2 Research Paradigm

This study adopts an empirical research paradigm. All primary outcome variables are measured quantitatively through deterministic tools: Trivy and Checkov provide scanner ground truth for fix verification, `helm template` rendering provides functional equivalence validation, and kubeconform provides Kubernetes schema validation. The use of deterministic scanners as the authoritative source of truth, rather than LLM self-assessment, ensures that measurements are objective and reproducible. Internal pipeline metrics such as token usage, iteration counts, and execution timing are recorded through a structured audit log. The LLM-authored verification report is a secondary output intended to support human review; it does not serve as ground truth for any reported metric.

3.3 Data Collection

3.3.1 Sampling

Helm charts were sampled from Artifact Hub [71], a public registry for cloud-native packages and configurations. The selection criteria were designed to maximise diversity of application type and structural complexity. Included charts must contain at least one Helm template generating Kubernetes workload resources, render successfully with `helm template` using default or minimal values, and produce at least one actionable security finding from either Trivy or Checkov after excluding findings classified as not applicable in the organisational context.

3.3.2 Sample Size

The final sample is organised into two complexity tiers. In total, 20 charts were included in the corpus for the evaluation set.

The `smaller_charts` category contains charts with fewer template files, typically ranging between 5-20 files, and simpler structures, without sub-charts. The `bigger_charts` category contains charts with multiple workload types, sub-charts, complex conditional logic, and extensive `values.yaml` configurations, typically 30+ template files.

3.4 Experimental Design and Planned Measurements

3.4.1 Test Environment

Each pipeline run operates on an isolated working copy of the chart under test, copied from the original chart directory into a clean working directory before execution begins. A sandbox mechanism restricts all file operations to the working directory, preventing the agent from modifying files outside its designated workspace. The working directory is destroyed and recreated between runs to ensure independence across observations.

3.4.2 Hardware and Software

The pipeline was implemented in Python 3.11 as a Jupyter notebook. Standard library modules, `subprocess`, `json`, `shutil`, `pathlib`, and `os` handled process management and file operations. The OpenAI Python SDK provided the interface for the API calls. `Pyyaml` handled YAML parsing for `values.yaml` validation and violation checking and `python-dotenv` manages API key configuration. `Pandas` and `matplotlib` were used for data manipulation and figure generation.

Path validation restricts all file system access except to the working directory. Reasoning effort is set to none for the first run and increased to medium for the second run. The maximum completion tokens parameter is set to 65,536 to accommodate larger charts. The token budget ceiling per run is 1,000,000 tokens.

The security scanning toolchain consists of three deterministic tools. *Trivy* is used to detect Kubernetes security misconfigurations in rendered YAML manifests, producing structured JSON output with finding IDs, severities,

titles, descriptions, and resolutions. *Checkov* is used in directory-scanning mode to evaluate Helm chart files against its built-in policy library, returning failed checks with check IDs, severities, resource targets, and file paths. *kubeconform* is used in strict mode to validate that post-remediation manifests conform to the Kubernetes API schema, catching structural errors that the security scanners would not detect. The Helm CLI is used for chart rendering.

The execution environment is WSL on Ubuntu, with all scanning tools installed natively. The pipeline invokes them via `subprocess.run()` with WSL path translation and structured output parsing.

3.5 Pipeline Architecture

The pipeline is the central artifact of this thesis. This section documents its design as a sequence of phases covering assessment, remediation, and verification.

3.5.1 Initialisation and Environment Setup

The initialisation phase establishes the execution context: loading dependencies, configuring API credentials, and consolidating token budgets and iteration caps into a set of runtime constants. Security scanners operate without knowledge of the deployment environment and may produce findings that are technically valid but operationally irrelevant. For instance, checks designed for cloud-managed infrastructure have no meaning in an on-premise setting. The pipeline addresses this by loading a requirements file that encodes two categories of deployment-context knowledge: findings classified as *not applicable*, which the pipeline suppresses entirely, and findings classified as *requiring manual review*, which represent genuine security gaps that fall outside the safe scope of automated remediation. Loading this classification at initialisation ensures consistent handling across all subsequent pipeline stages.

A sandboxed working environment is established in which all remediation takes place. Rather than operating directly on the source chart, the pipeline creates a clean copy in an isolated working directory at the start of every run, destroying any remnants of previous executions. All agent file operations are gated by a path-containment check that rejects any write target falling outside this directory, providing a guardrail against malformed LLM-generated paths escaping the sandbox. An audit log is initialised to capture a timestamped record of every agent action throughout the run. This log forms the primary data source for actions taken.

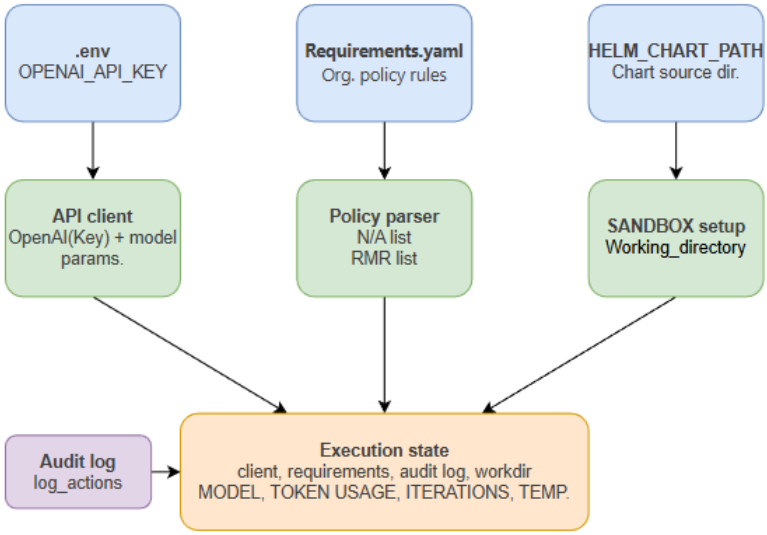

Figure 3.1: Initialisation phase of the pipeline. Three input sources are resolved. The environment file supplies the OpenAI API key and model parameters to the API client, the requirements file is parsed into a not-applicable list and a requires-manual-review list encoding deployment-context policy rules, and the chart source path is used to establish an isolated sandbox working directory. The outputs of all three paths are consolidated into a single execution state object carrying the client handle, policy requirements, audit log reference, working directory path, and runtime constants being model identifier, token budget, iteration caps for the pipeline run.

3.5.2 Tool Definitions

This phase establishes the interface through which the LLM agent interacts with the external environment. The pipeline defines six tool functions, `helm_template`, `run_trivy`, `run_checkov`, `read_file`, `write_file`, and `list_files`, each exposed to the model via OpenAI's function-calling schema.

The central security mechanism is path sandboxing. Every tool that accepts a file or directory path first resolves it to an absolute path and verifies that it falls within the designated working directory. This check prevents path traversal through hallucinated or adversarially injected paths, and rejections are recorded in the audit log.

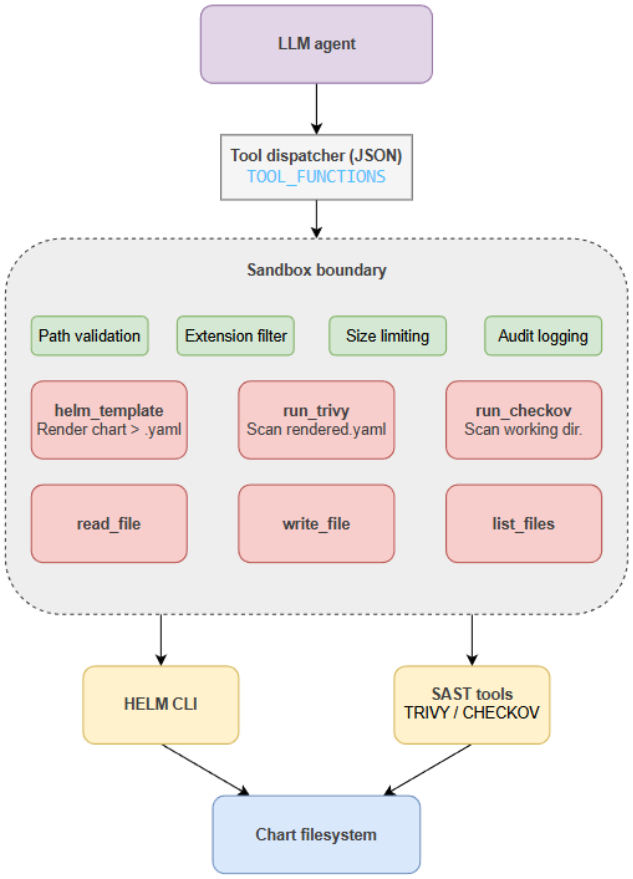

Figure 3.2: Tool definitions and sandbox boundary. The LLM agent issues function calls through a JSON tool dispatcher, invocations are routed through a sandboxed execution layer including path validation, extension filtering, size limiting, and audit logging. Within the sandbox, six tools are available, `helm_template`, `run_trivy`, `run_checkov`, `read_file`, `write_file`, and `list_files`.

The scanner tools are designed to be failsafe. `helm-template` invokes Helm's rendering engine to produce the flat Kubernetes YAML that downstream scanners consume. If the initial render fails due to missing required values, the function parses the error message, extracts the expected `.Values` references, and retries with placeholder values injected via `-set` flags. `run_trivy` and `run_checkov` execute their respective scanners, parse the JSON results, and return normalised finding objects with a consistent

schema including ID, severity, title, description, resolution, and source. This is essential because the two scanners use different output formats and severity vocabularies, and the agent must reason over a unified finding representation.

The filesystem tools, `read_file`, `write_file`, and `list_files`, give the agent the ability to inspect and modify chart source files. `write_file` applies additional guardrails beyond path validation: it enforces a maximum file size limit of 500 KB to prevent runaway generation and restricts writes to a whitelist of file extensions (`.yaml`, `.yml`, `.tpl`, `.json`, `.txt`) to prevent the agent from producing executable or binary artefacts. All tool invocations are recorded in the structured audit log.

Each tool is also described in an OpenAI-compatible JSON schema specifying the function name, a natural-language description, and the expected parameters with types. These schemas are passed to the model and serve as the sole mechanism through which the agent discovers what actions are available. Descriptions are deliberately concise and action-oriented, for example:

```
"type": "function",
"function": {
    "name": "run_trivy",
    "description": "Run␣Trivy␣security␣scanner␣on␣a␣
       ↪ rendered␣YAML␣file.␣Returns␣a␣list␣of␣
       ↪ security␣findings␣with␣IDs,␣severity,␣and␣
       ↪ descriptions.",
    "parameters": {
        "type": "object",
        "properties": {
            "yaml_path": {
                "type": "string",
                "description": "Path␣to␣the␣rendered␣
                   ↪ YAML␣file␣to␣scan"
            }
        }, "required": ["yaml_path"]
    }
}
```

3.5.3 Pre-Agent Scanning and Analysis

This step of the pipeline establishes the security baseline and constructs the structured task list that will guide the agent's remediation work.

The process begins by rendering the Helm chart to plain Kubernetes YAML. The rendered output collapses all template logic, conditionals,

and value substitutions into concrete manifests. Trivy then scans this rendered YAML for infrastructure misconfigurations, while Checkov scans the chart directory directly, performing its own internal Helm rendering and additionally inspecting subchart structures that Trivy's flat-file approach may miss. The use of two complementary scanners with different analysis strategies increases finding coverage and reduces the risk of false negatives in the baseline.

Findings from both scanners are normalised into a common schema and classified into one of three categories. Findings classified as *not applicable* represent scanner rules irrelevant to the deployment context. Findings classified as *requiring manual review* represent genuine security concerns that cannot be safely automated, for instance, network segmentation choices that depend on traffic flow, software architecture, or organisational context. Such findings are excluded from the agent's remediation scope but flagged in the output report with their rationale and recommended human action. The remaining findings constitute the *agent-fixable* set and define the remediation task.

The policy parser is deliberately tuned to the host organisation's requirements, since the operational motivation for the work is its specific remediation workload, and a concrete real-world policy produces a more representative in-scope and out-of-scope split than an ad hoc one would. The classification mechanism itself is general: the same parser can be driven by a different organisation's requirements without modification, and only the rule set it consumes is organisation-specific. The full requirements used here are not reproduced publicly, as the organisation's internal policy could not be made openly available.

Once classification is complete, the pipeline constructs a file task list that translates the abstract findings into actionable remediation instructions. Checkov findings are grouped by their originating file path, deduplicated within each file and annotated with concrete fix hints derived from the finding categories present. All hints follow an additive-only principle where the agent is explicitly instructed to add what is missing without removing existing configuration, a constraint that proved critical during development to prevent the agent from overwriting functional .Values references with hardcoded values.

The pipeline also handles a coverage gap between the two scanners. Trivy findings are reported against the rendered YAML and are not associated with specific source files, while Checkov findings reference individual templates. To ensure that workload templates not flagged by Checkov still receive Trivy-

relevant fixes, the pipeline scans the templates/ directory for any workload-producing files that do not appear in Checkov's target set, and appends them to the task list with instructions to apply all Trivy findings.

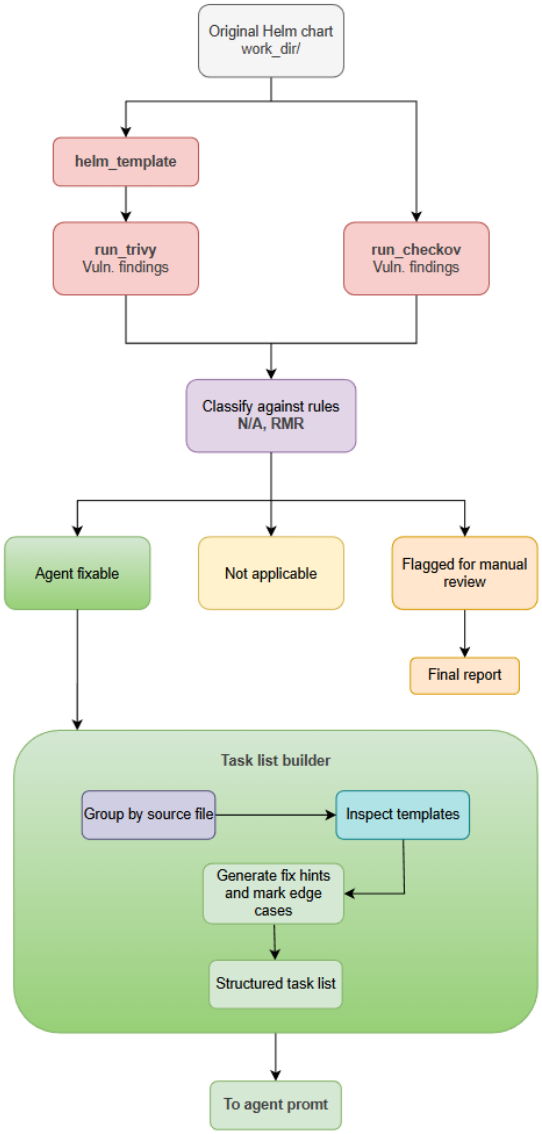

Figure 3.3: Pre-agent scanning and task list construction. The original Helm chart is rendered with `helm-template` and scanned in parallel by Trivy and Checkov, producing two sets of vulnerability findings. The combined findings are classified against the organisational policy rules into three categories, agent-fixable findings that proceed to remediation, not-applicable findings that are excluded, and findings flagged for manual review that are passed to the final report. Agent-fixable findings enter the task list builder, which groups them by source file, inspects the template structure to determine the appropriate fix pattern, and generates concrete fix hints. The structured task list is embedded in the agent's system prompt as the primary remediation brief.

The resulting task list is a structured text document that serves as the primary input to the agent's system prompt. By front-loading all file inspection, template classification, and hint generation into this deterministic pre-processing step, the pipeline minimises the agent's need to independently discover chart structure, reducing token consumption and the risk of the agent misidentifying which files to modify.

3.5.4 Agent Prompt Engineering

The system prompt governs the agent's behaviour throughout the remediation session and is an important design artifact in the pipeline. The prompt underwent continuous refinement across hundreds of runs, with revisions motivated by observed failure patterns. The prompt is structured around several distinct concerns, each addressing a category of errors observed during development. The opening section establishes the agent's identity and operational context, the organisation, the chart path, the security policy requirements, and the task list constructed in the preceding phase. This contextual grounding ensures the agent operates with awareness of the deployment environment rather than defaulting to generic recommendations.

The core of the prompt is organised around a pattern-matching framework for security context remediation as discussed in Section 1.1. During development, three recurring patterns emerged in how Helm charts implement security-relevant fields, and each requires a fundamentally different fix strategy.

1. **Pattern A**: The template uses `{{ toYaml .Values.securityContext | nindent N }}` to dump an entire values block into the manifest. Fixes must be applied in `values.yaml` only, because adding fields alongside a `toYaml` expression produces invalid YAML.

2. **Pattern B**: The template references individual `.Values` fields (e.g. `{{ .Values.securityContext.runAsNonRoot }}`). Fixes require merging missing fields into the existing template block while preserving existing references.

3. **Pattern C**: The template has no security context at all, requiring the agent to add a complete hardcoded block directly in the template.

This taxonomy is documented in the prompt since early experiments showed that without it, the agent would default to Pattern C behaviour

regardless of the chart's actual structure, overwriting functional *.Values* references with hardcoded values and breaking the chart's configuration interface.

The agent is instructed to check for existing .Values references before modifying any template file, and to fix values.yaml instead whenever such references exist. This principle was reinforced with explicit forbidden tasks and proved effective at preventing regressions. The preserve rules section enumerates modifications the agent must not make and includes changing container ports, removing existing probes or resource blocks, altering RBAC resources, modifying library chart templates, or injecting Helm template logic such as merge or mergeOverwrite functions. Each rule corresponds to a specific failure observed during development.

Probe handling receives particular treatment because it proved to be the most error-prone remediation category. The prompt implements a four-step decision tree. **First**, check whether the template already has probes, if so then do not touch. **Second**, check for conditional probe blocks gated by .Values flags. **Third**, check for commented-out probe configuration in values.yaml, if so uncomment and populate. **Lastly**, if none of these exist should probes be added directly to the template. Early agent runs would routinely add duplicate probes alongside existing conditional blocks, producing templates with two competing probe definitions.

The prompt specifies a structured JSON summary format that the agent must produce after completing all modifications. This summary requires the agent to categorise every finding ID from the task list into one of three buckets, fixed, not applicable, or unfixable, ensuring complete coverage and enabling automated comparison between the agent's self-report and the ground-truth scanner results in later pipeline stages.

3.5.5 Agentic Remediation Loop

The agentic remediation loop is the core execution phase of the pipeline, where the LLM agent iteratively applies security fixes to the Helm chart while receiving deterministic feedback on the correctness of each modification. The agent proposes changes through tool calls, the pipeline validates those changes against a comprehensive rule set, and violations are fed back to the agent as corrective instructions, creating a system that converges toward a correct remediation state.

Before the loop begins, the pipeline prepares the agent's initial context by selecting which files to include in the first message. Rather than sending

the entire chart directory, the pipeline looks for the set of files the agent needs such as, templates that appear as Checkov findings, all values.yaml files that are required for Pattern A and B fixes, helpers.tpl files which define template functions the agent needs to understand, and any workload-producing templates in the parent chart that were not flagged by Checkov but still require Trivy relevant fixes. Each file's complete content is embedded in the user message.

Alongside the file contents, the pipeline includes pre-computed hints for values.yaml changes, for instance, identifying all instances of *pullPolicy: IfNotPresent* that must be changed to *Always*, or empty *podSecurityContext:* blocks that need to be populated. These hints were introduced to compensate for the more limited analytical capacity of smaller models, which would otherwise fail to identify required values.yaml modifications without explicit guidance. Larger models such as GPT-5.4 are capable of finding these changes independently from the template context, but the hints are retained across all experimental runs to ensure a uniform pipeline configuration across model tiers.

Because this analysis is performed in the pre-processing stage, the agent can begin writing fixes immediately rather than spending its first several turns on file discovery and analysis. The loop itself follows an iteration-based interaction pattern with the chart. On each turn, the agent receives the accumulated conversation history of the system prompt, file contents, prior tool calls and results, any feedback messages, and produces either tool calls or a text response. When the agent makes tool calls, the pipeline executes them through the sandboxed tool layer, appends the results to the conversation, and triggers post-write validation before the next agent turn.

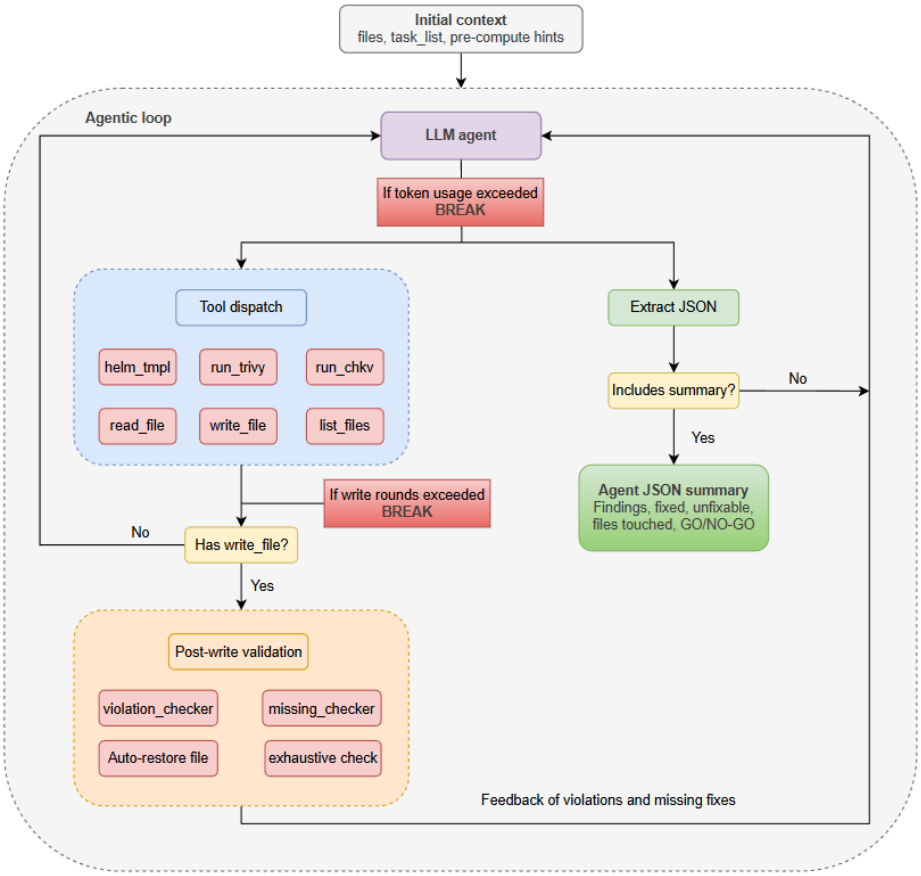

Figure 3.4: Main agentic remediation loop. The agent receives an initial context containing file contents and the structured task list, then enters an iterative loop. On each turn, the agent either issues tool calls or produces a text response. If the token budget is exceeded, the loop terminates immediately. Turns that include at least one write operation trigger the post-write validation stage, which runs the violation checker and missing-fix checker, auto-restores files where violations are detected, and performs an exhaustive integrity check. The combined checker output is returned to the agent as a feedback message, and the loop continues. If the write-round ceiling is reached, the loop terminates. When the agent produces a text response, the pipeline checks whether it contains the required JSON summary, if not, the agent is prompted again. Once a valid summary is extracted, reporting findings fixed, unfixable, files modified, and a GO/NO-GO deployment decision, the loop exits and the summary is retained for comparison against ground-truth scanner results.

There are three independent termination conditions that govern the loop to prevent runaway execution. A *maximum iteration* count caps the total number

of API calls. A *write-round* counter, incremented only on turns that contain at least one write_file call, enforces a ceiling on the number of modification attempts. A *token budget* ceiling prevents excessive cost accumulation. When any ceiling is reached, the pipeline injects a forcing message that instructs the agent to produce its final JSON summary immediately, regardless of whether all findings have been addressed.

One post-write validation mechanism is the **violation checker**. After every write round, a deterministic violation checker compares every modified file against its original content. The checker implements a comprehensive set of invariants, each derived from a specific failure mode observed during development. Structural integrity checks detect the removal of critical sections — `volumeMounts`, `volumes`, `env`, `args`, `ports` — whose absence would cause the workload to fail at deployment. For `values.yaml` files, a three-level key comparison detects key loss occurring when the agent rewrites the file and silently drops configuration. Duplicate top-level keys are also detected, since a duplicated key can silently override configuration without a parse error. Template checks verify that the agent has not replaced `.Values` references with hardcoded values, removed Helm control structures (`if`/`range`/`with`/`end` blocks), or changed resource kinds. Content policy checks prevent the agent from modifying protected files such as library chart templates in `charts/common/`, deleting comments, adding hallucinated top-level keys to `values.yaml`, or appending YAML fields after `toYaml` expressions.

The checker verifies that when `readOnlyRootFilesystem: true` is set, a corresponding `emptyDir` volume mount for `/tmp` is present because applications that cannot write to the root filesystem will crash without a writable temporary directory. A further check detects when the agent re-enables a feature the chart intentionally disabled, activating functionality that was deliberately left inactive.

Another post-write validation mechanism is the **missing-fix checker**. This helper inspects the current state of all files to identify security fixes that remain outstanding. For each Checkov finding, the checker determines whether the relevant template delegates to `values.yaml` via a `toYaml` expression and, if so, verifies that the fix has been applied in `values.yaml` rather than in the template file. Referenced `values.yaml` entries are also inspected directly.

After both checkers run, their outputs are combined into a feedback message sent into the conversation as a user-role message. This message categorises issues as either violations which the agent must fix by rewriting the affected file or missing fixes which the agent must address. The agent

processes this feedback in its next iteration. This convergence dynamic ensures that with each successive write round, the number of violations and missing fixes typically decreases.

However, convergence is not guaranteed. The agent may repeatedly attempt the same incorrect fix or fix one issue while introducing a regression. The pipeline addresses this by tracking the count of violations and missing fixes across write rounds. If no violations remain and the missing-fix count has not decreased for two consecutive rounds, the pipeline accepts the current state on the assumption that the agent has reached the limit for this chart. If the violation count remains unchanged for three consecutive rounds, the pipeline terminates unconditionally to prevent infinite loops.

The agent then forces its final summary of whatever fixes it has successfully applied. The loop ends when the agent produces a text response containing a JSON summary with `findings-fixed`, `findings-not-applicable`, `findings-unfixable`, `files-modified`, `decision`, and `decision-reasoning`. This self-assessment is subsequently compared against ground-truth scanner results to measure the self-report gap.

3.5.5.1 Agentic scope restriction

The current pipeline restricts the agent to `write_file` and `read_file`, shown in Figure 3.5. Although the pipeline implements sandboxed versions of all six tools and the agent could in principle be given access to all of them, extensive testing during development demonstrated that a fully agentic configuration produced substantially worse results.

A direct comparison was conducted on the full chart corpus using `GPT-5` and `GPT-5.4`, evaluating the constrained configuration with restricted tool usage against a variant in which the agent had unrestricted access to all tools while retaining the same system prompt, front-loaded file contents, fix hints, and task list. Across both models the constrained configuration matched or outperformed the fully agentic one on instance-level fix rate while consuming fewer tokens and fewer iterations on average, shown in Section 4.1. The gap was smaller on frontier models than on mini variants observed during development, where unrestricted tool access caused steeper degradation. For example, `GPT-5.4-mini` consumed nearly double the token budget (991998 vs 525818 input tokens), required nine additional iterations (23 vs 14), and achieved a fix rate of only 29.2 percent compared to 85.4 percent for the constrained configuration. In that run the fully agentic agent resolved zero Trivy findings and only three Checkov IDs, while the constrained agent

resolved 14 of 15 Trivy IDs and 15 of 18 Checkov IDs. The pattern indicates that the cost of unrestricted tool access scales with model capability when frontier models partially absorb the overhead by selecting a reduced tool surface, whereas smaller models exhaust their context budget on exploratory scanning and re-rendering. The additional render and scan tools offer no informational gain in this setting, since the agent already receives the full task list, fix hints, and file contents upfront, and the pipeline performs missing-fix detection and violation checking deterministically after each agent turn. The additional tool call appends its result to the conversation history, inflating the context window and degrading the model's ability to maintain focus on the remaining remediation tasks.

Constraining the agent's action space consistently outperformed giving it more autonomy. The additional tools offered no informational gain, but rather only inflated the context window and degraded remediation focus.

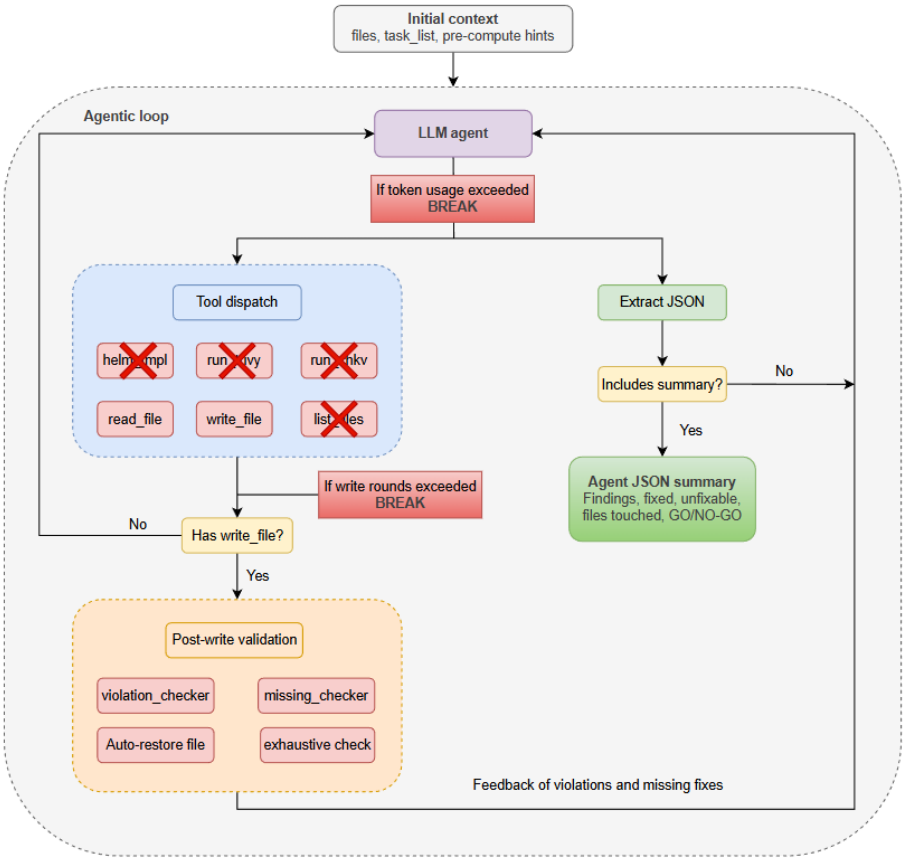

Figure 3.5: Constrained agentic remediation loop with read-write file only tool access. The loop structure is identical to Figure 3.4, but 4 of the six tools are disabled. This restriction prevents the agent from consuming iteration and token budget on re-rendering or re-scanning files whose contents were already supplied in the initial context. Testing showed this constrained configuration was substantially better than the fully agentic alternative.

3.5.6 Post Remediation Phase

Once the agentic remediation loop ends, the pipeline begins a multi stage verification phase in order to catch remaining issues, regressions, and implement improvements beyond the level of the static analysis tools. This phase consists of a re-scan to see if a second remediation pass is needed, a functional equivalence validation that compares the rendered manifests before and after remediation, and an adversarial red-team review to see changes from a deployment safety perspective.

3.5.6.1 Second-Pass Remediation

The remediated chart is rescanned by both Trivy and Checkov to obtain the count of remaining findings. If the chart fails to render, the second pass is skipped and the failure is recorded for the final report. Otherwise, the scanner results are filtered to exclude N/A and manual-review findings, and the remaining fixable findings are counted. If more than two findings remain, the pipeline triggers a second remediation pass. The second pass reuses the same system prompt, sandboxed tools, and violation checker as the first pass, but instead of the full task list, the agent only gets the remaining findings grouped by file, along with the partially fixed file contents. This narrowed context reduces token consumption and lets the agent concentrate on the remaining issues rather than re-examining files that have already been addressed. This instruction proved to be essential since otherwise the agent could sometimes rewrite files. Once the second pass is done, its metrics are merged with the first pass.

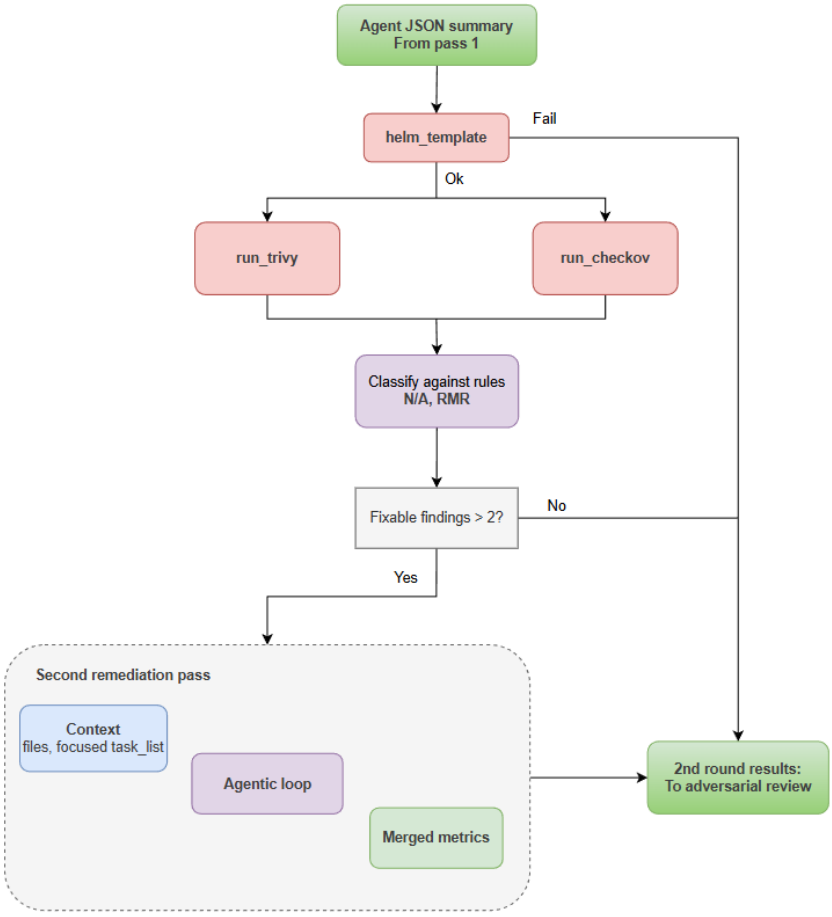

Figure 3.6: Second remediation pass. If more than two fixable findings remain, a second remediation pass is triggered, otherwise the pipeline proceeds directly to adversarial review. The second pass receives a narrowed context containing only the affected files and a focused task list restricted to outstanding findings, reducing token consumption and preventing the agent from revisiting already-corrected files. The agentic loop runs under the same constraints as the first pass, and its metrics are merged with the first-pass results before the combined output is forwarded to the adversarial review stage.

3.5.6.2 Adversarial Review and Remediation

The third verification step has a different approach, it sends the remediated chart to the LLM model, but now the agent is a deliberately adversarial red team reviewer and needs to assume the work about every change. This review targets issues that neither the violation checker or SAST scanners

can catch, implementations or changes that are syntactically valid, pass all misconfiguration checks but could nonetheless break the application at runtime.

The process begins by generating diffs between the original chart directory and the current modified state of the chart from the first two passes. If no files were changed meaning there were no fixable findings, the adversarial review is skipped, else, the pipeline constructs a detailed review prompt that looks at each modified file, the original content, the fixed content, and the diff, so that the reviewer has complete context to assess each change.

The reviewer assesses each change against five review dimensions:

- **Functional breakage scenarios** to identify concrete ways the changes could cause the application to fail, specifying where, the failure mode, and the triggering conditions.
- **Helm template integrity** checks whether Helm expressions were replaced, removed, or overridden. Explains whether conditional logic was altered and why it matters.
- **Resource constraint impact** evaluates whether resource limits are reasonable for the workload.
- **Security context compatibility** assesses whether added securityContext could conflict with the applications requirements.
- **Overall verdict** is given depending on the severity: SAFE, LIKELY SAFE, REVIEW NEEDED, or RISKY.

The structured output:

- **Remediation instructions.** Each issue is categorised as REVERT if the change should be undone entirely, ADJUST if the change is somewhat correct but needs modification, or ACCEPT if the change is safe.

If the adversarial review identifies REVERT or ADJUST items, a third and final remediation pass is triggered. This pass receives the reviewer's specific fix instructions alongside the current file contents of the affected files. It operates under a set of constraints where the agent is explicitly told to fix the implementation quality issues while keeping all security improvements. This prevents the adversarial feedback from being misinterpreted to weaken security rather than to improve the quality. For REVERT items, the agent is instructed to restore and move values to values.yaml and using .Values

references rather than simply deleting the security additions. The pass focuses on targeted corrections rather than performing broad remediation. When completed, a post-pass check verifies that the chart still produces valid output.

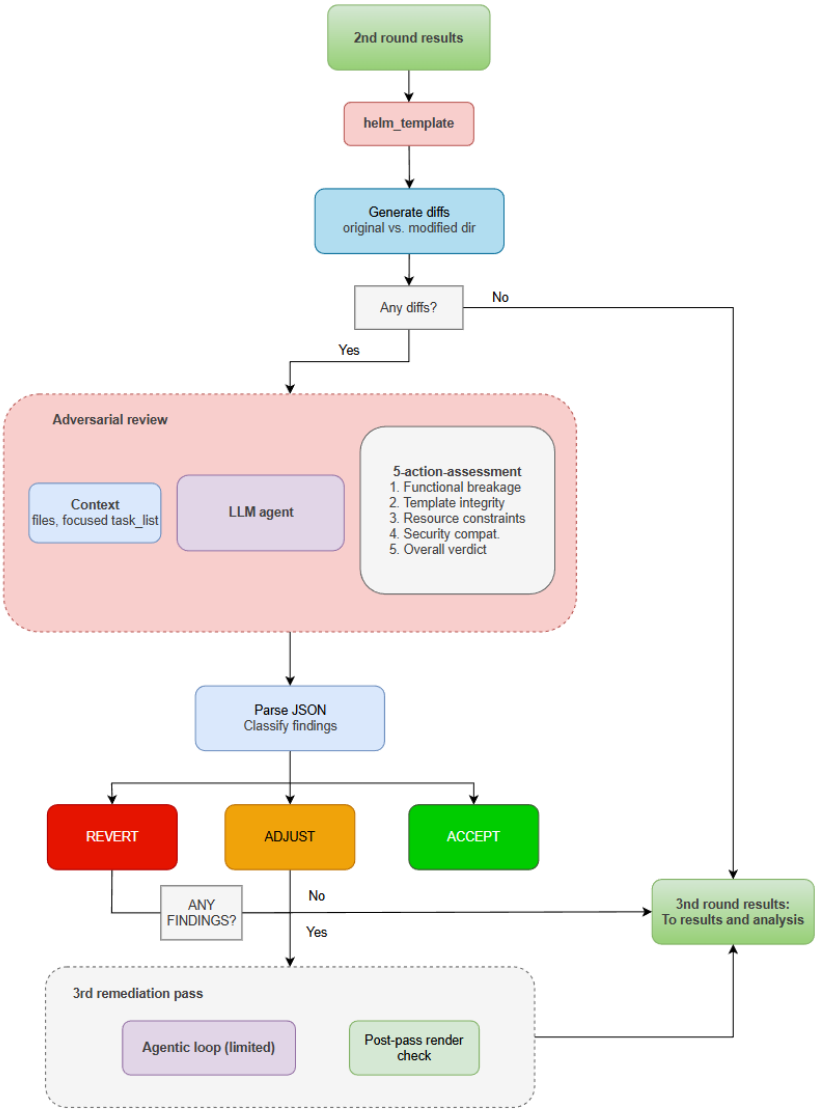

Figure 3.7: Adversarial review and third remediation pass. LLM agent operating under a red-team reviewer persona assesses each change across five dimensions. Changes classified as REVERT, ADJUST, or ACCEPT. If any REVERT or ADJUST items are identified, a third and final remediation pass is triggered, in which a scope-limited agentic loop applies the reviewer's specific instructions while preserving all security improvements. If no actionable items are found, or once the third pass completes, the combined results are forwarded to the deterministic verification and report generation stage.

3.5.7 Deterministic Verification and Report Generation

The final pipeline stage performs three actions. It establishes the ground-truth remediation results through deterministic scanning, generates a comprehensive analysis report for both human review and automated analysis, and validates that the modified chart remains functionally and structurally correct. The analysis consists of the fix rates, cost metrics, report discrepancies, and deployment readiness assessments. This step is arguably the most important output from the pipeline since it gives engineers or developers everything in order to evaluate the agent's work, which findings were fixed and how, which remain undone and why, what the agent changed and why, and whether the result is safe to deploy. This report is for human reviewers to see the verified fixes, check the changes and take a deployment decision.

This final phase starts by re-rendering the final modified chart and runs both Trivy and Checkov. Before trusting these results, the pipeline performs several integrity checks: verifying that rendering succeeded, and checking that the rendered output exceeds a minimum size threshold (to catch templates that silently produce empty output). If any of these checks fail, the pipeline treats the remediation as having produced zero fixes and falls back to the original scan results. This conservative approach prevents inflated fix rates from broken charts.

The results are then compared against the original baseline. At the *unique-ID level*, the pipeline tracks which unique IDs were fully resolved, partially resolved, or remain unfixed. At the *instance level*, it tracks individual finding occurrences, because a Checkov rule for example can alert against multiple containers across multiple templates, and the agent may fix some instances but not others. Checkov findings are tracked as (ID, target) pairs to enable instance-level tracking. Each ID is then categorised as fully fixed (all instances resolved), partially fixed (some instances resolved), or fully remaining (no instances resolved). The N/A and manual-review findings are excluded with the same classification mechanism used previously so this way the fix-rate percentage reflects only actual fixable issues. A key output is the discrepancy between the agent's self reported fix count and the ground truth results. It extracts the agent's fixed findings list from the JSON summary and compares the count against the verified fix count. Discrepancies are flagged as indicators of self-report drift.

Later the pipeline sends the complete set of original files, fixed files, and unified diffs for each modified file to the same LLM model for an independent

verification review. Unlike the adversarial review, which operates under a red-team persona, this review operates as a security auditor focused on traceability and risk classification.

The verification report consists of traces of each fixed finding to the specific diff section that resolve it and classifies each change into one of three categories. It assigns a risk tier with **LOW** for additive security hardening, **MEDIUM** for changes that alter runtime behaviour, and **HIGH** for changes that override user-configurable values or modify Helm template logic. The pipeline also performs a deterministic coverage check for the audit, checking whether the reviewer actually addressed every file that was modified. Files that are not mentioned in the review are appended as an unreviewed warning, ensuring that nothing slips through uncovered. The chart is then tested against *Helm lint* for template syntax issues and *Kubeconform* that validates the rendered YAML against official Kubernetes API schemas, catching invalid field names, incorrect types, and structural errors that helm template alone does not detect. *Functional equivalence* comparison renders both the original and modified charts, parses the output into structured Kubernetes resource objects keyed by kind and name, and performs a semantic diff. This comparison checks for removed or added resources, changes to resource kinds, removal of containers, changes to container ports, removal of environment variables, removal of volume mounts, changes to commands or args, and changes to image repositories. A separate check verifies that when readOnlyRootFilesystem: true is set, a corresponding /tmp emptyDir mount exists.

Two files are produced. The JSON report containing the complete dataset with metadata, cost metrics, results at both ID and instance level with full finding details, the agent's self-report, the structured verification and adversarial review outputs, the complete diff, the audit log, and the original file contents for every modified file. The Markdown report presents the same information in a readable format with tables, deployment decision rationale, and the full LLM verification and adversarial review.

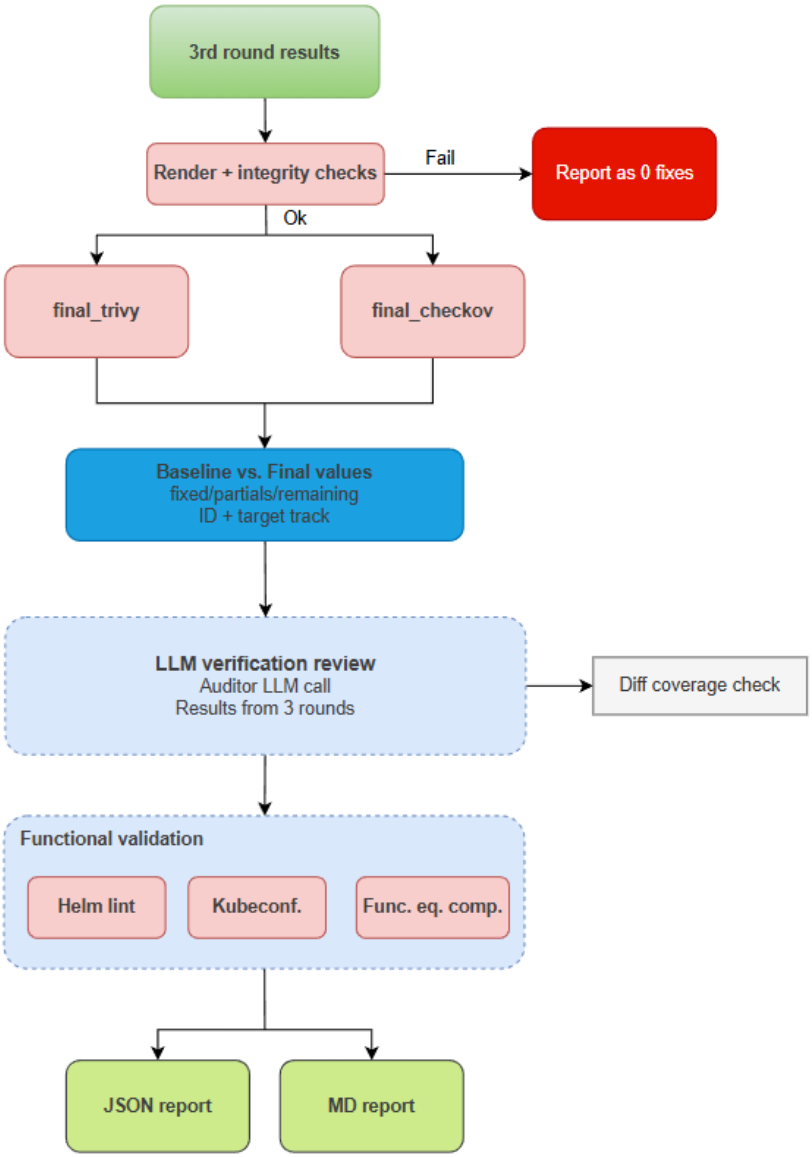

Figure 3.8: Deterministic verification and report generation. The final chart is subjected to integrity checks. Final Trivy and Checkov scans and their results compared against the pre-remediation baseline, tracking fixed, partially fixed, and remaining findings at both unique-ID and instance level. An LLM auditor review traces each fix to its corresponding diff, assigns risk tiers, and classifies behavioural changes, a deterministic diff coverage check verifies that every modified file was addressed. Functional validation, kubeconform schema validation, and functional equivalence comparison is done as well. The pipeline concludes by producing a structured JSON report and a Markdown report for human review.

3.6 Assessing reliability and validity of the data collected

3.6.1 Validity of method

The validity of this study rests on a multi-step verification architecture where every layer addresses a validity dimension, and the combination strengthens confidence in the reported results.

The foundational validity mechanism is the use of deterministic scanning as ground truth for fix-rate measurement. Rather than relying on the LLM's assessment of whether a fix was successful, the pipeline re-runs the same scanners that identified the original findings. A finding is counted as fixed if the scanner no longer reports it after remediation. Every remediated chart is also validated against Kubernetes resource schema using kubeconform, to catch structural imperfections that scanners would not detect. A chart that passes security scanning but fails schema validation would be undeployable, making this check essential for the validity of a deployment-readiness claim. The pipeline renders both the original and remediated charts and performs a structured comparison of the resulting Kubernetes manifests as an additional validity check. This check looks for removed resources, changed resource kinds, altered container ports, deleted environment variables, and missing volume mounts. Such changes can go unnoticed by a scanner, a high fix rate that comes at the cost of removing functionality or altering application behaviour would produce misleading results. Functional equivalence validation ensures that security improvements do not come at the cost of breaking the application.

During the agentic remediation loop itself, a violation checker runs after every write operation. It detects and if necessary, automatically reverts changes that would compromise the chart's integrity. This check operates actively in the remediation loop and prevents errors from compounding across iterations and ensures the agent cannot drift from the original chart during multi-round remediations. After the remediation passes complete, the pipeline initiates an adversarial review to evaluate the changes. If severe issues are identified, a third remediation pass is triggered to address them. An important validity mechanism in the pipeline is the verification report from the final stage.

This report serves as the **primary decision surface for human reviewers.** It is designed to provide everything an engineer needs to evaluate the agent's work and contains:

- Trace of each fixed finding to its corresponding change.
- A behavioural change analysis that classifies every modification.
- Remaining unfixed findings with remediation instructions and severity assessment.
- Manual review items that require human intervention with specific reasoning for why automation is inappropriate.
- A final deployment readiness verdict.

The verification report is a crucial validity contribution since it provides an evidence-based assessment that goes beyond simple pass or fail, and contributes with transparency necessary for human oversight.

In addition, the same pipeline architecture, scanner versions, prompt templates, organisational requirements file, and chart set are used across evaluation. The variable that changes between experimental runs is the LLM model itself, for cross-model comparisons. The observed differences are attributed to the model capabilities rather than to pipeline level variations.

The classification of findings is driven by an organisational requirements file that encodes actual deployment context, and where certain security decisions require human oversight. The scope of the automated remediation reflects organisational needs.

3.6.2 Reliability of method

The deterministic components of the pipeline such as scanning, violation checking, schema validation, functional equivalence validation, report generation are reproducible given identical input files. However, repeatability and reproducibility is somewhat limited by the non-deterministic nature of LLMs. Even with identical inputs and the same model, it can produce different outputs and different file modifications.

3.6.3 Data validity

Scanner findings are not all uniformly valid security issues. Three categories require different treatment. Not applicable findings are scanner checks that do not apply to the deployment context. These are excluded from fix-rate calculations entirely and identified through the organisational requirements file. Manual review findings are genuine security concerns that cannot be safely automated. These are excluded from the agent's remediation scope but reported in the output for human intervention.

3.6.4 Reliability of data

The non determinism of LLMs is a problem to data reliability. The study addresses this rather than try to eliminate it. All measurable variables such as variation in fix rates across charts are tracked and reported. The use of a single pipeline architecture across different models ensures that differences in performance are credited to the models rather than to implementation variations. Metrics such as token counts, timing data, and cost estimates are recorded by the pipeline's audit log.

3.7 Evaluation framework

The pipeline is evaluated by running it against every chart using GPT-5, GPT-5-mini, GPT-5.1, GPT-5.2, GPT-5.4, and GPT-5.4-mini. The chart–model runs output a JSON file containing pre-scan and post-scan findings at both ID and instance level, the agent's self-reported fixes, token usage, cost, iteration count, write rounds, violation checker outcomes, functional equivalence results, adversarial review verdicts, and the verification report.

The first evaluation aspect concerns the pipeline's capability and boundaries, whether an agentic LLM pipeline can meaningfully improve security validation of Helm charts, and where the reliability and safety limits of autonomous remediation lie. This is assessed with these principal metrics: Fix rate measures the proportion of findings remediated, computed as findings no longer reported by the deterministic scanners divided by total fixable findings, and is reported at both unique-ID level and instance level. Remediation quality, derived from verification report. Functional equivalence pass rate measures the proportion of runs where helm template diffing detects no regressions. A failure taxonomy documents what kinds of findings the pipeline consistently cannot fix and why, and model specific failures where the agent

attempts a fix but applies it incorrectly.

The second evaluation aspect concerns cross-model comparison, how different API models perform when used as the reasoning engine within the same pipeline architecture. Because the pipeline, prompts, scanners, and chart set are held constant, observed differences are attributable to the models themselves. This is assessed through five metrics. Fix rate per model measures remediation accuracy using the same ID-level and instance-level as the first dimension. Consistency is measured by the coefficient of variation in fix rates across charts within each model, a model that achieves similar fix rates across structurally different charts demonstrates more reliable behaviour than one with highly variable performance. Coverage measures the proportion of distinct findings that a model successfully addresses. self-report accuracy rates is derived from the self-report versus ground-truth discrepancy per model, capturing differences in self-assessment accuracy. Token usage and cost per chart measure efficiency, reported as total input tokens, output tokens, and estimated cost.

3.7.1 API Models Used in Comparison

GPT-5-mini: GPT-5-mini is a smaller version of GPT-5. It is faster and more cost-efficient, priced at $0.25/$2.00 per million input/output tokens, with a 400,000 context window, 128,000 max output tokens, and reasoning token support. As a cheap reasoning-capable model it functions at the cost-efficiency floor [89].

GPT-5: GPT-5 is a model for coding, reasoning, and agentic tasks, priced at $1.25/$10.00 per million input/output tokens, with a 400,000 context window, 128,000 max output tokens, and reasoning token support [90].

GPT-5.1: GPT-5.1 is a model for coding and agentic tasks, priced at $1.25/$10.00 per million input/output tokens, with a 400,000 context window, 128,000 max output tokens, and reasoning token support [91].

GPT-5.2: GPT-5.2 is a previous frontier model for complex professional work, priced at $1.75/$14.00 per million input/output tokens, with a 400,000 context window, 128,000 max output tokens, and reasoning token support with effort levels extended to *xhigh* [92].

GPT-5.4: GPT-5.4 is a frontier model for coding and professional work, priced at $2.50/$15.00 per million input/output tokens, with a 1,050,000 context window, 128,000 max output tokens, and reasoning token support [93].

GPT-5.4-mini: GPT-5.4-mini is a smaller version of GPT-5.4 designed for

high-volume workloads, priced at $0.75/$4.50 per million input/output tokens, with a 400,000 context window, 128,000 max output tokens, and reasoning token support [94].

Chapter 4

Results and Analysis

This chapter presents the empirical results of the evaluation and interprets what they reveal about the capabilities and limits of the pipeline. Section 4.1 reports the comparison between the constrained and fully agentic configurations that justifies the configuration used for the remaining results, after which the per-chart and per-model fix rates, validation-gate outcomes, and failure modes are reported and analysed.

4.1 Pipeline Configurations

Section 3.5.5.1 introduced the decision to restrict the remediation agent to file writing and reading operations while front loading rendered manifests, targeted files that need fixing, fix hints, and a structured task list into the initial context. A comparison was conducted that validates that decision. The constrained configuration is evaluated against a fully agentic variant in which the agent retains access to all tools and other elements of the pipeline are held constant. Both configurations were run on the full chart corpus using `GPT-5` and `GPT-5.4`, the two frontier models in the evaluation set.

Figure 4.1 presents the instance level fix-rate per chart for the four chart–configuration–model combinations. Across the corpus, the constrained configuration primarily matches or outperforms the fully agentic variant on the majority of charts, and the gap is most pronounced under `GPT-5`. Under `GPT-5.4` the average gap is smaller, with the fully agentic variant even outperforming the constrained one on five charts. This is plausibly attributed to frontier models partially absorbing the overhead of unrestricted tool access. The gap widened drastically on the smaller-tier models tested during development (`GPT-5-mini`, `GPT-5.4-mini`), indicating that the

cost of unrestricted tool access scales inversely with model capability.

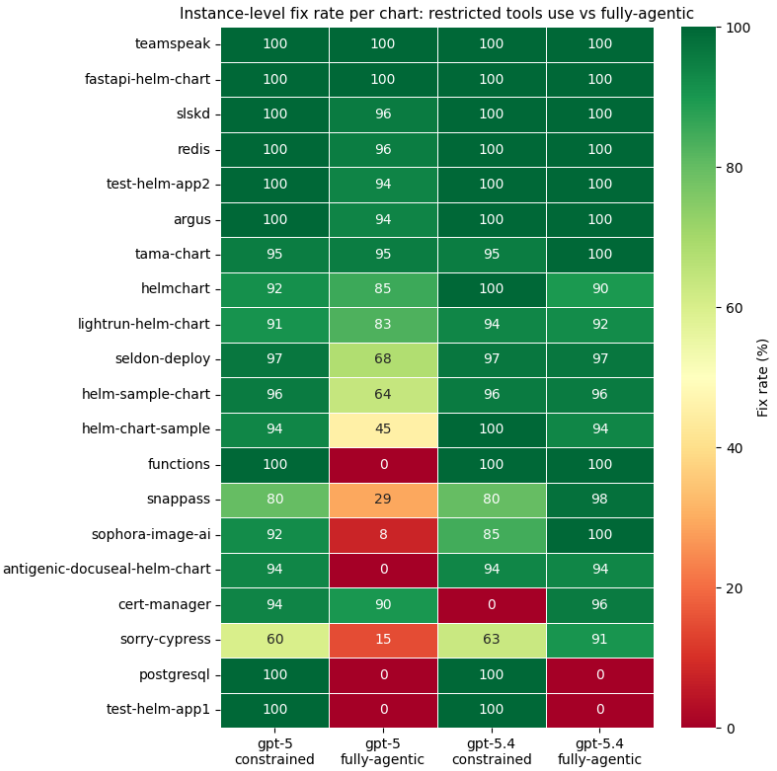

Figure 4.1: Instance level fix-rate per chart for `GPT-5` and `GPT-5.4` under the constrained (restricted tool use) and fully agentic (all tools) configurations.

The token consumption in Figure 4.2 reveals the cost of unrestricted tool access. Under `GPT-5.4`, the unrestricted configuration consumed a mean of 352.7k tokens per run versus 252.4k for the constrained configuration, an overhead without a corresponding accuracy gain. For `GPT-5`, the mean token consumption is closer between configurations (291.7k versus 242.7k), and the constrained configuration additionally yields more accurate remediation and higher overall fix rates.

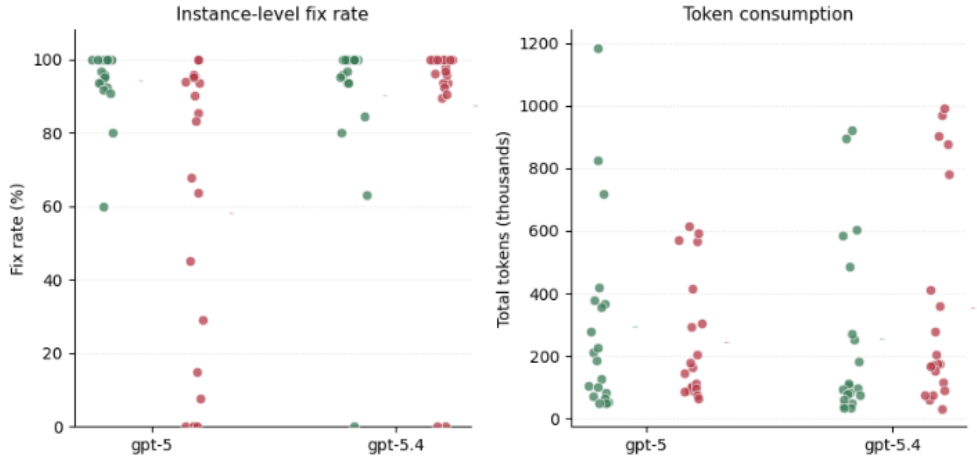

Figure 4.2: Per-chart performance of the constrained (green dots) and fully agentic (red dots) pipelines on `GPT-5` and `GPT-5.4`, showing instance level fix-rate and total token consumption. Each dot represents one chart.

The constrained design was originally adopted during development, when the pipeline was iterated primarily against the smaller models in the GPT-5 family (`GPT-5-mini`, `GPT-5.4-mini`, `GPT-5`). On these models the gap between configurations was substantial where the unrestricted agentic runs frequently exhausted token budget and produced poor results compared to the constrained configuration. Therefore reported results for the rest of this section are from the constrained configuration.

4.2 Remediation Pipeline Results

Figure 4.3 presents the instance level and ID level fix rates for each chart–model combination across 20 charts and 6 models in 120 runs.

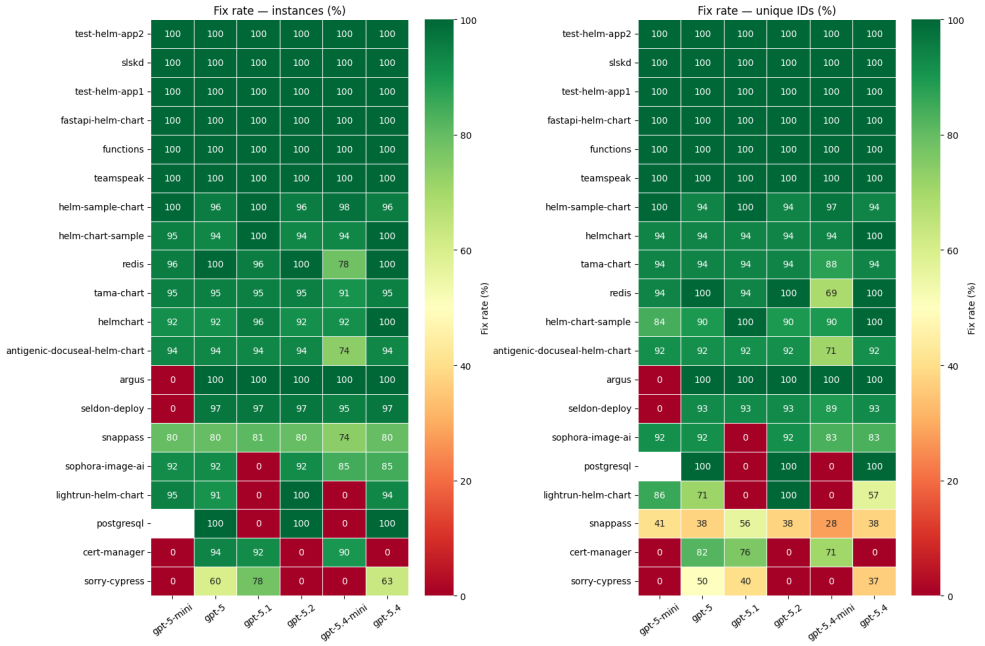

Figure 4.3: Fix rates at instance level (left) and unique-ID level (right) for all 20 chart–model combinations across six models and 120 runs. Green indicates high fix rates, orange indicates medium fix rates, and red indicates low or zero.

4.2.1 Fix Rates Across Chart–Model Pairs

The pipeline achieved an overall median instance-level fix rate of 95.7% and a mean of 84.5% across 120 runs. The gap between the fix-rates reflects the bimodal character of the distribution, the majority of runs cluster between 90–100%, while a small number of zero-fix runs caused by structural YAML corruption, pipeline timeouts, token usage limits, or iteration count limits. The pipeline can be characterised as reliable on charts it can handle, with failure concentrated in a distinct set of hard cases rather than distributed as degraded performance across the corpus.

The six charts achieving 100% across all models are all single-workload deployments with straightforward template structures, where security fields either use direct `.Values` references or have no existing security configuration at all. These represent the simplest fix patterns and serve as a reliability baseline. The subsequent six charts, all above 90% mean fix rate, share similar structural properties but introduce modest complexity such as partial security context definitions or conditional blocks. Together, these twelve charts demonstrate that the pipeline handles the large majority of

standard Helm chart architectures with high consistency.

4.2.2 Instance-Level vs. Unique-ID-Level Fix Rates

Instance-level fix rates are consistently higher than unique-ID-level fix rates for the same chart–model combination. The distinction is that a single finding ID may appear as multiple instances across different workload resources within the same chart. When the agent fixes the finding in some files but not others, the instance-level rate registers this partial progress, while the ID-level rate counts the finding as unresolved because at least one instance remains. The ID-level fix rate therefore reflects what a security reviewer would consider fully remediated and is the more conservative and operationally relevant metric of the two.

The missing entry for `postgresql` under `GPT-5-mini` represents a timeout. Remediation exceeded the 20-minute ceiling and was recorded as a non-result rather than a zero-fix run. Aggregate statistics in this section are therefore computed over 119 runs rather than 120.

4.2.3 Model Comparison

Figure 4.4 shows the instance-level fix rate distribution for each model, with the mean marked as a horizontal bar. The results reveal a clear performance hierarchy, but also a consistent distributional pattern.

GPT-5 is the most consistent performer, achieving the highest mean instance fix rate of 94.28% and the lowest coefficient of variation at 9.80%. It is the only model whose worst-performing run still achieves at 60%, indicating that it degrades gracefully on difficult charts rather than failing outright. **GPT-5.4** ranks second in mean fix rate at 90.41%, but with a substantially larger CV of 25.00%, meaning that while its average performance approaches GPT-5, it does so less predictably.

The mini-tier models, **GPT-5-mini** and **GPT-5.4-mini**, exhibit the highest variance, with CVs of 51.60% and 43.10% respectively. Both achieve high fix rates on simpler charts but collapse to 0% on complex charts where the larger models succeed, suggesting that complex multi-file reasoning tasks exceed their capacity. **GPT-5.1** and **GPT-5.2** occupy the middle ground, with mean fix rates of 82.11% and 87.57% and CVs in the 33–42% range.

4.2.3.1 Fix Rate Distribution

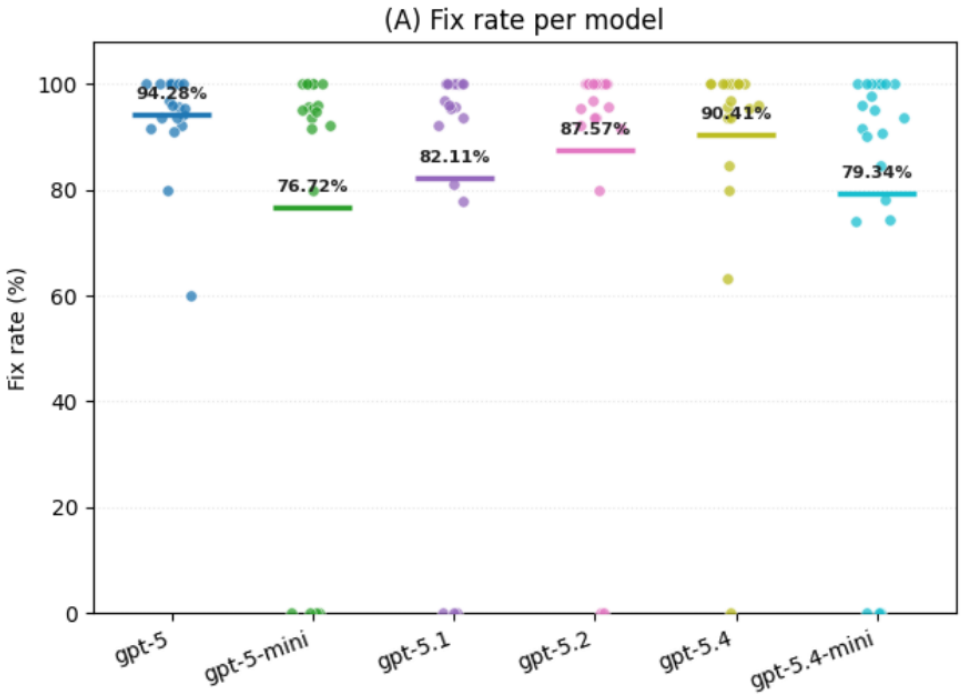

Figure 4.4: Instance-level fix rate per chart run for each model. Each dot represents one chart–model combination. The horizontal bar denotes the mean.

The plot in Figure 4.4 reveal that the distribution of fix rates is bimodal rather than continuous. For GPT-5, dots cluster tightly in the 90–100% range with only two outliers at approximately 60% and 80%. For all other models, the pattern is a dense cluster near 80–100% accompanied by a smaller cluster at or near 0%, with relatively few observations in between.

This bimodal structure has an important implication for interpreting mean fix rates. A mean of 87.57% for GPT-5.2, for instance, does not indicate that a typical run resolves around 87% of findings. It indicates that most runs resolve close to 100% while a minority resolve close to 0%. The dominant failure mode is therefore binary, the agent either produces a near-complete fix set or fails to make meaningful progress at all. Partial degradation as chart complexity increases is not the primary pattern, instead it becomes a complete failure on a subset of charts.

4.2.3.2 Model Consistency Across Charts

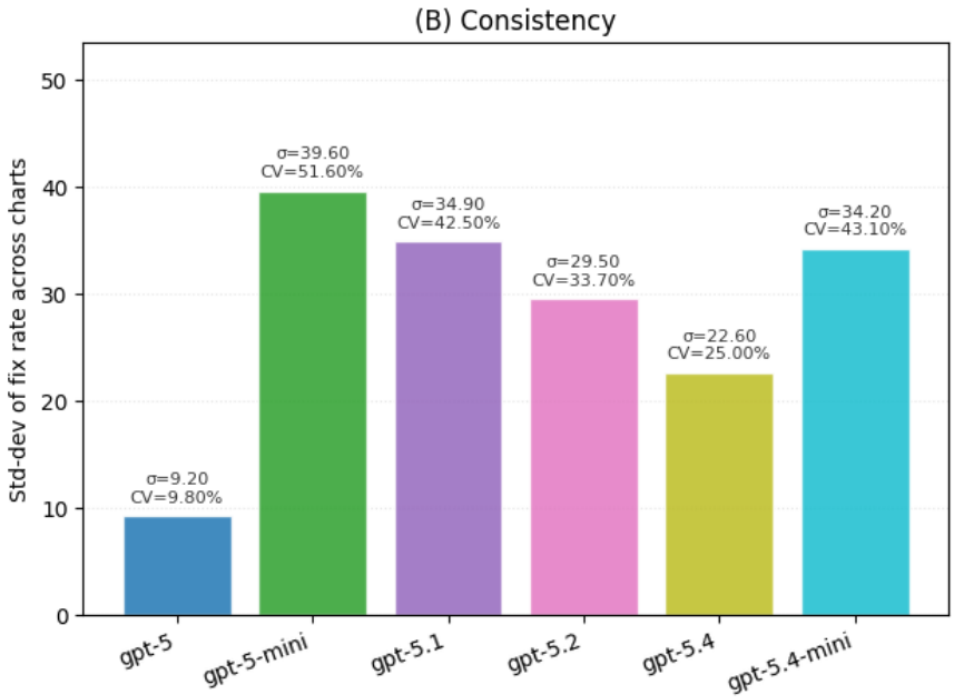

Figure 4.5: Consistency of fix rates across charts per model, measured as standard deviation (σ) and coefficient of variation (CV). Lower values indicate more predictable performance.

Figure 4.5 shows the standard deviation of instance-level fix rates across charts for each model, with the coefficient of variation annotated above each bar. The standard deviation captures how much a model's fix rate varies from chart to chart, a high σ means the model produces very different outcomes on different charts, for instance, near-perfect results on some and complete failures on others. The CV normalises this by the mean, expressing variability as a proportion of average performance. This makes it an appropriate comparison metric across models with different mean fix rates, since a model with a high σ but also a high mean may actually be more reliable in relative terms than a model with a lower σ but a much lower mean.

GPT-5 is substantially more consistent than all other models, with $\sigma = 9.20$ and CV $= 9.80\%$, more than four times lower CV than the next best model. This means that GPT-5's fix rate is not only high on average but also stable, an engineer deploying it can expect similar performance across charts of varying complexity. All other models show CVs between 25% and 52%,

indicating that their average fix rates are driven by a mix of near-perfect runs and complete failures rather than uniformly good performance. A clear pattern also emerges across model tiers, in both GPT-5 and GPT-5.4 families, the full-size model has a substantially lower CV than its mini counterpart, suggesting that model scale contributes meaningfully to remediation reliability and not only to raw accuracy.

4.2.3.3 Cost per Fixed Instance

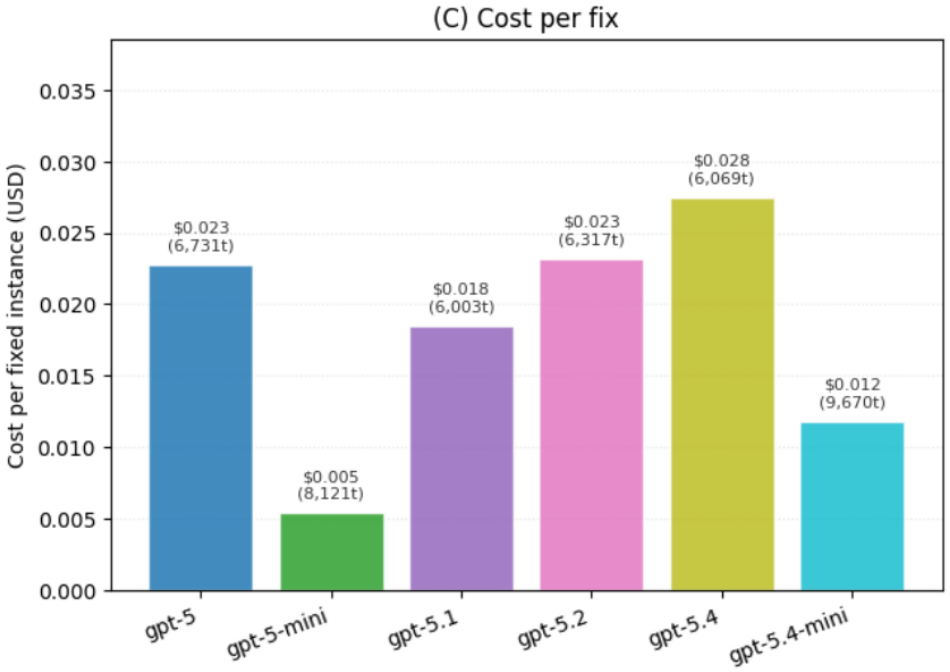

Figure 4.6: Cost per fixed instance (USD) for each model, annotated with total tokens consumed per fixed instance.

Figure 4.6 shows the cost per fixed instance, computed as total USD spent divided by total instances fixed across all charts. GPT-5-mini and GPT-5.4-mini achieve the lowest absolute cost per fix at $0.005 and $0.012 respectively, while GPT-5.4 is the most expensive at $0.028. The token annotations reveal a pattern that the mini models consume *more* tokens per fixed instance than their full-size counterparts, 8121 and 9670 tokens per fix versus 6731 and 6069, yet cost less because of their lower price per token. This suggests that smaller models require more iterations and produce more redundant output to achieve

the same fix, and that their cost advantage is a function of pricing rather than efficiency.

The cost-performance relationship that emerges from combining fix rate, consistency, and cost per fix is not what a straightforward capability-pricing assumption would predict. GPT-5.4, the most expensive model per fix, does not deliver the highest fix rate or the highest consistency. GPT-5, priced in the middle of the evaluated range, achieves the highest mean fix rate, the lowest coefficient of variation, and a cost per fix lower than GPT-5.4. When deploying this pipeline operationally, GPT-5 represents the dominant choice on all three dimensions simultaneously: accuracy, reliability, and cost-efficiency.

4.2.3.4 Token Consumption, Cost, and Fix Rate

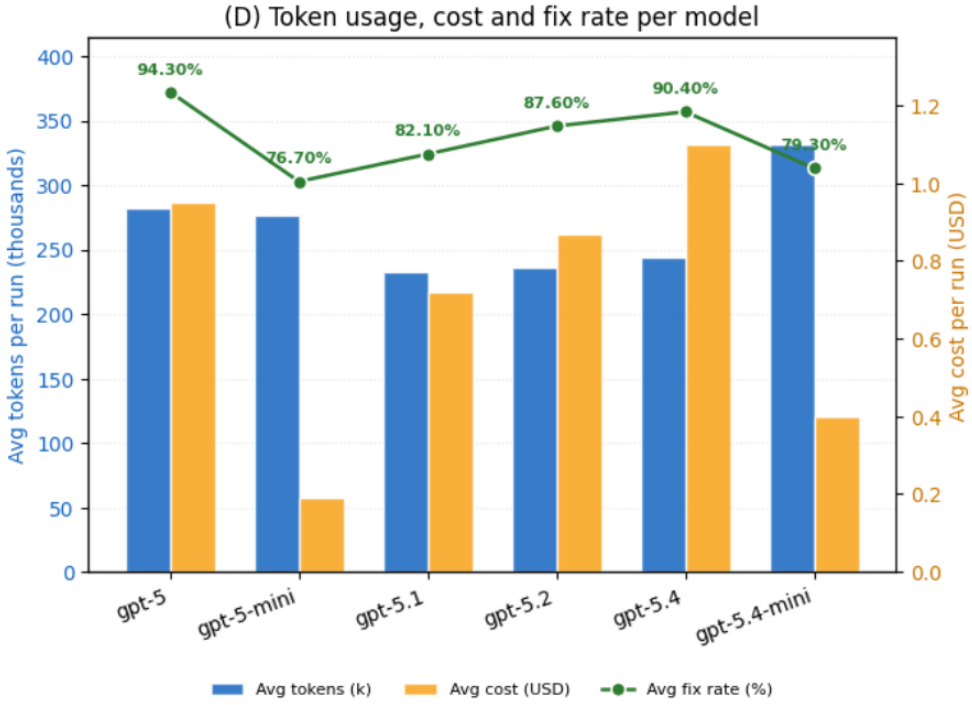

Figure 4.7: Average tokens per run (blue bars, left axis), average cost per run in USD (orange bars, left axis), and average fix rate (green line) for each model.

Figure 4.7 overlays token consumption, cost, and fix rate to examine the relationship between resource usage and remediation effectiveness. Average token consumption per run is broadly similar across models, despite differences in fix rate and cost. This convergence is expected. Input tokens

are dominated by chart file contents and the task list, which are identical across models for the same chart, while output tokens vary with each model's verbosity and iteration count.

The GPT-5 versus GPT-5-mini comparison illustrates this dynamic clearly. The two models consume on average nearly identical token volumes per run yet differ by nearly 18 percentage points in mean fix rate. GPT-5 costs approximately five times more per run, but converts that budget into substantially more verified fixes and far fewer complete failures. The takeaway here is that token volume is not a useful proxy for pipeline effectiveness, a less capable model can consume an equivalent context budget while making little or no meaningful progress. What differs between models is not how much they read but how effectively they reason over what they are given.

4.2.4 Failure Modes

The 0% results can be attributed to several factors. Pipeline-level failures where the model exhausted its token budget or time limit, structurally incorrect fixes that corrupted the chart and caused rendering to fail, or chart-specific issues where the agent needed to reason across multiple template layers simultaneously.

Several 0% runs do not represent failed remediation attempts but rather cases where the agent applied fixes that structurally corrupted the chart. In the GPT-5-mini run on `argus`, the agent modified `values.yaml` in a way that produced invalid YAML, preventing the chart from rendering. (This run was successfully recovered with *reasoning_effort=medium* in the retry experiment described in Section 4.3. In the GPT-5-mini run on `seldon-deploy`, the agent introduced a template syntax error that manifested as an unexpected EOF. In both cases the agent claimed numerous successful fixes, but because the corrupted chart could not be rendered by `helm template`, no scanner verification was possible and the fix rate was recorded as 0%. Both runs would have reported high fix rates based on the agent's own self-assessment alone.

A separate failure mode was observed in the cert-manager runs with GPT-5.4 and GPT-5-mini, where the agent applied numerous fixes but the remediated chart failed schema validation. Cert-manager ships with a `values.schema.json` file that constrains which fields may appear in `values.yaml` and what types they must have. The agent's modifications produced a chart that is valid YAML and syntactically correct Helm, but rejected by the chart's own validation layer. This failure could be mitigated by including `values.schema.json` in the pre-loaded file set provided to the

agent.

A further failure mode is context-window saturation, where the cumulative size of file contents, tool outputs, and conversation history grows large enough to start degrading the models ability to conduct the tasks properly. The `sorry-cypress` run with `GPT-5` consumed approximately 1.005 M input tokens and 0.179 M output tokens across 14 iterations and 12 write rounds, appending its result to the conversation history. The run completed with valid output, but achieved only a 60% instance-level fix rate. This degradation can be due to longer input increases the amount of information the model must reason over, which can decrease accuracy [95].

A severe failure mode observed is destructive rewriting, where the agent rewrites entire template files rather than applying targeted fixes, deleting large amounts of functional logic to produce templates that are simpler and pass security checks but would deploy a non-functional application. The `postgresql` run with GPT-5-mini exhibited this behaviour, which is particularly dangerous because, unlike YAML corruption or template syntax errors, a destructively rewritten chart may render successfully and pass all security scans, meaning only the functional equivalence check or human review would catch the regression. Similarly, examining the audit log for the `postgresql` run with GPT-5.4-mini showed that on the first iteration the agent rewrote four template files simultaneously, in subsequent iterations it repeatedly rewrote `values.yaml` in its entirety, regenerating the file from scratch rather than making targeted modifications, partially losing content with each pass until the run terminated at the 700k token ceiling. Rather than populating the `.Values` references that feed into the chart's security context blocks, the agent hardcoded values directly and rewrote template logic to match.

4.2.4.1 The Mediocre Runs of `snappass`

Another distinct failure pattern emerges on charts that complete successfully on every model yet plateau at mediocre fix rates uniformly across the fleet. The clearest example is `snappass`, which produced no zero-rate runs but converged to the same approximate ceiling at roughly 80% of instances and 26–56% of unique IDs, regardless of which model was driving the agent. This is different from the bimodal pattern seen on charts where some models score around 100% and others 0%. The `snappass` pattern reflects a structural property of the chart itself that no model in the suite was able to overcome.

`snappass` is a parent chart that bundles another subchart configured

for replication mode. The remediation surface spans six workload-bearing files across the parent and subchart, and each subchart statefulset hosts four container specifications. The pre-scan baseline records 185 finding instances across 43 unique IDs.

Inspection of, for instance, the GPT-5.1 run (81% instance, 56% ID) shows the agent executed three write rounds, edited all six target files, and produced structurally valid YAML throughout. The remaining findings requires coordinated edits across multiple containers in multiple files, with no single `values.yaml` key that toggles them all. Resource findings were resolved for the master, slave, and test pod but left untouched on the exporter and Sentinel containers, and a single uncovered container is sufficient to keep the ID flagged. High-UID findings were applied to the snappass pod but left in the inherited subchart, where changing it risks breaking filesystem ownership assumptions. Probe findings remained because the agent had no reliable way to infer endpoints for containers without fabricating health checks that could break the workload. The pattern is one of *distributed coverage*. Each individual edit is within reach but what defeats it is the container multiplicity within a template, where the agent reliably patches the dominant container but tends to leave something untouched once the principal fix is applied.

4.2.5 Self-Report Accuracy and Discrepancies

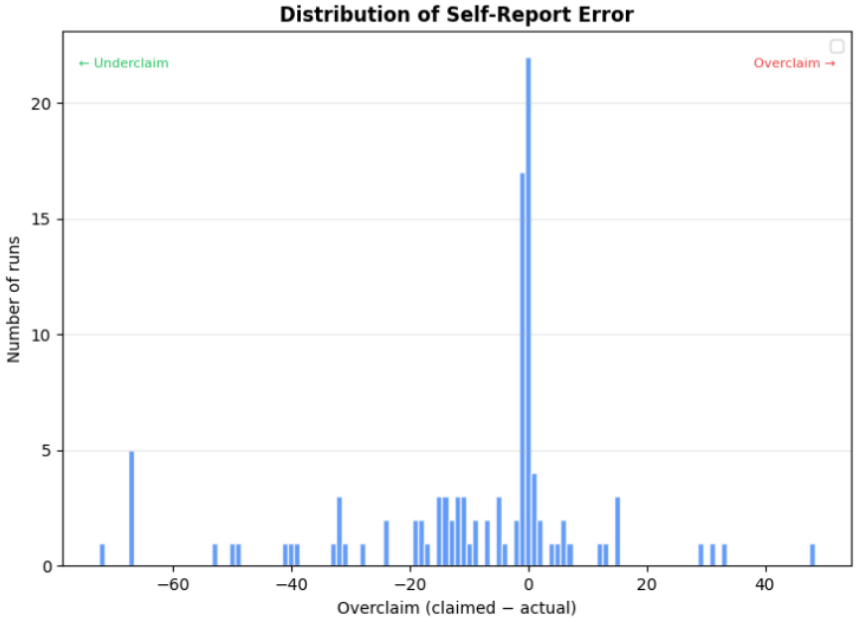

Figure 4.8: Distribution of self-report error (claimed fixes - ground-truth fixes) across runs. Negative values indicate underclaiming, positive values indicate overclaiming.

The pipeline is predominantly conservative rather than hallucinatory. Across all runs, 61.5% underclaim and only 18.3% overclaim. Underclaiming is the safer failure mode, an agent that underestimates what it has fixed prompts unnecessary human review, whereas an agent that overclaims may give false confidence in the security of an unverified chart.

Several underclaiming patterns have a clear structural cause. The `functions` chart shows every model claiming 32 fixed findings against a ground-truth count of 99, because the agent counts unique finding IDs addressed rather than the total number of instances resolved. The same pattern appears in `snappass` and `lightrun-helm-chart`. These are not hallucinations in the conventional sense but rather a mismatch between the agent's internal counting convention and the instance-level metric used for ground-truth verification.

Table 4.1: Worst overclaim cases.

Chart	Model	Claimed	Actual	Overclaim
functions	GPT-5.1	147	99	+48
sorry-cypress	GPT-5.4-mini	33	0	+33
argus	GPT-5-mini	31	0	+31
seldon-deploy	GPT-5-mini	29	0	+29
lightrun-helm-chart	GPT-5.1	15	0	+15
cert-manager	GPT-5.2	15	0	+15
cert-manager	GPT-5.4	15	0	+15
cert-manager	GPT-5-mini	13	0	+13
seldon-deploy	GPT-5.1	72	60	+12
docuseal-helm-chart	GPT-5	36	29	+7

Overclaims are the more consequential error type. The worst case, `functions` with **GPT-5.1**, claimed 147 fixed instances against a ground-truth count of 99, an overclaim of 48 caused by the agent counting each fix attempt independently, including re-applications after reverts, rather than counting net resolved findings. The remaining severe overclaims share a common cause, the agent corrupted the chart during remediation, preventing `helm template` from rendering and reducing the verified fix count to zero, while the agent's JSON summary still reported a large number of successful fixes. These cases demonstrate why deterministic re-scanning is essential at every verification stage — the agent's self-report cannot be used as a substitute for ground-truth scanner results.

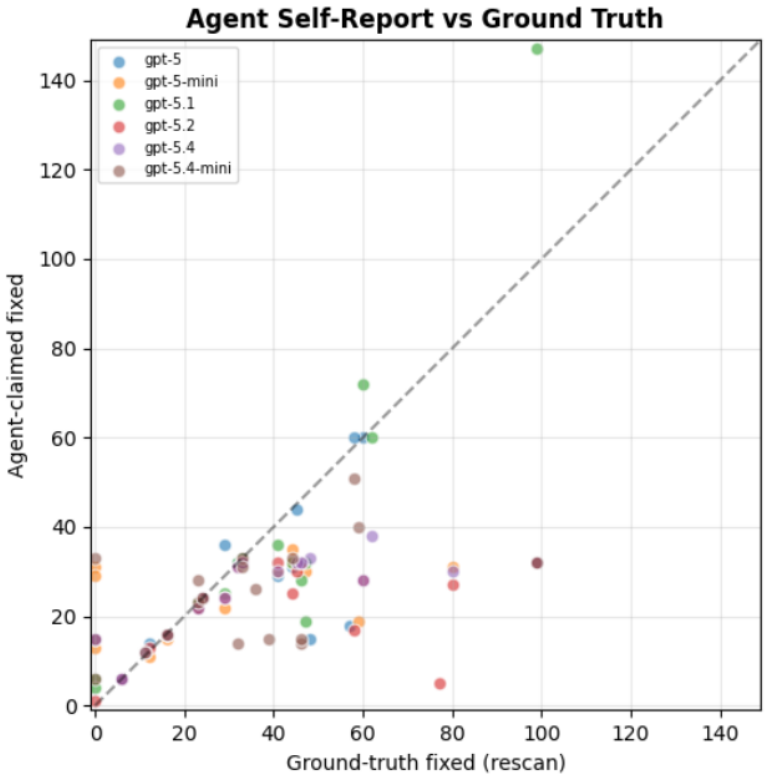

Figure 4.9: Agent self-reported fix count versus ground-truth fix count from deterministic re-scanning, per run. Points on the dashed diagonal indicate perfect self-report accuracy. Points below the diagonal represent underclaiming and points above represent overclaiming.

Figure 4.9 plots each run's agent-claimed fix count against the ground-truth count from deterministic re-scanning. Most runs fall below or on the diagonal, consistent with the overall underclaiming tendency. The cluster of points at zero ground-truth with non-zero claimed fixes corresponds to the chart corruption cases in Table 4.1.

4.2.6 Validation Gate Analysis

Before producing a deployment recommendation, the pipeline subjects each remediated chart to four validation gates: `helm lint` for template syntax validation, `helm template` for rendering correctness, kubeconform for Kubernetes API schema validation, and a functional equivalence check that performs a semantic diff of the rendered manifests against the original. Table 4.2 summarises the pass rates across all 119 runs.

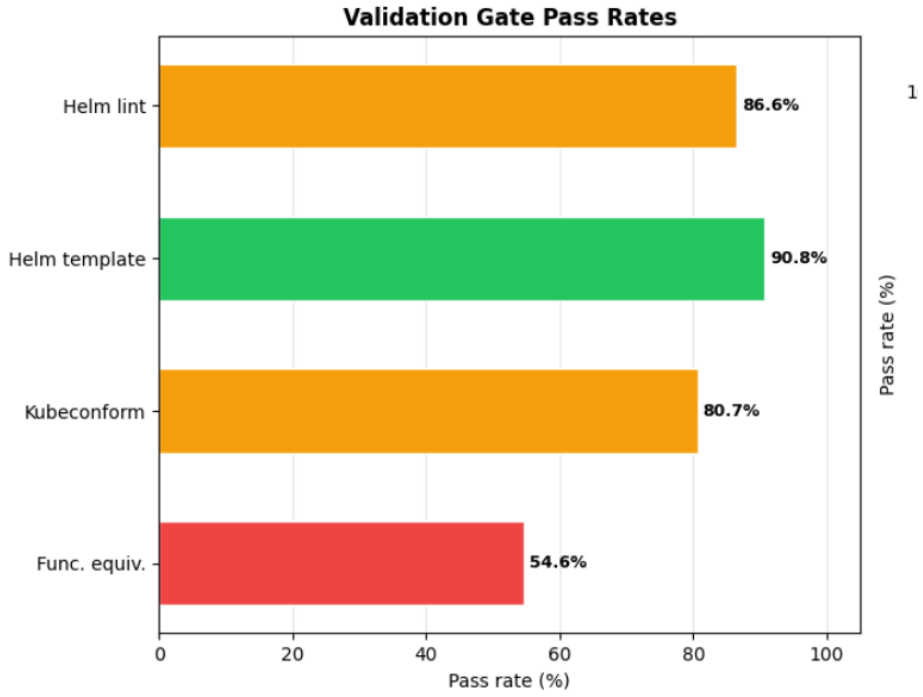

Figure 4.10: Validation gate pass rates across all 119 pipeline runs.

Table 4.2: Validation gate pass rates across all pipeline runs.

Gate	Pass	Fail	Rate (%)
Helm template	108	11	90.8
Helm lint	103	16	86.6
Kubeconform	96	23	80.7
Functional equivalence	65	54	54.6

The three structural gates indicate that the pipeline preserves chart integrity in the majority of runs. The 11 rendering failures correspond exactly to the runs that achieved a 0% fix rate, where the agent's modifications corrupted the chart structure sufficiently to prevent valid output. These runs propagate as failures across all downstream gates. Of the 108 runs that rendered successfully, 103 also passed `helm lint`. The 23 kubeconform failures combine two distinct groups, the 11 structurally broken runs, and 12 runs that rendered correctly but produced manifests violating the Kubernetes API schema.

4.2.6.1 Functional Equivalence and Behavioural Change

The functional equivalence gate has the lowest pass rate at 54.6%, but this figure requires careful interpretation. Security hardening by definition alters runtime behaviour, setting `readOnlyRootFilesystem: true`, changing UID and GID defaults, or modifying image pull policies are all intentional behavioural changes introduced by the pipeline. The functional equivalence gate therefore does not measure whether the pipeline failed, but rather classifies which runs made additive-only changes versus which made changes that modify runtime defaults.

Of the 54 functional equivalence failures, the most recurring issue was `readOnlyRootFilesystem: true` being set without a corresponding writable `emptyDir` mount for `/tmp`. This constitutes an incomplete fix, the security finding is resolved from the scanner's perspective, but applications that write temporary files will fail at runtime without a writable temporary directory. A rarer issue was port remapping from 80 and 443 to 8080 and 8443, likely introduced to comply with the non-root UID constraint, since binding ports below 1024 requires elevated privileges.

4.2.6.2 Validation Pass Rates by Model

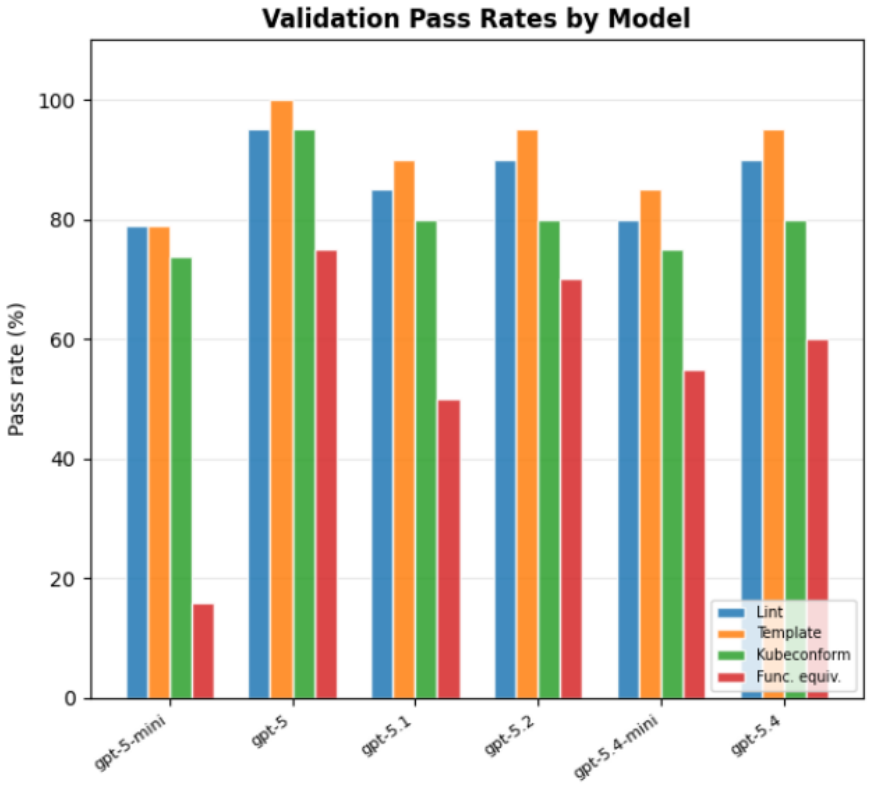

Figure 4.11: Validation gate pass rates by model.

Table 4.3: Per-model validation gate pass rates.

Model	Runs	Lint	Template	Kubeconform	Func. equiv.
GPT-5	20	19	20	19	15
GPT-5.2	20	18	19	16	14
GPT-5.4	20	18	19	16	12
GPT-5.1	20	17	18	16	10
GPT-5.4-mini	20	16	17	15	11
GPT-5-mini	19	15	15	14	3

GPT-5 achieves the highest pass rates across all four gates, with 20/20 renders, 19/20 on lint and kubeconform, and 15 of 20 runs maintaining functional

equivalence. GPT-5-mini performs notably worse across the board, with only 3 of 19 runs passing the functional equivalence check. This is consistent with its lower fix rate and the failure mode analysis where smaller models more frequently produce incomplete or structurally corrupted fixes that either prevent rendering or alter application behaviour beyond the scope of the intended security changes.

4.2.7 Behavioural Classification of Changes

The verification stage classifies every applied change by its behavioural impact on the chart. *Category A* covers additive-only security hardening, changes that add a missing field without altering any existing runtime behaviour. *Category B* covers behavioural modifications, changes that alter how the application runs, for instance tightening an existing security context field or changing a default value. *Category C* covers values overrides, changes that hardcode a value directly in a template rather than populating the corresponding `.Values` reference.

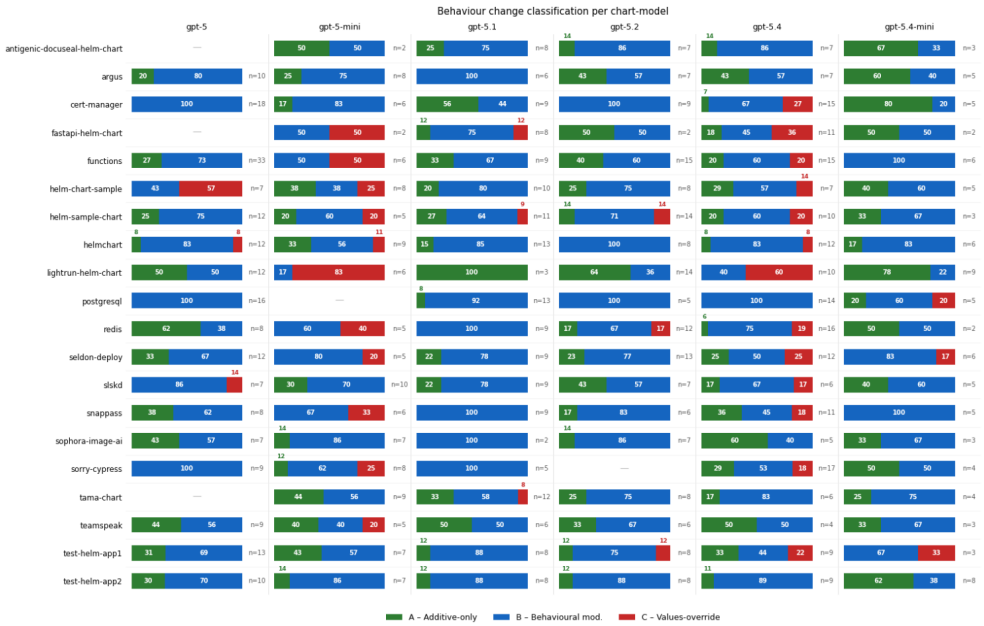

Figure 4.12: Behavioural change classification per chart–model combination. Each bar shows the proportion of applied changes classified as additive-only (A, green), behavioural modification (B, blue), or values override (C, red). Five runs produced no classifiable changes and appear as empty rows.

Figure 4.12 presents the classification breakdown for every chart–model

combination. Two patterns stand out. First, Category B dominates across all six models. This reflects the composition of the findings, since many of Trivy and Checkov findings concern existing fields that must be tightened. The pipeline therefore operates predominantly in a modification mode rather than an extension mode, and every Category B change represents a runtime-affecting edit that a human reviewer should examine before deployment.

Second, the Category C rate is strongly model-dependent, with **GPT-5-mini** and **GPT-5.4** producing the highest proportion of values overrides. Charts remediated by these models are less configurable than the original even when the fixes are technically correct, because hardcoded template values bypass the `.Values` interface that operators would use to customise behaviour. This is a safety concern distinct from fix correctness, a chart can pass all scanner checks while silently losing its configurability through Category C changes. For production deployment, Category C changes should be treated as requiring mandatory human review regardless of fix rate, since the scanner has no visibility into the loss of operator configurability they introduce.

4.3 Retrying Failed Runs with Higher Reasoning Effort

To investigate whether the 0% outcomes in Section 4.2 reflected a fundamental incapability of the smaller models or a single unlucky trajectory, every chart–model pair that scored 0% in run 1 was retried with `reasoning_effort=medium`, holding the rest of the pipeline constant. Twelve such pairs were re-executed across the five affected models. The retry was deliberately scoped to the failed cells only.

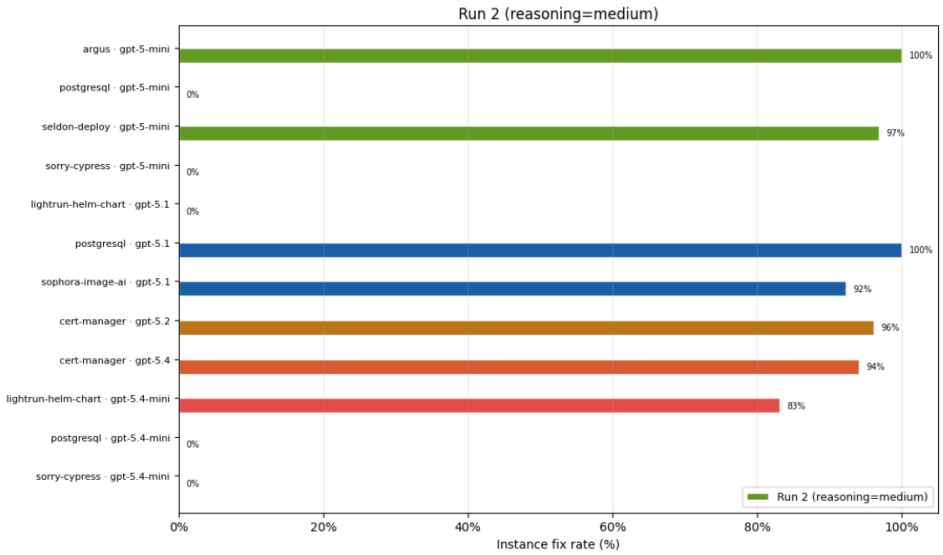

Figure 4.13: Instance-level fix rates for the 12 chart–model pairs that scored 0% in the primary evaluation, re-run with `reasoning_effort=medium`.

Seven of the twelve retried pairs recovered, six of them to fix rates above 90%. Five remained at 0%, postgresql under two different models and sorry-cypress under two. These two charts resist remediation regardless of model and reasoning budget, which separates two failure modes that the original run-1 data alone could not distinguish. The first is reasoning-bound failure, where the default effort is insufficient for the agent to plan a coherent multi-file edit but a higher effort succeeds. The second is architectural failure, where the chart structure exceeds what the present pipeline can remediate at any effort level.

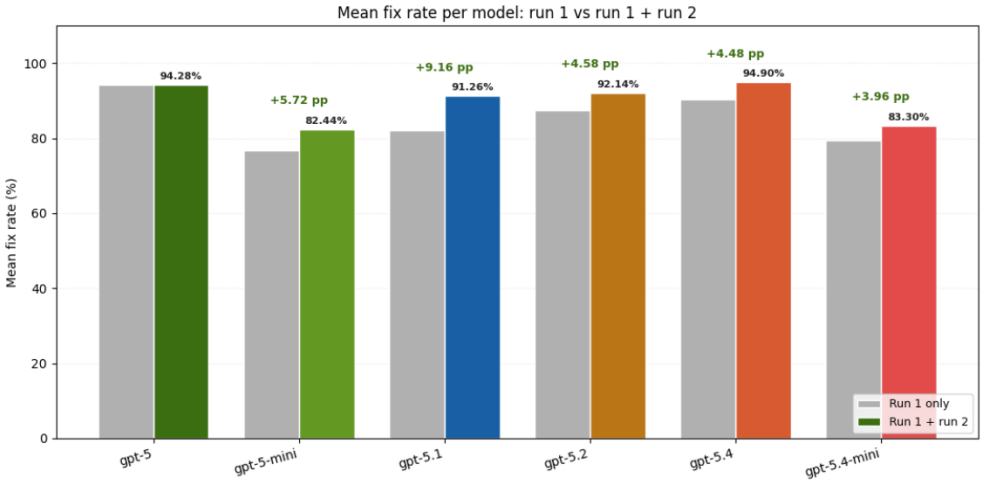

Figure 4.14: Per-model mean fix rate, run 1 only versus run 1 with the run-2 retries overlaid.

Table 4.4: Per-model mean instance fix rate before and after the run-2 overlay.

Model	n_{run2}	Mean before (%)	Mean after (%)
GPT-5	0	94.28	94.28
GPT-5-mini	4	76.72	82.44
GPT-5.1	3	82.11	91.26
GPT-5.2	1	87.57	92.14
GPT-5.4	1	90.41	94.90
GPT-5.4-mini	3	79.34	83.30

The recovery is uneven across models. GPT-5.1 gains the most (+9.16 pp) because two of its three retries flipped from 0% to roughly 100%. The smaller-tier models, GPT-5-mini and GPT-5.4-mini, also gain in fix rate, but their overall mean stays below the larger models because their failures concentrate on the structurally hardest charts, which remain unfixable even with more reasoning.

Per-model mean fix rates with 95% confidence intervals are summarised in Table 4.5.

Table 4.5: Mean instance-level fix rate per model with 95% confidence intervals. Run-1 figures use a single run per chart at default reasoning effort. Post-retry figures overlay the run-2 results, in which the run-1 zero-rate cells were re-executed at `reasoning_effort=medium`.

Model	Run-1 mean (95% CI)	Run-2 mean (95% CI)
GPT-5	94.28 [90.0, 97.5]	94.28 [89.9, 97.5]†
GPT-5-mini	76.72 [58.0, 91.9]	82.44 [67.2, 95.9]
GPT-5.1	82.11 [66.4, 95.4]	91.26 [80.9, 97.7]
GPT-5.2	87.57 [73.7, 97.7]	92.14 [82.0, 98.2]
GPT-5.4	90.41 [79.5, 97.6]	94.90 [90.7, 98.1]
GPT-5.4-mini	79.34 [63.8, 92.1]	83.30 [69.9, 93.7]

†GPT-5 had no zero-rate cells and therefore no retries, so its run-1 and run-2 figures are identical

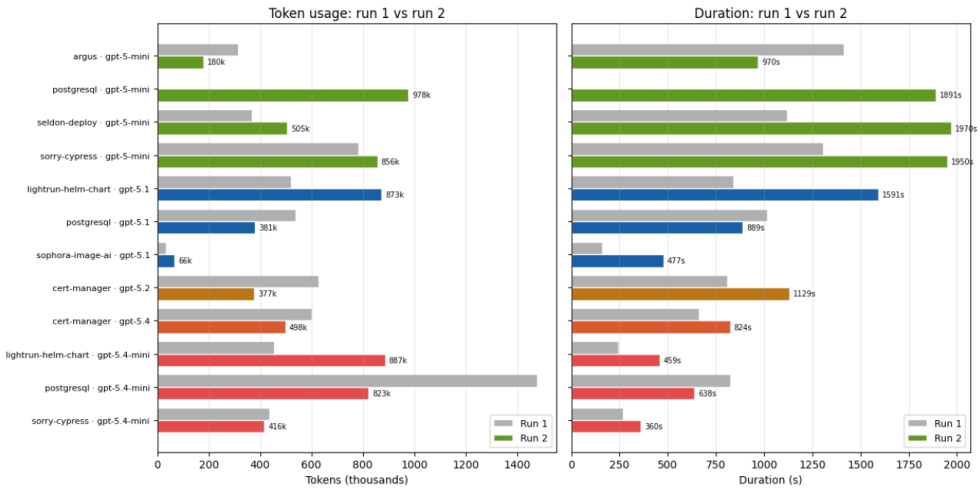

Figure 4.15: Token usage and duration per pair, run-1 versus run-2.

Figure 4.15 shows that the cost of the second run does not behave as a simple multiplier of the first. Token usage moves in both directions across the twelve pairs: in some cases it increases substantially, in others it decreases. Wall-clock duration is more consistently elevated in run-2, which matches the documented behaviour, the model spends additional hidden reasoning tokens before producing visible output, lengthening per-iteration latency even when total tokens are similar or lower. Token usage and successful remediation are therefore decoupled, more tokens do not guarantee a better outcome, and a

successful retry can be cheaper than the original failed run if the failure mode was a stale loop.

4.3.1 Implications for Pipeline Deployment

The retry experiment has two concrete implications for pipeline deployment. First, `reasoning_effort=medium` is a viable recovery mechanism for a non-trivial fraction of zero-rate failures, but it is not a general substitute for the default configuration. The cost increase is real even when token usage is similar, and the majority of charts already succeed at default effort. The appropriate deployment pattern is therefore two-tier, the pipeline runs at default effort and escalates to medium reasoning only on detected failure, avoiding the overhead on the charts where it is unnecessary.

Second, the persistent failures on postgresql and sorry-cypress across all models and both effort levels constitute a useful negative result. They identify the chart architectures that the present single-agent constrained configuration cannot remediate, and therefore mark the practical boundary of the pipeline's applicability. Addressing these charts in a future iteration would require extending the agent's awareness of dependency loaders and multi-workload coordination, neither of which is addressable through reasoning effort alone.

4.4 Reliability Analysis

The deterministic components of the pipeline, pre- and post-remediation SAST scanning, schema validation, functional equivalence checking, and the violation checker — are fully reproducible given identical input files. Any variance across runs therefore reflects variance in the LLM agent's behaviour, not in the measurement instruments. The OpenAI models do not guarantee identical outputs for identical inputs. For reasoning models, the principal controllable parameter is reasoning effort, which is held constant across all primary runs and increased to medium only in the retry experiment described in Section 4.3.

The consistency analysis in Section 4.2.3.2 serves as the primary reliability signal available within the constraints of this study. In the primary evaluation each chart–model combination was executed once, so CV values reflect how reliably a model performs across charts of varying complexity rather than run-to-run stability on a fixed input. The retry experiment in Section 4.3 provides limited additional data on within-pair stability, but only for the 12 pairs that originally failed. The bimodal fix rate distribution where runs cluster near

100% or near 0% with few intermediate outcomes indicates that the dominant failure mode is binary, which has direct consequences for how mean fix rates should be interpreted operationally.

The single-run-per-combination design is a reliability limitation. A 0% result on a particular pairing may reflect either a consistent inability of that model to handle that chart or a single unlucky trajectory that would resolve on a second attempt; the two cannot be distinguished without repeats. This constraint is primarily one of cost, the cumulative expense of the full model–chart matrix approached the budget available within the thesis timeframe. Repeated runs are a valuable extension of this evaluation.

4.5 Validity Analysis

The fix-rate metric is defined as the proportion of pre-scan findings no longer reported by the same tools after post-remediation scanning. This definition is narrow by design: it measures whether the pipeline resolves the findings that Trivy and Checkov report, so its scope is bounded by the rule coverage of those two scanners. Findings outside their coverage, such as application-layer vulnerabilities, supply-chain compromises, or organisation-specific policies not encoded as scanner rules, are invisible to the pipeline and absent from the reported fix rates. The metric is internally consistent because the same scanners define both the problem and the success criterion, but it does not support any claim that a chart achieving 100% fix rate is secure in a broader sense. The verification report's categorisation of findings into fixable, not applicable, and requires-manual-review partially compensates for this by surfacing non-automatable findings explicitly, and the deployment readiness verdict incorporates a judgement that goes beyond the fix-rate number.

The reporting gap proxy compares the agent's self-reported fix count against the deterministic scanner ground truth. It measures the degree to which the agent's self-assessment matches the artefact that the pipeline actually produces, rather than hallucination in the broader LLM sense. It captures only the overclaim and underclaim failure mode against a verifiable tool. Other failure modes, such as generating syntactically valid but functionally incorrect YAML, are captured by separate validation stages and are not conflated with the self-report metric. The functional equivalence diff check addresses the specific threat of a pipeline that achieves high fix rates by stripping functionality, catching removed resources, changed resource kinds, deleted environment variables, and altered container ports.

The adversarial red-team review and the verification report are both LLM-

based, which is perhaps the most significant validity threat in the pipeline. Two design choices partially mitigate this. First, the deterministic scanner ground truth and the violation checker are independent of any model and cannot be compromised by shared LLM failure modes; the fix-rate metric depends solely on these. Second, the verification report produces its verdict in a structured JSON schema with explicit reasoning tied to specific chart changes, making each classification auditable: a human reviewer can check any classification against the diff without having to trust the LLM summary. Even so, the LLM-based validation stages should be interpreted as secondary evidence that complements the deterministic measurements rather than as independent truth.

A further consideration is that the agent prompt was held constant throughout the evaluation. The same system and task instructions were issued for every run, with only the model and the per-chart findings varying. Holding the prompt fixed is what makes the cross-model comparison fair, since each model is assessed under identical instructions, but it also confounds the contribution of the prompt with that of the pipeline structure. The development-phase observation that targeted prompt revisions produced the largest gains, noted in Section 6.1, comes from uncontrolled iterative testing rather than a controlled comparison, and should be read as a development heuristic rather than a measured effect. Because models can respond differently to the same instructions, a different prompt could shift the fix rates and, in principle, reorder the models relative to the ranking in Section 4.2.3.2. The reported ordering is therefore valid for this prompt and should not be assumed to hold under prompt variations that were not tested.

Finally, the test corpus covers publicly available Helm charts spanning a range of application categories, sizes, and template complexities, and the model set is restricted to OpenAI API models. The deployment context is on-premise, and several not-applicable classifications are specific to that context. In a cloud-native deployment, a portion of the currently excluded findings would become fixable and the reported fix rates would shift accordingly.

Chapter 5

Discussion

This chapter interprets the empirical results presented in Chapter 4, explains what they imply about the feasibility of agentic LLM remediation for Helm charts, where the boundaries of that feasibility lie, and which design decisions in the pipeline either held up or proved questionable under measurement.

5.1 Restricted Agency Over Full Tool Access

Before interpreting the empirical results, it is worth examining the design choice that conditions them. The pipeline restricts the agent to a file-writing primitive while front-loading targeted files, scanner-derived fix hints, and a structured task list into the initial prompt. Section 4.1 evaluated this choice against a fully agentic variant in which the agent retained access to all six tools while also being given the same initial context.

During development and testing, we observed that unrestricted tool access does not improve outcomes. Across the paired chart corpus, the constrained configuration matched or outperformed the fully agentic variant on the majority of charts, under both `GPT-5` and `GPT-5.4`. The development-time observation on smaller-tier models was even sharper, there the fully agentic configuration collapsed more frequently while the constrained configuration remained competitive. However, the unrestricted tool access scales with model capability, frontier models absorb most of the overhead and smaller models do not. The mechanism behind this is straightforward, the additional render and scan tools offer minimal gain in this pipeline, since the agent already receives the full task list, fix hints, and file contents upfront, and the pipeline performs missing-fix detection and violation checking deterministically after each agent turn.

These tool calls therefore consume context and tokens without contributing new signal, and the additional content can in fact degrade accuracy by enlarging the volume of information the model must reason over.

5.2 Pipeline Capabilities and Boundaries

The headline finding from Section 4.2 is that the pipeline resolves the majority of scanner findings on the majority of charts: a median instance-level fix rate of 95.7% and a mean of 84.5% across 119 successful runs. Six of the twenty charts reach 100% across every model, and a further six charts fall in the range of 74% to 100%.

The more important result, however, is the structure of the failures rather than the volume of the successes. Section 4.2.3.1 shows that fix rates are bimodally distributed, where runs cluster either near 100% or near 0%, with comparatively little intermediate territory. A mean of 87.57% for **GPT-5.2** does not describe a typical run that resolves 87% of findings, it describes a population of runs that mostly succeed completely and occasionally fail completely. Operationally this matters because the pipeline's reliability cannot be summarised by an average.

A user deploying the pipeline does not get an 87% chance of an 87%-fixed chart. They get a high probability of a near fully-fixed chart and a smaller probability of a chart that did not move at all or, worse, was actively corrupted. This bimodality is the most important boundary condition for production use.

The corruption cases documented in Section 4.2.4 give the failure mode a concrete shape. Three patterns recur. First, structural YAML corruption, where the agent writes values files or templates that no longer parse, the agent reports numerous successful fixes against a chart that `helm template` cannot render. Second, schema-level rejection, where the chart renders but violates a project-specific values schema. Third, destructive rewriting, where the agent regenerates entire template files from scratch rather than applying targeted edits. The first two are loud failures where the chart breaks, the scanner cannot run, and the 0% fix rate makes the failure immediately visible. The third is the quiet failure, a destructively rewritten chart can render, pass all scanner checks, and would only be caught by the functional equivalence diff or by a human reviewer examining the resulting deployment behaviour. This is precisely the failure pattern that Section 4.2.4 identified as a risk of LLM-based code repair, and it confirms in practice that scanner-only verification is insufficient.

These failure patterns echo findings from the broader automated program

repair literature. Bouzenia et al. [86] observe that an agentic system with selective tool autonomy occasionally produces overly complex fixes for simple bugs and edits only a subset of required files in multi-file repairs. Both failure modes appear in this study. The partial fixes observed across charts with multiple interacting templates reflect the same incomplete cross-file reasoning, while the destructive rewriting cases represent the inverse problem, where the agent overcorrects by regenerating entire files from scratch rather than making targeted edits, trading structural correctness for a simpler output it can reason about completely.

5.2.1 Addressing RQ1

The empirical results demonstrate that an agentic LLM pipeline can meaningfully improve the security posture of Helm charts. A median instance-level fix rate of 95.7% across 119 runs, with six charts reaching 100% across all models, establishes that autonomous remediation is feasible for the majority of standard chart architectures.

The reliability boundary lies at chart structural complexity. Fix rates are bimodally distributed: the pipeline either resolves nearly all findings or fails completely. Failure concentrates in three patterns, structural YAML corruption, schema-level rejection, and destructive rewriting, rather than being spread as degraded performance across the corpus.

The safety boundary is that scanner-only verification is insufficient. Destructively rewritten charts can pass all security checks while being functionally broken, which is why the functional equivalence gate and the human-readable verification report are necessary complements to the fix-rate metric. This reliability profile is also conditional on pipeline configuration: full tool access does not improve outcomes, and a controlled pipeline that combines deterministic stages with constrained agent action is preferable to an unrestricted agent.

Autonomous remediation is therefore most reliably deployed as a decision-support tool that automates the tractable majority of findings while flagging the remainder for human review, rather than as a fully autonomous deployment gate.

5.3 Cross–Model Comparison and the Cost–Performance Factor

The cross-model results in Section 4.2.3 produced one finding that cuts against the intuitive ordering of the OpenAI tier list. At default reasoning effort, the most expensive model in the family, **GPT-5.4**, was not the best performer. The model that delivered the highest mean fix rate *(94.28%)*, the lowest coefficient of variation *(9.80%)*, the only worst-case run at 60%, and a cost per fix in the middle of the pack was **GPT-5**. Based on accuracy, consistency, and cost, **GPT-5** was the most reliable choice in run 1. The mini-tier models achieved lower cost per fix but at the price of a much wider variance, they succeed on simple charts and collapse to 0% on complex ones, which makes them unsuitable for an unattended deployment where the cost of a structurally corrupted chart is far higher than the savings from cheaper tokens.

The picture changes once the run-2 retries from Section 4.3 are folded in. GPT-5.4 closes its gap to GPT-5 and reaches effectively the same mean instance fix rate (94.90% vs. 94.28%), with GPT-5.2 and GPT-5.1 also gaining ground. This re-ordering is within measurement noise rather than a measured lead, and the two models' fix-rate intervals in Table 4.5 overlap. The two larger reasoning models are therefore best read as statistically indistinguishable once reasoning effort is escalated, rather than as GPT-5.4 taking a measurable lead. This re-ordering does not contradict the original finding so much as refine it. A meaningful share of GPT-5.4's run-1 disadvantage was an effort-budget artefact rather than a capability limit, the model could remediate the failed chart but not at default reasoning effort. GPT-5 had no zero-rate cells to retry and therefore no headroom in this comparison, so the ordering at `reasoning_effort=medium` is partial evidence.

Crucially, this asymmetry runs against GPT-5 rather than in its favour. Every other model was given a second attempt on its hardest cells and GPT-5 was not, so folding in run 2 can understate GPT-5's relative standing. The claim that GPT-5 leads at default effort therefore rests on the run 1 comparison, which is symmetric in that every model was executed once per chart under identical instructions and the same default reasoning effort, rather than on the mixed run-1/run-2 picture.

A symmetric experiment, retrying every chart with every model at medium effort, would be needed to claim that GPT-5.4 dominates at that setting. What the present data does support is the weaker but operationally useful claim that the gap between the larger reasoning models is narrower than the run-

1 numbers suggested. A deployment that standardises on GPT-5 at default effort and GPT-5.4 with reasoning escalation on failure is consistent with the evidence in either direction.

The interpretation of why GPT-5 led at default effort is unchanged. GPT-5 is plausibly better calibrated to multi-file YAML reasoning at the depth this task requires, and GPT-5.4 trades some of that default-effort calibration for capabilities that are not exercised by this remediation task. A fair reading is that GPT-5 is the strongest model within the pipeline as configured at default settings, which is a meaningful operational claim but a narrower scientific one.

The token-consumption pattern in Section 4.2.3.4 reinforces this reading. GPT-5 and GPT-5-mini consume nearly identical token volumes but differ by 18 percentage points in mean fix rate. The mini variant is using the same context less effectively and remediation tasks that require cross-file reasoning over multi-template Helm charts can be reasoning-bound. Run 2 reinforces the same point from a different angle: at higher reasoning effort the mini-tier models still trail the larger ones because their failures concentrate on the structurally hardest charts, where additional reasoning cannot substitute for the architectural understanding that scale provides. The practical consequence is that the cheapest viable strategy is not the cheapest model. A pipeline operator paying for GPT-5-mini is paying roughly five times less per run for a model that fails completely on the harder charts, and the cost of remediating a corrupted chart by hand far exceeds the token savings.

5.3.1 Addressing RQ2

Across the six models evaluated within the same pipeline, GPT-5 is the strongest choice at default reasoning effort, it achieves the highest mean fix rate (94.28%), the lowest coefficient of variation (9.80%), the only worst-case run that remains at 60%, and a mid-range cost per fix. The mini-tier models achieve lower absolute cost but at the price of high variance and complete failure on structurally complex charts, making them unsuitable for unattended deployment. This ordering holds for the single prompt used throughout the evaluation. Because the prompt was held constant to keep the cross-model comparison fair, its contribution cannot be separated from that of the pipeline structure, and a different prompt could in principle shift the ordering (Section 4.5).

GPT-5.4 closes the gap with GPT-5 when reasoning effort is increased, suggesting that the run-1 ordering is partly an effort-budget artefact rather than a capability ceiling. Because the retry experiment only ran the 0% cells

and GPT-5 had none, every model except GPT-5 received a second attempt, so the post-retry ordering can understate GPT-5. The dominance claim is therefore grounded in the symmetric run-1 comparison at default effort, not in the post-retry numbers. The margin over GPT-5.4 in particular is modest and sensitive to the single-run-per-cell design, whereas the gap to the mini-tier models is large and stable. On self reporting gap rate, the dominant signal is underclaiming driven by an ID-versus-instance counting mismatch rather than fabrication. Genuine overclaiming concentrates in chart-corruption cases across all models. Token consumption is similar across models for the same chart. Cost differences between tiers are determined by output pricing and write-round count rather than context volume, and that token budget alone is not a proxy for remediation effectiveness.

5.4 The Self-Report Gap

The empirical pattern observed here partially contradicts that expectation. Across all 119 runs, 61.5% underclaim, only 18.3% overclaim, and the agent-claimed fix rate is on average 9–22 percentage points *below* the scanner-measured ground truth.

The apparent self-report gap in this study is a unit-of-measurement mismatch rather than a hallucination phenomenon. Once the unit is corrected, the residual underclaiming is small, and the residual overclaiming concentrates in a recognisable subset of runs. The worst overclaim cases listed in Table 4.1 fall into two categories: runs where the chart was structurally corrupted so scanner-ground-truth is 0 but the agent reports its intended fixes, and runs that genuinely overstated success on a renderable chart.

The methodological consequence is that deterministic re-scanning is not optional. The pipeline's self-report is unreliable in exactly the failure modes where reliability matters most, the cases where the chart no longer renders. Without an external verification, a pipeline operator would be told that a corrupted chart was successfully remediated. With it, the corruption is visible as a 0% fix rate. This is the mechanism by which the multi-layer verification architecture earns its complexity, and Section 4.2.6 confirms that the structural validation gates catch the corruption that the self-report misses.

5.5 Behavioural Change and the Functional Equivalence

The functional equivalence gate has the lowest pass rate of any validation stage at 54.6%, and the temptation is to read this as the pipeline's weakest performance. However, it may be the most informative result in the chapter. Security hardening, by construction, changes runtime behaviour.

> A pipeline that produced 100% functional equivalence would be a pipeline that did not actually harden anything. The 54.6% number is therefore not a measure of failure — it is a measure of how often the pipeline's hardening was additive enough to leave runtime behaviour untouched.

The Category A/B/C classification in Section 4.2.7 makes this concrete. Category B, behavioural modifications, dominates the population of changes across every model, because the underlying Trivy and Checkov findings concern existing fields that must be tightened. The majority of Category B changes are a runtime-affecting edit that a deployment reviewer should examine, and the verification report is the mechanism by which those edits are surfaced for review. Category C being the values overrides, the Helm anti-pattern of hardcoding into templates is rarer but model-dependent, with GPT-5-mini and GPT-5.4 contributing disproportionately.

The operational implication of these results is that the fix-rate metric, taken alone, is not a sufficient acceptance criterion. A chart can achieve a 100% fix rate through Category B changes that materially alter runtime behaviour, or through Category C overrides that silently strip the chart's configurability. In both cases the scanner is satisfied while the chart that reaches the operator is no longer equivalent to the one that entered the pipeline. The functional equivalence gate detects the first class of change but cannot, on its own, indicate which behavioural modifications are intended hardening and which are unintended regressions.

5.6 Threats to validity

Overfitting to the corpus. The pipeline was developed and evaluated on the same set of 20 Helm charts. The prompt, the violation-checker rules, the retry thresholds, and the fix hints were all refined by repeated testing against this corpus, and no separate validation set was held back. However, the testbed

consist of a wide range of charts with different structural complexities, ranging from single workload charts to multi container workloads and subcharts, as well as the refinements being structural and aimed at general issues faced with LLM augemented remediation.

Shared reviewer model. The adversarial review and security-auditor stages use the same model that produced the edits. A model may share the blind spots that caused a misconfiguration to be missed or mis-fixed in the first place, which limits the independence of these verification stages. An independent reviewer model would give a stronger check (Section **??**).

Chapter 6

Conclusions and Future work

6.1 Conclusions

This thesis set out to determine whether an agentic LLM pipeline could autonomously remediate static-analysis findings in Kubernetes Helm charts to a standard usable in a regulated environment, and to characterise the conditions under which it can and cannot. The contribution of this work is empirical evidence that, across the twenty public Helm charts, an LLM-driven pipeline can resolve the large majority of scanner-reported misconfigurations using OpenAI models. Whether these fix rates hold on enterprise umbrella charts, other deployment contexts, or other model families remains open. This finding cuts against the prevailing trajectory in agentic systems research, which tends toward expanding agent autonomy and tool access. The thesis demonstrates that for narrow, high-stakes tasks like security remediation, the inverse design, minimising autonomy for the agent and enforcing a semi-autonomous agentic pipeline structure, produces measurably better outcomes at lower cost. Within the six OpenAI models evaluated, remediation quality did not scale monotonically with model price. GPT-5 leads on accuracy and consistency at default reasoning effort while remaining mid-range on cost per fix, and the gap to GPT-5.4 narrows only once reasoning effort is escalated on the runs it first failed.

A key contribution for Finansinspektionen is that this thesis provides a concrete answer to the operational question that motivated it. An LLM pipeline can substantially reduce the manual remediation load on a security team, provided that the pipeline is treated as a generator working within guardrails rather than as an autonomous agent. The boundary between what the pipeline can close and what it correctly defers to human review is itself

a useful artefact that converts a uniform queue of findings into a triaged one, which is where most of the operational value lies.

If undertaken again, the project would prioritise prompt engineering earlier. The largest gains observed during development appeared to come from targeted prompt revisions rather than from architectural changes, such as small adjustments to how findings were framed, how the agent was instructed to reason, and how the task list was structured each produced large improvements. Prompt engineering is an important design lever and could have shortened the path to the final architecture considerably. Second, the majority of development and iteration should have been conducted on stronger models rather than smaller ones. Substantial effort went into accommodating failure modes specific to the smaller-tier models, only for many of those failures to disappear once frontier models were used. Anchoring development on a capable model from the outset would have prevented the pipeline from absorbing complexity it did not actually need.

6.2 Limitations

The single-run-per-combination constraint is consequential. Each chart–model pair was executed once in the primary evaluation, with the exception of the 0%-rate pairs from the first run, which were re-executed at higher reasoning effort in the retry experiment. A 0% fix rate on a particular pairing might reflect a consistent inability of that model to handle that chart, or it might reflect a single trajectory that would have resolved on a second attempt. The cost ceiling approached the budget for the full matrix testing available within the thesis timeframe. The consequence is that the model-level CV reported in Section 4.2.3.2 is partly a measurement of within-model variance and partly a measurement of cross-chart sensitivity. A second pass over the matrix would be interesting to examine.

The validity of the LLM-based verification stages is the second meaningful threat. The verification report and the adversarial red-team review are themselves model outputs, and a shared failure mode in the underlying model could in principle compromise both the remediation and its validation. The deterministic ground-truth scan and the deterministic violation checker are the load-bearing protections against this, and the fix-rate metric depends only on them. The deployment readiness verdict and the behavioural classification do not. A reader interpreting those classifications should treat them as auditable secondary evidence rather than as independent ground truth. The structured JSON schema of the verification report is the mechanism by which they remain

auditable, since each classification is tied to a chart change a human reviewer can inspect, but this is a mitigation rather than a guarantee.

A narrower limitation is the corpus. Twenty Helm charts span a useful range of application categories and template architectures, but they do not span enterprise umbrella charts. The reported fix rates are valid for the corpus and the deployment context evaluated, they do not generalise to umbrella deployments without further evaluation.

6.3 Future Work

Several directions emerge naturally from the findings of this study.

In-loop render validation. Adding a render check at the final agent iteration would address the structural-corruption failure mode that produces the 0% runs. The agent currently has no opportunity to react because rendering is verified after the loop terminates. Returning the render error to the agent as a feedback message on failure would convert a non-trivial fraction of the current 0% runs into successful or partially successful outcomes without changing any other pipeline component.

Structured edit actions. Rather than allowing the agent to write arbitrary file content, the pipeline could expose a set of predefined, semantically typed edit operations such as `set-field-value(field, value)` and `delete-field(field)` instead of a raw `write_file` tool. Each operation would enforce syntactic correctness by construction, eliminating the structural-corruption failure mode. This would likely improve consistency on the weaker models and may make locally deployed open-source models viable for the task.

Tiered model escalation. An optimised deployment strategy could start each chart with a mini-tier model and escalate to a full-size model only on failure, and further escalate to higher reasoning effort if the full-size model also fails. Given the bimodal distribution of outcomes, the majority of charts would resolve at the cheapest tier, and escalation cost would only be incurred on the minority of structurally complex charts that actually require it.

Locally deployed open-source models. A natural extension is evaluating whether the same pipeline can be driven by locally hosted open-source models. This would eliminate the data-egress concern that constrains LLM use in regulated environments, where sending production chart contents to an external API is forbidden.

Taken together, these results position agentic LLM remediation as a practically deployable component of a security review workflow, rather than as

a speculative capability. The pipeline does not eliminate the human reviewer, and it does not need to. It converts a uniform queue of scanner findings into a triaged one, fixes the tractable majority autonomously, and surfaces the remainder, together with the runtime-affecting changes it does make, for explicit review. The evidence presented in this thesis supports the use of constrained agentic remediation as a meaningful reduction of manual security work in regulated infrastructure environments.

References

[1] M. Kerrisk. namespaces(7) — Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html

[2] Kubernetes. Cluster networking. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/

[3] ——. About cgroup v2. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cgroups/

[4] Distr. Oci container and artifact registry. [Online]. Available: https://distr.sh/glossary/oci-container-artifact-registry/

[5] Google Cloud. Troubleshooting image pulls. [Online]. Available: https://docs.cloud.google.com/kubernetes-engine/docs/troubleshooting/image-pulls

[6] Docker Inc. docker manifest. [Online]. Available: https://docs.docker.com/reference/cli/docker/manifest/

[7] Google Cloud. Containers vs. vms. [Online]. Available: https://cloud.google.com/discover/containers-vs-vms

[8] Scale Computing. Container virtualization explained. [Online]. Available: https://www.scalecomputing.com/resources/container-virtualization-explained

[9] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-190, 2017. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/190/final

[10] Sysdig. CVE-2022-0185: Container escape. [Online]. Available: https://www.sysdig.com/blog/cve-2022-0185-container-escape

[11] Kubernetes. Cluster architecture. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[12] etcd Authors. etcd FAQ. [Online]. Available: https://etcd.io/docs/v3.6/faq/

[13] LearnKube. etcd in kubernetes. [Online]. Available: https://learnkube.com/etcd-kubernetes

[14] N. Nuwanwe. Understanding the kubernetes controller manager. [Online]. Available: https://medium.com/@nuwanwe/understanding-the-kubernetes-controller-manager-b34db5b92dd4

[15] Kubernetes. kubelet. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[16] ——. kube-proxy. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/

[17] ——. Pods. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[18] ——. Deployments. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[19] ——. Service. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[20] ——. Service. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[21] ——. Configmaps. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/

[22] ——. Secrets. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/

[23] Helm. Helm – the kubernetes package manager. [Online]. Available: https://helm.sh/

[24] ——. Using helm. [Online]. Available: https://helm.sh/docs/intro/using_helm/

[25] Red Hat. What is helm? [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-helm

[26] Helm. Charts. [Online]. Available: https://helm.sh/docs/topics/charts/

[27] ——. Template functions and pipelines. [Online]. Available: https://helm.sh/docs/chart_template_guide/functions_and_pipelines/

[28] Go. Package template. [Online]. Available: https://pkg.go.dev/text/template

[29] Masterminds. Sprig function documentation. [Online]. Available: https://masterminds.github.io/sprig/

[30] Helm Authors. Control structures. [Online]. Available: https://helm.sh/docs/chart_template_guide/control_structures/

[31] ——. Values files. [Online]. Available: https://helm.sh/docs/chart_template_guide/values_files

[32] Startup Defense. Helm chart injection. [Online]. Available: https://www.startupdefense.io/cyberattacks/helm-chart-injection

[33] Red Canary. Kubernetes security. [Online]. Available: https://redcanary.com/blog/security-operations/kubernetes-security/

[34] Cloud4C. 4cs of cloud-native security. [Online]. Available: https://www.cloud4c.com/blogs/4cs-of-cloud-native-security-blog

[35] MITRE. ATT&CK matrix for containers. [Online]. Available: https://attack.mitre.org/matrices/enterprise/containers/

[36] Microsoft Security. Attack matrix for kubernetes. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2020/04/02/attack-matrix-kubernetes/

[37] ——. Secure containerized environments with updated threat matrix for kubernetes. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2021/03/23/secure-containerized-environments-with-updated-threat-matrix-for-kubernetes/

[38] IoT Security Institute. Kubernetes exploitation: Understanding attack vectors and hardening your clusters. [Online]. Available: https://iotsecurityinstitute.com/iotsec/iot-security-institute-cyber-security-articles/239-kubernetes-exploitation-understanding-attack-vectors-and-hardening-your-clusters

[39] OWASP. Kubernetes security cheat sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Kubernetes_Security_Cheat_Sheet.html

[40] Kubernetes. Security checklist. [Online]. Available: https://kubernetes.io/docs/concepts/security/security-checklist/

[41] ——. RBAC good practices. [Online]. Available: https://kubernetes.io/docs/concepts/security/rbac-good-practices/

[42] Sysdig. Sysdig usage and security report: More than 75 percent of running containers have high-risk vulnerabilities. [Online]. Available: https://www.sysdig.com/press-releases/sysdig-usage-and-security-report-finds-more-than-75-of-running-containers-have-high-risk-vulnerabilities

[43] Dynatrace. Kubernetes misconfiguration: Attack paths and mitigation. [Online]. Available: https://www.dynatrace.com/news/blog/kubernetes-misconfiguration-attack-paths-and-mitigation/

[44] CrowdStrike. CVE-2022-0185: Kubernetes container escape using linux kernel exploit. [Online]. Available: https://www.crowdstrike.com/en-us/blog/cve-2022-0185-kubernetes-container-escape-using-linux-kernel-exploit/

[45] Aikido Security. Kubernetes security vulnerabilities. [Online]. Available: https://www.aikido.dev/blog/kubernetes-security-vulnerabilities

[46] Microsoft. Threat matrix for kubernetes. [Online]. Available: https://microsoft.github.io/Threat-Matrix-for-Kubernetes/

[47] The Kubernetes Authors. Configure a security context for a pod or container. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[48] Kubernetes. Configure a security context for a pod or container. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[49] OneUptime. Read-only root filesystem: Immutable containers. [Online]. Available: https://oneuptime.com/blog/post/2026-02-09-readonly-root-filesystem-immutable-containers/view

[50] Kubernetes. Assign pod-level resources. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/assign-pod-level-resources/

[51] The Kubernetes Authors. Limit ranges. [Online]. Available: https://kubernetes.io/docs/concepts/policy/limit-range/

[52] Kubernetes. Resource quotas. [Online]. Available: https://kubernetes.io/docs/concepts/policy/resource-quotas/

[53] ——. Using RBAC authorization. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[54] StrongDM. Kubernetes RBAC: Role-based access control. [Online]. Available: https://www.strongdm.com/blog/kubernetes-rbac-role-based-access-control

[55] Red Hat. What is kubernetes role-based access control (RBAC)? [Online]. Available: https://www.redhat.com/en/topics/containers/what-kubernetes-role-based-access-control-rbac

[56] Trilio. Kubernetes RBAC best practices. [Online]. Available: https://trilio.io/kubernetes-best-practices/kubernetes-rbac/

[57] Apono. Kubernetes RBAC: How to enable and manage access. [Online]. Available: https://www.apono.io/blog/kubernetes-rbac-how-to-enable-and-manage-access/

[58] SentinelOne. Climbing the ladder: Kubernetes privilege escalation, part 1. [Online]. Available: https://www.sentinelone.com/blog/climbing-the-ladder-kubernetes-privilege-escalation-part-1/

[59] The Kubernetes Authors. Network policies. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[60] ——. Network plugins. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

[61] NetworkPolicy. Networkpolicy tutorial. [Online]. Available: https://github.com/networkpolicy/tutorial

[62] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, “Security misconfigurations in open source Kubernetes manifests: An empirical

study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 99:1–99:36, 2023. doi: 10.1145/3579639. [Online]. Available: https://par.nsf.gov/servlets/purl/10415562

[63] Red Hat. State of kubernetes security report 2024. [Online]. Available: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview

[64] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices," in *Proceedings of the 2020 IEEE Secure Development Conference (SecDev)*, 2020, pp. 58–64. [Online]. Available: https://arxiv.org/pdf/2006.15275

[65] TechTarget. Static analysis (static code analysis). [Online]. Available: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis

[66] S. Bhattacharjee. (2025) Static vs. dynamic code analysis: A comprehensive guide. vFunction. [Online]. Available: https://vfunction.com/blog/static-vs-dynamic-code-analysis/

[67] Qt Group. Static vs dynamic code analysis. [Online]. Available: https://www.qt.io/quality-assurance/blog/static-vs-dynamic-code-analysis

[68] A. G. Bardas, "Static code analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010. [Online]. Available: https://scispace.com/pdf/static-code-analysis-539k986pv3.pdf

[69] Bridgecrew. Checkov: Prevent cloud misconfigurations during build-time. [Online]. Available: https://github.com/bridgecrewio/checkov

[70] Aqua Security. Trivy documentation. [Online]. Available: https://trivy.dev/

[71] Artifact Hub. Artifact hub. [Online]. Available: https://artifacthub.io/packages/search?kind=0&category=7

[72] OWASP. OWASP application security verification standard (ASVS). [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/

[73] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Transactions on Software Engineering and Methodology*, 2026. doi: 10.1145/3747588. [Online]. Available: https://dl.acm.org/doi/10.1145/3747588

[74] The Kubernetes Authors. Structural schemas for custom resources. [Online]. Available: https://kubernetes.io/blog/2019/06/20/crd-structural-schema/

[75] Prompting Guide. Function calling. [Online]. Available: https://www.promptingguide.ai/agents/function-calling

[76] B. Yang, Z. Cai, F. Liu, B. Le, L. Zhang, T. F. Bissyandé, Y. Liu, and H. Tian, "A survey of LLM-based software repair: Taxonomies, design paradigms, and applications," 2025. [Online]. Available: https://arxiv.org/abs/2506.23749

[77] OpenAI. Prompt engineering guide. [Online]. Available: https://developers.openai.com/api/docs/guides/prompt-guidance

[78] ——, "GPT-4 system card," Tech. Rep., 2023. [Online]. Available: https://cdn.openai.com/papers/gpt-4-system-card.pdf

[79] ——, "OpenAI GPT-5 system card," 2025. [Online]. Available: https://arxiv.org/abs/2601.03267

[80] V. Dongre, R. A. Rossi, V. D. Lai, D. S. Yoon, D. Hakkani-Tür, and T. Bui, "Drift no more? context equilibria in multi-turn LLM interactions," 2025. [Online]. Available: https://arxiv.org/html/2510.07777v1

[81] E. L. HARTZ, "Development of an AI-driven DevOps engineer: Automating workflows with an LLM based multi-agent system," Master's thesis, KTH Royal Institute of Technology, 2025. [Online]. Available: https://kth.diva-portal.org/smash/get/diva2:1987584/FULLTEXT01.pdf

[82] J. N. Müller, "Static source code analysis of container manifests," Master's thesis, Technical University of Munich, 2020. [Online]. Available: https://teamscale.com/hubfs/26978363/Publications/2020-static-source-code-analysis-of-container-manifests.pdf

[83] F. Minna, A. Blaise, K. Tuma, and F. Massacci, "Automated analysis of security policy violations in Helm charts," *IEEE Transactions on Dependable and Secure Computing*, 2025. doi: 10.1109/TDSC.2025.3628213. [Online]. Available: https://ieeexplore.ieee.org/document/11223885

[84] F. Minna, A. Blaise, F. Massacci, and K. Tuma, "Automated security repair for Helm charts," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2024. doi: 10.1145/3639478.3643534 pp. 412–413. [Online]. Available: https://dl-acm-org.focus.lib.kth.se/doi/pdf/10.1145/3639478.3643534

[85] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024. doi: 10.1145/3650212.3680323 pp. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[86] I. Bouzenia, P. Devanbu, and M. Pradel, "RepairAgent: An autonomous, LLM-based agent for program repair," 2024. [Online]. Available: https://arxiv.org/abs/2403.17134

[87] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "GenKubeSec: LLM-based Kubernetes misconfiguration detection, localization, reasoning, and remediation," 2024. [Online]. Available: https://arxiv.org/abs/2405.19954

[88] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004. doi: 10.2307/25148625. [Online]. Available: https://www.researchgate.net/publication/201168946_Design_Science_in_Information_Systems_Research

[89] OpenAI. GPT-5 mini model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5-mini

[90] ——. GPT-5 model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5

[91] ——. GPT-5.1 model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5.1

[92] ——. GPT-5.2 model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5.2

[93] ——. GPT-5.4 model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5.4

[94] ——. GPT-5.4 mini model. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5.4-mini

[95] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, 2023.

TRITA-EECS-EX-2026:399
Stockholm, Sverige 2026

www.kth.se