Towards Practical Realization of Intent-Based Security Paradigm: Design and Implementation of an LLM-Powered Host-Level Security System

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Nokia Bell Labs
July 2026
Elson Arampillykudy Eldo

Supervisors:
Tahir Mohammad (PhD)
University of Turku
Antti Hakkala (D.Sc. Tech)
University of Turku
Siddharth Prakash Rao (D.Sc. Tech)
Nokia Bell Labs

UNIVERSITY OF TURKU
Department of Computing

Elson Arampillykudy Eldo: Towards Practical Realization of Intent-Based Security Paradigm: Design and Implementation of an LLM-Powered Host-Level Security System

Master of Science (Tech) Thesis, 94 p., 26 app. p.
Nokia Bell Labs
July 2026

As digital technologies continue to grow, computer applications are becoming an essential part of everyday life. These applications need a host machine to run, whether it is a physical machine or a virtual machine. In existing host machines, security settings are typically configured manually by users, using low-level configurations. This process can be time-consuming and error-prone, and often requires specialized technical knowledge. This can lead to security gaps and make security management difficult. One way to simplify this kind of manual configuration is an intent-based approach, where users describe what they want the system to achieve instead of specifying the low-level implementation details. This approach has already been applied in networking, where users state their network goals and the system works out the configurations needed to meet them. However, the concept is underexplored in the domain of host-level security. This thesis addresses that gap by extending the intent-based approach to host-level security.

The study used literature review, case study, requirements engineering, and user journey mapping. Existing intent-based systems were examined to understand their capabilities and limitations. Based on the findings, this thesis proposes an architecture for a system that receives security intent expressed by a user in natural language and fulfills it on a host machine. The proposed architecture interprets the intent using large language models, translates it into executable actions, and executes them on the host machine. The thesis also introduces a unified structure to represent security intent, the steps required to fulfill that intent, and the information needed for intent-lifecycle management in a machine-friendly form.

The proposed architecture was validated through both scenario-based and empirical evaluations performed with a proof-of-concept prototype. The validation results suggest that an intent-based approach is a promising solution for simplifying host-level security management. By allowing the user to define security intent in natural language, the proposed architecture makes host-level security management easy for a user without in-depth technical knowledge. This thesis lays the groundwork for further research into more user-friendly ways of managing host-level security.

Keywords: intent-based security, intent-based systems, host-level security, Intent Representation Model, large language models

Contents

Appendices

Usage of AI

AI has been used to suggest alternative wordings, correct spelling, and expressions. NotebookLM has been helpful in summarizing research papers. LLMs such as ChatGPT and Gemini have been utilized in brainstorming design strategies and refining certain technical aspects. Cursor has been used to support the development of the proof-of-concept prototype. However, all the ideas, analysis, and conclusions remain my own original work, and I have critically reviewed and edited all AI-generated suggestions.

List of Figures

List of Tables

List of acronyms

ACL Access-control list

AI Artificial Intelligence

CFG Context-Free Grammar

CTI Cyber Threat Intelligence

GUI Graphical User Interface

HP Hewlett-Packard

HSPL High-level Security Policy Language

I2NSF Interface to Network Security Functions

IBCS Intent-Based Cloud Service

IBN Intent-Based Networking

IRM Intent Representation Model

KPI Key Performance Indicator

LAI Language for ACL Intents

LLM Large Language Model

LSM Linux Security Module

LSTM Long Short-Term Memory

LTL Linear Temporal Logic

MSPL Medium-level Security Policy Language

MTTR Mean Time to Respond

NBI North-Bound Interface

NeMo Network Modeling

NFV Network Functions Virtualization

NILE Network Intent LanguagE

NLP Natural Language Processing

NSF Network Security Function

PGA Policy Graph Abstraction

QoS Quality of Service

RAG Retrieval-Augmented Generation

RDF Resource Description Framework

SAT Satisfiability

SDN Software-Defined Networking

SELinux Security-Enhanced Linux

SMT Satisfiability Modulo Theories

TEE Trusted Execution Environment

TPM Trusted Platform Module

Acknowledgement

My supervisor, Sid from Nokia Bell Labs, has shaped this thesis more than anyone else. Our weekly brainstorming sessions and his constant feedback on my drafts kept me moving forward, especially on the days I felt stuck. He taught me what good research actually looks like, and that standard will always stay with me. Beyond research, Sid was also a fantastic friend. Thank you Sid, for every bit of advice you gave me, both professional and personal. I will always cherish the time we spent working together.

I am also thankful to my supervisors at the University of Turku, Tahir Mohammad and Antti Hakkala, for their guidance and constructive feedback on my thesis drafts throughout this process.

At Nokia Bell Labs, I want to thank my manager, Liliann Miche, and all my colleagues, for making my short stint there such a memorable one. Those days would have been boring without all of you around.

Much of my time as a master's student was spent wandering the campuses of the University of Rennes and the University of Turku. Looking back, those two years were some of the richest of my life, full of good friends and countless people who left their mark on me. I am grateful for every one of those interactions.

Last but not least, I want to thank my friends back home, especially Anju, for being my support system remotely. Our weekly calls and rants kept me going and lifted me up on the days when things felt hardest.

1 Introduction

People naturally seek ways to simplify tasks and achieve their goals with minimal effort. This tendency is evident in many aspects of our daily lives and is equally relevant in the use and management of computer systems. As computing environments have evolved, they have become more complex, consisting of numerous interconnected components, services, and configuration parameters. While these advancements have expanded the capabilities of modern systems, they have also increased the burden on users, who are often required to understand extensive technical details to achieve relatively simple objectives. One key factor contributing to this burden is the gap between the language used by users and systems.

Human–machine interactions have always been limited by language more than anything else, as both speak different languages. In most of the earlier systems, the users had to speak the language of the machine, such as machine code and assembly language, not the other way around. Consequently, there has been a growing demand for approaches that enable users to focus on desired outcomes rather than the technical steps required to achieve them. Intent-based systems represent one such approach that simplifies the management of complex technical environments [1].

Over the past decade, intent-based systems have gained attention as a new paradigm for human-computer interaction and system administration. Reducing the semantic gap between human requirements and machine-level configurations is

the primary goal of intent-based systems. Instead of requiring users to define low-level implementation details, intent-based systems allow them to specify high-level objectives and desired outcomes. Traditionally, users have been required to translate their goals into numerous technical commands, rules, or configuration parameters. This process introduces the possibility of human error. By providing a higher level of abstraction, intent-based systems help bridge this gap and allow users to communicate their objectives in a manner closer to natural human conversation. For example, a user may express what they want the system to achieve rather than how each component should be configured. The system is then responsible for interpreting the user's intent, determining the necessary actions, and executing those actions automatically. By automating the translation of objectives into executable actions, these systems can reduce manual effort and increase consistency in system behavior.

Growing interest in intent-based systems has led to research as well as their implementation in various domains. In networking, an intent-based system allows administrators to specify desired network goals, while the system automatically configures and manages network resources [1]. In infrastructure management, intent-based systems handle tasks related to resource orchestration based on user requirements [2]. Similar concepts have also been explored in blockchain systems and conversational agents, where high-level objectives are translated into low-level actions [3], [4].

1.1 Problem Statement

Despite the progress made in intent-based systems, achieving host-level security using intents remains underexplored. Although there have been attempts to achieve security through intents, most of these research efforts have focused on networking environments [5]–[7]. In this context, administrators typically specify objectives related to traffic control, communication restrictions, and access between systems. Existing studies have shown that high-level security requirements can be translated

into effective network configurations without requiring administrators to manage every implementation detail.

However, security management is not limited to the network alone. Many security-critical activities, such as file access control and user privilege enforcement, occur at the host level. Therefore, host-level security is critical. We define a host as a desktop operating system that runs on a physical or virtual machine. Managing host security often requires administrators to work with platform-specific permission systems such as access control lists. They need to translate high-level security requirements into multiple low-level configuration settings. This translation process can be both time-consuming and error-prone [8], [9].

Despite the importance of host security, it has received relatively limited attention in the literature on intent-based security. Although researchers have explored how high-level intents can be used to manage network security, to the best of our knowledge, none of the previous studies have investigated how similar concepts can be applied to the host-level security. This gap highlights the need for further research into approaches that enforce security intents at the host level while reducing administrative effort. Large Language Models (LLMs) are good at natural language processing [10]. Therefore, it is useful to explore how LLMs can be utilized in intent-based systems for host-level security.

1.2 Research Objectives and Questions

This thesis explores how intent-based concepts can be extended to host-level security. Specifically, this thesis aims to understand how system users, including administrators, can provide security requirements using simple, high-level instructions, and how these instructions can be automatically converted into executable actions. By reducing the need for low-level configuration, this approach has the potential to simplify security administration.

To achieve this goal, the thesis addresses the following research questions:

- **RQ1: How is security currently achieved in intent-based systems?**
- **RQ2: How should an intent-based security system be designed and implemented at the host level?**
- **RQ3: How can LLMs serve as an interface for handling security requirements, thereby reducing the need for manual configuration expertise?**

Based on these research questions, the objectives of this thesis are:

1. **To examine existing intent-based systems and approaches from the perspectives of systems security.**
2. **To identify the requirements for an intent-based security system at the host level, followed by designing and implementing a proof-of-concept system sufficing those requirements.**
3. **To methodologically evaluate the proposed intent-based security system for functional correctness.**

1.3 Scope

We define the scope of this thesis work as follows:

- The thesis focuses on Linux-based desktop operating systems (e.g., Ubuntu) as the host environment where the security policies are executed. No other host operating systems are considered.
- We limit the intents to those related to file-system access-control policies. Even though the thesis focuses on the file-system, since everything in the

Linux operating system is a file [11], the system can potentially be extended to satisfy the requirements of other system internals such as hardware devices and inter-process communication in Linux-based operating systems.

- The thesis focuses on security intents expressed in natural language by human users. While we are aware of machine-to-machine intents [12], we exclude them to reduce the complexity of system design.

1.4 Structure

The remainder of the thesis is structured as follows. Chapter 2 provides the necessary background for the thesis, including the literature review. Chapter 3 presents the methodology used for this research. Chapter 4 describes the architecture of the proposed system and the implementation details of the proof-of-concept prototype. Chapter 5 provides the evaluation details of the proof-of-concept prototype. Chapter 6 discusses the limitations of our work and the directions for future work. Finally, Chapter 7 concludes the thesis by summarizing the key contributions.

2 Background

2.1 Host-Level Security

Host-level security refers to the security measures that operate on individual computing devices to protect them from potential threats. Early host-level protection mainly relied on basic operating system controls, such as file permissions and user accounts. Over time, host-level security has grown into a layered approach that also includes kernel-level protection, application restrictions, firmware checks, and hardware-backed trust mechanisms.

2.1.1 Operating System-level Mechanisms

Operating system-level mechanisms are one of the layers in host-level security. In Linux-based systems, access to files and directories is mainly controlled through permission bits. Each file normally has nine permission bits, which define what the file owner, the owning group, and other users are allowed to do. This model is simple and useful, but it becomes limited when several users, groups, or services need different types of access to the same file or directory.

Access-control lists (ACLs) extend the traditional Linux permission model when basic permission bits are insufficient [13]. ACLs allow administrators to assign specific permissions to additional users and groups without changing the main owner or group of a file. However, ACLs still work within the general idea that a valid user

or process is allowed to use the permissions assigned to it. If a trusted account or process becomes compromised, file permissions and ACLs alone will not prevent an attacker from misusing the access already granted to that account or process.

2.1.2 Kernel-level Mechanisms

Kernel-level mechanisms provide stronger control to address some of the weaknesses of basic file permissions and ACLs. The Linux Security Modules (LSM) framework provides hooks inside the kernel that security modules can use to check whether an operation should be allowed. Security-Enhanced Linux (SELinux) and AppArmor are two well-known examples of security modules that use this framework. SELinux is an implementation of a flexible mandatory access control architecture in the Linux operating system [14]. SELinux assigns labels to subjects, such as processes, and objects, such as files. When a process requests access to a file or another resource, SELinux checks the labels and the policy rules before allowing or denying the request [15]. AppArmor also provides mandatory access control, but it follows a different style. AppArmor confines programs by using profiles that describe what each program is allowed to access and perform. The per-application profiles define which operations are allowed based on file paths, and every access to system resources by the application must be explicitly permitted in the profile. The per-profile limitations are enforced by the kernel, and an application cannot simply change or bypass its own profile, limiting the damage when the application becomes compromised [16].

Network filtering is another important aspect of kernel-level security mechanisms. Network filtering at the kernel level protects the host by controlling network packets before they reach local services or leave the machine. In Linux-based systems, netfilter is a framework inside the kernel that provides hooks other modules can use to register callback functions at different locations of the Linux network stack [17].

The traditional user-space tool for configuring many of these rules has been iptables. With iptables, rules are arranged in chains, and each rule matches packets based on selected conditions before applying an action such as accepting, dropping, or forwarding the packet [18]. nftables is a successor to iptables and also uses netfilter. nftables provides in-kernel packet classification based on a virtual machine, and the rulesets are compiled into bytecode [19].

Some of the kernel mechanisms focus more on isolation and attack-surface reduction. Secure Computing Mode, commonly called seccomp, filters and restricts the system calls that a process can make, thereby reducing the number of kernel interfaces available to that process and limiting the effect of a compromised program. Namespaces isolate what a process can see by giving it a separate view of system resources, such as process IDs, network interfaces, mount points, and user IDs [20]. Control groups, or cgroups, allow the system to limit and manage resource use for groups of processes, which helps prevent one process or container from consuming too much CPU, memory, or other host resources [21]. All these mechanisms work on the assumption that the kernel is trusted, but they are not reliable when the kernel itself is compromised.

2.1.3 Hardware-level Mechanisms

Hardware-level mechanisms add a lower layer of trust below the kernel. They do not replace operating system and kernel controls, but they can protect sensitive information and support trust decisions even when parts of the software stack are not fully trusted. Two important examples are the Trusted Platform Module (TPM) and the Trusted Execution Environment (TEE). TPMs mainly support secure storage, cryptographic operations, and platform measurement, while TEEs provide protected environments for running sensitive code and processing sensitive data.

A TPM is a technology that provides hardware-based security functions. While

usually implemented as a discrete hardware chip (dTPM), it can also be implemented via firmware (fTPM) or as a virtualized TPM (vTPM). TPM works as a secure cryptographic processor that can support protected key storage, cryptographic operations, and platform integrity measurements [22]. Software can ask the TPM to perform cryptographic operations or read public measurement values, such as Platform Configuration Registers. However, the TPM architecture prevents the plaintext extraction of sensitive cryptographic material, such as private keys. A TPM helps establish trust in the boot process through Measured Boot by storing cryptographic hashes of boot components. A TPM is not designed to run arbitrary application logic. Instead, its main goals are to protect keys, support cryptographic operations, and record measurements.

A TEE addresses the need to protect sensitive computations while they are running. Sabt et al. [23] define TEE as "a tamper-resistant processing environment that runs on a separation kernel". A TEE is an isolated area of the CPU and memory where sensitive programs can run. Any code outside the isolated area cannot read or modify the data inside the isolated area [24]. Sensitive application code is normally executed inside the TEE. Therefore, even in a compromised host, the sensitive data is not easily accessible to the operating system.

2.1.4 Other Mechanisms

Application-level mechanisms add another layer of host-level protection by controlling what software is allowed to run and what it is allowed to access. Code signing is one such mechanism. In code signing, a software binary or file is signed using a digital signature from the software author or publisher [25]. Depending on the operating system and policy, this signature may be checked before installation, before execution, or during software updates. Application sandboxing is another security mechanism, which isolates an application from the rest of the system by restricting

the application's access to files, memory, and network resources. This containment prevents malicious code or vulnerabilities within the sandboxed application from compromising other parts of the environment [26]. Code signing improves trust in software distribution, but it does not guarantee that signed software is safe, because a trusted signer can still make mistakes or have its signing key compromised. Similarly, the strength of a sandbox depends on the rules it applies and on whether the sandbox itself contains weaknesses.

Firmware-level mechanisms protect the host before the operating system fully starts. Secure Boot is one such mechanism that ensures that a device boots using only trusted software. When the device starts, Secure Boot checks the signature of each important boot component before allowing it to run. If the signature check fails, Secure Boot may block the boot process or start a recovery process instead, making it difficult to exploit the host by hijacking the boot process. BIOS write protection is another security mechanism that prevents writes to security-sensitive regions, such as the BIOS bootblock and recovery regions [27]. This reduces the risk of attackers modifying the firmware.

Despite these advances, host-level security still has limitations. Basic file permissions and ACLs can fail when a trusted process itself is compromised. Kernel-level controls, such as SELinux, AppArmor, seccomp, namespaces, and cgroups, depend on correct policies, proper configuration, and a trustworthy kernel. TPMs can protect keys and support Measured Boot, but they do not monitor every action an application performs after the system starts. TEEs reduce the amount of trust that must be placed in the host operating system, but they can still face side-channel attacks. Furthermore, most of these mechanisms require the host-system administrator to have deep knowledge to carry out the cumbersome configuration process [28], [29].

2.2 Large Language Models

Language Models are systems that learn patterns in natural language and predict the probability of the next token in a sequence. Tokens may be characters, subwords, words, symbols, or other text units, depending on the tokenization method used. Earlier language models were based on statistical methods. Large Language Models (LLMs) extend this idea using very large neural networks trained on a large corpus of data. The main objective of an LLM is to predict the next token given the previous context, although there are exceptions such as BERT and T5. Although the objective is simple, large-scale training allows LLMs to perform many language-related tasks, such as text generation, summarization, and translation. This has made them useful in areas where large amounts of textual or semi-structured data need to be interpreted.

Most modern LLMs are built on the Transformer architecture introduced by Vaswani et al. [30]. The main idea behind the Transformer is the attention mechanism, which allows the model to connect one token with other tokens in the same context. Self-attention assigns different levels of importance to different tokens, so that the model can focus more on the parts of the context that are most relevant. This is useful because the meaning of a word or symbol often depends on what appears around it.

LLMs are usually trained through more than one stage. During pre-training, the model learns from large datasets in a self-supervised manner, with the objective of predicting the next token from the previous context. After pre-training, the model can be adapted for more specific tasks or user expectations through fine-tuning. Fine-tuning may include instruction tuning, where the model learns to follow natural language instructions, and reinforcement learning from human feedback, where human preferences are used to improve the model's responses [31]. Some systems also use Retrieval-Augmented Generation (RAG) during inference, where

relevant external information is retrieved and passed to the model [32]. The retrieved data can help the model give more grounded responses, especially when the answer depends on external or changing information.

Prompting is the process of providing an LLM with instructions and context at inference time so that it can produce a useful response. A prompt may contain a question, a task description, background information, examples, or rules that the LLM is expected to follow. Different prompting methods, such as zero-shot prompting, one-shot prompting, and few-shot prompting, are available. They differ mainly in the number of examples provided to the LLM. In zero-shot prompting, the LLM is provided with a natural-language instruction describing the task it is expected to perform. In one-shot prompting, the LLM is provided with a demonstration of the task it is expected to perform as an example along with the instructions in natural language, while in few-shot prompting, instead of one demonstration, a few demonstrations are included in the instruction.

2.2.1 LLMs in Host-Level Security

Researchers have started exploring the use of LLMs for host-level security. A host system often produces large amounts of log data. The logs can contain useful details but are often noisy, long, and difficult for analysts to review manually. LLMs are useful here because they can summarize text, connect related events, and explain suspicious behavior. For the same reason, much of the recent work on LLMs in host-level security has focused on intrusion detection and alert interpretation.

Sun et al. [33] propose a multi-level LLM framework for host-level intrusion detection called Secure Host-based Intrusion dEtection with LLM Defense (SHIELD). The system addresses the problems that traditional host-level intrusion detection systems often face, such as high false-positive rates. SHIELD first parses raw audit logs to extract event identifiers, event attributes, and entity information related

to activities such as command execution and application behavior. Because raw log data can exceed the context length of an LLM, the authors use masked autoencoders to identify time windows that are more likely to contain attack-related events. Identifying the relevant time windows helps reduce the amount of information that must be sent to the LLM and allows the later stages of analysis to focus on the most relevant parts of the log. Then, SHIELD uses a focus-and-expand process to investigate suspicious events in more detail, where the LLM first identifies events that appear highly suspicious and uses a breadth-first search method to build an evidence neighborhood of events that are directly or indirectly related to them. The system also uses a method called Deterministic Data Augmentation to create benign profiles for processes based on attack-free training data. These benign profiles help the LLM compare suspicious behavior with normal process behavior. Finally, the LLM receives attack evidence, evidence neighborhood, and benign profiles as input and is prompted to support attack investigation. Specifically, the LLM is instructed to provide three different types of output: entities exploited by the attacker and events associated with those entities, each step of the attack chain labeled with the appropriate MITRE ATT&CK tactic, and a detailed narrative that outlines the entire attack in plain text. Sun et al. [34] also introduce HIDBench as a benchmark for evaluating how well LLMs can support host-level intrusion detection.

Maseno et al. [35] explore the integration of LLMs with intrusion detection systems for analyst-guided automated threat response to reduce the Mean Time to Respond (MTTR) and analyst fatigue. Their work connects the Suricata intrusion detection system with an LLM, specifically the GPT-4 model, to support alert classification and mitigation recommendation. In this system, Suricata works as the primary detection engine and produces real-time security alerts in a structured JSON format. The alerts are stored in a file, which is continuously monitored by a Python-based auto-responder engine. The engine filters alerts based on selected

criteria, such as IP address or severity level, and places them into a queue in a persistent state store for analyst review, along with their pending response actions. The system also sends a Slack notification to the analyst and makes the alert available through a dashboard. When the analyst selects an alert in the dashboard, the alert and its related context are sent to the LLM through an API. The LLM classifies the severity, explains the potential threats, and recommends mitigation. The analyst can then approve or deny the mitigation action from the dashboard itself, based on that the system parses the pending action from the state store and executes the corresponding mitigation. However, the authors did not explicitly mention what happens with the LLM-recommended mitigation if it is different from the pending action in the persistent state store.

2.2.2 Limitations of LLMs

LLMs still have important limitations that must be considered before using them in security-related tasks. One of the most discussed limitations is hallucination, where an LLM produces an answer that sounds convincing but is factually wrong, especially when the LLM is uncertain [36]. LLMs are also sensitive to the way prompts are written. Small changes in wording, ordering, or examples can sometimes lead to different outputs. In few-shot prompting, even the order of examples can affect the model's performance [37]. In ordinary writing tasks, issues such as hallucination may lead to inaccurate explanations. In security tasks, the same problem can be more serious because a wrong answer may cause an analyst to miss a threat, trust vulnerable code, or take an unsafe action.

LLMs also have different kinds of risks associated with them, such as technical risks [38] and ethical risks [39], [40]. Prompt injection is one such technical risk, which occurs when user prompts alter the LLM's behavior or output in unintended ways [41]. This is especially relevant when the LLM reads logs, alerts, or other

untrusted content. An attacker may place malicious instructions inside the input so that the model ignores the original task, leaks information, or recommends unsafe actions. Another technical risk is data poisoning, where training, fine-tuning, or retrieval data is manipulated to introduce bias, vulnerabilities, or hidden backdoors into the model or system [42]. One ethical risk is that LLMs may act in conflict with ethical standards or human goals and values, especially the goals of designers or users. Another risk is that an LLM may memorize and leak sensitive personal data or infer private information about individuals without their consent. These risks are especially important in security contexts because logs and alerts may contain sensitive details such as usernames, IP addresses, commands, file paths, and system configurations. Other similar risks are described in detail by Slattery et al. [40].

The limitations of LLMs are particularly problematic when it comes to security-related tasks. For example, when LLMs are used in host-level security tasks, a hallucinated answer may incorrectly state that a vulnerable configuration is safe or that a malicious event is harmless. Prompt injection may cause the model to ignore security rules or produce instructions that benefit the attacker. Poisoned data may cause the model to hide certain attack patterns or recommend unsafe actions. For these reasons, even though LLMs can improve security workflows, they should not be treated as fully reliable security authorities.

2.3 Overview of Intent-Based Systems

2.3.1 Definition of an Intent

Intent-based systems represent a fundamental shift in how people envision future machines. Hewlett-Packard (HP) can be seen as one of the early contributors to this shift, having taken the initiative to bring together the work on Intent-Based Networking (IBN) in 2013 by hosting the first IBN summit in Palo Alto [43]. Many

major networking companies have continued this work through the North-Bound Interface (NBI) working group, which has since published the definition and technical overview of IBN. Over time, the same idea has also been adopted in other domains beyond networking.

The meaning of the word intent changes depending on its context. In the context of Android, an intent is an abstract description of an operation to be performed, and it requests an action from another application component [44]. It also supports runtime binding between components within the same application or across different applications [45]. In blockchain, intent refers to a concept that allows the user to define their goals in abstract terms rather than executing specific instructions. A few researchers have developed a system in which users can specify their expectations from a chosen blockchain without needing to know the implementation details [3]. In the context of chatbots, an intent refers to what a person wants when interacting with a chatbot or virtual agent. In other words, it represents what the user wants to do or ask about [46].

In contrast to all these, in the case of autonomous AI agents, intent has several layers. For example, user intent is the goal or task that the user is trying to accomplish. Developer intent refers to the purpose for which the agent is designed and built, whereas role-based intent refers to the specific function the agent performs within an organization. Another layer called organization intent relates to business policies, standards, and operational constraints. Here, a successful AI agent must satisfy what the user intended to accomplish, while operating within the bounds of what the developer, role, and organization intended it to do [47].

Therefore, in a broader sense, intent can be defined as a high-level abstraction that specifies the end goals a system should meet without mentioning how to achieve them or delving into lower-level details [2], [48].

2.3.2 Types of Intent-Based Systems

Intent-Based Networking Systems

The most prominent intent-based systems are IBN systems. These are the systems that allow an operator to control a network through a set of operational goals that a network is supposed to meet and outcomes that a network is supposed to deliver, without specifying how to achieve or implement them [1]. Traditionally, operators manually configure each network device whenever a change is required. The configuration process is time-consuming and prone to errors, and IBN addresses this problem. Advances in Software-Defined Networking (SDN) have played an important role in making this possible, as many IBN systems are built on SDN.

There are different types of IBNs. Commercial solutions, such as Cisco Catalyst Center [49] target enterprise environments, whereas solutions like Juniper Apstra [50] target mainly data centers. When these commercial solutions are not enough, organizations with global-scale infrastructure have developed their own proprietary systems, such as Robotron by Facebook [51] and Orion by Google [52]. Robotron [51] is a top-down management system that uses standardized data models to convert high-level operator goals into device settings and provides tools for safe deployment and constant health checks. The results from long-term use at Facebook have shown that this model-driven method reduces operational mistakes and allows for the smooth update of network designs while ensuring that the system stays in its intended state. Orion [52] is a modular system that coordinates independent services, which translate high-level operator goals into device settings. It operates as a real-time control plane designed for traffic engineering within large-scale networks.

In addition to commercial and proprietary systems, several open-source and research-oriented IBN solutions have been developed. An early attempt to create IBN specifically for scientific networks is iNDIRA [53], developed by the Energy Sciences Network. Its developers aimed to address the difficulty faced by non-technical

scientists when trying to manually provision complex, high-performance network resources for large-scale data transfers and collaborative experiments. iNDIRA is a tool that uses natural language processing and structured data models to translate simple user requests into automated network configurations and service commands. There are also open-source frameworks that are used to build IBN systems, such as ONOS, developed by the Open Networking Foundation [54], and the OpenDaylight SDN controller [55].

Intent-Based Infrastructure Management Systems

Inspired by developments in the domain of IBN, the idea of intent-based systems has also been adopted for infrastructure management. There are mainly two types of systems based on how they convert high-level intents to infrastructure: deterministic rule-based systems and LLM-based systems.

Rule-based systems make use of predefined grammars, rules, and templates. The system described in [56] uses a language parser based on parsing rules and Part-of-Speech tagging to identify requirements from an intent to deploy cloud resources accordingly. Another framework, Int2IT [2], maps intents to standardized infrastructure-as-code (IaC) templates. Int2IT is an autonomic infrastructure management platform that realizes high-level business requirements expressed in the natural language. It converts user intents into standardized application descriptions and uses an autonomic control loop to continuously monitor and adjust the system state. The platform uses TOSCA [57] to manage the infrastructure by generating configurations from natural language intents. The platform then uses these configurations to deploy and manage infrastructure.

A few other studies have used advancements in LLMs for infrastructure management. In AppleSeed [58], the authors used generic LLMs to translate natural-language intents into Python code for infrastructure provisioning. The platform uses

in-context learning to translate natural language into an intermediate high-level execution plan. The platform then automatically converts this high-level execution plan into a parallelized execution graph using Just-in-Time compilation. Finally, it converts the execution graph to domain-specific execution plans, which an executor uses for actual execution. Similarly, some researchers have proposed the use of LLMs to build microservices by decomposing an intent into multiple hierarchical steps, such as high-level design and low-level design [59].

Intent-Based Security Systems

Security is becoming an important area within intent-based systems, where high-level user objectives are translated into enforceable security policies and configurations. Existing research has explored several approaches to achieve security through intent, including formal methods and AI-driven frameworks. These approaches differ in how intents are interpreted and deployed, but they share the common goal of reducing the complexity of security management while ensuring that security requirements are correctly enforced.

Methods such as formal verification and finite automata have been used in some works as supporting mechanisms to achieve security through intents. The Intent-Based Cloud Service (IBCS) framework, described in [60], uses a formal method. It focuses on the intent-driven deployment of cloud security services. The IBCS framework uses finite automata to understand the security intent and then identifies and deploys suitable Network Security Functions (NSFs) to satisfy the intent. A system called INTPOL, described in [61], uses formal verification to identify conflicts between intents. Conflict detection is crucial because conflicting intents, if implemented, can make the system unstable. INTPOL works by translating high-level security goals into a unified representation and applying mathematical verification techniques to identify policy conflicts before deployment. The framework combines

Linear Temporal Logic (LTL) with bounded model checking and uses a SAT solver together with the symbolic model checker NuSMV [62] to perform the verification.

Several recent studies have also explored the possibility of using the ability of LLMs to process unstructured information in systems that handle security intents. RefinementEngine, presented in [5], expects users to express their intents in High-level Security Policy Language (HSPL), while using LLMs to extract security-relevant entities, such as malicious IP addresses, from Cyber Threat Intelligence (CTI) reports written in natural language. The system then uses this information to generate device-specific commands for deployment in a deterministic manner. Su et al. [63] used an LLM to generate policies for Kubernetes [64] deployments based on data privacy and residency requirements expressed in natural language. The system uses intents expressed in natural language to select compliant Kubernetes nodes for pod placement and deploy traffic management rules that ensure network traffic is confined to compliant paths only. Here, the LLM analyzes the requirements expressed in natural language and generates ready-to-deploy policies. Zacarias et al. [65] proposed a framework that is more autonomous than the systems mentioned in [63] and [5]. They introduced a hierarchical multi-agent architecture that incorporates a dedicated Security Plane. The system relies on AI agents to translate natural language requests into technical security configurations and deploy them accordingly. A central coordinator AI agent coordinates task execution and delegates responsibilities to specialized sub-agents. The results from their functional prototype showed that even small, on-premises language models can effectively automate complex security tasks with high accuracy.

Although all three works mentioned previously in this subsection used LLMs, they were utilized differently. In RefinementEngine [5], the user must provide the intent in HSPL, and the system uses an LLM only as an information-extraction component to analyze the CTI reports. In contrast, the framework described in [63]

allows the user to specify their intent in natural language, and the LLM generates a ready-to-deploy Kubernetes policy based on the extracted intent. Zacarias et al. [65] extend this approach further by giving AI agents more autonomy to control the entire pipeline, from capturing user intents to their deployment.

Other Intent-Based Systems

The concept of intent is not limited to the technical domain. The intent-based paradigm has also been extended to several other domains. It has found its way to the military. Researchers from the US military are working toward a framework for intent-based orchestration of distributed command-and-control centers to achieve a commander's intent [66]. The framework applies automated networking principles to translate a commander's goals into machine-readable tasks using standardized mission models and mathematical reasoning. Researchers have also proposed intent-based systems to orchestrate the networking and computing requirements of vehicular edge computing applications [67]. In addition, there is an intent-based system that generates reinforcement learning environments that can be used to train AI agents to satisfy user intents expressed in natural language [68].

2.3.3 Stages of an Intent-Based System

There are different stages associated with the lifecycle of an intent in an intent-based system. This section discusses those stages. Since the most widely used intent-based paradigm is IBN, much of the literature referenced here is from that domain. However, at a high level, these stages are common to all intent-based systems in general.

Intent Expression

The first stage of any intent-based system is intent expression. At this stage, the user conveys their intent, that is, what they want the system to achieve. This intent can be expressed in several ways, depending on what the system supports. The easiest and most natural way is through natural language [4], [48], [63]. This allows users to clearly express their needs in their own language. Especially with the advancements in natural language processing and LLMs, natural language has become the preferred option in several recent works.

Controlled languages are also used to express intents. Although natural language allows users to express their requirements freely, processing natural language is challenging because of many factors, including ambiguity and contradictions. Therefore, other forms of expression, such as domain-specific controlled languages, are used instead. A controlled language is a subset of natural language with limited vocabulary and grammar. This reduces ambiguity and enables more efficient text processing. Some of the controlled languages are Network Intent LanguagE (NILE) [4], Network Modeling (NeMo) [69], and Language for ACL Intents (LAI) [7]. NILE is an intent language designed to capture connectivity and middlebox intents through a human-friendly syntax. Bezahaf et al. [70] have developed a system that uses natural language processing to translate simple human instructions into an intermediate representation and then into network settings. They used an extended version of NILE in their work. Similarly, NeMo [69] is a controlled language used mainly in networking. Researchers at Alibaba Cloud have developed an automated system called Jinjing that allows network operators to update ACL configurations using a high-level language. They used LAI as the high-level language for this system [7]. Researchers have also used controlled languages for intent expression to support the selection of suitable blockchains [3].

Both natural and controlled languages are best suited for humans to express

their intents, but the question of how machines can express intents to other machines still remains. Researchers at Nokia Bell Labs addressed this question in their work [12]. They introduced the concept of emergent communication for intents, in which machines autonomously agree on the meaning of the vocabulary used for intent expression between them.

In addition to the language of intent expression, methods for capturing intent are also important. One common method is a Graphical User Interface (GUI), which is usually paired with predefined templates that map user choices to low-level configurations. This method has several advantages. For example, it simplifies the configuration process for inexperienced users, limits the action space to supported operations, and eliminates certain conflicts. It also makes intent extraction simpler, since the system just needs to parse input that is already structured. A related approach lets users type free-form text instead of selecting from a fixed template, giving them more freedom to express their intents. Chatbots can also be used for this purpose. For example, Jacobs et al. [4] used Dialogflow [71] (which is currently part of Google's Customer Engagement Suite) to capture intent from the user. Similarly, the system described in [72] used a Rasa-based chatbot as an interface for interacting with an intent management layer. Researchers have also explored voice-based intent expressions [73].

Intent Extraction

Once the user intent is captured, the next step is to extract the main requirements expressed in that intent. The extraction process largely depends on how the intent is captured. When the system collects intents through a template-based GUI, the extraction process is relatively straightforward because the required information can be directly parsed from the template fields. However, extracting requirements from natural language is more challenging. Natural language intents may contain

ambiguous expressions or may not use commonly accepted technical terminology.

Previous studies have explored several approaches for extracting requirements from user intents. The approaches include methods based on deterministic finite automata, NLP, and LLMs. These techniques identify relevant information from user inputs and transform them into structured representations that support intent realization. An overview of different types of methods and associated research works is presented in Table 2.1.

Deterministic finite automata have been used in previous works for data extraction from intents. For example, researchers have proposed a framework that combines standardized security interfaces, such as Interface to Network Security Functions (I2NSF), with network function virtualization to automatically translate high-level user intents into executable technical policies for virtualized security tools [60]. The framework uses deterministic finite automata in the data extractor module of the system to parse the high-level intent delivered by the user and extract meaningful data points. Similarly, in another intent-based system for the selection of suitable blockchains, the system uses deterministic finite automata for parsing intents expressed in a controlled language [3].

NLP techniques have been widely used for requirement extraction from natural-language intents. The system described by Jacobs et al. [4] uses Named Entity Recognition to identify important entities from user intent captured through Dialogflow. Similarly, the parser module in iNDIRA uses NLP methods for identifying user-provided terms and maps them to specific network service commands to generate Resource Description Framework (RDF) graphs for intent realization. EVIAN [74] uses a natural language chatbot and machine learning to extract user requirements and then converts them into structured data models for automated network configuration. This system also uses NLP techniques to generate RDF graphs that map user intents to network commands.

More recent studies have explored the use of LLMs for extraction tasks. The hierarchical multi-agent framework for performing security actions discussed by Zacarias et al. [65] uses an LLM. Here, the LLM is used to extract the required information from natural-language intents to convert them into structured representations using predefined data structures. Capova et al. [68] used an LLM to decompose vague business intents into a set of manageable sub-intents and extract relevant node labels from associated infrastructure knowledge graphs. Similarly, Su et al. [63] also used an LLM to extract the requirements expressed in natural language and link them to specific low-level Kubernetes labels.

Table 2.1: Methods of Intent Extraction

Type of Method	**Method**	**Related Works**
Formal Method	Deterministic Finite Automata	[3], [60]
Natural Language Processing	Named Entity Recognition	[4], [74]
Large Language Models	ReAct paradigm, Few-Shot Prompting	[63], [68]

Intent Translation

After requirements are extracted, they must be converted into a machine-friendly structured form. Researchers have proposed different methods for this transformation process, such as using context-free grammars, custom translation modules, and large language models.

A few approaches rely on custom translators or external providers [70] to support requirement translation. The Int2IT [2] framework, described in Section 2.3.2, includes a custom translator module that converts user intent into machine-friendly TOSCA YAML definitions. Similarly, RefinementEngine [5] has a custom translator that translates high-level requirements into a Medium-level Security Policy Language (MSPL). Some researchers have also used context-free grammar (CFG) to translate user requirements into machine-friendly representations. The policy

generator component of the IBCS [60] creates an XML-based low-level security policy from user intent. This component uses CFGs to transform the extracted intent information into structured XML documents.

Large language models are also widely used in this stage because of their ability to understand and structure natural language inputs. NFV-Intent is a system that deploys virtual network functions and service chains based on natural language instructions. The system converts user instructions into structured JSON configurations and then uses these configurations to call the APIs of the underlying Network Intelligence Network Function Virtualization Orchestrator. The system achieves this transformation process using the in-context learning capability of LLMs [75]. AppleSeed [58], described in Section 2.3.2, also uses in-context learning to translate high-level intents into execution graphs for infrastructure provisioning. Emergence [48] is a system that automates the deployment and monitoring of applications through natural language. Emergence uses an LLM to progressively translate natural language instructions into a policy tree with policies as JSON objects. The system uses prompt engineering and few-shot learning techniques to map user intent into policies.

Intent Fulfillment

Once the requirements are translated into a machine-friendly structured form, they must be deployed as actual actions to fulfill the intent. Before this, the system must verify that no active intents conflict with the new requirements. If conflicts exist, they must be resolved through a process known as conflict detection and resolution.

Graph-based approaches have been widely explored for representing and analyzing relationships. Prakash et al. [76] introduced Policy Graph Abstraction (PGA) to represent ACL policies as graphs, allowing the system to identify and resolve conflicts through graph analysis techniques. Building on this idea, JANUS [77] extends

PGA using heuristic algorithms to resolve conflicts and optimize QoS-related policies in networks. Another graph-oriented direction for policy analysis was presented by McNamara et al. [78]. Their method structures intents using five descriptors and applies a specialized dictionary to convert values into a common representation that supports direct comparison between policies. These approaches have shown that graph abstractions can simplify the analysis of complex policy relationships while supporting conflict management.

Formal verification methods have also been applied for intent conflict detection. INTPOL [61], described in Section 2.3.2, converts intents into LTL and applies bounded model checking to detect conflicts before deployment. The evaluation results demonstrate that detecting conflicts at the intent level is significantly faster than checking the rules individually at the network-device level. In Jinjing [7], a Satisfiability Modulo Theories (SMT)-based module detects and resolves ACL update conflicts while validating policy consistency during updates. Similarly, Li et al. [79] address the limitations of earlier systems that evaluate only individual flow rules while ignoring complete user intents and temporal constraints, such as delayed requests. The mechanism they propose uses an SMT solver to analyze logical expressions across nine policy characteristics and identify five relationship categories between translated policy sets. All these studies highlight the value of formal methods in providing precise and verifiable conflict analysis for intent-driven systems.

Recent studies have increasingly used AI and machine learning techniques for conflict detection. The authors of AGIR [80] use natural language processing to translate business objectives into network instructions for open radio access networks. Its conflict resolver module uses an LSTM-based classification technique to detect semantically conflicting intents by treating conflict detection as a binary classification problem. The authors reported more than 80 percent precision in identifying conflicting instructions. Perepu et al. [81] proposed a framework for

handling multiple network performance goals in cellular systems using multi-agent reinforcement learning. The framework treats conflicts as situations where the actions of one control loop negatively affect the Key Performance Indicators (KPIs) of another loop. During centralized training, agents learn coordination strategies through a shared global reward function that discourages greedy actions, which lead to conflicting behavior. Natarajan et al. [82] introduced a logistic-regression-based approach for conflict detection. They first translate natural-language intents into a controlled language called NILE. Then, they transform each NILE intent into a vector embedding and process the modulus difference between embeddings using a logistic regression model to estimate the probability of conflict. An overview of conflict detection and assurance mechanisms in the domain of IBN can be found in the survey paper by Gharbaoui et al. [83].

After conflicts have been resolved, translated intents must be deployed using executable actions. Some systems rely on deterministic rule-based mechanisms to invoke the required applications and services [6], [70]. Other studies have explored AI-driven orchestration methods that dynamically route tasks and coordinate execution workflows [63], [68]. For instance, the Int2IT [2] framework fulfills intents by converting them into standardized infrastructure-as-code files, which are then processed by a resource manager to provision and manage cloud resources through external connectors. In contrast, Zacarias et al. [65] used an orchestrator AI agent that distributes tasks among specialized subordinate AI agents. These deployment approaches demonstrate the growing shift from static execution pipelines toward more adaptive and intelligent orchestration frameworks in intent-based systems.

Intent Assurance

Intent assurance is fundamental to closed-loop automation in intent-based systems, since deployed intents must continue to satisfy user-defined requirements throughout

their lifecycle. This closed-loop automation relies on continuous monitoring. Commonly, intent assurance is carried out by observing KPIs and applying corrective actions whenever the observed behavior deviates from the expected behavior. Such deviations are commonly referred to as intent drift, a condition in which the operational state of the system no longer matches the original intent specification. Several studies have explored different approaches for detecting, analyzing, and mitigating intent drift.

Graph-based assurance mechanisms have been widely used to maintain intent requirements. In the work by Edeline et al. [84], the open-source tool DxAgent collects data from multiple sources to construct assurance graphs that the system later uses to detect intent drift. An orchestrator then plans the rectifying actions based on the identified drift. Fernández et al. [85] presented a different monitoring-oriented approach, which uses the multi-pronged monitoring framework. Their approach aggregates metrics gathered from multiple network segments to establish a ground-truth view of the current network state. These metrics are continuously monitored and compared with predefined threshold values. When a metric deviates from its expected value, a state transition is triggered in an extended finite state machine. This transition activates a self-healing mechanism to restore normal system operation. Both of these studies demonstrate that continuous monitoring and state-based analysis can support automated assurance in closed-loop systems.

SAFLA [86] follows a different approach for intent reconstruction. Instead of relying on predefined models, it uses a bottom-up method that examines the flow tables currently deployed in the network. By analyzing the installed flow rules, the framework derives the effective network intent and compares it with the original intent to identify any differences. When intent drift is detected, SAFLA uses an *Event-Condition-Action* mechanism to correct the deviation. Herbaut et al. [87] proposed another approach using a conformance-checking algorithm that uses intent-

based models as an intermediate layer between management policies and observed network traffic. This approach helps identify missing or unnecessary flow rules that violate existing SDN security policies.

Machine learning techniques have also been integrated into intent assurance frameworks to improve prediction and adaptive decision-making capabilities. The closed-loop automation architecture presented by Leivadeas et al. [88] uses telemetry data to train machine learning models that can predict KPI deviations before they occur. Zheng et al. [89] explored a similar predictive approach, where they used neural networks in data center IBN environments to estimate CPU usage patterns that can result in intent drift. These studies show a shift from reactive assurance, which addresses problems after they occur, to proactive assurance, which seeks to identify and prevent potential deviations in advance.

Recent research has explored the use of LLMs and reinforcement learning techniques for intent assurance. Dzeparoska et al. [90] used few-shot learning with LLMs to define KPIs and KPI-related policies. In addition to LLM-based approaches, reinforcement learning has also been investigated for adaptive service assurance. The framework by Njah et al. [91] uses deep reinforcement learning agents to support intent assurance operations. A broader overview of assurance mechanisms, architectures, and research challenges in this area is presented in the survey paper by Gharbaoui et al. [83].

2.4 Related Works

Since intent itself is a relatively new concept, there are not a lot of works that deal with intent-based security. To the best of our knowledge, there has been no academic research on intent-based security systems at the host level. A few works that can be considered closely related to our work are Emergence proposed by Dzeparoska et al. [48], RefinementEngine proposed by Colaiacomo et al. [5], and the natural

language interface to configure enterprise firewalls proposed by Taghiyev et al. [92].

Emergence, proposed by Dzeparoska et al. [48], is an intent-based management system for application deployment. The main goal of Emergence is to automate application management by decomposing a user's intent in natural language into smaller policy steps that can be mapped to system APIs. It uses a generic LLM with few-shot prompting to generate policies whose actions follow the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) closed control loop principles. The system uses a three-stage LLM pipeline and policy-based abstraction to achieve this. The first stage is intent classification, where the LLM classifies the user intent into one or more supported intent types, such as creating a resource or deploying a service. The second stage of the pipeline is progressive intent decomposition, where the LLM decomposes intent into policies with actions to deploy the intent in a MAPE-based fashion. Instead of generating all policies at once, the LLM outputs one policy at a time. After each policy is generated, the system maps it to an API, executes it in the digital twin environment, and sends the result back to the LLM. The LLM then uses this result to decide the next policy, which allows the policy sequence to adapt to the current state of the system. The final stage of Emergence focuses on validation before deployment. In this stage, an LLM reviews the generated policy tree to find possible problems such as formatting errors, missing attributes, or incorrect ordering of actions. If the validation process changes any policy, the updated policy tree is tested again in the digital twin environment before it is used for deployment. Once the validation is successful, the policy tree is used to deploy the intent. However, Emergence mainly focuses on application management and does not address security.

RefinementEngine, described by Colaiacomo et al. [5], is also close to our work because it converts high-level security intents into deployment-ready configurations. The RefinementEngine focuses on network-level filtering policies under topology and device constraints. The system starts by processing CTI reports using an extrac-

tor module. This module uses LLM-based processing to identify security-relevant entities, such as malicious IP addresses and indicators of compromise, and represents them as facts in an expert language called CLIPS [93]. The refiner module then combines CTI-derived facts with security objectives defined by the user using HSPL, and a network topology model described using TOSCA. The refiner module then analyzes possible traffic paths between endpoints and identifies where filtering rules should be placed in the network to ensure full coverage of all traffic paths while minimizing the number of devices configured. The module also considers the security capabilities of available devices defined using the Security Capability Model [94] while selecting the enforcement points for filtering rules. The converter module then generates device-specific policy definitions through an intermediate representation in MSPL, which are represented as XML documents. The translator module then converts the MSPL policies into low-level rules in the native language of the target device, such as iptables commands or other filtering rules. The RefinementEngine is useful for understanding how high-level security goals can be refined into concrete configurations. However, the RefinementEngine focuses on network-level enforcement and does not address host-level security intent.

Taghiyev et al. [92] present a system that uses natural language to configure enterprise firewalls. Their system allows administrators to express access control intent in natural language. The natural language intent is then translated into a vendor-specific firewall configuration. A key idea in this work is the use of a structured Intermediate Representation (IR), which separates the administrator's intent from the final device-level syntax. This makes the intent easier to validate and audit before any configuration is applied. The natural language frontend allows administrators to provide access control intent and upload a network context file containing information about existing objects, zones, and services. An LLM agent known as the resolver agent parses the natural language access control intent and replaces

ambiguous names, such as HR laptops or Payroll SaaS, with concrete entities in the network context, such as specific subnets or address objects. Another LLM agent called the IR builder agent then takes the resolved access control intent and converts it into the predefined IR format. After the IR has been produced, it undergoes verification and analysis. A general linter and a vendor-specific linter check the IR for structural issues and hygiene rules, providing non-blocking warnings to the administrator. Then the safety gate inspects the final IR for inherently unsafe rules, such as any-to-any permit rules or rules with empty source/destination lists. Once the IR passes these checks, it is translated deterministically into vendor-specific commands. Before the configuration reaches a live firewall, an offline simulator checks for issues such as parsing errors, undefined references, and unused structures. Although this work deals with access control intent, the focus remains on the network level rather than the host level.

Both the RefinementEngine and the natural language interface to configure enterprise firewalls deal with security at the network level, whereas Emergence focuses on intent-based application management. Even though all the works deal with intent-based systems, none of them specifically target intent-based security systems at the host level. To address this gap, we designed an architecture for an intent-based security system at the host level, taking inspiration from these works and a few others. We took inspiration from the few-shot intent decomposition approach used in Emergence [48]. We also adopted the idea of human-in-the-loop clarification for vague or incomplete intents from the work of Zacarias et al. [65]. The decision to use an intermediate representation was influenced by the firewall configuration system of Taghiyev et al. [92] and RefinementEngine [5]. In addition, the idea of using an execution queue based on dependency analysis over an intent graph was inspired by AppleSeed [58].

3 Methodology

This thesis followed a mixed-methods approach combining a literature review [95], a case study [96], requirements engineering [97], user journey mapping [98], and AI-assisted software development methods. The goal of using these methods together was to develop a thorough understanding of the existing work and use these insights to design, build, and evaluate an intent-based security system. An overview of the research workflow is presented in Figure 3.1.

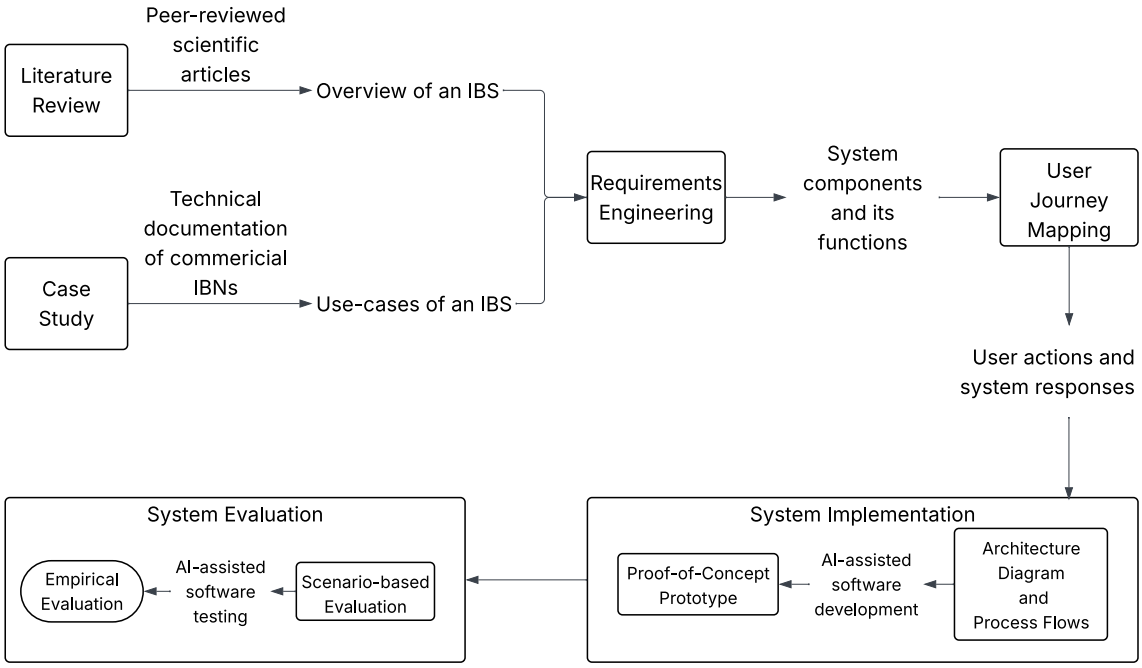


Figure 3.1: Overview of Research Workflow

To understand the problem space, we conducted a literature review of peer-reviewed articles and research preprints, focused on understanding how security is

achieved through intent. The review covered research papers indexed in Google Scholar and DBLP. The search was performed using the keywords "intent-based systems", "intent-based networks", "intent-based security", "intent-based management", and "host-level intents". For each keyword, the search results were limited to the first 50 results. After collecting the papers, we examined their abstracts to identify studies relevant to this research. Papers were selected for detailed review if they discussed any of the following: systems that use user-defined intents in any application domain, the security aspects of intent-based systems, or systems that implement security requirements through intents. The selected literature was then analyzed to identify the main components of intent-based systems and the stages of intent processing. The review also examined how existing intent-based security systems handled security requirements.

In addition to the literature review, we conducted a case study to understand how intent-based systems are applied outside of a research environment. We studied the technical documentation of commercial IBN products, specifically Cisco Catalyst Center and Juniper Apstra. This helped us understand the different use cases associated with an intent-based system. Together, the literature review and case study provided an understanding of the generalized architecture of an intent-based system and the state of the art in the domain.

Using the information gathered from the literature review, we applied requirements engineering to identify the different modules of the system and the functions associated with those modules. The functional requirements identified from the literature were converted into architectural components and their interactions. This step helped shape the structure of the proposed architecture.

Building on the architecture identified through requirements engineering, we used user journey mapping to understand how a user would interact with the system and to further refine the system modules. This helped us understand the system from

the perspective of the user and identify all the actions a user would perform while interacting with the system.

Alongside user journey mapping, we built process flows to show how the system would respond to each action in the user journey. This allowed us to follow an intent as it moved through the system, from the moment it was entered to the moment it was fulfilled, giving us a clear picture of the intent's lifecycle from the perspective of the system. Using all these insights, we designed an architecture for the intent-based security system at the host level.

Based on the architecture diagram, we developed a proof-of-concept prototype of the system using AI-assisted software development. Cursor served as the main development environment. However, using Cursor extensively could get expensive. To keep the cost in check, we adopted metaprompting for the more demanding parts of the build, such as the individual system modules. Metaprompting [99] is the process of using one LLM to generate a detailed task prompt that is then provided as input to another LLM. For this step, we used Google AI Studio and its Gemini 3.1 Pro model to generate a detailed implementation prompt. The generated prompt was then provided to Cursor to carry out the coding task. The approach reduced repeated experimentation within Cursor and provided more detailed instructions for larger tasks. For smaller tasks, such as limited code changes or adjustments to existing functions, we prompted Cursor directly.

After developing the proof-of-concept prototype, we evaluated its functional correctness. We used two methods for the evaluation, namely scenario-based evaluation and empirical evaluation. The scenario-based evaluation used two representative scenarios and assessed their outcomes manually. For the empirical evaluation, we utilized AI-assisted software testing. We divided the intents into seven different categories and manually created two examples for each category. We then used ChatGPT, specifically the GPT-5.6 Sol model, to generate 28 additional intents for

each category based on those examples. We used random sampling to select eight generated intents from each category and combine them with the two manually created example intents. These intents were used for empirical evaluation by providing them to the system one at a time and manually comparing each result with its expected outcome. Both the scenario-based and empirical evaluations helped us validate our proposed architecture.

4 System Design

As mentioned in Section 2.3.2, although intent-based paradigms have been extensively studied in networking and service orchestration, their application to host-level security management remains unexplored. To bridge this gap, we propose an intent-based security system at the host level.

4.1 System Overview

The proposed system analyzes security intents expressed in natural language and converts them into executable actions. Rather than directly executing the intents without verification, the system processes them through validation and planning stages. At a high level, the architecture consists of four modules: Intent Processing Module, Task Deployment Module, User Interaction Module, and Logger and Retriever Module. The system also has three databases for data storage: a graph database, an unstructured database, and a vector database. A high-level architecture diagram of the system is shown in Figure 4.1.

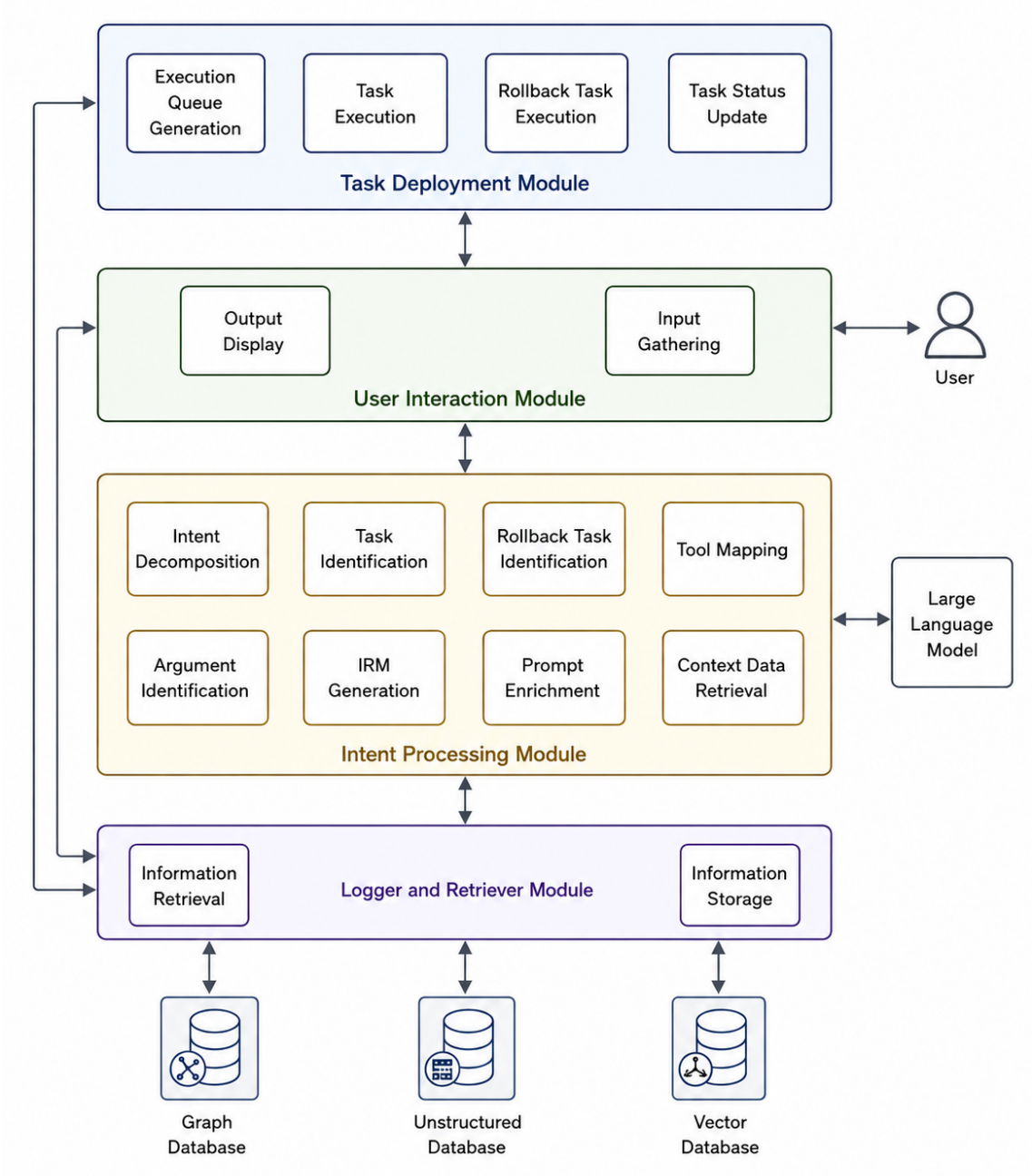


Figure 4.1: Architecture Diagram of the System

User communication and data handling are kept separate from the rest of the system through two shared modules, namely, the User Interaction Module and the Logger and Retriever Module. All interactions between the system and the user are coordinated through the User Interaction Module, so individual modules never talk to the user directly. The workflow begins when a user submits a security intent in natural language through the User Interaction Module. The submitted intent is forwarded to the Intent Processing Module.

The Intent Processing Module serves as the central module for interpreting user security requirements and preparing them for execution. It uses an LLM to analyze the intent, decompose it into smaller executable tasks, and identify the appropriate tools on the host machine that are required for each task. After the intent is processed, the system converts the extracted information into a machine-friendly Intent Representation Model (IRM). This IRM is then passed to the Task Deployment Module.

The Task Deployment Module acts as the execution layer of the architecture. The module converts the IRM into a graph that maps out how the different tasks and intents depend on one another. The module then analyzes this graph to determine the execution sequence and builds a task execution queue from it. Following the successful execution of all the tasks in the queue, the intent is marked as active, and the intent graph is saved in the graph database, marking the completion of the deployment workflow. As mentioned earlier, all storage and retrieval operations to the databases go through the Logger and Retriever Module, which sits between the system's modules and the databases themselves.

The LLM is treated as an external service that is outside the system boundary. This choice is made to keep the system LLM-agnostic, allowing the implementer to use any LLM of their choice. The design also provides the flexibility to modify the system to use specific models for specific operations during intent processing.

4.2 System Modules

4.2.1 User Interaction Module

The User Interaction Module is responsible for managing all user-facing communications within the system. It supports interactions such as intent clarification requests, task execution confirmations, and execution-related status updates. Acting as an intermediary between the user and system modules, it collects user responses and returns them to the module that initiated the request. The module serves as a shared service that can be used by other system modules. This approach ensures that information is presented to users in a consistent manner.

In addition, the module supports auditing and traceability by maintaining records of user interactions. Information related to approvals, clarifications, feedback, and decisions is stored in an unstructured database using the Logger and Retriever Module. These records provide a clear history of actions involving human input and help ensure a consistent audit trail throughout the system.

The first step in realizing a security intent is capturing the intent from the user. As mentioned in Section 2.3.3, there are multiple ways to express and capture user intents. We chose natural language as the language of expression because it allows the user to express their requirements more flexibly. The system allows users to express their intent through a GUI with a free-form text box. The User Interaction Module captures the intent entered in the text box.

The module workflow starts when a system module requests input or confirmation from the user. In such situations, the requesting module sends relevant information to the User Interaction Module. The module then displays the information to the user through the GUI. After the information is displayed, the module waits for the user to provide a response. When the user provides a response, the module captures the response and forwards it back to the module that initiated the request.

The requesting module can then continue its execution using the received input, allowing user decisions to be incorporated into the automated system processes.

The workflow also includes logging as part of the response handling process. User responses, approval decisions, and clarification-related interactions are stored in the unstructured database through the Logger and Retriever Module for later reference.

4.2.2 Intent Processing Module

The Intent Processing Module serves as the central point for all LLM-related operations within the system. The module is responsible for processing the intents, including decomposing them into different tasks and identifying the tools required for each task. It also manages prompt preparation and the collection of reasoning summaries related to LLM decision-making.

The module consists of several components responsible for prompt generation, context enrichment, and data handling. The module prepares the prompts for the LLM depending on the stage of processing. When additional context is required, these prompts are enhanced using information such as few-shot examples or available tool details. This additional context helps the LLM produce more consistent and reliable outputs.

The module also has a mechanism for handling LLM interactions that require user involvement. When clarification or confirmation is needed during intent processing, the module communicates with the User Interaction Module. This enables the system to obtain the necessary user input before continuing with the process. To support this interaction, the module requires the LLM to return outputs in a structured format that contains an attribute indicating whether further user input is required. This mechanism supports iterative interactions that allow the LLM to resume processing the request based on the user's response.

In addition, the module supports reasoning-summary logging for auditability.

During certain operations, such as task planning and tool identification, the module instructs the LLM to generate summaries explaining the reasoning behind its decisions. These reasoning summaries are preserved for later auditing purposes. Logging operations associated with the reasoning summaries are managed through the Logger and Retriever Module. The system maintains an unstructured database where reasoning summaries associated with LLM-related operations are stored.

The final output of the module is a structured intent representation. Once processing is complete, the LLM uses the available details associated with the intent to generate a machine-friendly representation. We propose a model called the Intent Representation Model (IRM) to represent the intent in a machine-friendly manner, as described in Listing 1. Thus, the input to the module is the intent in natural language, and the output is an IRM.

The design of our representation model started with an evaluation of existing security and privacy policy languages. We reviewed several policy languages discussed in the study [100] and languages such as NILE [4], NeMo [69], and LAI [7], with the goal of finding a suitable way to represent the intents related to security requirements. In particular, we needed a representation that can capture the intent expressed in natural language, how that intent can be fulfilled, and the details to support its lifecycle management. To the best of our knowledge, none of the existing representations fully supported our requirements. Most of the languages support only a single use case, such as access control policies, accountability-related policies, or monitoring policies. Because no existing solution provides all the capabilities we needed in a single representation, we propose a new representation model that offers a unified structure for representing the intent, the process flow needed to realize that intent, and the information needed for intent-lifecycle management. It serves as an intermediate layer between security requirements in natural language and the process flow that implements them. It is important to note that the IRM is not a

Listing 1 Intent Representation Model

```
{
  "intent_id": "<intent_id>",
  "intent_status": "<active | deleted | provisioning>",
  "action": "<allow | deny | create | delete | update>",
  "scope": [
    {
      "type": "<scope_type>",
      "value": "<scope_value>"
    }
  ],
  "conditions": {
    "combiner": "<AND | OR | NOT>",
    "rules": [
      {
        "field": "<field_name>",
        "operator": "<operator>",
        "value": "<value>"
      },
      {
        "combiner": "<AND | OR | NOT>",
        "rules": [
          {
            "field": "<field_name>",
            "operator": "<operator>",
            "value": "<value>"
          },
          {
            "field": "<field_name>",
            "operator": "<operator>",
            "value": "<value>"
          }
        ]
      }
    ]
  },
  "owner": "<user_id>",
  "parent_id": "<intent_id>",
  "depends_on": ["<intent_id>"],
  "sub_intents": ["<sub_intent_in_IRM_structure>"],
  "metadata": {
    "raw_intent_in_NL": "<natural_language_description>",
    "version": "<intent_version_string>",
    "timestamp": "<timestamp>"
  },
  "associated_tasks": [{
      "task_id": "<task_id>",
      "tool": "<tool_id>",
      "arguments": [],
      "task_status": "<pending | executing | successful | failed>",
      "depends_on": ["<task_id>"],
      "reasoning_id": "<reasoning_id>",
      "task_type": "<primary | rollback>",
      "rollback_tasks": ["<task_id>"]
    }],
  "reasoning_id": "<reasoning_id>"
}
```

security-policy language. Instead, it is an intent representation model for security requirements.

Figure 4.2 illustrates the workflow of the module. It begins when a user submits an intent through the User Interaction Module using natural language. The submitted intent is first forwarded to the LLM for intent analysis and understanding. Because natural language input may contain ambiguous or missing details, the LLM evaluates whether the provided information is sufficient for further processing. If the intent is unclear, the LLM generates clarification questions and requests additional information from the user through the User Interaction Module. The clarification cycle continues iteratively until the LLM determines that the intent is sufficiently precise for processing. To prevent an infinite clarification loop, intent processing is terminated if the intent remains unclear after five iterations. The threshold of five iterations is selected to balance usability and processing efficiency. All clarification interactions are stored in an unstructured database through the Logger and Retriever Module.

The next stage of processing focuses on breaking down complex intent into simpler and more manageable parts. During this stage, the LLM analyzes the intent and identifies whether it contains multiple objectives. When necessary, the original intent is broken down by the LLM into smaller sub-intents that can be processed more effectively in later stages. Along with the generated sub-intents, the LLM provides brief reasoning summaries that explain the basis for decomposition. These reasoning summaries are stored in the unstructured database through the Logger and Retriever Module.

The next stage focuses on converting the intent into executable tasks. The intent, along with any corresponding sub-intents, is sent back to the LLM for task decomposition. The LLM determines the set of tasks required to fulfill each intent and identifies the corresponding rollback tasks that can be used for recovery purposes.

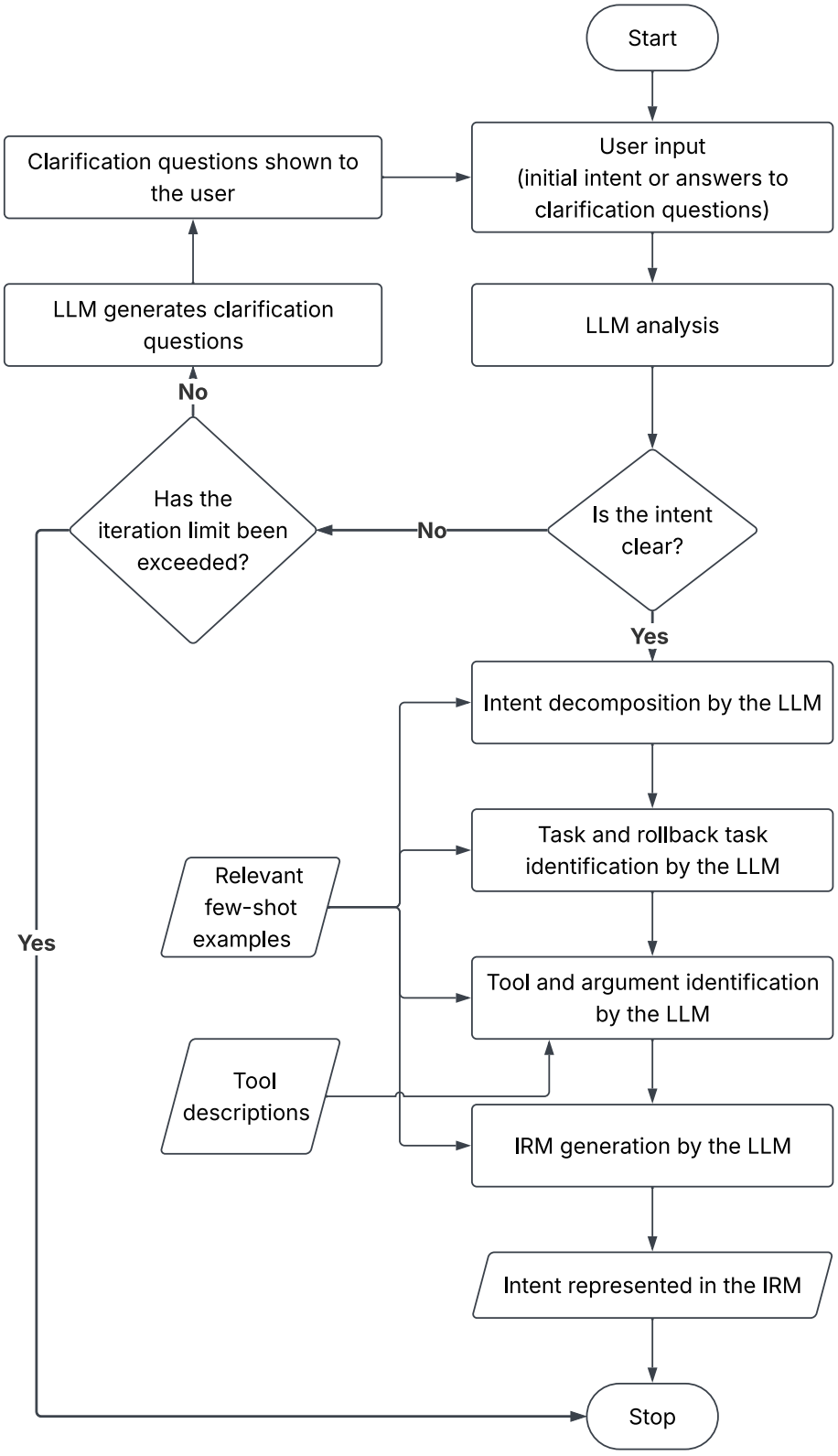


Figure 4.2: Intent Processing Workflow

If the tasks are irreversible, the rollback tasks are indicated as none. To improve consistency in the generated outputs, the prompt used at this stage is enriched with few-shot examples of task decomposition. The LLM then returns the list of tasks, rollback tasks, and reasoning summaries associated with its decisions. Similar to the previous stage, all reasoning summaries produced during task decomposition are stored in the unstructured database for future reference.

Once the sub-intents are decomposed into associated tasks, the workflow then moves to tool identification and argument generation. The generated tasks associated with each sub-intent are submitted to the LLM to identify the required tool for execution and generate the arguments required for that specific tool. During this step, the module enriches the prompt using few-shot examples for tool identification and tool descriptions. Following the principles of Retrieval-Augmented Generation, data for enrichment are retrieved by performing a similarity search in the vector database. This enrichment process helps maintain consistency in tool mapping across different intents. After task-to-tool mapping is completed, the LLM returns a list of tools and arguments associated with each task. If the LLM cannot identify a suitable tool for a particular task, the system informs the user and terminates further processing of the intent. Reasoning summaries generated during tool identification are also stored in the unstructured database.

After the tool identification process is complete, all the information about the intent is sent to the LLM for IRM generation. The LLM uses available details, such as identified tasks, rollback tasks, tools, and arguments, to generate the IRM. Once the IRM is generated, it is passed to the Task Deployment Module. This marks the completion of the intent processing workflow.

4.2.3 Task Deployment Module

The Task Deployment Module acts as the execution layer of the system and is responsible for carrying out the tasks defined in the IRM. Its primary role is to transform the structured intent representation into an execution sequence and execute the tasks while ensuring that task dependencies are respected during execution. The module receives the IRM as input and converts it into an in-memory graph representation. This graph structure allows the system to model the relationships between tasks, including execution dependencies and rollback mappings.

The module is responsible for execution coordination. Based on the dependency graph, the module generates an execution queue containing an ordered list of tasks. This queue serves as the execution plan used for deploying the intent. The module also manages execution state tracking. During task execution, the status of tasks is updated in the graph. After deployment, the intent status is also updated in the graph.

Figure 4.3 depicts the workflow of the module. The workflow starts after an IRM becomes available for deployment. The module converts it into an in-memory graph. Once the graph is constructed, the module analyzes the dependency links between tasks and generates an execution queue based on that. The queue contains the tasks to be executed, along with the tools and arguments associated with each task.

After the execution queue is prepared, it is shown to the user for approval through the User Interaction Module. If the user rejects the execution queue, the processing of the intent is stopped, and the intent is discarded. If the user approves the execution queue, the module begins processing tasks sequentially. The tool corresponding to each task is invoked with the associated arguments. Once execution completes, the module updates the task status in the graph and adds the task to a completed tasks queue before moving to the next task in the execution queue.

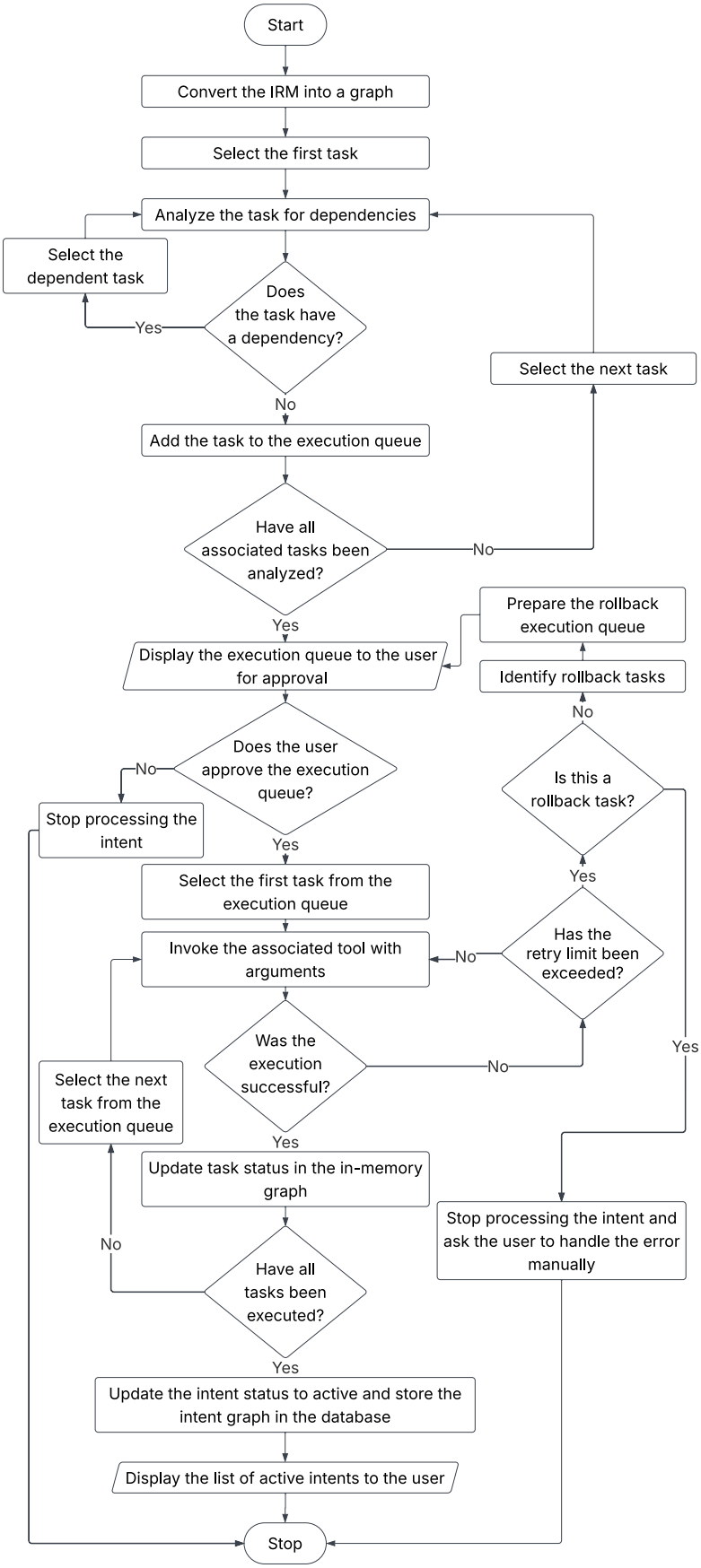


Figure 4.3: Task Deployment Workflow

The execution cycle continues until all tasks associated with the intent have completed successfully or the retry limit is reached. When all associated tasks are executed successfully, the intent status is updated to `active` in the graph. The graph is then stored in the graph database. The updated intent status is shown to the user through the User Interaction Module. Finally, the completed tasks queue is cleared, which marks the successful completion of the deployment workflow for the given intent.

A separate workflow is triggered when the task execution fails. If a task is not completed successfully, the module retries the task according to the retry limit configured during the system implementation. When the retry limit is exceeded, the module identifies the rollback tasks associated with all tasks that have already been executed and prepares a new execution queue for the rollback tasks. The system then informs the user of the failure and related error details. It also shows the rollback task details for the approval. The system notifies the user if any of the deployed tasks are irreversible and asks whether the user still wants to proceed with the planned rollback tasks. Once approved, the rollback tasks are executed to revert the changes introduced by previously executed tasks associated with the intent.

The user can initiate the deletion of an active intent through the User Interaction Module. The User Interaction Module then forwards the intent identifier to the deletion workflow for processing. The deletion workflow begins when it receives the identifier of the intent that must be removed. The Task Deployment Module then uses the Logger and Retriever Module to query the graph database and retrieve all the sub-intents, tasks, and rollback-related information associated with the specified intent. This retrieval stage provides the details required for the deletion planning.

Once the relevant entities are retrieved, the system performs a dependency analysis on the related tasks to determine the correct execution order. During this process, the rollback task associated with each task is added to the execution queue

together with the required tool details and arguments. After the dependency analysis is completed, the system reverses the execution queue. This ensures that the rollback task related to the most recently executed task is performed first during deletion. Following this step, the system presents the complete list of tasks planned as part of the intent deletion for user approval. The deletion tasks are executed only after the user approves them. These tasks are carried out sequentially to ensure that dependency relationships remain valid throughout the deletion process. When all rollback tasks are completed successfully, the status of the corresponding intent in the graph database is updated to `deleted`. This status change indicates that the deletion workflow is complete.

If any rollback or deletion task fails during execution, the system immediately stops further processing and notifies the user of the failure. In such cases, the user is asked to handle the situation manually. Although this approach requires manual intervention, it prevents the system from continuing in an inconsistent state.

4.2.4 Logger and Retriever Module

The Logger and Retriever Module is responsible for managing data storage, retrieval, and audit logging across the system. It acts as a central interface through which other system modules access the available databases. By centralizing data storage, the module ensures consistency in data management across the system. The details of the databases are described in Section 4.2.5.

A key responsibility of the module is to determine how an incoming data-storage or data-retrieval request should be processed. It uses a rule-based mechanism that evaluates each request and selects the most suitable database. The selection is based on factors such as the source of the request and the type of data involved. This approach allows different types of data to be stored in and retrieved from the appropriate databases through a single module. Another important function

of the module is the abstraction of database operations. It provides the necessary implementations for interacting with the different databases used by the system while hiding the underlying technical details.

The module also supports the use of multiple database technologies within the same architecture. Because all database interactions are handled through the module, additional storage backends can be integrated without requiring changes to other system modules. This improves flexibility and simplifies future system extensions.

The workflow begins whenever a module requires either storage or retrieval of information during its execution. The module with a storage or retrieval requirement invokes the Logger and Retriever Module along with the relevant data and operation details. Subsequently, the rule-based mechanism within the module processes the parameters received during the invocation to determine the correct database. It takes into account the invoking module and the type of data while deciding the database to use. Once the correct database is identified, the module invokes the corresponding storage or retrieval function associated with that database. These functions perform the actual read or write operations and return the results to the Logger and Retriever Module. After completing the storage or retrieval operation, the Logger and Retriever Module returns the output to the module that requested the operation so that the system workflow can continue.

4.2.5 Databases

The system uses multiple databases to manage different categories of data, such as operational, reasoning, and contextual data. Each database associated based on the type of data it handles. This separation helps improve scalability and maintainability by allowing each database to be optimized for a specific type of data.

A graph database is used to store intents, tasks, tools, and the relationships between them. These elements are represented as nodes, each with its own at-

tributes. The relationships and dependencies between these elements are stored as edges connecting the nodes. This graph-based representation makes it easier to perform dependency analysis and task sequencing. In addition to the relationships, the graph database also stores the current status of the intents, enabling state tracking.

An unstructured database is used to store the reasoning summaries generated during LLM-assisted operations. The stored reasoning summaries are primarily used for auditing, allowing system decisions to be reviewed and analyzed when required. Storing reasoning summaries in a dedicated database ensures that unstructured data is managed independently from other intent data. The database is also used to store the logging data associated with the User Interaction Module.

A vector database is used to support context enrichment that is part of the Intent Processing Module. The LLM context is enriched with few-shot examples, tool documentation, and domain-specific knowledge during reasoning tasks. Input documents with the required data are first converted into vector embeddings, and these embeddings are then stored in the vector database. During retrieval, relevant embeddings are identified, and related text is passed to the LLM to provide additional contextual information. This improves the relevance and consistency of responses generated by LLMs.

4.3 System Implementation

We have built a proof-of-concept prototype of the intent-based security system. We have deliberately narrowed the scope to only handle security intents that are related to file-system access control. This limited scope has allowed us to test the main ideas of the system without building functions that are not necessary for the initial evaluation.

4.3.1 Implementation Strategy and Technology Stack

We relied on AI-assisted methods to build the prototype, with Cursor serving as our main development environment. We used metaprompting to develop individual modules and gave Cursor direct prompts for smaller code changes. Although Cursor supported the coding process, it did not control the main development decisions. Every architectural choice and every detail of the technology stack was decided by us. These details were included explicitly in the instructions provided during each stage of development. Therefore, the AI tools acted as coding assistants rather than independent system designers.

The UI has been kept intentionally simple because the system is a proof-of-concept prototype. On the frontend, the interface was built with plain HTML and CSS, with vanilla JavaScript handling the dynamic parts of the interaction. Frameworks such as React have been considered but set aside in favor of a simpler setup. The interface communicates with a set of session-scoped API endpoints on the backend, which drive the step-by-step nature of the user workflow, moving the user from one stage of intent processing to the next.

The backend has been implemented in Python, using the Django framework. The choice is grounded less in a technical comparison between frameworks and more in familiarity. Our prior experience with Python has made Django a natural starting point. Because parts of the workflow need to run asynchronously, we used asyncio and httpx to support them. The application runs on the Uvicorn ASGI server, which supports asynchronous behavior. Graph-related logic in the system relies on the NetworkX library for in-memory graph creation and manipulation. For task execution, we chose to keep things simple by relying on Python's native asyncio library rather than adding a workflow engine such as Celery to the stack. The IRM is represented using Pydantic v2 models, as they support the validation of the JSON structures returned by the LLM. In total, the backend exposes 10 API endpoints,

and their details are listed in Table 4.1. We have designed the UI such that IRM approval and execution queue approval can be carried out as a single step instead of two separate steps. This has been done to improve the user experience. A few helper components have also been developed to support file-system access control. One of them is responsible for enriching prompts sent to the LLM with metadata about the file path and the user. Another carries out a deterministic check of whether the access being requested is already granted or denied, based on existing permission bits, ACLs, and group membership. The metadata and the result of the deterministic checks become part of the information used in the later stages of the workflow.

For the LLM-related workflows, we evaluated two LLMs. Initially, we used a locally hosted gemma4 model. Although the model produced usable outputs, inference was slow because of the limitations of our hardware setup. We therefore decided to use Google as an external model provider and used the gemini-3.1-flash-lite model. Google was selected because Google AI Studio provided access to a free-tier API. In both cases, we set the model temperature to 0.1 and the top-p value to 0.9. The prompts used in the system are provided in Appendix A.

We chose Neo4j, MongoDB, and ChromaDB as the databases for the system. For the graph database, we selected Neo4j over alternatives such as TigerGraph and ArangoDB, mainly because of its native graph support and because it is operationally simpler to run than the others considered. MongoDB was chosen as the unstructured database. Even though Elasticsearch is a potential candidate for storing unstructured data, the prototype does not require a dedicated full-text indexing process and the overhead that comes with it. For the vector database, ChromaDB was selected over options such as Milvus and Qdrant because it is comparatively light on resources and suits prototyping. All three databases run as Docker containers.

Table 4.1: API Endpoints

Method	Endpoint	Input	Description
POST	`/api/intents/start/`	`raw_intent` and `user_id` in the request body.	Creates a new intent-processing session and starts intent analysis.
GET	`/api/intents/{session_id}/status/`	The session identifier is provided as a path parameter.	Returns the current workflow state. The response may include clarification questions, the generated IRM, execution information, or rollback details.
POST	`/api/intents/{session_id}/clarify/`	`user_response` in the request body.	Records the user's clarification response and continues intent analysis using the updated conversation.
POST	`/api/intents/{session_id}/approve-irm/`	The boolean field `approved` and optional `feedback` in the request body.	Approves or rejects the generated IRM.
POST	`/api/intents/{session_id}/approve-rollback/`	The boolean field `approved` in the request body.	Approves or rejects a rollback plan created after task execution fails.
GET	`/api/intents/active/`	No input is required.	Returns active intents, including their identifiers, original intent text, action, owner, and creation time.
GET	`/api/intents/{intent_id}/logs/`	The intent identifier is provided as a path parameter.	Returns execution logs, including task identifiers, tool identifiers, arguments, outputs, errors, and execution status.
GET	`/api/intents/{intent_id}/audit/`	The intent identifier is provided as a path parameter.	Returns audit information, including sub-intents, reasoning summaries, user interactions, and execution logs.
DELETE	`/api/intents/{intent_id}/`	The intent identifier is provided as a path parameter.	Starts the intent-deletion workflow.
POST	`/api/intents/{intent_id}/confirm-delete/`	`interaction_id` and the boolean field `approved` in the request body.	Confirms or cancels a proposed deletion and performs any required rollback operations.

4.3.2 Implemented Features

Capture Intent

The system captures security intent through its UI. Users describe what they want to achieve in natural language, using the text box in the UI. A screenshot of the intent-capturing UI can be seen in Figure 4.4.

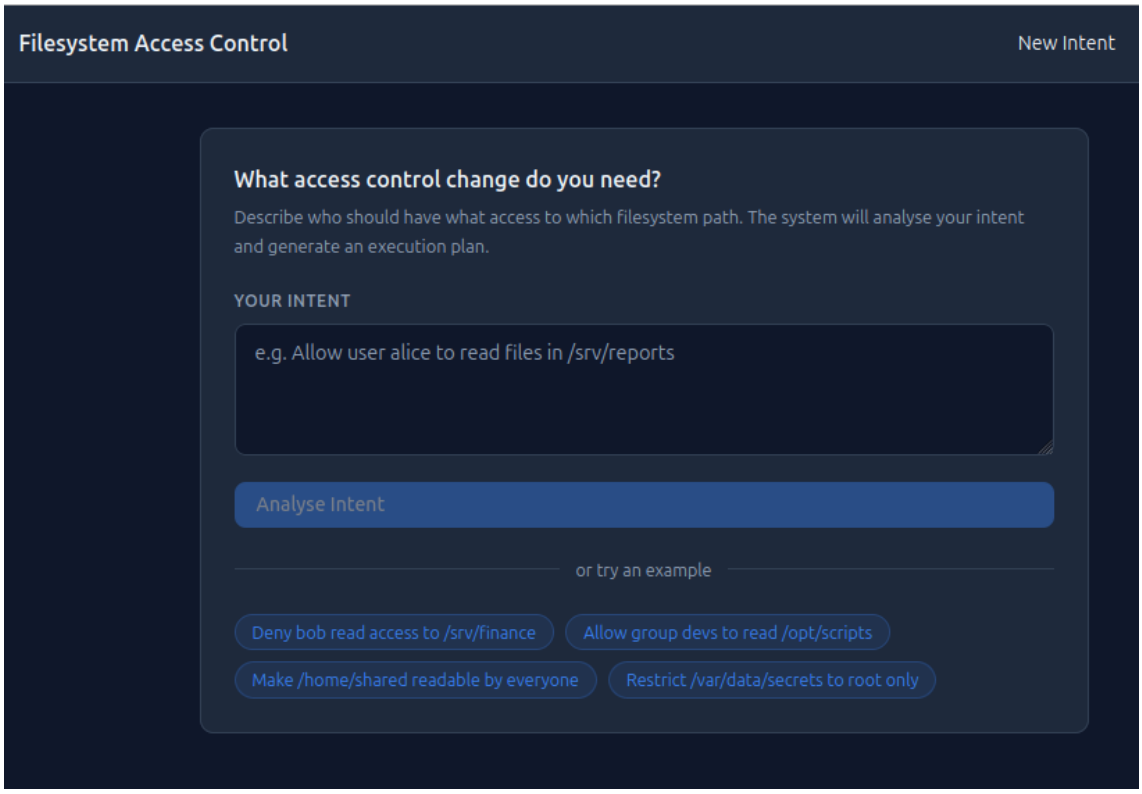


Figure 4.4: Capture Intent

Clarify Intent

Natural-language intent can sometimes be ambiguous. In these cases, the system asks the user for clarification through the UI before processing continues, as shown in Figure 4.5. The system only proceeds with intent processing once it has determined that the intent is clear.

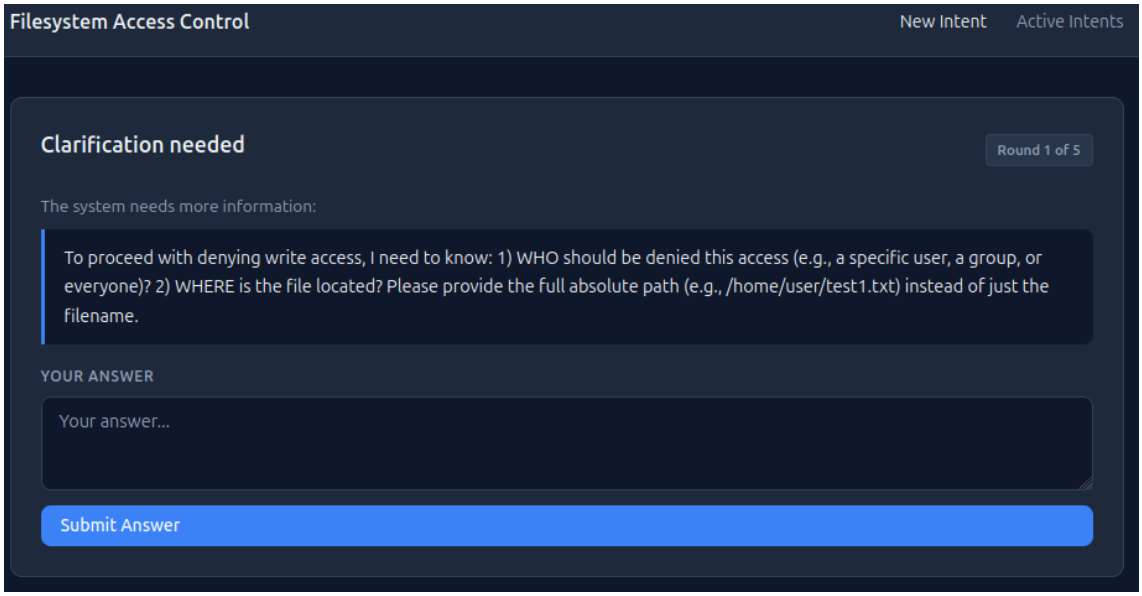


Figure 4.5: Clarify Intent

Review and Approve Tasks

After the system has understood the intent, it plans the tasks needed to fulfill it. The user must then review and approve these tasks. To support this, the UI displays the tasks along with the IRM. The user sees a summary of the tasks, as shown in Figure 4.6, as well as the complete IRM, as shown in Figure 4.7.

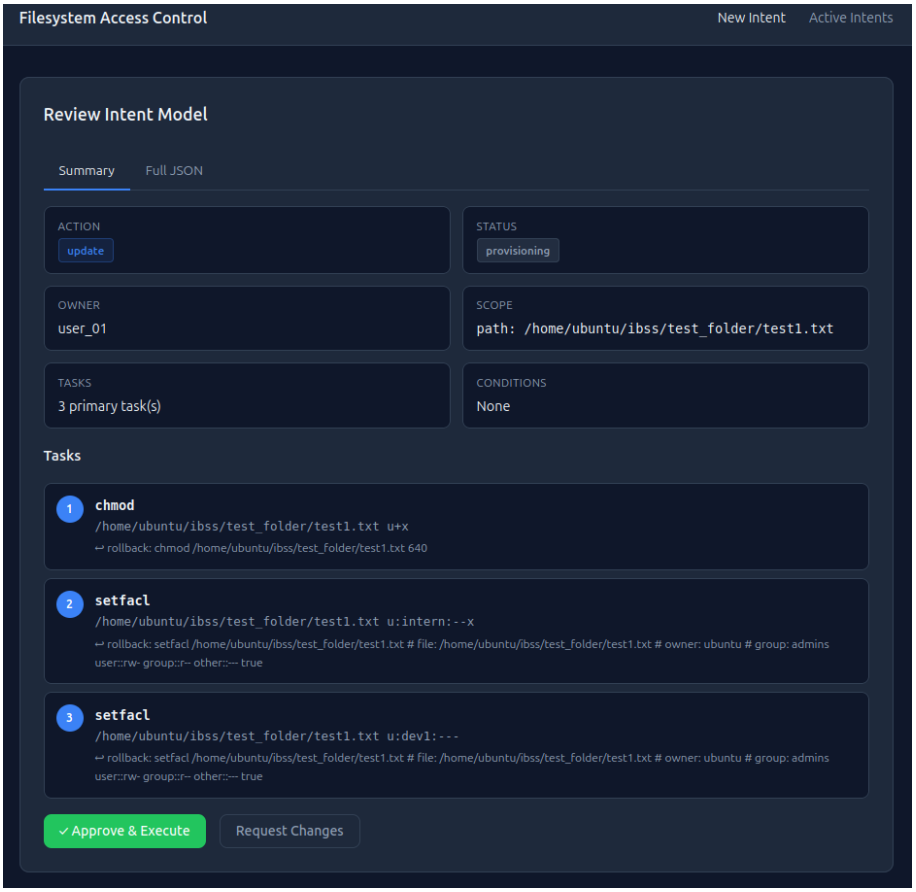


Figure 4.6: Task Summary

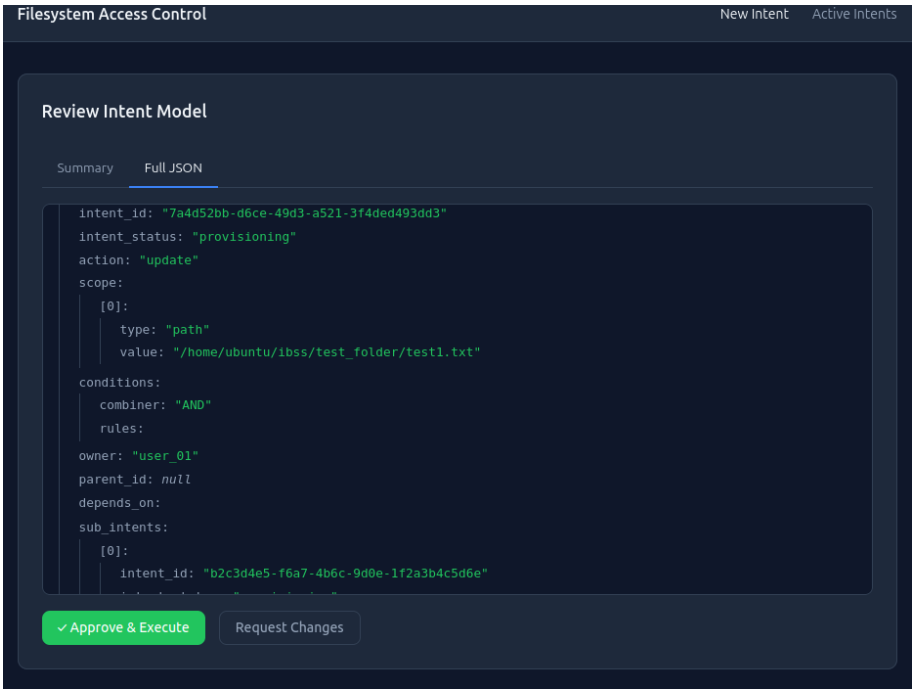


Figure 4.7: Complete IRM

Request Changes

After reviewing the tasks and the IRM, the user can request changes through the text box shown in Figure 4.8, describing the required change in natural language. The system then applies the requested changes and returns an updated IRM, as shown in Figure 4.9.

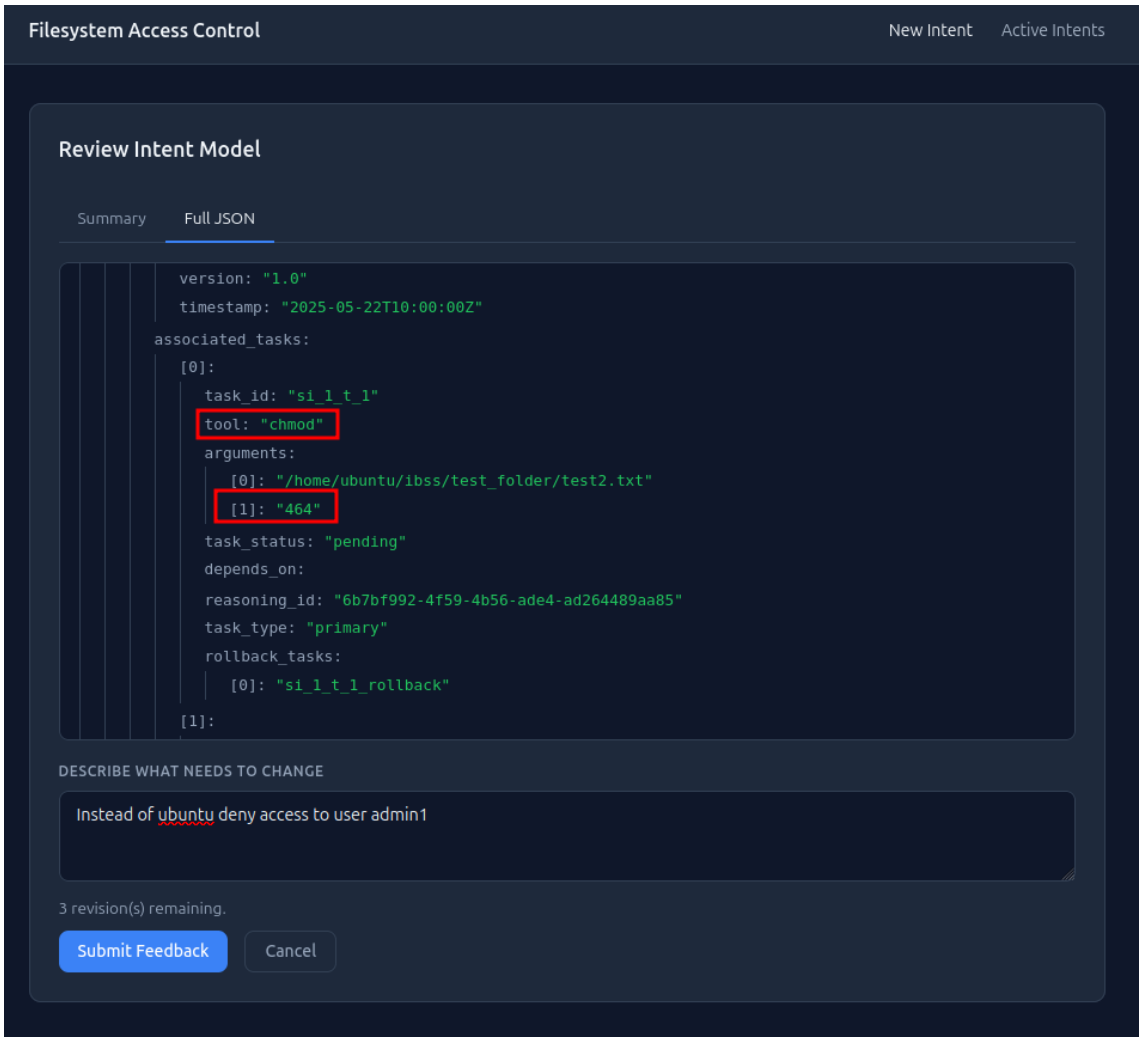


Figure 4.8: IRM Before Requested Changes

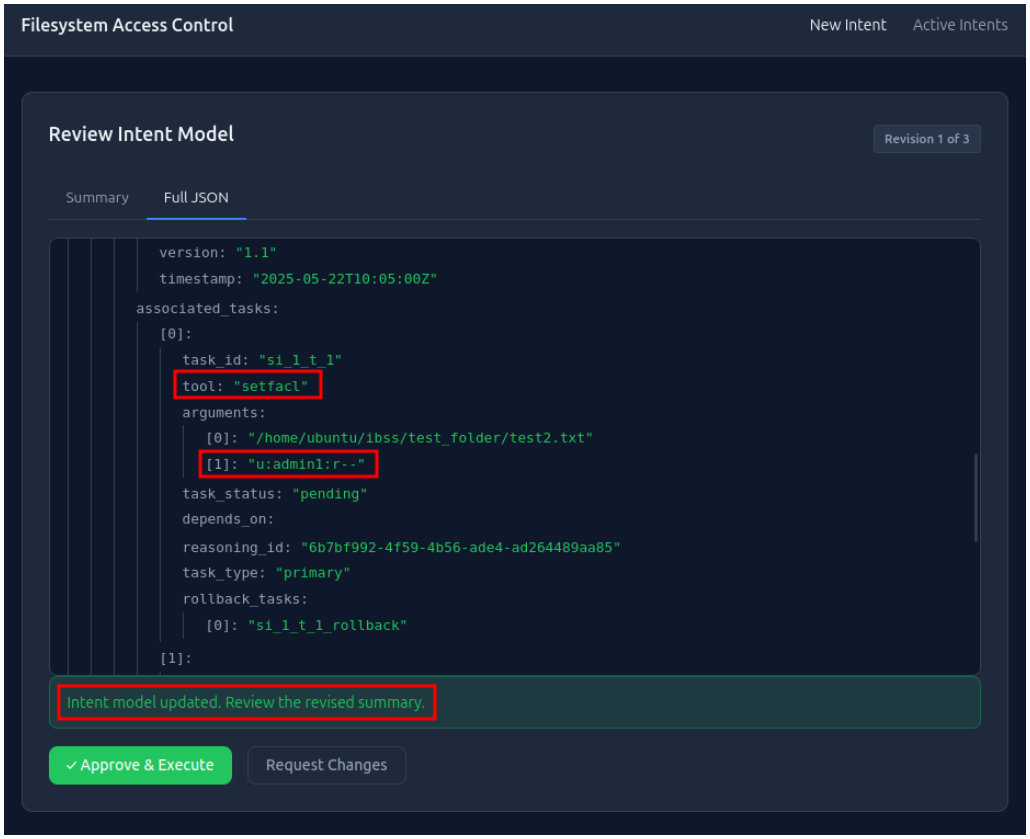


Figure 4.9: IRM After Requested Changes

List Active Intents

Once all tasks associated with an intent have executed successfully, the intent moves into an active state. The Active Intents screen lists all active intents, as shown in Figure 4.10.

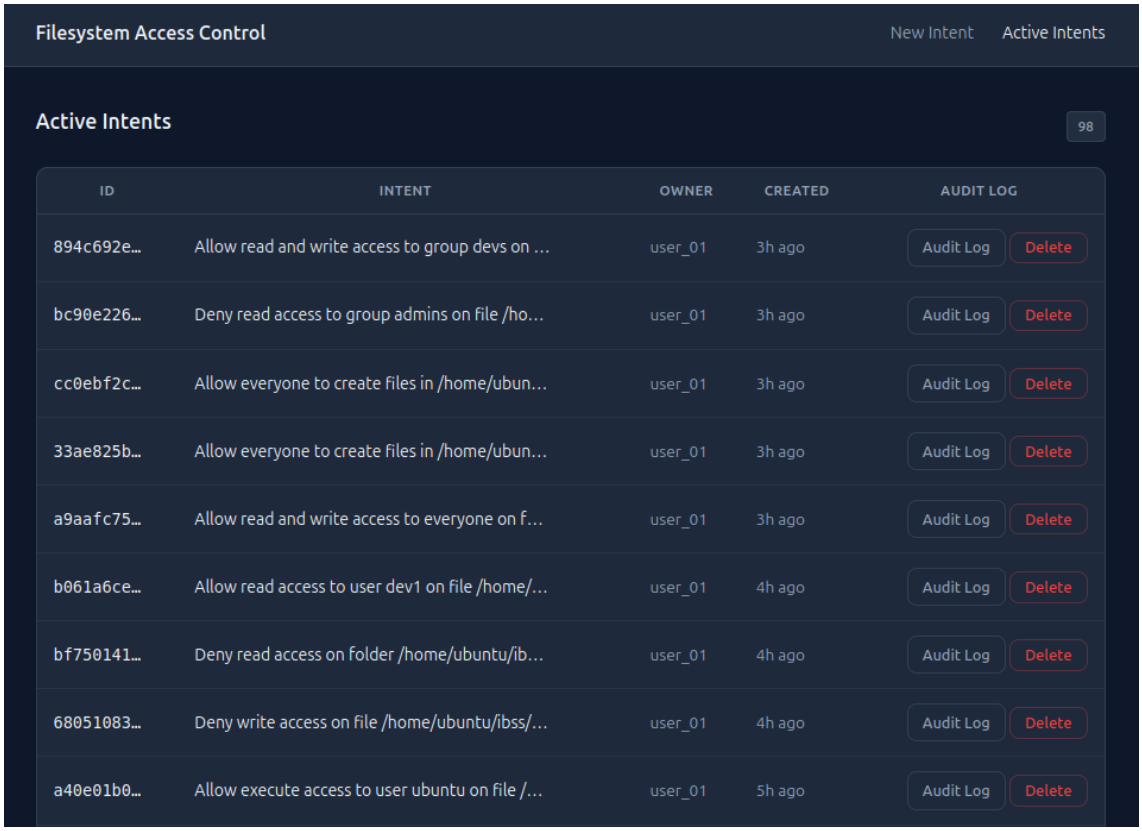
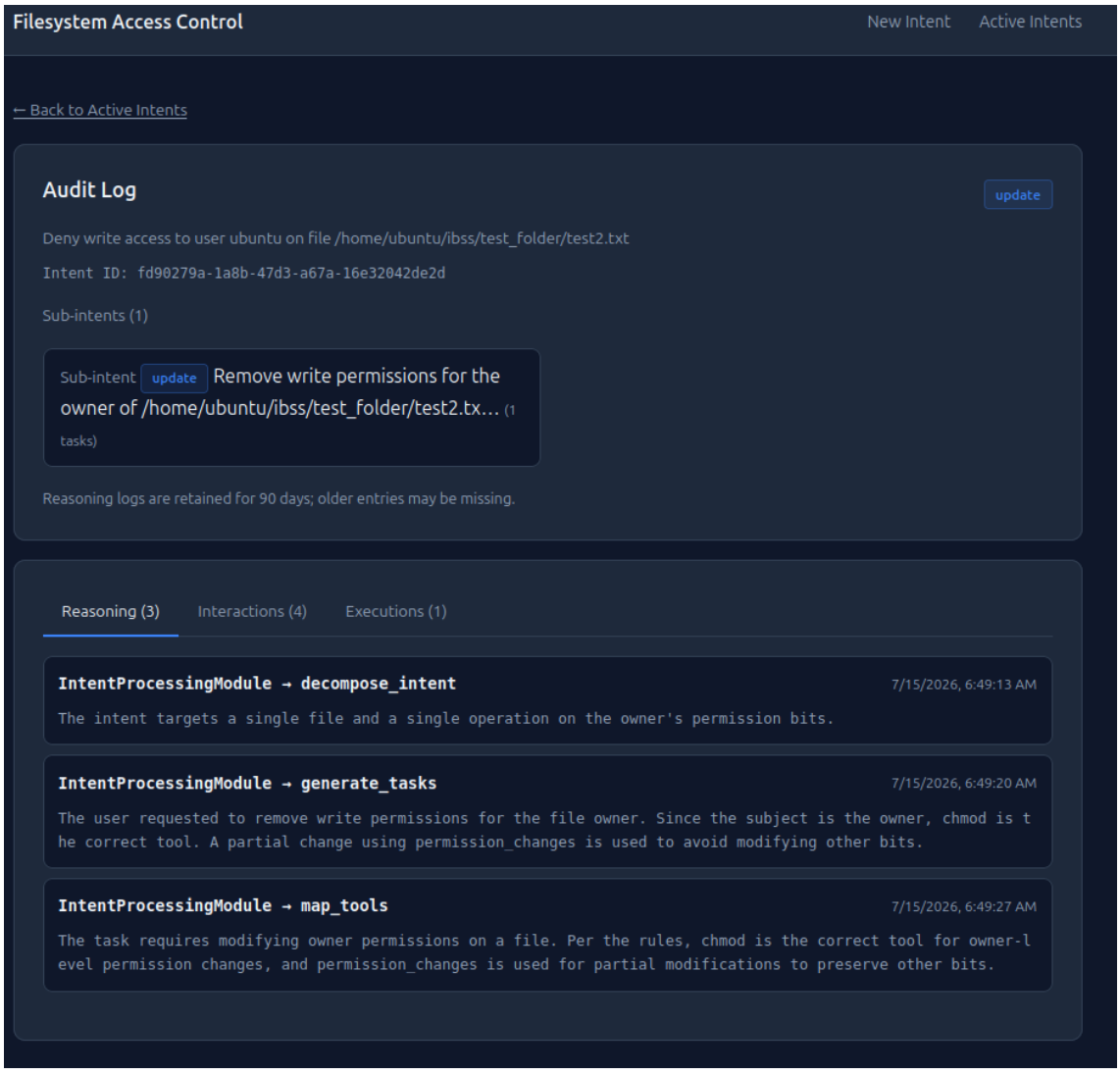


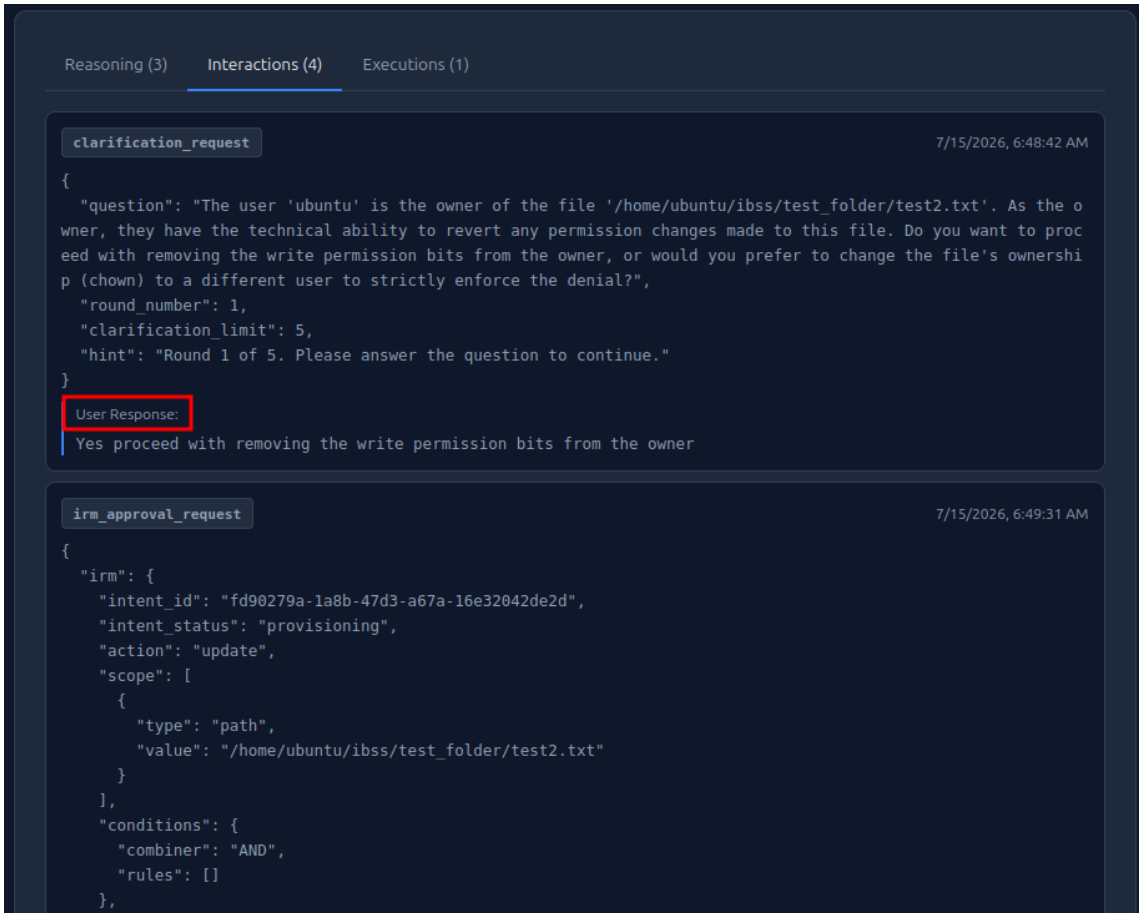
Figure 4.10: Active Intents List

View Audit Log

Users can view the audit logs associated with each intent, as shown in Figure 4.11. The audit log screen presents three types of logs. The first is the reasoning log, which records the reasoning summaries behind each LLM decision. The second is the interaction log, which records user interactions at different stages of intent processing, such as approving the planned tasks or requesting changes to the IRM. The third is the execution log, which shows the tools and arguments used for each task.



(a) Reasoning Log



(b) Interaction Log

(c) Execution Log

Figure 4.11: Audit Log

Failure Handling

When a task associated with an intent fails to execute after multiple retries, the system initiates failure handling. During this process, the system executes rollback tasks for each task that has already been completed successfully. Figure 4.12 shows a screenshot of failure handling for an intent.

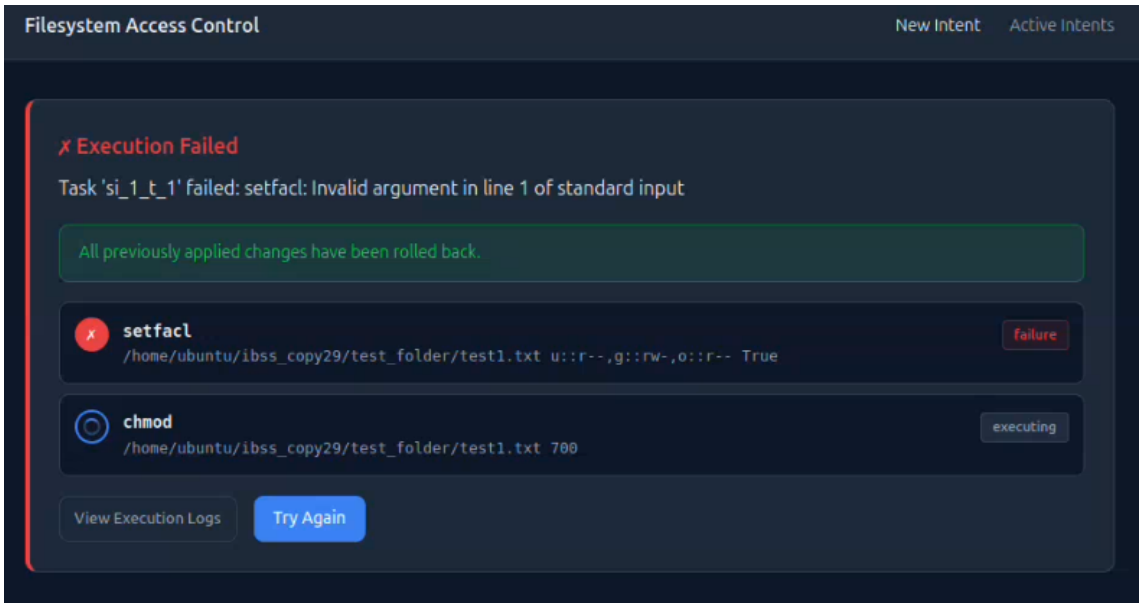


Figure 4.12: Failure Handling

Delete Intent

Users can delete an active intent using the delete button on the Active Intents screen. A pop-up window then shows the tasks that deletion will involve, as shown in Figure 4.13. Once the user approves, the system executes these tasks and removes the intent from the active intents list.

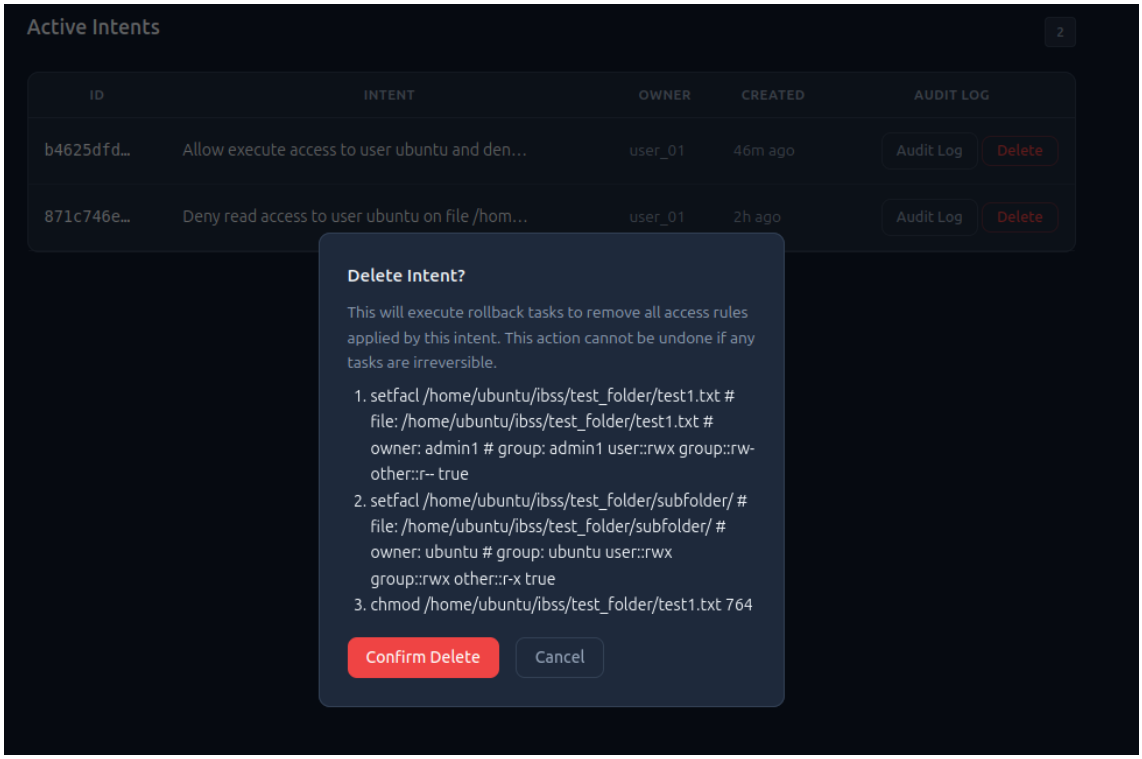


Figure 4.13: Delete Intent

5 System Evaluation

This chapter presents the evaluation of the proof-of-concept prototype. The evaluation uses two methods, which are scenario-based evaluation and empirical evaluation. The scenario-based evaluation includes two representative scenarios whose outcomes are assessed manually. The empirical evaluation covers seven intent categories and includes 70 intents, with 10 intents in each category. The outcome of each intent is assessed manually.

5.1 Scenario-Based Evaluation

This section evaluates the proof-of-concept prototype using two representative scenarios: a simple scenario and a complex scenario. The simple scenario tests an unambiguous intent containing one action, one subject, one object, and no additional conditions. The complex scenario tests an ambiguous intent containing multiple subjects and objects, and requiring clarification and decomposition. The aim of these scenarios is to determine whether the system correctly identifies and executes all tasks required to fulfill each intent.

5.1.1 Simple Scenario

The simple scenario tests whether the system can interpret an unambiguous request, generate the required task, obtain user approval, execute the task, and enforce the requested access restriction. Figure 5.1 shows the initial permission state of the

target file. At the start of the evaluation, the user `ubuntu` had read access to `/home/ubuntu/ibss/test_folder/test1.txt`. This initial state provides a baseline for evaluating the permission change.

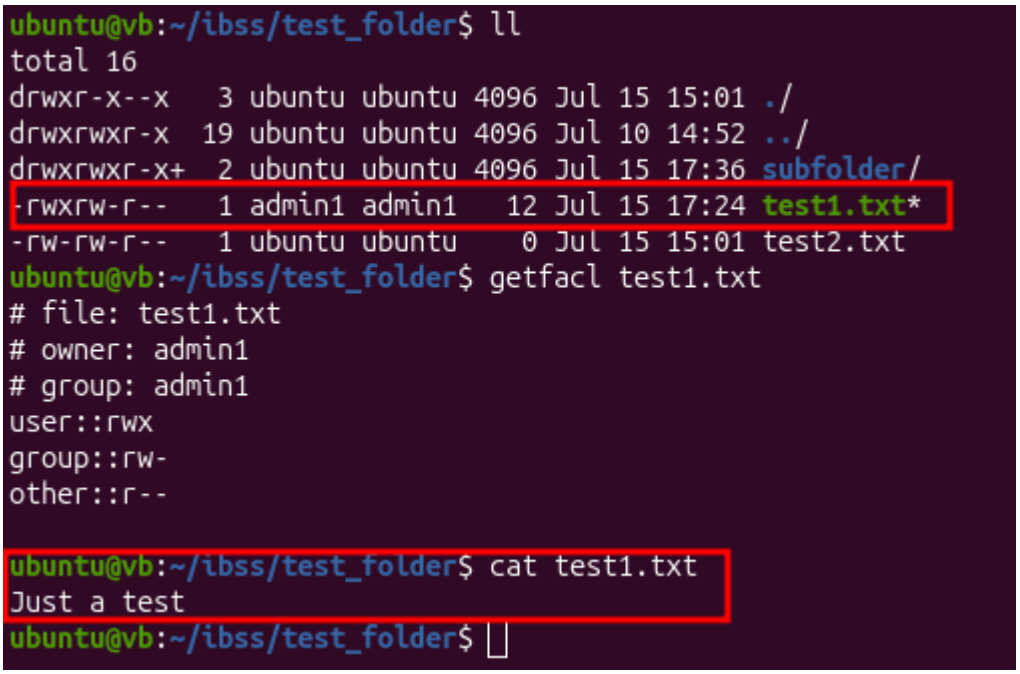


Figure 5.1: File Permissions Before Intent Deployment

The evaluation workflow begins when the user submits a natural-language intent through the UI. Listing 2 shows the submitted intent.

Listing 2 Intent Submitted in the Simple Scenario

Deny read access to user ubuntu on file /home/ubuntu/ibss/test_folder/test1.txt

The intent clearly stated the access-control operation, user, permission, and target file. Therefore, the system did not require any additional clarification.

The system then generated the corresponding IRM and list of tasks required to deploy the intent. It displayed the IRM and the planned tasks through the UI. Figure 5.2 presents the summary of the planned tasks, and Figure 5.3 presents the generated IRM.

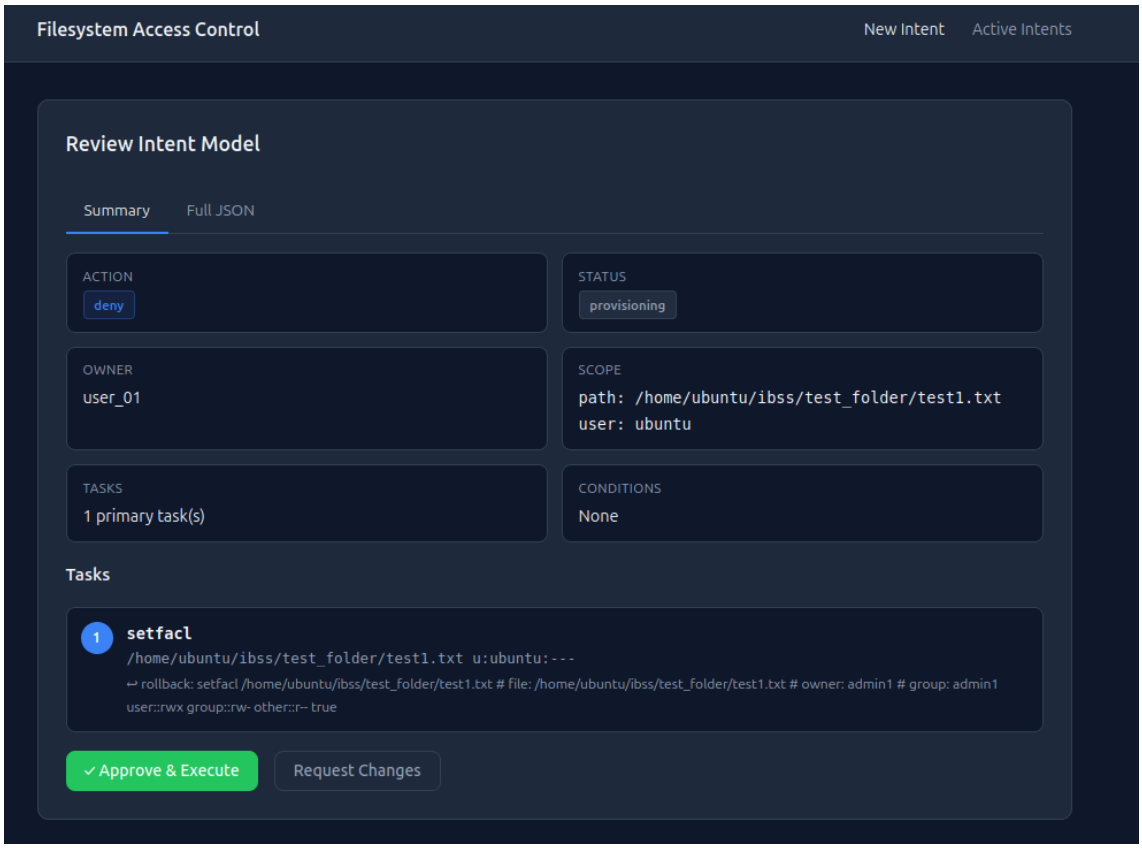


Figure 5.2: Summary of Planned Tasks

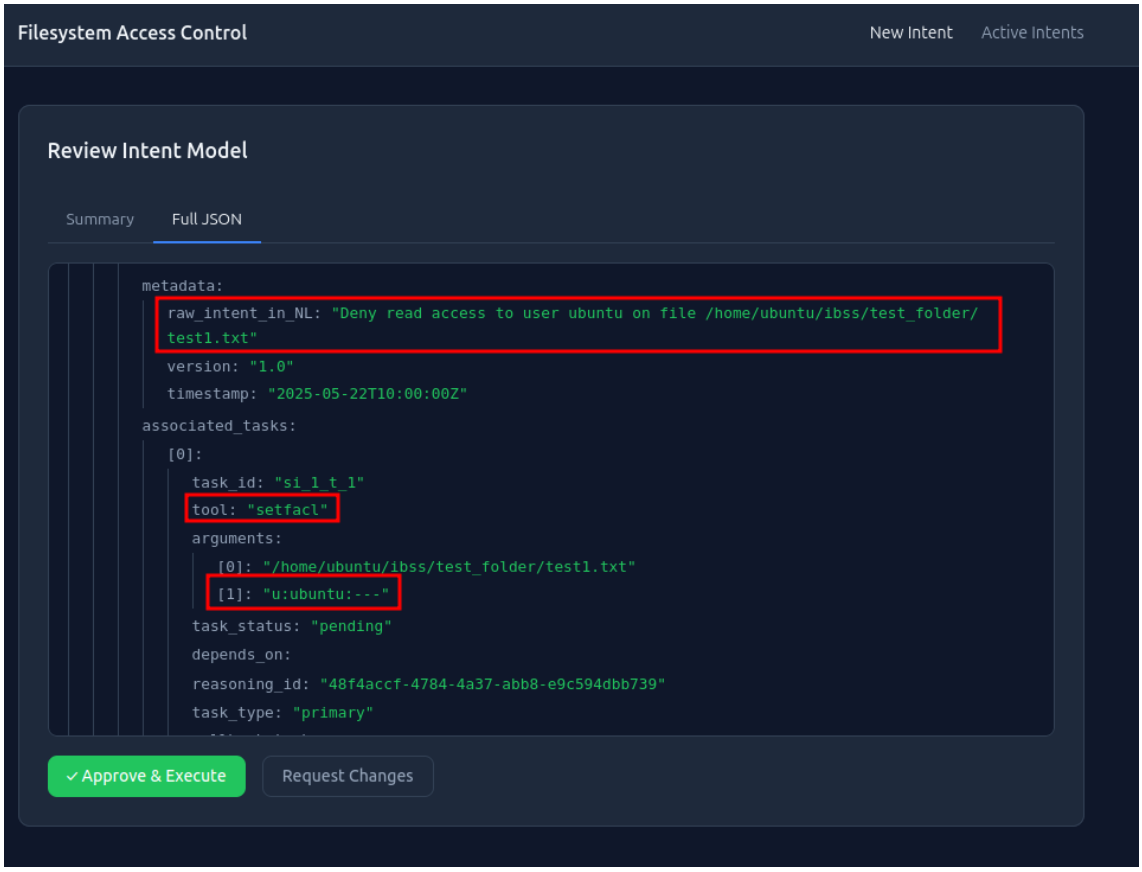


Figure 5.3: Generated IRM

Before changing the file permissions, the system asked the user to review the planned tasks. The user approved the tasks using the `Approve & Execute` button. The system then executed the tasks and displayed their current status. Figure 5.4 shows the execution status.

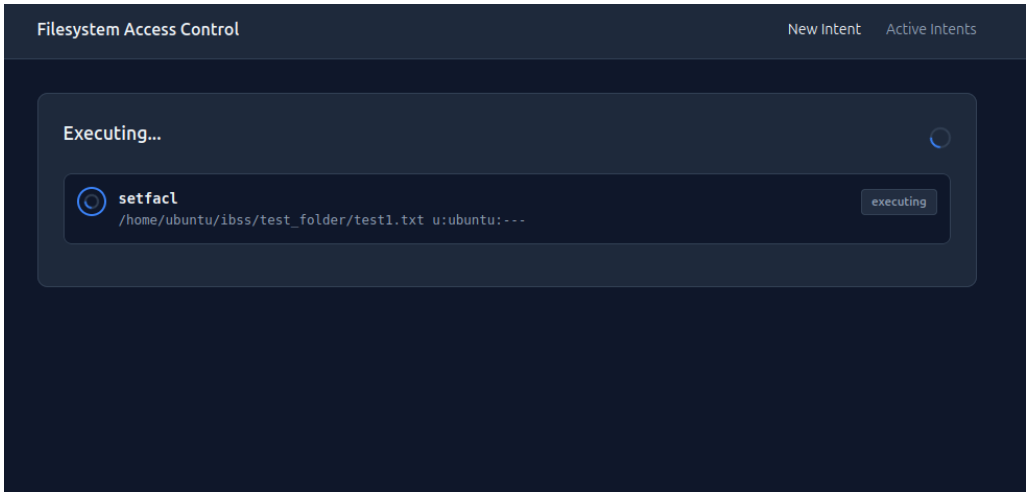


Figure 5.4: Status of Task Execution

After completing the execution, the system displayed a summary of the execution results, as shown in Figure 5.5. The system also added the deployed intent to the list of active intents. Figure 5.6 shows the updated list of active intents.

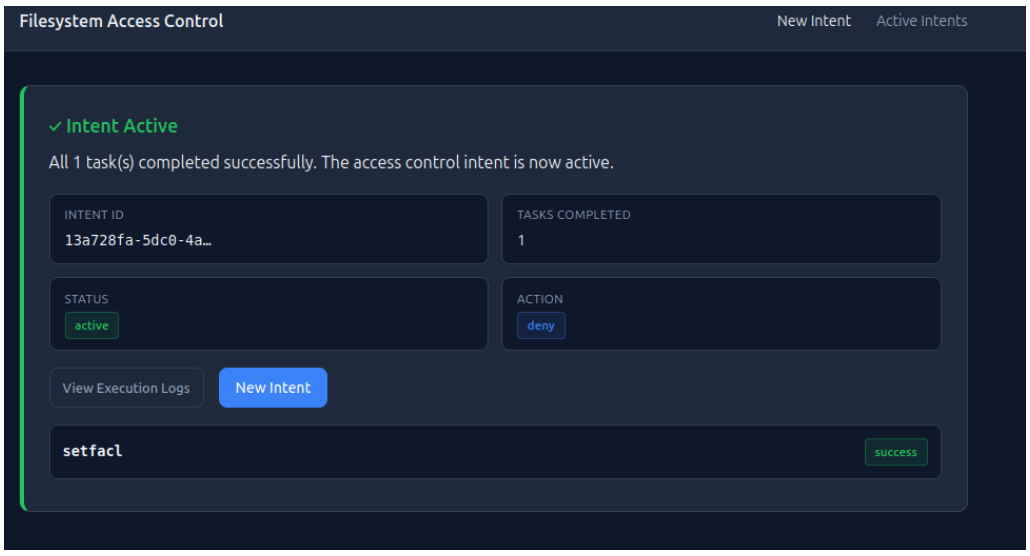


Figure 5.5: Completion of Intent Deployment

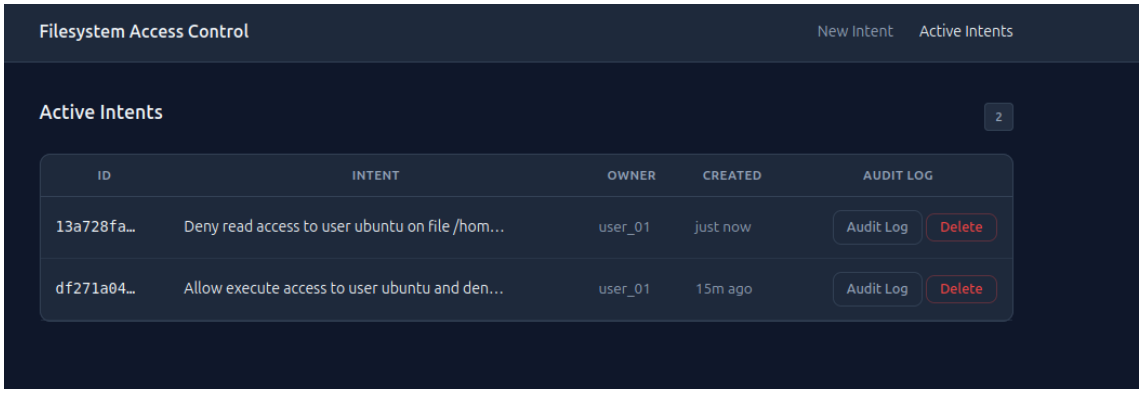


Figure 5.6: List of Active Intents

The final stage tested whether the resulting access-control state matched the submitted intent. Before enforcement, as shown in Figure 5.1, a read attempt performed as the user `ubuntu` succeeded. After enforcement, the same read attempt failed. Figure 5.7 shows the final permission state. The failed read attempt confirms the effective access restriction. The comparison between the initial and final states shows that the system removed read access for `ubuntu` from the specified file. Therefore, the simple scenario evaluation met its expected outcome.

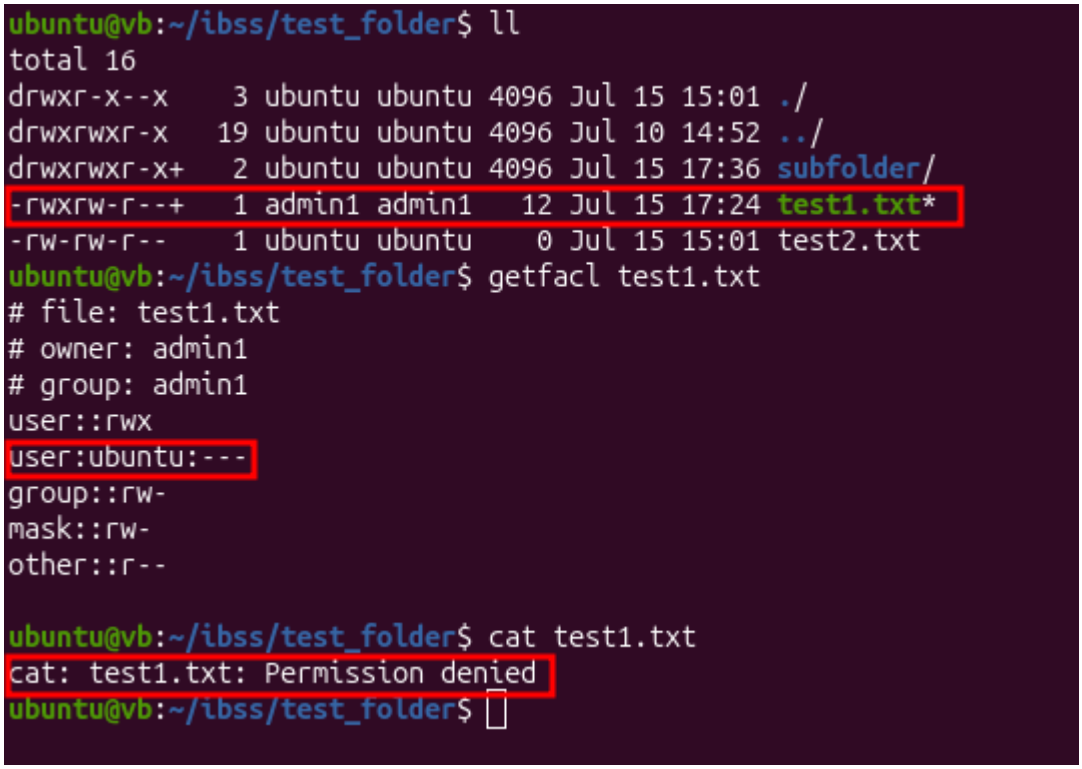


Figure 5.7: File Permissions After Intent Deployment

5.1.2 Complex Scenario

The complex scenario examines how the system handles an intent that is ambiguous, contains more than one subject or object, and requires several tasks. The scenario is designed to determine whether the system can identify missing information, request clarification, divide the request into separate sub-intents, and execute all required tasks. Figure 5.8 presents the initial permission states of the file and directory used in this evaluation. At the start of the evaluation, the user `ubuntu` did not have execute permission on `/home/ubuntu/ibss/test_folder/test1.txt`. The user `admin1`, who owned the file, did have read, write, and execute permissions. The group `devs` did not have write permission on the directory `/home/ubuntu/ibss/test_folder/subfolder`.

```
ubuntu@vb:~/ibss/test_folder$ cat /etc/group | grep devs
devs:x:1004:dev1
ubuntu@vb:~/ibss/test_folder$ ll
total 16
drwxr-x--x  3 ubuntu ubuntu 4096 Jul 15 15:01 ./
drwxrwxr-x 19 ubuntu ubuntu 4096 Jul 10 14:52 ../
drwxrwxr-x  2 ubuntu ubuntu 4096 Jul 15 17:13 subfolder/
-rwxrw-r--  1 admin1 admin1   12 Jul 15 17:21 test1.txt*
-rw-rw-r--  1 ubuntu ubuntu    0 Jul 15 15:01 test2.txt
ubuntu@vb:~/ibss/test_folder$ getfacl subfolder/
# file: subfolder/
# owner: ubuntu
# group: ubuntu
user::rwx
group::rwx
other::r-x

ubuntu@vb:~/ibss/test_folder$ getfacl test1.txt
# file: test1.txt
# owner: admin1
# group: admin1
user::rwx
group::rw-
other::r--

ubuntu@vb:~/ibss/test_folder$ ./test1.txt
bash: ./test1.txt: Permission denied
ubuntu@vb:~/ibss/test_folder$ sudo su admin1
admin1@vb:/home/ubuntu/ibss/test_folder$ echo 'Just a test' > test1.txt
admin1@vb:/home/ubuntu/ibss/test_folder$
exit
ubuntu@vb:~/ibss/test_folder$ sudo su dev1
dev1@vb:/home/ubuntu/ibss/test_folder$ touch subfolder/testfile.txt
touch: cannot touch 'subfolder/testfile.txt': Permission denied
dev1@vb:/home/ubuntu/ibss/test_folder$
exit
ubuntu@vb:~/ibss/test_folder$
```

Figure 5.8: File and Directory Permissions Before Intent Deployment

As in the previous scenario, the evaluation began with the user submitting a natural-language intent through the UI. Listing 3 shows the submitted intent.

Listing 3 Intent Submitted in the Complex Scenario

Allow execute access to user ubuntu and deny write access to user admin1 on file test1.txt, also allow write access to group devs on folder /home/ubuntu/ibss/test_folder/subfolder/

The intent specified the users, the group, the requested permissions, and the corresponding access-control operations. However, the first target is incomplete because the intent provides only the filename `test1.txt`, rather than its full path. The system therefore requested the missing path through the UI, as shown in Figure 5.9. The system also warned the user and asked for confirmation before removing write permission from the file owner, as shown in Figure 5.10, because in a discretionary access-control system, the owner can normally restore the permissions if they want.

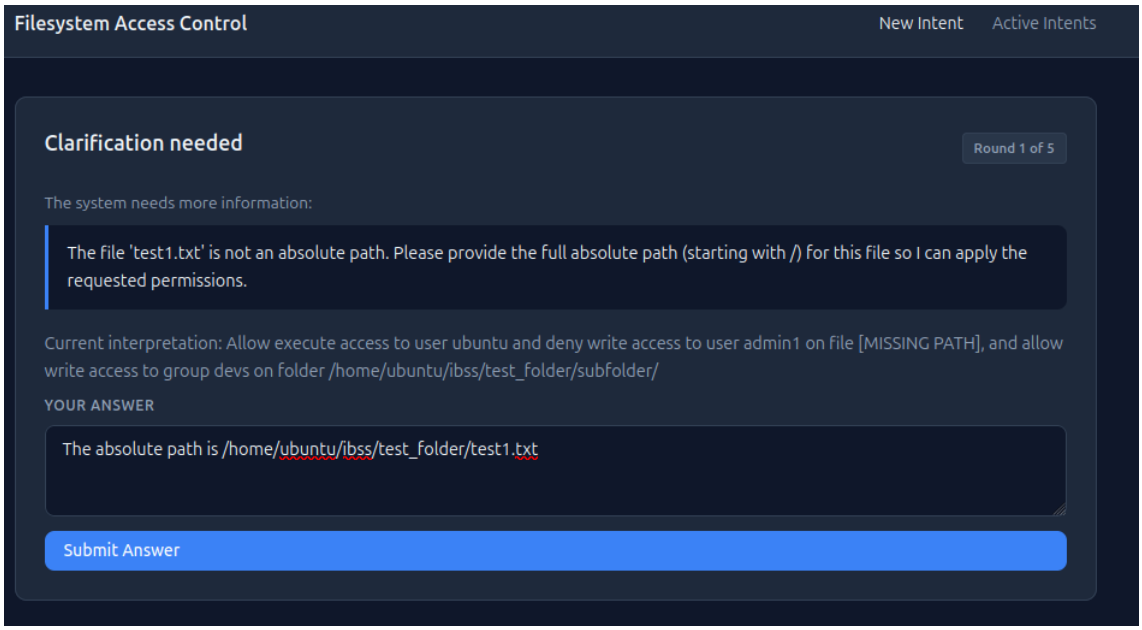


Figure 5.9: Clarification Request Regarding Missing File Path

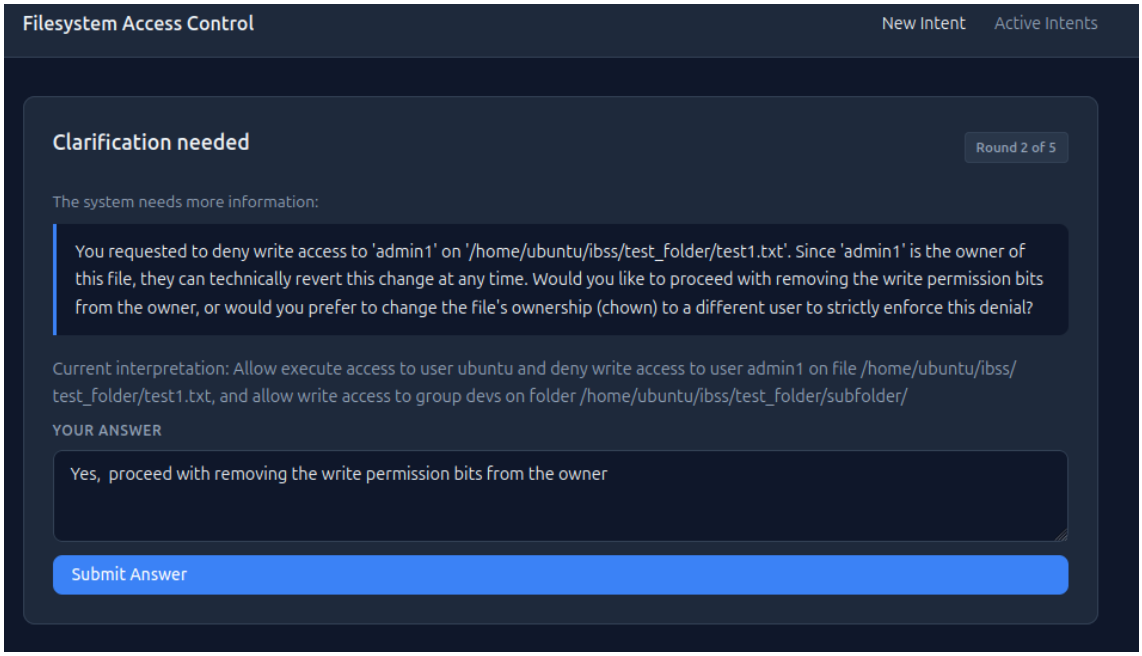


Figure 5.10: Warning Regarding File Ownership

After the user provided the full file path and confirmed that write permission should be denied for the owner, the system updated its interpretation of the intent. The system then generated the IRM along with the required tasks. Figure 5.11 presents the summary of planned tasks, while Figure 5.12 presents the generated IRM. The system divided the request into three sub-intents. The first sub-intent is to grant execute permission to `ubuntu` on the specified file. The second sub-intent is to remove write permission from `admin1` on the same file. The third sub-intent is to grant write permission to the group `devs` on the specified directory. Figure 5.13 shows these three sub-intents.

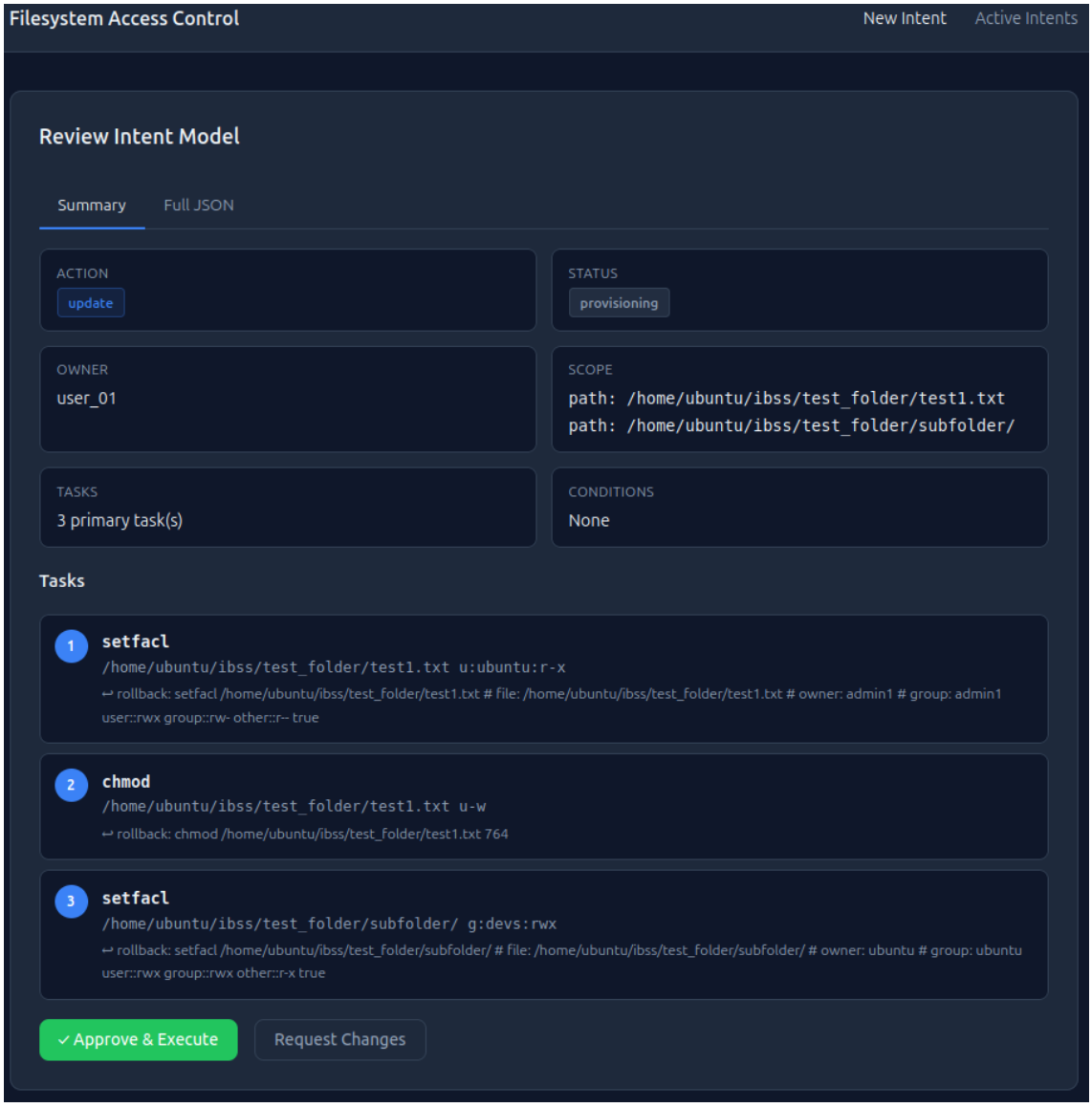


Figure 5.11: Summary of Planned Tasks

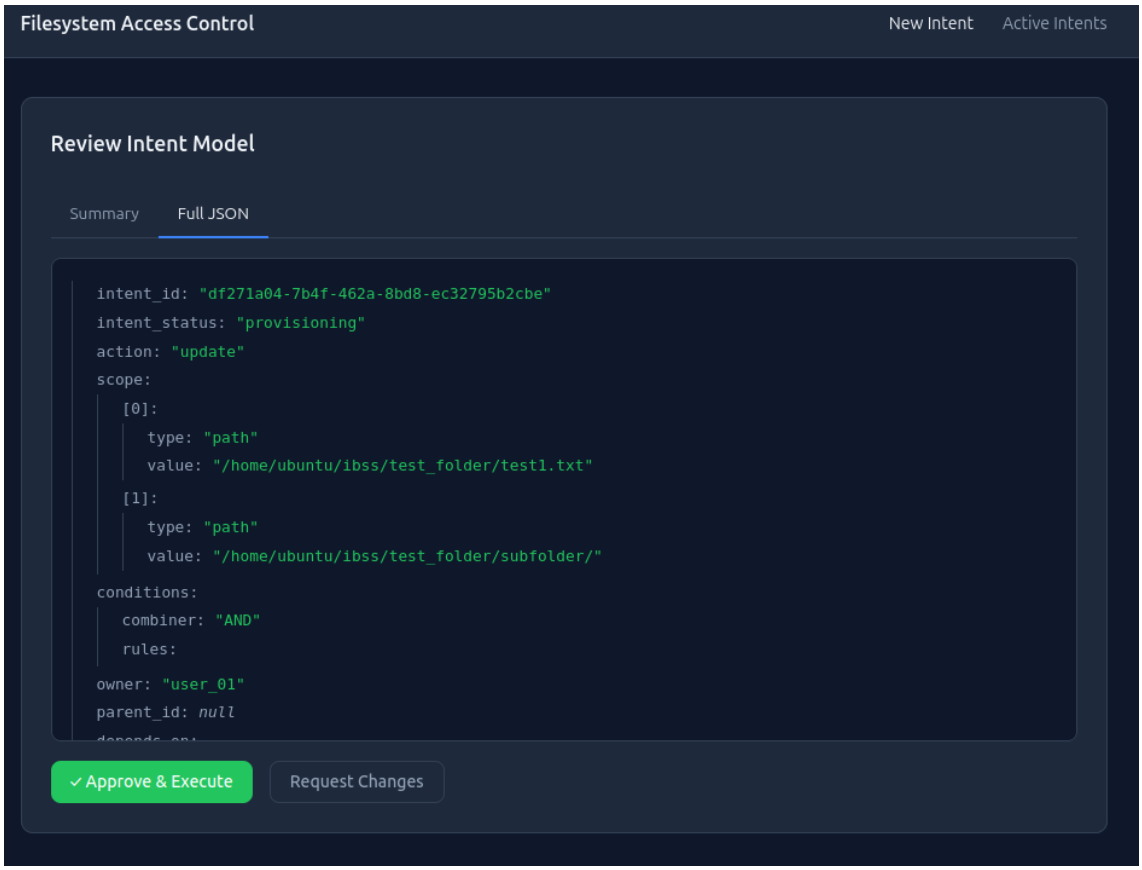
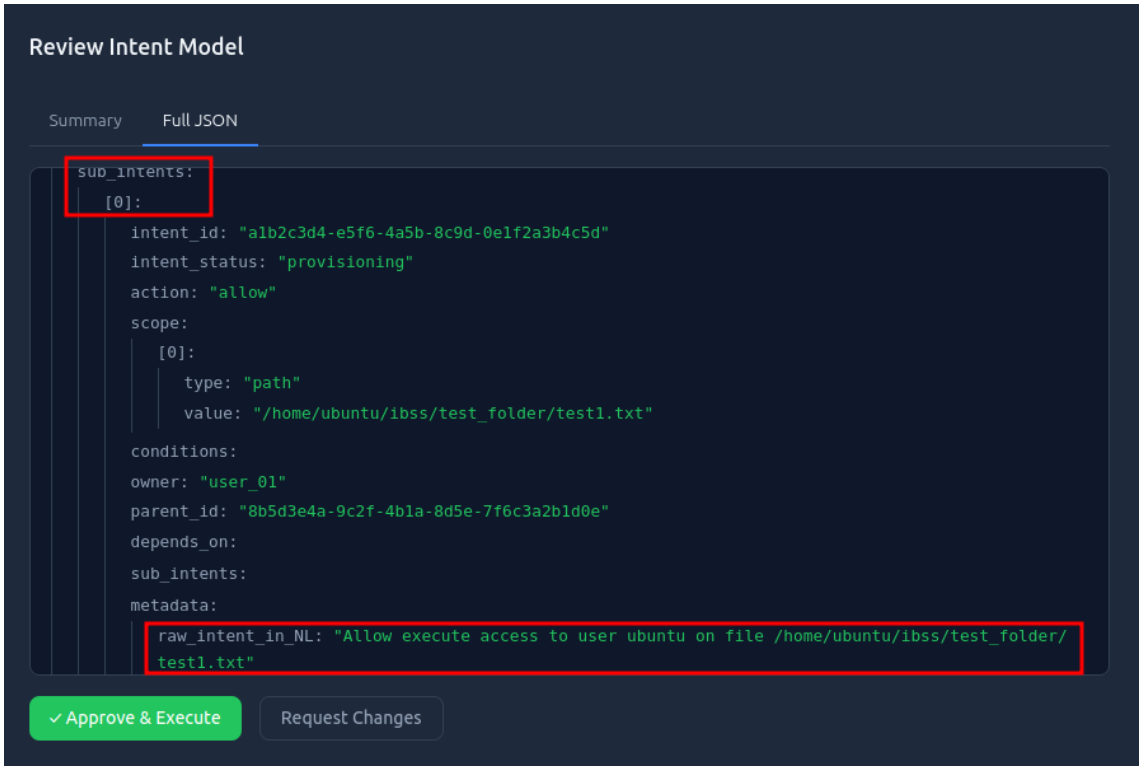
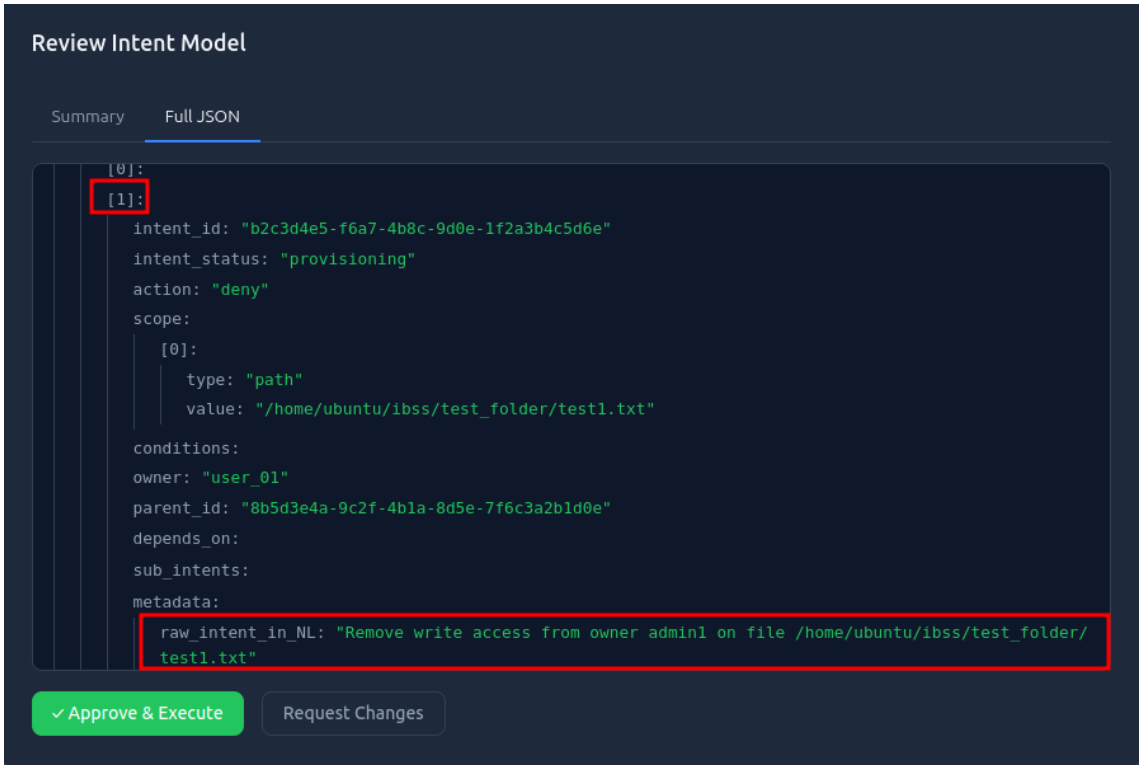


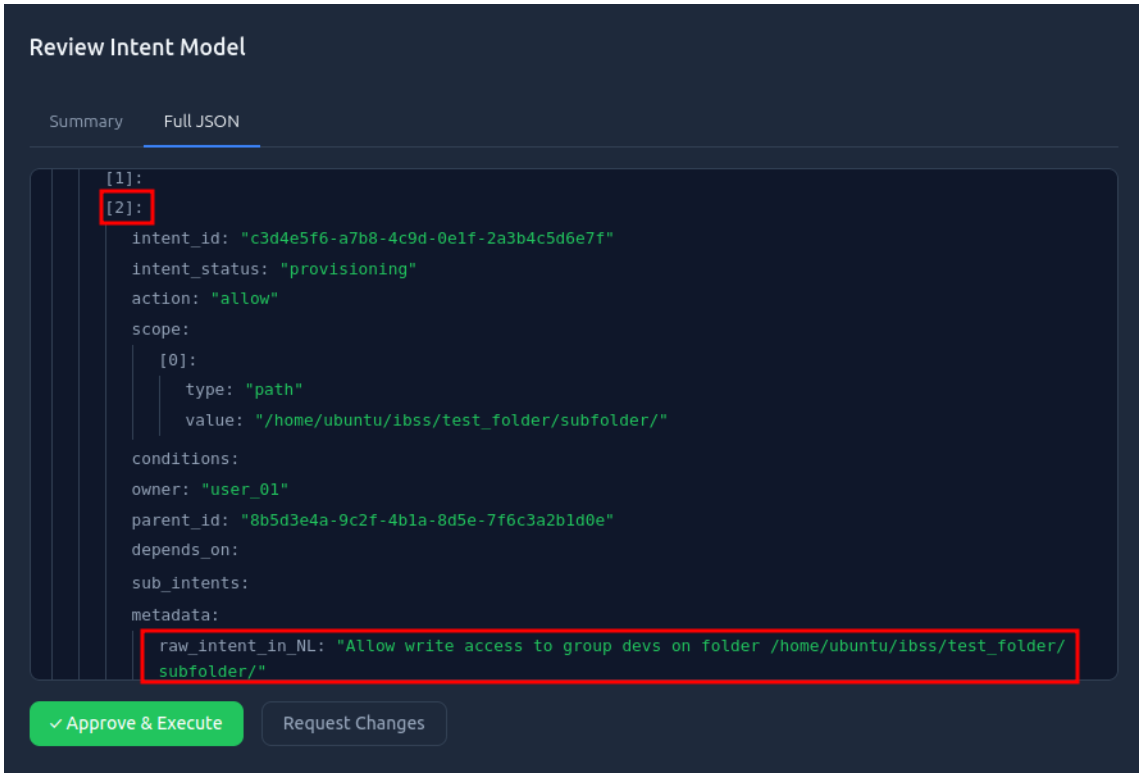
Figure 5.12: Generated IRM



(a) Sub-Intent 1



(b) Sub-Intent 2



(c) Sub-Intent 3

Figure 5.13: Decomposed Sub-Intents in IRM

The user reviewed the planned tasks before their execution. After the user approved the tasks, the system executed each task and displayed its status through the UI. Figure 5.14 shows the execution status of the individual tasks.

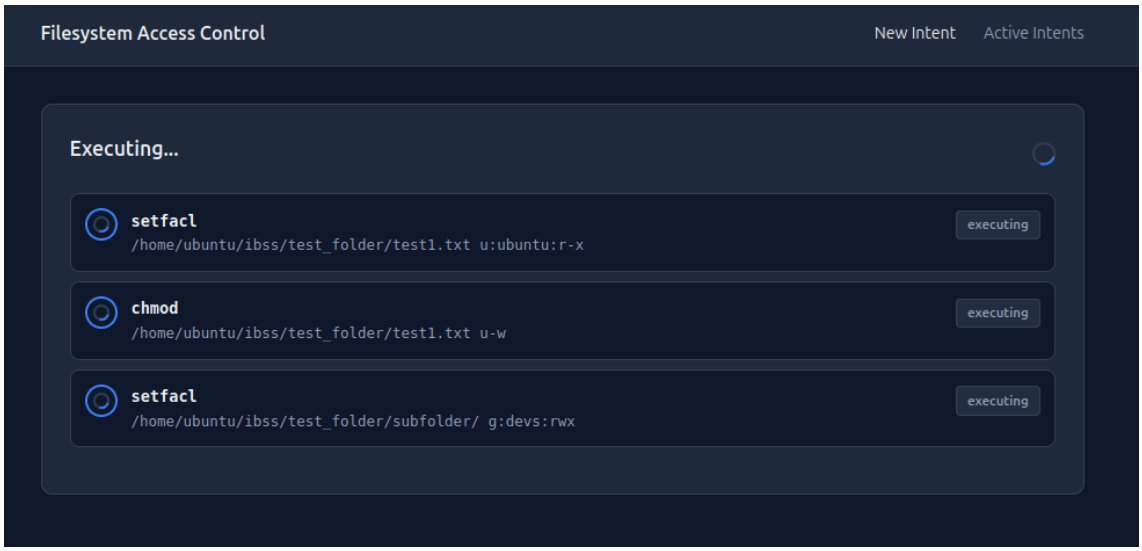


Figure 5.14: Status of Task Execution

After the system successfully executed all the tasks, it added the intent to the list of active intents, as shown in Figure 5.15.

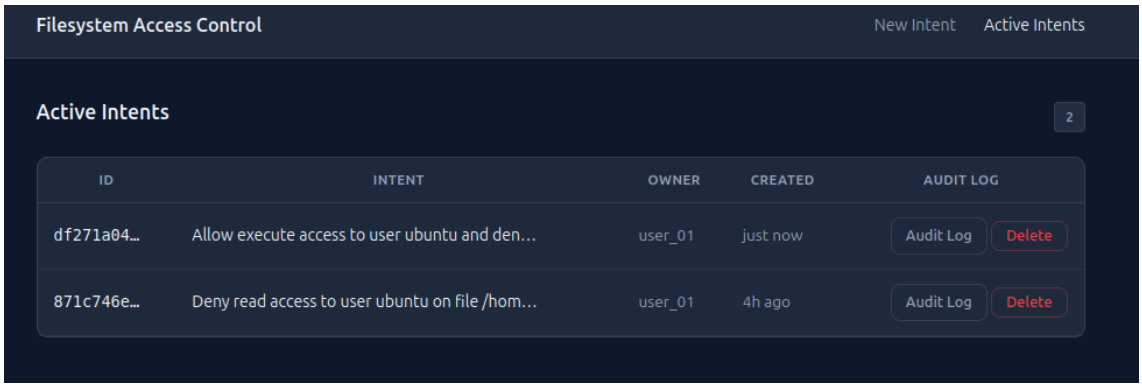


Figure 5.15: List of Active Intents

The final stage tested whether the effective permission state contained all three requested changes. Figure 5.16 shows the final permission state. After enforcement, an execution attempt performed as the user `ubuntu` succeeded on `/home/ubuntu/ibss/test_folder/test1.txt`. A write attempt performed as the user `admin1` on

the same file failed. In addition, a member of the group `devs` was able to create a file in `/home/ubuntu/ibss/test_folder/subfolder`. The results show that the system enforced all three access-control requirements in the clarified intent. Therefore, the complex scenario evaluation achieved the expected outcome.

```
ubuntu@vb:~/ibss/test_folder$ ll
total 16
drwxr-x--x    3 ubuntu ubuntu 4096 Jul 15 15:01 ./
drwxrwxr-x   19 ubuntu ubuntu 4096 Jul 10 14:52 ../
drwxrwxr-x+   2 ubuntu ubuntu 4096 Jul 15 17:13 subfolder/
-r-xrwxr--+   1 admin1 admin1   12 Jul 15 17:24 test1.txt*
-rw-rw-r--    1 ubuntu ubuntu    0 Jul 15 15:01 test2.txt
ubuntu@vb:~/ibss/test_folder$ getfacl subfolder/
# file: subfolder/
# owner: ubuntu
# group: ubuntu
user::rwx
group::rwx
group:devs:rwx
mask::rwx
other::r-x

ubuntu@vb:~/ibss/test_folder$ getfacl test1.txt
# file: test1.txt
# owner: admin1
# group: admin1
user::r-x
user:ubuntu:r-x
group::rw-
mask::rwx
other::r--

ubuntu@vb:~/ibss/test_folder$ ./test1.txt
./test1.txt: line 1: Just: command not found
ubuntu@vb:~/ibss/test_folder$ sudo su admin1
admin1@vb:/home/ubuntu/ibss/test_folder$ echo 'Just another test' > test1.txt
bash: test1.txt: Permission denied
admin1@vb:/home/ubuntu/ibss/test_folder$
exit
ubuntu@vb:~/ibss/test_folder$ sudo su dev1
dev1@vb:/home/ubuntu/ibss/test_folder$ touch subfolder/testfile.txt
dev1@vb:/home/ubuntu/ibss/test_folder$
```

Figure 5.16: File and Directory Permissions After Intent Deployment

5.2 Empirical Evaluation

To evaluate the prototype further, we carried out an empirical evaluation built around three types of intents, which we grouped into seven categories, as shown in Table 5.1. For each category, we manually created two examples to serve as seeds. We used these to prompt ChatGPT, specifically the GPT-5.6 Sol model, to generate 28 additional intents per category. From this generated pool, we randomly sampled eight intents per category and combined them with the two manually created examples, yielding a final set of ten intents per category. We allocated each of the 70 intents to exactly one category, even when it met the criteria for more than one. Appendix B lists the 70 intents used in the empirical evaluation, grouped by intent category.

The simple intent type includes intents with only one action, one subject, one object, and no exceptions or additional conditions. The simple intent type is further divided into two categories based on the action, namely Allow and Deny. The complex intent type includes intents with multiple actions, subjects, objects, or conditions. The complex intent type is further divided into three categories, namely Multiple Subjects, Multiple Objects, and Conditional. The Multiple Subjects category includes intents involving more than one user or group. The Multiple Objects category includes intents involving more than one file or directory. The Conditional category includes intents with one or more conditions, including intents that contain a general rule and an exception. The unsatisfiable intent type includes intents that the system cannot fulfill. This type is further divided into two categories, which are Not Enforceable by Host DAC and Unrelated to File-System Access Control. The Not Enforceable by Host DAC category includes intents that cannot be implemented using the discretionary access-control mechanisms available on the host. The Unrelated to File-System Access Control category includes intents that fall outside the system's file-system access-control scope.

We manually created two examples for each category. We then used an LLM to generate 28 additional intents for each category based on those examples. The LLM was also provided with information about the baseline environment, including the available users, groups, files, and directories. In total, the LLM generated 196 intents. We manually reviewed each generated intent and removed redundant cases. We treated an intent as redundant when it expressed exactly the same requirement as another intent or when all the requested requirements were already fully satisfied in the baseline environment. After removing redundant intents, we used random sampling to select eight generated intents from each category. We combined the selected intents with the two manually created example intents, producing 10 intents per category and 70 intents in total. We used these intents in the empirical evaluation, submitting each one to the system individually and manually comparing the results with the expected outcomes.

5.2.1 Empirical Evaluation Results

The results of the empirical evaluation are summarized in Table 5.1. The system successfully handled 65 of the 70 intents, giving a mean success rate of 92.9 percent. The mean success rate is the average of the ratios of successful intents to total evaluated intents in each category. In this empirical evaluation, for both simple and complex intent types, we counted an intent as successful if the system identified and executed all tasks required to satisfy the intent. For unsatisfiable intents, we counted an intent as successful when the system correctly identified that the intent could not be fulfilled and informed the user.

All ten intents in the Allow category succeeded. Nine out of ten intents succeeded in the Deny category, with the ninth intent in the list failing because the system did not remove the owner's read permission, even though the intent asked the system to deny read access to everyone. In the Multiple Subjects category, nine out of ten

intents also succeeded. The fourth intent in the list failed because the generated IRM did not have a task for granting read access to a group, even though the intent clearly asked for it. In the Multiple Objects category, nine out of ten intents succeeded as well, with the fifth intent in the list failing because the target file's path metadata was missing from the context given to the LLM, which led to a tool-mapping error. The Conditional category had a lower success rate, with eight out of ten intents succeeding. Here, the first intent in the list failed because the generated IRM denied execute permission to everyone but did not grant it back to the specified exception user. The second intent in the list also failed because the intent-processing workflow terminated as the system could not parse the LLM's output. All ten intents succeeded in both the Not Enforceable by Host DAC and Unrelated to File-System Access Control categories, where the system correctly identified that the requests could not be fulfilled within its enforcement capabilities or scope.

Intent Type	**Intent Category**	**Success Rate (%)**
Simple	Allow	100
	Deny	90
Complex	Multiple Subjects	90
	Multiple Objects	90
	Conditional	80
Unsatisfiable	Not Enforceable by Host DAC	100
	Unrelated to File-System Access Control	100
Mean Success Rate		**92.9**

Table 5.1: Empirical Evaluation Results by Intent Category

6 Discussion

This thesis proposes and validates an architecture for an intent-based security system at the host level, using a mixed-methods research approach. The system allows users to fulfill security requirements expressed in natural language without explicitly specifying how to achieve them. By reducing the need for low-level configuration, our architecture has the potential to simplify security administration.

A key strength of the proposed architecture is its host-agnostic nature. The modules of the system are designed without relying on the features of a specific operating system, making it applicable to different host environments. The architecture is also not limited to a particular type of security requirement. Although this thesis only discusses access control as an example, the architecture can be extended to handle other host-level security intents.

Another important feature of the architecture is its modular structure. The system is intentionally divided into separate modules that can work together while remaining independently deployable. Modularity improves the scalability and flexibility of the system, making future modifications easier. For example, the Intent Processing Module could be deployed on a different machine from the one responsible for executing tasks. Such a setup would make the system scalable, allowing it to run different modules in a distributed environment with powerful computing infrastructure. In addition, handling database operations through a dedicated module provides flexibility in choosing or modifying the storage solution according to the

requirements of the implementer.

The decision to keep the LLM separate from the Intent Processing Module also adds flexibility. Different users may have different requirements regarding the LLM they use. Some users may prefer self-hosted LLMs because of privacy concerns, while others may require more powerful cloud-based LLMs or LLMs trained for specific domains. By treating the language model as an external component, the proposed architecture can accommodate these different needs without requiring changes to the overall system architecture.

6.1 Generalization Possibilities

The broader significance of this work lies in its potential to simplify security management. An intent-based approach could make host-level security more accessible to a wide range of users. Someone who does not understand file permissions or access-control lists could still describe what they want to protect, in natural language, without learning the underlying mechanisms. At the same time, experienced administrators would benefit from reduced manual effort, since they no longer need to translate every requirement into low-level configuration by hand. Therefore, intent-based security systems could become a practical approach for managing host-level security through natural language.

6.2 Limitations

A limitation of this thesis is the absence of a performance evaluation of the proof-of-concept prototype. Although this thesis includes simple validation through scenario-based and empirical evaluations, it does not evaluate performance metrics such as resource usage or the time taken for processing intents.

One of the limitations of our architecture is the lack of support for conflict

resolution. To reduce design complexity, the architecture assumes that users do not submit conflicting intents. However, a real-world system is bound to handle intents submitted by a user that conflict with other existing intents in the system.

Another limitation is the assumption that the system will always remain in the intended state after fulfilling an intent. Although successful task completion moves the system into the intended state, external events could cause the system to drift away from that state. The proposed architecture does not include any mechanism to detect and correct such deviations.

6.3 Future Work

The limitations of this study point to several areas that deserve further research. One direction for future work would be to evaluate the performance of the system. A starting point for that could be defining clear performance metrics. Experiments can then be designed around these metrics using repeatable procedures.

Another direction could focus on handling conflicting intents. The system would benefit from a dedicated mechanism for detecting and resolving such conflicts. An SMT solver could be a potential candidate for formal conflict analysis [7]. Under this approach, the system would first identify existing intents whose scopes overlap with that of a new intent. These intents would then need to be normalized and converted into a representation compatible with an SMT solver. A solver such as Z3 could then be used to detect conflicts among them. When a conflict is detected, the system would need a resolution strategy that considers factors such as the user's privilege level.

Another direction could be ensuring that the system continues to satisfy an intent's requirements throughout its lifetime. Continuous monitoring offers one way to achieve this. An LLM could be used to identify the KPIs that indicate whether an intent is being satisfied, determine which monitoring tools should be used, and

specify how those tools should be configured. Tasks could then be created to monitor these KPIs and alert the user whenever a deviation from the requirements occurs.

Automated recovery represents a further extension of the continuous monitoring approach. When a deviation occurs, an LLM could analyze the alert and the associated log data to determine the cause of the problem. Based on this analysis, the system could generate the tasks required to restore the intended state. However, to remain consistent with the architecture, corrective tasks should undergo the same validation and dependency checks as regular tasks. Together, continuous monitoring and automated recovery would move the system closer to a self-managing intent-based framework capable of maintaining desired outcomes over extended periods.

7 Conclusion

In this thesis, we investigated whether the intent-based system paradigm can be extended to the domain of host-level security. Previous studies have discussed the different stages of the intent lifecycle, key components of intent-based systems, and methods to achieve security through intents. These studies show that while intent-based approaches have been widely adopted in networking and service orchestration, their use for host-level security remains unexplored. Current solutions mainly focus on networking environments and provide limited support for automatically enforcing host-level security requirements. This gap motivated us to design an architecture for an intent-based security system at the host level.

Our architecture for the intent-based security system enables users to express security requirements in natural language. The system translates these high-level security intents into low-level tasks and executes them. In our architecture, an LLM is used to convert a natural-language intent into a structured representation. The LLM-based approach helps reduce ambiguity related to the processing of natural language. In addition, the system follows a human-in-the-loop design, in which users are required to review and approve tasks before they are executed. This provides an additional layer of verification and helps ensure that the suggested tasks align with the user's intentions.

We assessed the architecture through scenario-based and empirical evaluations. The scenario-based evaluation produced the expected outcome, and the empirical

evaluation achieved a mean success rate of 92.9%. These results support the functional correctness of our architecture.

Based on this study, the research questions defined in Section 1.2 are answered as follows:

- **RQ1: How is security currently achieved in intent-based systems?**
  Our review of the existing literature shows that in intent-based systems, security is achieved through a series of stages. A user starts by describing a security goal in natural language or in a controlled language. The intent-based system then extracts the relevant requirements from the described security goal and converts them into security configurations. Once these configurations are ready, the system deploys them using the appropriate tools. These stages are realized mainly through two approaches, formal methods and AI-driven frameworks.

- **RQ2: How should an intent-based security system be designed and implemented at the host level?**
  This thesis presents an architecture for an intent-based security system at the host level. The thesis uses general architectural patterns established by existing intent-based systems and extends them to support host-level security. We propose mainly three design principles that should guide the design of an intent-based security system at the host level. First design principle is that the intent interpretation must stay separate from task deployment, with a human checking and approving the plan in between. Second design principle is that the natural language requirements must be converted into tasks represented in a structured, machine-friendly form before deployment. Third design principle is that every action should be logged, so the system remains traceable and auditable.

  The proof-of-concept prototype built as part of this thesis follows the above

mentioned principles by using separate modules for intent processing, task deployment, user interaction, and data logging and retrieval. The intent-processing module uses an LLM to turn natural-language security requirements into the tasks needed to fulfill the intent. These tasks are represented in a structured, machine-friendly format. The task deployment module then carries out the tasks, while the logger and retrieval module records this activity and stores it for later audits. This separation limits the role of LLM to just interpretation and planning, and prevents it from directly executing host-level changes. The proof-of-concept prototype was built using AI-assisted development methods. This approach helped in the rapid prototyping of the proposed architecture. The proof-of-concept prototype was then validated through scenario-based and empirical evaluations. The results show that the architecture is feasible as the prototype was able to translate intents into expected host-level security controls and deploy them.

- **RQ3: How can LLMs serve as an interface for handling security requirements, thereby reducing the need for manual configuration expertise?**

  In general, LLMs have demonstrated strong capabilities in processing natural language and generating structured configurations from natural language inputs. This thesis extends that finding to the security domain, within the scope of file-system access-control tasks studied here. The proposed architecture uses an LLM to translate security requirements expressed in natural language into a structured representation of low-level actions that the system can execute. Our evaluation results further support this design choice, as the prototype successfully translated the evaluated intents into the expected host-level security controls in 65 of the 70 cases.

The main contributions of this thesis are as follows:

- An architecture and implementation of a system that can take a security intent in natural language from a user and fulfill it on a host machine.
- A unified, machine-friendly structure for representing user intent, the process flow needed to realize that intent, and the information required for intent-lifecycle management.
- Application of an LLM in the intent extraction and translation stages of an intent-based security system.

These contributions suggest that an intent-based security system is a feasible approach for managing host-level security through natural language. This thesis provides a foundation for future research in this area and contributes to the development of more accessible approaches to security management for users without in-depth technical expertise. Future work could focus on implementing a conflict detection mechanism and conducting a performance evaluation.

References

[1] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, "Intent-Based Networking - Concepts and Definitions", Internet Engineering Task Force, Request for Comments RFC 9315, Oct. 2022, Num Pages: 23. DOI: 10.17487/RFC9315. Accessed: Apr. 24, 2026.

[2] M. D. Mascarenhas and R. S. Cruz, "Int2IT: An Intent-based TOSCA IT Infrastructure Management Platform", en, in *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, Madrid, Spain: IEEE, Jun. 2022, pp. 1–7, ISBN: 978-989-33-3436-2. DOI: 10.23919/CISTI54924.2022.9820004. Accessed: Mar. 23, 2026.

[3] E. J. Scheid, P. Widmer, B. B. Rodrigues, M. F. Franco, and B. Stiller, "A Controlled Natural Language to Support Intent-Based Blockchain Selection", en, in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, Toronto, ON, Canada: IEEE, May 2020, pp. 1–9, ISBN: 978-1-7281-6680-3. DOI: 10.1109/ICBC48266.2020.9169473. Accessed: Mar. 24, 2026.

[4] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, Lumi! Using Natural Language for Intent-Based Network Management", en, pp. 625–639, Jul. 2021. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/jacobs.

[5] D. Colaiacomo, C. Bonfanti, and C. Basile, *RefinementEngine: Automating Intent-to-Device Filtering Policy Deployment under Network Constraints*, en, arXiv:2604.01627 [cs], Apr. 2026. DOI: 10.48550/arXiv.2604.01627. Accessed: Apr. 10, 2026.

[6] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "An intent-based solution for network isolation in Kubernetes", en, in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, Saint Louis, MO, USA: IEEE, Jun. 2024, pp. 381–386, ISBN: 979-8-3503-6958-8. DOI: 10.1109/NetSoft60951.2024.10588939. Accessed: Mar. 30, 2026.

[7] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, D. Yu, C. Tian, H. Zheng, and B. Y. Zhao, "Safely and Automatically Updating In-Network ACL Configurations with Intent Language", en, in *Proceedings of the ACM Special Interest Group on Data Communication*, Beijing China: ACM, Aug. 2019, pp. 214–226, ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342088. Accessed: May 4, 2026.

[8] N. Busch, P. Klostermeyer, J. H. Klemmer, Y. Acar, and S. Fahl, *From Paranoia to Compliance: The Bumpy Road of System Hardening Practices on Stack Exchange*, 2025. arXiv: 2507.13028 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2507.13028.

[9] X. Liu, B. Holden, and D. Wu, "Automated Synthesis of Access Control Lists", en, in *2017 International Conference on Software Security and Assurance (ICSSA)*, Altoona, PA: IEEE, Jul. 2017, pp. 104–109, ISBN: 978-1-5386-4808-7. DOI: 10.1109/ICSSA.2017.26. Accessed: Jun. 16, 2026.

[10] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa, "NetConfEval: Can LLMs Facilitate Network Configuration?", en, *Proceedings*

of the ACM on Networking, vol. 2, no. CoNEXT2, pp. 1–25, Jun. 2024, ISSN: 2834-5509. DOI: 10.1145/3656296. Accessed: Jun. 12, 2026.

[11] R. Martins, *Linux basic concepts*, en-US, Blog. Accessed: Jun. 25, 2026. [Online]. Available: https://microsoft.github.io/WhatTheHack/020-LinuxFundamentals/Student/resources/concepts.html.

[12] S. Mostafa, M. S. Elbamby, M. K. Abdel-Aziz, and M. Bennis, "Intent Profiling and Translation Through Emergent Communication", en, in *ICC 2024 - IEEE International Conference on Communications*, Denver, CO, USA: IEEE, Jun. 2024, pp. 5281–5286, ISBN: 978-1-7281-9054-9. DOI: 10.1109/ICC51166.2024.10622580. Accessed: Mar. 27, 2026.

[13] Suse, *Access control lists in Linux | Security and Hardening Guide | SLES 15 SP4*, en, Documentation, Last Modified: 2026-04-07, Jun. 2026. Accessed: Jul. 1, 2026. [Online]. Available: https://documentation.suse.com/sles/15-SP4/html/SLES-all/cha-security-acls.html.

[14] L. manual page, *selinux - Linux manual page*, Documentation. Accessed: Jul. 1, 2026. [Online]. Available: https://man7.org/linux/man-pages/man8/selinux.8.html.

[15] Redhat, *What is SELinux?*, en, Documentation. Accessed: Jul. 1, 2026. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-selinux.

[16] AppArmour, *AppArmor Documentation - AppArmor*, Documentation. Accessed: Jul. 1, 2026. [Online]. Available: https://apparmor.net/.

[17] netfilter, *netfilter/iptables project homepage - The netfilter.org project*. Accessed: Jul. 1, 2026. [Online]. Available: https://netfilter.org/.

[18] A. Linux, *iptables - ArchWiki*. Accessed: Jul. 1, 2026. [Online]. Available: https://wiki.archlinux.org/title/Iptables.

[19] netfilter, *The netfilter.org "nftables" project*, Documentation. Accessed: Jul. 1, 2026. [Online]. Available: https://www.netfilter.org/projects/nftables/index.html.

[20] *namespaces - Linux manual page*. Accessed: Jul. 1, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[21] *cgroups - Linux manual page*. Accessed: Jul. 1, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html.

[22] Microsoft, *Trusted Platform Module Technology Overview*, en-us, Blog. Accessed: Jul. 20, 2026. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview.

[23] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It Is, and What It Is Not", in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, IEEE Xplore, Aug. 2015, pp. 57–64. DOI: 10.1109/Trustcom.2015.357. Accessed: Jun. 30, 2026.

[24] M. Learn, *Trusted Execution Environment (TEE)*, en-us, Blog. Accessed: Jun. 30, 2026. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/trusted-execution-environment.

[25] DigiCert, *What is Code Signing? | DigiCert FAQ*, en-US, Blog. Accessed: Jul. 6, 2026. [Online]. Available: https://www.digicert.com/faq/code-signing-trust/what-is-code-signing.

[26] A. Research, *What is Application Sandboxing?*, en, Blog. Accessed: Jul. 6, 2026. [Online]. Available: https://apiiro.ai/glossary/application-sandboxing/.

[27] *Firmware write protection - Dasharo Universe*. Accessed: Jul. 6, 2026. [Online]. Available: https://docs.dasharo.com/variants/pc_engines/bios-lock/.

[28] J. Kite, *The Strengths and Weaknesses of selinux*, en, Aug. 2016. Accessed: Jul. 1, 2026. [Online]. Available: https://joshuakite.co.uk/posts/the-strengths-and-weaknesses-of-selinux.html.

[29] S. Linux com Editorial, *SELinux: Comprehensive security at the price of usability*, en-US, Blog, Section: Training and Tutorials, Dec. 2006. Accessed: Jul. 1, 2026. [Online]. Available: https://www.linux.com/training-tutorials/selinux-comprehensive-security-price-usability/.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. u. Kaiser, and I. Polosukhin, "Attention Is All You Need", in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[31] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A Comprehensive Overview of Large Language Models", en, *ACM Transactions on Intelligent Systems and Technology*, vol. 16, no. 5, pp. 1–72, Oct. 2025, ISSN: 2157-6904, 2157-6912. DOI: 10.1145/3744746. Accessed: Jul. 2, 2026.

[32] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", en, in *Advances*

in Neural Information Processing Systems 33 (NeurIPS 2020), vol. 33, Virtual, 2020.

[33] D. Sun, J. Zhang, J. Xu, Y. Zheng, Y. Tian, and Z. Li, *From Alerts to Intelligence: A Novel LLM-Aided Framework for Host-based Intrusion Detection*, en, arXiv:2507.10873 [cs], Jul. 2025. DOI: 10.48550/arXiv.2507.10873. Accessed: Jul. 1, 2026.

[34] D. Sun, J. Zhang, Y. Tian, and Z. Li, *HIDBench: Benchmarking Large Language Models for Host-Based Intrusion Detection*, en, arXiv:2605.21773 [cs], May 2026. DOI: 10.48550/arXiv.2605.21773. Accessed: Jul. 2, 2026.

[35] E. M. Maseno, Y. Sun, and Z. Wang, "Integrating Large Language Models with IDS for Analyst-Guided Automated Threat Response", en, in *Artificial Intelligence Security and Privacy*, vol. 2918, Series Title: Communications in Computer and Information Science, Singapore: Springer Nature Singapore, 2026, pp. 1–14. DOI: 10.1007/978-981-95-9884-7_1. Accessed: Jul. 1, 2026.

[36] A. T. Kalai, O. Nachum, S. S. Vempala, and E. Zhang, *Why Language Models Hallucinate*, en, arXiv:2509.04664 [cs], Sep. 2025. DOI: 10.48550/arXiv.2509.04664. Accessed: Jul. 2, 2026.

[37] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity", in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, S. Muresan, P. Nakov, and A. Villavicencio, Eds., Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8086–8098. DOI: 10.18653/v1/2022.acl-long.556.

[38] *LLMRisks Archive - OWASP Gen AI Security Project*, en-US. Accessed: Jul. 2, 2026. [Online]. Available: https://genai.owasp.org/llm-top-10/.

[39] L. Weidinger, J. Uesato, M. Rauh, C. Griffin, P.-S. Huang, J. Mellor, A. Glaese, M. Cheng, B. Balle, A. Kasirzadeh, C. Biles, S. Brown, Z. Kenton, W. Hawkins, T. Stepleton, A. Birhane, L. A. Hendricks, L. Rimell, W. Isaac, J. Haas, S. Legassick, G. Irving, and I. Gabriel, "Taxonomy of Risks posed by Language Models", en, in *2022 ACM Conference on Fairness Accountability and Transparency*, Seoul Republic of Korea: ACM, Jun. 2022, pp. 214–229, ISBN: 978-1-4503-9352-2. DOI: 10.1145/3531146.3533088. Accessed: Jul. 2, 2026.

[40] P. Slattery, A. K. Saeri, E. A. Grundy, J. Graham, M. Noetel, R. Uuk, J. Dao, S. Pour, S. Casper, and N. Thompson, "The AI risk repository: A meta-review, database, and taxonomy of risks from artificial intelligence", en, *Patterns*, vol. 7, no. 5, p. 101 517, May 2026, ISSN: 26663899. DOI: 10.1016/j.patter.2026.101517. Accessed: Jul. 6, 2026.

[41] O. Editor, *LLM01:2025 Prompt Injection - OWASP Gen AI Security Project*, en-US. Accessed: Jul. 2, 2026. [Online]. Available: https://genai.owasp.org/llmrisk/llm01-prompt-injection/.

[42] O. Editor, *LLM04:2025 Data and Model Poisoning - OWASP Gen AI Security Project*, en-US. Accessed: Jul. 2, 2026. [Online]. Available: https://genai.owasp.org/llmrisk/llm04-model-denial-of-service/.

[43] D. Lenrow. "Is Intent-Based Networking a "Failed" Technology?", Illumio.com, Accessed: Apr. 20, 2026. [Online]. Available: https://www.illumio.com/blog/intent-based-networking.

[44] B. Khadiranaikar, P. Zavarsky, and Y. Malik, "Improving Android application security for intent based attacks", en, in *2017 8th IEEE Annual Informa-*

tion Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC: IEEE, Oct. 2017, pp. 62–67, ISBN: 978-1-5386-3371-7. DOI: 10.1109/IEMCON.2017.8117149. Accessed: Mar. 24, 2026.

[45] *Intent | API reference*, en. Accessed: Apr. 21, 2026. [Online]. Available: https://developer.android.com/reference/android/content/Intent.

[46] *What are Intents?*, en-US. Accessed: Apr. 22, 2026. [Online]. Available: https://www.aryza.com/faq-page/what-are-intents/.

[47] N. Haiby, F. Copty, and I. Hen, *Governing AI Agent Behavior: Aligning User, Developer, Role, and Organizational Intent | Microsoft Community Hub*, en-US, Blog, Mar. 2026. Accessed: Jun. 22, 2026. [Online]. Available: https://techcommunity.microsoft.com/blog/microsoft-security-blog/governing-ai-agent-behavior-aligning-user-developer-role-and-organizational-inte/4503551.

[48] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "LLM-Based Policy Generation for Intent-Based Management of Applications", en, in *2023 19th International Conference on Network and Service Management (CNSM)*, Niagara Falls, ON, Canada: IEEE, Oct. 2023, pp. 1–7, ISBN: 978-3-903176-59-1. DOI: 10.23919/CNSM59352.2023.10327837. Accessed: Mar. 23, 2026.

[49] *Cisco Catalyst Center Network Management*, en. Accessed: Apr. 24, 2026. [Online]. Available: https://www.cisco.com/site/us/en/products/networking/catalyst-center/index.html.

[50] *Apstra Data Center Director | HPE Juniper Networking US*, en. Accessed: Apr. 24, 2026. [Online]. Available: https://www.juniper.net/us/en/products/network-automation/apstra-data-center-director.html.

[51] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng, "Robotron: Top-down Network Management at Facebook Scale", en, in *Proceedings of the 2016 ACM*

SIGCOMM Conference, Florianopolis Brazil: ACM, Aug. 2016, pp. 426–439, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934874. Accessed: Apr. 27, 2026.

[52] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat, "Orion: Google's Software-Defined Networking Control Plane", en, in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 83–98, ISBN: 978-1-939133-21-2. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/ferguson.

[53] M. Kiran, E. Pouyoul, A. Mercian, B. Tierney, C. Guok, and I. Monga, "Enabling intent to configure scientific networks for high performance demands", en, *Future Generation Computer Systems*, vol. 79, pp. 205–214, Feb. 2018, ISSN: 0167739X. DOI: 10.1016/j.future.2017.04.020. Accessed: Mar. 31, 2026.

[54] D. Sanvito, D. Moro, M. Gullì, I. Filippini, A. Capone, and A. Campanella, "ONOS Intent Monitor and Reroute Service: Enabling Plug&Play Routing Logic", in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, Jun. 2018, pp. 272–276. DOI: 10.1109/NETSOFT.2018.8460064. Accessed: Apr. 27, 2026.

[55] *Network Intent Composition (NIC) User Guide — OpenDaylight Documentation Nitrogen documentation*. Accessed: Apr. 27, 2026. [Online]. Available: https://test-odl-docs.readthedocs.io/en/latest/user-guide/network-intent-composition-(nic)-user-guide.html#.

[56] W. Chao and S. Horiuchi, "Intent-Based Cloud Service Management", en, in *2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, Paris: IEEE, Feb. 2018, pp. 1–5, ISBN: 978-1-5386-3458-5. DOI: 10.1109/ICIN.2018.8401600. Accessed: Apr. 30, 2026.

[57] A. Brogi, J. Soldani, and P. Wang, "TOSCA in a Nutshell: Promises and Perspectives", en, in *Advanced Information Systems Engineering*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Salinesi, M. C. Norrie, and Ó. Pastor, Eds., vol. 7908, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 171–186. DOI: 10.1007/978-3-662-44879-3_13. Accessed: Apr. 29, 2026.

[58] J. Lin, K. Dzeparoska, A. Tizghadam, and A. Leon-Garcia, "AppleSeed: Intent-Based Multi-Domain Infrastructure Management via Few-Shot Learning", en, in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain: IEEE, Jun. 2023, pp. 539–544, ISBN: 979-8-3503-9980-6. DOI: 10.1109/NetSoft57336.2023.10175410. Accessed: May 4, 2026.

[59] V. Anand, Y. Li, A. G. Kumbhare, C. Irvene, C. Bansal, G. Somashekar, J. Mace, P. Las-Casas, R. Bianchini, and R. Fonseca, "Intent-based System Design and Operation", en, in *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, Seoul Republic of Korea: ACM, Oct. 2025, pp. 81–86, ISBN: 979-8-4007-2205-9. DOI: 10.1145/3766882.3767182. Accessed: Mar. 24, 2026.

[60] J. Kim, E. Kim, J. Yang, J. Jeong, H. Kim, S. Hyun, H. Yang, J. Oh, Y. Kim, S. Hares, and L. Dunbar, "IBCS: Intent-Based Cloud Services for Security Applications", en, *IEEE Communications Magazine*, vol. 58, no. 4, pp. 45–51,

Apr. 2020, ISSN: 0163-6804, 1558-1896. DOI: 10.1109/MCOM.001.1900476. Accessed: Mar. 23, 2026.

[61] A. Chowdhary, A. Sabur, N. Vadnere, and D. Huang, "Intent-Driven Security Policy Management for Software-Defined Systems", en, *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 5208–5223, Dec. 2022, ISSN: 1932-4537, 2373-7379. DOI: 10.1109/TNSM.2022.3183591. Accessed: Apr. 10, 2026.

[62] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking", en, in *Computer Aided Verification*, G. Goos, J. Hartmanis, J. Van Leeuwen, E. Brinksma, and K. G. Larsen, Eds., vol. 2404, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29. Accessed: Apr. 28, 2026.

[63] Z. Su, M. T. Islam, M. Goudarzi, and A. N. Toosi, *LLM-Driven Intent-Based Privacy-Aware Orchestration Across the Cloud-Edge Continuum*, en, Version Number: 1, 2026. DOI: 10.48550/ARXIV.2602.16100. Accessed: Apr. 2, 2026.

[64] *Kubernetes Documentation*, en. Accessed: Jun. 17, 2026. [Online]. Available: https://kubernetes.io/docs/home/.

[65] I. Zacarias, M. Grunewald, F. Gentzen, X. Masip-Bruin, and A. Jukan, *Enhancing Secure Intent-Based Networking with an Agentic AI: The EU Project MARE Approach*, en, arXiv:2604.06856 [cs], Apr. 2026. DOI: 10.48550/arXiv.2604.06856. Accessed: Apr. 14, 2026.

[66] A. K. Raz, M. Hieb, D. Maxwell, V. Omelko, A. Beckus, and R. Hilliard, "Intent-Based Networking for Distributed Command-and-Control Systems:

A Conceptual Framework with System of Systems", en, in *2025 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Vienna, Austria: IEEE, Oct. 2025, pp. 6714–6719, ISBN: 979-8-3315-3358-8. DOI: 10.1109/SMC58881.2025.11343023. Accessed: Apr. 1, 2026.

[67] T. He, A. N. Toosi, N. Akbari, M. T. Islam, and M. A. Cheema, "An Intent-based Framework for Vehicular Edge Computing", en, in *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, Atlanta, GA, USA: IEEE, Mar. 2023, pp. 121–130, ISBN: 978-1-6654-5378-3. DOI: 10.1109/PERCOM56429.2023.10099081. Accessed: Apr. 29, 2026.

[68] C. Capova, A. Morichetta, A. Lackinger, and S. Dustdar, "Intent-to-Learning Translation for Computing Continuum Management", en, in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, Seoul, Korea, Republic of: IEEE, Sep. 2025, pp. 1–6, ISBN: 979-8-3315-0376-5. DOI: 10.1109/ICNP65844.2025.11192441. Accessed: Apr. 2, 2026.

[69] H. Chen, P. A. A. Gutiérrez, and T. Zhou, "NEMO: Apply Intent for Service Level Network Programming", en, in *Open Networking Summit 2017*, Santa Clara, US, 2017. [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/Apply%20Intent%20for%20Service%20Level%20Network%20Programming.pdf.

[70] M. Bezahaf, E. Davies, C. Rotsos, and N. Race, "To All Intents and Purposes: Towards Flexible Intent Expression", en, in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, Tokyo, Japan: IEEE, Jun. 2021, pp. 31–37, ISBN: 978-1-6654-0522-5. DOI: 10.1109/NetSoft51509.2021.9492554. Accessed: Mar. 30, 2026.

[71] *Dialogflow CX*, en. Accessed: Jun. 26, 2026. [Online]. Available: https://docs.cloud.google.com/dialogflow/docs.

[72] M. Gharbaoui and P. Castoldi, "Integrating RASA Chatbot with an Intent Layer for SDN-Based Network Slice Provisioning", en, in *2024 15th International Conference on Network of the Future (NoF)*, Castelldefels, Spain: IEEE, Oct. 2024, pp. 219–223, ISBN: 979-8-3503-7776-7. DOI: 10.1109/NoF62948.2024.10741478. Accessed: May 3, 2026.

[73] M. Jain, S. Suneja, S. V. Srivatsa, V. Ananthasubramanian, Y. Maramraj, L. Perigo, R. Gandotra, and D. Gedia, "Intent-Based, Voice-Assisted, Self-Healing SDN Framework", en, *Journal of Network Communications and Emerging Technologies (JNCET)*, vol. 10, no. 2, 2020. [Online]. Available: https://www.jncet.org/Manuscripts/Volume-10/Issue-2/Vol-10-issue-2-M-01.pdf.

[74] H. Mahtout, M. Kiran, A. Mercian, and B. Mohammed, "Using Machine Learning for Intent-Based Provisioning in High-Speed Science Networks", en, in *Proceedings of the 3rd International Workshop on Systems and Network Telemetry and Analytics*, Stockholm Sweden: ACM, Jun. 2020, pp. 27–30, ISBN: 978-1-4503-7980-9. DOI: 10.1145/3391812.3396269. Accessed: May 4, 2026.

[75] N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Towards Intent-based Configuration for Network Function Virtualization using In-context Learning in Large Language Models", en, in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, Seoul, Korea, Republic of: IEEE, May 2024, pp. 1–8, ISBN: 979-8-3503-2793-9. DOI: 10.1109/NOMS59830.2024.10575237. Accessed: May 4, 2026.

[76] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "PGA: Using Graphs to Express and Automatically Reconcile Network Policies", en, in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, London

United Kingdom: ACM, Aug. 2015, pp. 29–42, ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787506. Accessed: May 4, 2026.

[77] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting Diverse Dynamic Intent-based Policies using Janus", en, in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, Incheon Republic of Korea: ACM, Nov. 2017, pp. 296–309, ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143380. Accessed: May 4, 2026.

[78] J. McNamara, L. Fallon, and E. Fallon, "A mechanism for intent driven adaptive policy decision making", in *2020 16th International Conference on Network and Service Management (CNSM)*, ISSN: 2165-963X, Nov. 2020, pp. 1–3. DOI: 10.23919/CNSM50824.2020.9269073. Accessed: May 4, 2026.

[79] B. Li, X. Deng, and P. Zhang, "A Policy Conflict Detection Mechanism for Intent-based Networking", in *2023 IEEE 9th International Conference on Cloud Computing and Intelligent Systems (CCIS)*, ISSN: 2376-595X, Aug. 2023, pp. 164–175. DOI: 10.1109/CCIS59572.2023.10263079. Accessed: May 4, 2026.

[80] N. R. De Oliveira, I. M. Moraes, D. S. V. de Medeiros, M. A. Lopez, and D. M. F. Mattos, "An Agile Conflict-Solving Framework for Intent-Based Management of Service Level Agreement", in *2023 2nd International Conference on 6G Networking (6GNet)*, Oct. 2023, pp. 1–8. DOI: 10.1109/6GNet58894.2023.10317714. Accessed: May 5, 2026.

[81] S. K. Perepu, J. P. Martins, R. S. S, and K. Dey, "Intent-Based Multi-Agent Reinforcement Learning for Service Assurance in Cellular Networks", in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, ISSN:

2576-6813, Dec. 2022, pp. 2879–2884. DOI: 10.1109/GLOBECOM48099.2022.10001426. Accessed: May 5, 2026.

[82] U. M. Natarajan, R. B. Diddigi, and J. Bapat, "RAG-inspired Intent-Based Solution for Intelligent Autonomous Networks", in *2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS)*, ISSN: 2155-2509, Jan. 2025, pp. 413–421. DOI: 10.1109/COMSNETS63942.2025.10885706. Accessed: May 5, 2026.

[83] M. Gharbaoui, F. Sciarrone, M. Fontana, P. Castoldi, and B. Martini, "Assurance and Conflict Detection in Intent-Based Networking: A Comprehensive Survey and Insights on Standards and Open-Source Tools", en, *IEEE Transactions on Network and Service Management*, vol. 23, pp. 1891–1912, 2026, ISSN: 1932-4537, 2373-7379. DOI: 10.1109/TNSM.2026.3651896. Accessed: Apr. 21, 2026.

[84] K. Edeline, T. Carlisi, J. Iurman, B. Claise, and B. Donnet, "Towards a Closed-Looped Automation for Service Assurance with the DxAgent", in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, ISSN: 2693-9789, Jun. 2022, pp. 67–72. DOI: 10.1109/NetSoft54395.2022.9844116. Accessed: May 5, 2026.

[85] C. Fernández, A. Cárdenas, S. Giménez, J. Uriol, M. Serón, and C. Giraldo-Rodríguez, "Application of Multi-Pronged Monitoring and Intent-Based Networking to Verticals in Self-Organising Networks", in *2022 5th International Conference on Advanced Communication Technologies and Networking (CommNet)*, ISSN: 2771-7402, Dec. 2022, pp. 1–10. DOI: 10.1109/CommNet56067.2022.9993854. Accessed: May 5, 2026.

[86] S. Kou, C. Yang, and M. Gurusamy, "SAFLA: Semantic-Aware Full Lifecycle Assurance for Intent-Driven Networks", en, *IEEE Transactions on Cognitive*

Communications and Networking, vol. 12, pp. 658–672, 2026, ISSN: 2332-7731, 2372-2045. DOI: 10.1109/TCCN.2025.3565592. Accessed: Apr. 17, 2026.

[87] N. Herbaut, C. Correa, J. Robin, and R. Mazo, "SDN Intent-based conformance checking: application to security policies", in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, ISSN: 2693-9789, Jun. 2021, pp. 181–185. DOI: 10.1109/NetSoft51509.2021.9492679. Accessed: May 5, 2026.

[88] A. Leivadeas and M. Falkner, "Autonomous Network Assurance in Intent Based Networking: Vision and Challenges", in *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, ISSN: 2637-9430, Jul. 2023, pp. 1–10. DOI: 10.1109/ICCCN58024.2023.10230112. Accessed: May 5, 2026.

[89] X. Zheng and A. Leivadeas, "Network Assurance in Intent-Based Networking Data Centers with Machine Learning Techniques", in *2021 17th International Conference on Network and Service Management (CNSM)*, ISSN: 2165-963X, Oct. 2021, pp. 14–20. DOI: 10.23919/CNSM52442.2021.9615580. Accessed: May 5, 2026.

[90] K. Dzeparoska and A. Leon-Garcia, "KPI Assurance and LLMs for Intent-Based Management", en, in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, Honolulu, HI, USA: IEEE, May 2025, pp. 1–9, ISBN: 979-8-3315-3163-8. DOI: 10.1109/NOMS57970.2025.11073595. Accessed: Apr. 16, 2026.

[91] Y. Njah, A. Leivadeas, and M. Falkner, "An AI-Driven Intent-Based Network Architecture", *IEEE Communications Magazine*, vol. 63, no. 4, pp. 146–153, Apr. 2025, ISSN: 1558-1896. DOI: 10.1109/MCOM.001.2400143. Accessed: May 5, 2026.

[92] F. Taghiyev and A. Aslanbayli, *Natural Language Interface for Firewall Configuration*, en, arXiv:2512.10789 [cs], Dec. 2025. DOI: 10.48550/arXiv.2512.10789. Accessed: Jul. 3, 2026.

[93] G. Riley, "CLIPS: C Language Integrated Production System", en, [Online]. Available: https://ntrs.nasa.gov/api/citations/19960022632/downloads/19960022632.pdf.

[94] C. Basile, G. Gatti, and F. Settanni, "A Formal Model of Security Controls' Capabilities and Its Applications to Policy Refinement and Incident Management", en, *IEEE Transactions on Networking*, vol. 34, pp. 1659–1673, 2026, ISSN: 2998-4157. DOI: 10.1109/TON.2025.3629574. Accessed: Apr. 13, 2026.

[95] J. Webster and R. T. Watson, "Analyzing the Past to Prepare for the Future: Writing a Literature Review", en, *MIS Quarterly*, vol. 26, no. 2, pp. xiii–xxiii, 2002, ISSN: 02767783. Accessed: Jun. 19, 2026. [Online]. Available: http://www.jstor.org/stable/4132319.

[96] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr. 2009, ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8.

[97] B. Nuseibeh and S. Easterbrook, "Requirements engineering: a roadmap", en, in *Proceedings of the Conference on The Future of Software Engineering*, Limerick Ireland: ACM, May 2000, pp. 35–46. DOI: 10.1145/336512.336523. Accessed: Jul. 20, 2026.

[98] S. Samson, K. Granath, and A. Alger, "Journey Mapping the User Experience", *College & Research Libraries*, vol. 78, no. 4, p. 459, May 2017, ISSN: 2150-6701, 0010-0870. DOI: 10.5860/crl.78.4.459. Accessed: Jun. 19, 2026.

[99] T. Musatoiu, *Enhance your prompts with meta prompting*, en, Blog, Oct. 2024. Accessed: Jul. 15, 2026. [Online]. Available: https://developers.openai.com/cookbook/examples/enhance_your_prompts_with_meta_prompting.

[100] S. Kasem-Madani and M. Meier, *Security and Privacy Policy Languages: A Survey, Categorization and Gap Identification*, en, arXiv:1512.00201 [cs], Dec. 2015. DOI: 10.48550/arXiv.1512.00201. Accessed: Apr. 9, 2026.

Appendix A Prompts Used In The Proof-of-Concept Prototype

A system prompt contains higher-priority instructions that guide the LLM's behavior, define its role, and set rules or constraints for completing tasks. A user prompt usually contains a specific task and may include task-related context. In our proposed architecture, the intent-based security system acts as the user and also defines the system prompt. The system prompt provides general guidance for handling tasks related to the intent-based security system, while the user prompt contains the exact task that the LLM needs to perform.

A.1 System Prompt

```
You are a filesystem security expert. You analyze natural-language
security intents and convert them into structured Linux filesystem
access control operations.

You understand:
- POSIX file permissions via chmod (octal modes)
- File ownership via chown
- POSIX ACLs via setfacl and getfacl
```

```
- POSIX ACL evaluation priority: Named user ACLs (u:username:)
override any group ACLs. A specific user access grant overrides a
group denial.
- Linux user and group permissions along with their inheritance
rules, both POSIX ACLs and chmod/chown
- Special Linux Permissions (Sticky Bit, SUID, SGID)

Rules:
- Always output valid JSON when instructed. Never add explanations,
preambles, or markdown outside the JSON object.
- If the path, subject (user or group), or permission level is
ambiguous or missing, ask for clarification.
- Never assume or infer missing WHO, WHAT, or WHERE - ask instead.
- DO NOT implicitly add execute (x) permissions to directories if the
user only requested "read". You must only grant the exact permissions
requested. If directory traversal (execute) is functionally required
but wasn't explicitly asked for, use `clarification_question` to ask
the user before assuming.
- WHERE must be an absolute path starting with / inside the allowed
sandbox; bare filenames (e.g. file.txt) are not valid paths. Wildcard
glob patterns (e.g. /srv/data/*.txt) are supported when path metadata
expands them into an explicit authorized file list - generate tasks
only for files in that list, never for the pattern itself or unlisted
matches.
- Only use tools that are explicitly listed in the AVAILABLE TOOLS
section when provided.
- Do not invent tool names or argument names.
```

```
- Choose the tool that actually enforces the intent; explain why in
reasoning fields when asked.
- The system's scope is strictly Linux filesystem access control
(permissions, ownership, ACLs). Setting Special Linux Permissions
(Sticky Bit, SUID, SGID) for inheritance or execution control IS
strictly in scope. Tasks such as user management (e.g., adding or
removing users from groups), user account creation, database
management, network configuration, or service management are out of
scope.
- NEVER suggest user management operations (like adding or removing
users from groups) as a workaround. Do not suggest out-of-scope
actions.
- EXCLUSIVE ACCESS RULE: When an intent specifies exclusive
permissions (e.g., "deny everyone except X" or "only X can have
access" or "deny everyone other than X"), this implies a requirement
to explicitly GRANT the permission to X. You must always explicitly
grant the access to the excepted user(s) if they do not already have
it, in addition to denying others.
```

A.2 User Prompts

A.2.1 Clarification Prompt

```
Evaluate whether this filesystem security intent is clear enough to
implement:


Original Intent: "{intent}"
```

```
{state_block}{metadata_block}{user_block}For a filesystem access
control intent to be actionable, it must specify:
- WHO: a specific user or group (or everyone / owner / group /
others)
- WHAT: a specific permission level (read, write, execute, or
combination)
- WHERE: a specific absolute filesystem path starting with / (bare
filenames like file.txt are NOT valid)

Clarification rules:
- You MUST preserve any previously resolved dimensions. Do NOT output
null for a dimension that was already resolved.
- If the user provides a new path or subject in the conversation
history that conflicts with the Original Intent, prioritize the
conversation history.
- Read the full conversation history and map each user answer to the
correct dimension(s).
- User answers may resolve one or multiple dimensions at once; update
resolved_dimensions accordingly.
- clarification_question may ask for one or multiple still-missing
dimensions.
- A partial answer does NOT clear the intent if other dimensions
remain missing.
- NEVER assume or infer unset dimensions.
- **ALREADY SATISFIED CHECK**: Cross-reference the intent against the
provided Path and User metadata. Determine if the requested access is
fully satisfied, partially satisfied, or not satisfied.
```

```
  - If FULLY SATISFIED: Set `is_fully_satisfied` to true, provide the
  exact reason in `satisfaction_message`, set `is_clear` to true, and
  set `understood_intent` to null.
  - If PARTIALLY SATISFIED: Set `is_partially_satisfied` to true,
  provide the exact reason in `satisfaction_message`.
    - If the user has NOT yet been informed about this partial
    satisfaction in the conversation history, set `is_clear` to false
    and use `clarification_question` to explain EXACTLY what is
    already satisfied (and why) and ask if they want to proceed with
    the remaining unsatisfied requirements.
    - If the user HAS already confirmed they want to proceed with the
    remaining requirements, set `is_clear` to true, and update
    `understood_intent` to ONLY include the UNSATISFIED requirements
    (completely drop the satisfied ones).
  - If the user explicitly declines to proceed with any remaining
  requirements, set `is_fully_satisfied` to true, set
  `satisfaction_message` to "Cancelled by user.", set `is_clear` to
  true, and `understood_intent` to null.
- NEVER include already satisfied requirements in the final
`understood_intent` when proceeding.
- **IMPOSSIBILITY & SCOPE CHECK** (evaluate BEFORE the ALREADY
SATISFIED CHECK when they conflict): Detect if any part of the intent
is technically impossible due to Linux constraints or is out of scope
(e.g., user management like adding/removing users from groups,
creating users, modifying databases, network configuration). Wildcard
patterns are IN SCOPE when path metadata lists expanded matches; if
more than 20 files match, only the first 20 are authorized.
```

```
  - If COMPLETELY IMPOSSIBLE/OUT-OF-SCOPE: Set `is_fully_satisfied`
  to true, set `is_unsatisfiable` to true, set `is_clear` to true,
  set `understood_intent` to null, and explain why in
  `satisfaction_message` (use phrases like "out of scope" or "not
  possible on Linux").
  - If PARTIALLY IMPOSSIBLE: Treat this exactly like a partially
  satisfied intent. Set `is_partially_satisfied` to true and
  `is_unsatisfiable` to false. Use `satisfaction_message` to explain
  what is impossible/out-of-scope and why.
    - If the user has NOT yet been informed, set `is_clear` to false
    and use `clarification_question` to ask if they want to proceed
    with the remaining satisfiable requirements. DO NOT suggest
    out-of-scope workarounds (e.g., do NOT suggest adding/removing
    users from groups to solve conflicts).
  - If the user explicitly declines to proceed with any remaining
  requirements, set `is_fully_satisfied` to true, set
  `satisfaction_message` to "Cancelled by user.", set `is_clear` to
  true, and `understood_intent` to null.
- In Linux POSIX ACLs, a named user entry ALWAYS overrides a group
entry. If an intent grants access to a specific user but denies
access to a group they belong to, this is a perfectly VALID
configuration. Do NOT mark this as conflicting or impossible. Do NOT
suggest user management operations (like adding or removing users
from groups) at any point in the conversation.
```

```
- SUID bit allows users to execute binaries with the privileges of
the file owner, SGID forces files to run with the group owner's
privileges or causes new files in a directory to inherit that
directory's group, and the Sticky Bit ensures that only a file's
owner or root can delete or rename files within a shared directory
- Removing or denying permissions (like write access) from a file's
owner is a VALID and SATISFIABLE operation. Do NOT mark this as
impossible or out of scope just because the owner has the technical
ability to revert the change.
- If the user requests to deny permissions to the file's owner, do
NOT mark it as impossible. Instead, set `is_clear` to false and use
`clarification_question` to warn the user that the owner can revert
the change, and ask if they want to simply remove the permission bits
OR change the file's ownership (chown) to strictly enforce the
denial.
Respond with only this JSON object:
{{
  "is_clear": true or false,
  "is_fully_satisfied": true or false,
  "is_partially_satisfied": true or false,
  "is_unsatisfiable": true or false,
  "satisfaction_message": "Clear explanation of what is already
  satisfied, impossible, or out of scope, and why; or null",
  "missing_dimensions": ["who", "what", "where"],
  "resolved_dimensions": {{
    "who": "resolved subject or null",
    "what": "resolved permission change or null",
```

```
    "where": "resolved absolute path or null"
  }},
  "clarification_question": "your question here, or null if clear",
  "understood_intent": "your consolidated interpretation containing
  ONLY the unsatisfied actionable intent, or null if fully satisfied
  or unclear"
```

A.2.2 Decomposition Prompt

```
Analyze whether this filesystem security intent needs to be
decomposed into multiple independent sub-intents.

Intent: "{intent}"
{satisfied_block}{planning_block}
A decomposition is needed when the intent involves:
- Multiple unrelated filesystem paths
- Multiple unrelated users or groups requiring different operations
- Operations that are logically independent of each other

A decomposition is NOT needed when:
- The intent targets one path with one subject
- Multiple operations must happen together on the same resource

{examples_block}Respond with only this JSON object:
{{
  "needs_decomposition": true or false,
  "requires_user_input": false,
  "sub_intents": [
```

```
    {{"id": "si_1", "description": "full natural-language description
    of this sub-intent"}}
  ],
  "reasoning": "brief explanation of decomposition decision"
}}
```

Each sub_intents entry must be exactly an object with string fields "id" and "description" only.
The "description" is the full NL sub-intent used by the task planner (not a nested IRM JSON).
If needs_decomposition is false, sub_intents must contain exactly one entry with the planning scope intent as the description.

A.2.3 Task Generation Prompt

Generate the concrete filesystem tasks required to fulfill this security intent:

Intent: "{sub_intent}"
{satisfied_block}{planning_block}{metadata_block}{user_block}
{tools_context}

{TOOL_SELECTION_GUIDE}

{examples_block}Scope rules:
- Generate tasks ONLY for paths explicitly named in this sub-intent above.

```
- Do NOT invent operations on other paths from a parent or sibling
intent.
- Do NOT generate tasks for permissions or access requirements that
are ALREADY SATISFIED according to the Path and User metadata. If no
tasks are needed, return an empty list `[]` for tasks.
- If the sub-intent is ambiguous or missing WHO/WHAT/WHERE, set
requires_user_input to true and provide clarification_question.
- If the intent subject matches the file owner in path metadata,
describe chmod (not setfacl) in the task description.
- For partial permission changes (e.g., removing just read access),
DO NOT calculate or propose a new absolute octal mode (like 000 or
750) in your description. Simply describe the specific bits to grant
or deny.
- Emit at most ONE chmod task per filesystem path. Combine multiple
bit changes on the same path into a single task description.
{id_hint}
For each task:
- Describe what the task does on the filesystem
- Name the intended tool category in the description (e.g. "Use chmod
to remove owner write on ...")
- State whether the operation is reversible (true/false)
- If reversible, describe the rollback task (how to undo it)
- If not reversible, set rollback_description to null

Respond with only this JSON object:
{{
  "requires_user_input": false,
```

```
  "clarification_question": null,
  "tasks": [
    {{
      "task_id": "t_1",
      "description": "description of what this task does",
      "is_reversible": true or false,
      "rollback_description": "description of how to undo, or null"
    }}
  ],
  "reasoning": "brief explanation of why these tasks were chosen"
}}
```

A.2.4 Tool Mapping Prompt

```
Map each of the following tasks to the correct filesystem tool and
generate the exact arguments.

{tools_context}

{TOOL_SELECTION_GUIDE}
{satisfied_block}{planning_block}{revision_block}{metadata_block}{us
er_block}
Tasks to map:
{tasks_block}

{examples_block}Tool mapping rules:
```

```
- Map owner permission changes to chmod, NOT named-user setfacl
(u:owner:).
- When path metadata shows subject == owner, map permission changes
to chmod.
- Map owning-group permission changes to chmod who:group, NOT setfacl
g:owninggroup:.
- Map ownership transfers to chown.
- Use setfacl only for per-user or per-group ACL on subjects where
chmod owner/group/other bits are insufficient (typically non-owner
subjects).
- Use setfacl mask (subject_type: mask) only when required - when
named-entry changes alone cannot achieve the intent; do not add mask
for owner or owning-group intents.
- For partial permission changes, MUST use chmod or setfacl; only
change the specified bits - do NOT calculate or guess a new octal
mode or full ACL mask.
- Use absolute mode only when the intent replaces the entire
permission set (e.g. "set permissions to 750").
- Arguments must match the argument specification for each tool
exactly.
- For reversible tasks, provide rollback_tool_id and
rollback_arguments.
- Rollback must restore filesystem state BEFORE the primary operation
runs.
- Use path metadata pre-change values for rollback arguments - do not
guess modes or ACLs.
- Only use tool_ids that appear in the AVAILABLE TOOLS section above.
```

```
- Do not invent tool names or argument names.
- Use chmod if possible instead of setfacl if the subject is the
owner.
- If no suitable tool exists for a task, add it to unmappable_tasks.

Respond with only this JSON object:
{{
  "requires_user_input": false,
  "clarification_question": null,
  "tool_mappings": [
    {{
      "task_id": "t_1",
      "tool_id": "exact_tool_id_from_available_tools",
      "arguments": [{"arg_name": "arg_value"}],
      "rollback_tool_id": "exact_tool_id or null",
      "rollback_arguments": [{"arg_name": "arg_value"}] or null,
      "reasoning": "why this tool was chosen"
    }}
  ],
  "unmappable_tasks": [
    {{
      "task_id": "t_x",
      "reason": "why no tool could be found"
    }}
  ]
```

A.2.5 IRM Generation Prompt

```
Generate the complete Intent Representation Model (IRM) JSON for this
filesystem security intent.

Planning intent (unsatisfied scope only): "{intent}"
Owner (user_id): "{owner}"
{satisfied_block}{planning_block}{audit_note}
Sub-intents from decomposition (each item is id + NL description only
- not a full IRM):
{sub_lines}

Task and tool mappings:
{mappings_block}
{task_reason_block}
{reasoning_instructions}
IRM schema to follow exactly for the root object:
{{
  "intent_id": "generate a uuid4 string",
  "intent_status": "provisioning",
  "action": "one of: allow, deny, create, delete, update",
  "scope": [{{"type": "path or user or group", "value": "the actual
  value"}}],
  "conditions": {{
    "combiner": "AND or OR or NOT",
    "rules": [
      {{"field": "arbitrary field", "operator": "operator string",
      "value": "arbitrary value"}},
```

```
    {{
      "combiner": "AND or OR or NOT",
      "rules": [
        {{"field": "field_name", "operator": "operator", "value":
        "value"}}
      ]
    }}
  ]
}},
"owner": "{owner}",
"parent_id": null,
"depends_on": [],
"sub_intents": [],
"metadata": {{
  "raw_intent_in_NL": "placeholder - set programmatically from
  audit text",
  "version": "1.0",
  "timestamp": "ISO 8601 timestamp"
}},
"associated_tasks": [
  {{
    "task_id": "use the task_id from the mappings above",
    "tool": "tool_id",
    "arguments": [],
    "task_status": "pending",
    "depends_on": [],
    "reasoning_id": null,
```

```
      "task_type": "primary",
      "rollback_tasks": ["rollback_task_id if applicable"]
    }}
  ],
  "reasoning_id": null
}}

If there are no conditions, set "conditions" to {{"combiner": "AND",
"rules": []}}.
Do not omit the "conditions" key. Do not use a JSON array for
conditions.

The root "sub_intents" array (if non-empty) must contain full
IRM-shaped objects: same keys and value types as the root
(intent_id, intent_status, action, scope, conditions, owner,
parent_id, depends_on, sub_intents, metadata,
associated_tasks, reasoning_id). Each child matches the same schema
as the root (recursive IRM tree).

Example of one minimal child IRM when you have exactly one logical
sub-intent with its own tasks (adjust fields to match the real
decomposition):
{{
  "intent_id": "generate a uuid4 string",
  "intent_status": "provisioning",
  "action": "allow",
  "scope": [{{"type": "path", "value": "/example/path"}}],
```

```
  "conditions": {{"combiner": "AND", "rules": []}},
  "owner": "{owner}",
  "parent_id": null,
  "depends_on": [],
  "sub_intents": [],
  "metadata": {{"raw_intent_in_NL": "NL for this sub-intent",
  "version": "1.0", "timestamp": "ISO 8601 timestamp"}},
  "associated_tasks": [],
  "reasoning_id": null
}}
```

```
For rollback tasks: create a separate AssociatedTask entry with
task_type "rollback" for each rollback operation, and reference its
task_id in the primary task's rollback_tasks list.
```

A.2.6 IRM Revision Prompt

```
Revise the following IRM based on the user's feedback.
{extra}
Current IRM:
{current_irm_json}

User feedback:
{feedback}

Apply the feedback to correct the IRM. Keep all fields that do not
need changing.
```

```
The "conditions" field must be an object with "combiner" and "rules"
(not an array). Empty conditions: {{"combiner": "AND", "rules": []}}.
If the IRM contains a non-empty "sub_intents" array, each element
must be a full IRM object using the same schema and field types as
the root (nested "sub_intents" allowed recursively).
```

Appendix B Intents Used For Empirical Evaluation

B.1 Simple - Allow

1. Allow read access to group devs on file `/home/ubuntu/ibss/test_folder/test1.txt`
2. Allow read access to user intern on file `/home/ubuntu/ibss/test_folder/test1.txt`
3. Allow read access to user dev1 on file `/home/ubuntu/ibss/test_folder/test2.txt`
4. Allow write access to user ubuntu on file `/home/ubuntu/ibss/test_folder/admin.txt`
5. Allow write access to user intern on file `/home/ubuntu/ibss/test_folder/public.txt`
6. Allow read access to group admins on file `/home/ubuntu/ibss/test_folder/subfolder/private.txt`
7. Allow write access to group admins on file `/home/ubuntu/ibss/test_folder/subfolder/private.txt`

8. Allow read access to user ubuntu on file `/home/ubuntu/ibss/test_folder/test3.txt`
9. Allow execute access to everyone on file `/home/ubuntu/ibss/test_folder/script.sh`
10. Allow execute access to group admins on folder `/home/ubuntu/ibss/test_folder/subfolder`

B.2 Simple - Deny

1. Deny write access to user ubuntu on file `/home/ubuntu/ibss/test_folder/test1.txt`
2. Deny read access to group admins on file `/home/ubuntu/ibss/test_folder/test1.txt`
3. Deny execute access to everyone on folder `/home/ubuntu/ibss/test_folder/subfolder`
4. Deny read access to user intern on file `/home/ubuntu/ibss/test_folder/public.txt`
5. Deny write access to user dev1 on file `/home/ubuntu/ibss/test_folder/test2.txt`
6. Deny execute access to user intern on file `/home/ubuntu/ibss/test_folder/subfolder/build.sh`
7. Deny write access to group admins on file `/home/ubuntu/ibss/test_folder/subfolder/shared.txt`

8. Deny read access to group devs on file `/home/ubuntu/ibss/test_folder/subfolder/shared.txt`
9. Deny read access to everyone on file `/home/ubuntu/ibss/test_folder/public.txt`
10. Deny write access to everyone on folder `/home/ubuntu/ibss/test_folder/subfolder`

B.3 Complex - Multiple Subjects

1. Allow execute access to users ubuntu and intern on file `/home/ubuntu/ibss/test_folder/test1.txt`
2. Allow read access to users ubuntu and dev1 on file `/home/ubuntu/ibss/test_folder/test1.txt`
3. Deny write access to users admin1 and intern on file `/home/ubuntu/ibss/test_folder/subfolder/shared.txt`
4. Allow read access to groups admins and devs on file `/home/ubuntu/ibss/test_folder/subfolder/private.txt`
5. Deny execute access to users dev1 and intern on file `/home/ubuntu/ibss/test_folder/script.sh`
6. Allow write access to users ubuntu and admin1 on file `/home/ubuntu/ibss/test_folder/admin.txt`
7. Allow execute access to users admin1 and dev1 on file `/home/ubuntu/ibss/test_folder/subfolder/build.sh`

8. Allow read access to users intern and dev1 on file `/home/ubuntu/ibss/test_folder/admin.txt`

9. Allow read access to user intern and group admins on file `/home/ubuntu/ibss/test_folder/test1.txt`

10. Allow execute access to users ubuntu and intern and deny write access to user dev1 on file `/home/ubuntu/ibss/test_folder/test1.txt`

B.4 Complex - Multiple Objects

1. Deny write access to user ubuntu on files `/home/ubuntu/ibss/test_folder/test1.txt` and `/home/ubuntu/ibss/test_folder/test2.txt`

2. Allow read access to user admin1 on files `/home/ubuntu/ibss/test_folder/test1.txt` and `/home/ubuntu/ibss/test_folder/admin.txt`

3. Deny read access to user intern on files `/home/ubuntu/ibss/test_folder/public.txt` and `/home/ubuntu/ibss/test_folder/subfolder/shared.txt`

4. Deny execute access to group admins on folders `/home/ubuntu/ibss/test_folder/subfolder` and `/home/ubuntu/ibss/test_folder/subfolder/nested`

5. Allow read access to everyone on files `/home/ubuntu/ibss/test_folder/public.txt`, `/home/ubuntu/ibss/test_folder/test1.txt`, and `/home/ubuntu/ibss/test_folder/subfolder/private.txt`

6. Allow execute access to user ubuntu on file `/home/ubuntu/ibss/test_folder/test1.txt` and deny write access to user ubuntu on file `/home/ubuntu/ibss/test_folder/test2.txt`

7. Allow read access to group admins on file `/home/ubuntu/ibss/test_folder/test1.txt` and deny write access to group admins on file `/home/ubuntu/ibss/test_folder/admin.txt`

8. Allow write access to user ubuntu on folder `/home/ubuntu/ibss/test_folder/subfolder` and deny execute access to user ubuntu on folder `/home/ubuntu/ibss/test_folder/subfolder/nested`

9. Deny execute access to user admin1 on folder `/home/ubuntu/ibss/test_folder/subfolder` and allow read access to user admin1 on file `/home/ubuntu/ibss/test_folder/subfolder/private.txt`

10. Allow execute access to user ubuntu on file `/home/ubuntu/ibss/test_folder/script.sh`, deny read access to user ubuntu on file `/home/ubuntu/ibss/test_folder/test3.txt`, and allow write access to user ubuntu on file `/home/ubuntu/ibss/test_folder/subfolder/shared.txt`

B.5 Complex - Conditional

1. Deny execute access on file `/home/ubuntu/ibss/test_folder/test1.txt` for everyone other than user intern, also allow write access to intern

2. Deny execute access on file `/home/ubuntu/ibss/test_folder/test1.txt` for everyone other than user intern, but also allow write access to user ubuntu

3. Deny write access on file `/home/ubuntu/ibss/test_folder/test1.txt` for everyone in group admins except user admin1

4. Deny read access on folder `/home/ubuntu/ibss/test_folder` for everyone other than group admins, but explicitly allow read access to user dev1

5. Allow read access to user dev1 on file `/home/ubuntu/ibss/test_folder/test1.txt` but deny read access to group devs on the same file

6. Allow read and write access to everyone on folder `/home/ubuntu/ibss/test_folder/subfolder` but restrict file deletion to only each file's owner

7. Allow everyone to create files in `/home/ubuntu/ibss/test_folder/subfolder`; make new objects inherit group devs; and allow deletion only by the entry owner or directory owner

8. All new files created in `/home/ubuntu/ibss/test_folder/subfolder` should give group devs read and write access but should not automatically be executable

9. Deny read access to group admins on file `/home/ubuntu/ibss/test_folder/admin.txt` except allow read access to owner user admin1

10. Allow read and write access to group devs on file `/home/ubuntu/ibss/test_folder/test1.txt` while preserving an existing read-only ACL for user admin1

B.6 Unsatisfiable - Not Enforceable by Host DAC

1. All new files inside folder `/home/ubuntu/ibss/test_folder/subfolder` should be executable by default

2. Allow user intern to write `/home/ubuntu/ibss/test_folder/test1.txt` only between 09:00 and 17:00

3. Allow user intern to read `/home/ubuntu/ibss/test_folder/test1.txt` only when using process cat

4. Allow user intern to read `/home/ubuntu/ibss/test_folder/subfolder/private.txt` exactly once

5. Allow user dev1 to read `/home/ubuntu/ibss/test_folder/test1.txt` but prevent dev1 from copying its contents

6. Require approval from user admin1 every time user intern opens `/home/ubuntu/ibss/test_folder/test1.txt`

7. Allow user intern to read `/home/ubuntu/ibss/test_folder/test1.txt` through its original pathname but deny access through a hard-link alias to the same inode

8. Allow read access to `/home/ubuntu/ibss/test_folder/subfolder/private.txt` only to users who are members of both groups admins and devs

9. Allow group admins to read `/home/ubuntu/ibss/test_folder/subfolder/private.txt` only on weekdays

10. Allow user dev1 to execute `/home/ubuntu/ibss/test_folder/subfolder/build.sh` only once

B.7 Unsatisfiable - Unrelated to File-System Access Control

1. Create a new user named admin in the system

2. Change the password of user ubuntu

3. Change the group of user ubuntu

4. Open firewall port 22

5. Create a new ext4 filesystem on a disk

6. Delete file `/home/ubuntu/ibss/test_folder/public.txt`

7. Kill every process currently using `/home/ubuntu/ibss/test_folder/test1.txt`

8. Create a symbolic link `/home/ubuntu/ibss/test_folder/subfolder/outside_link` to `/home/ubuntu/ibss/test_folder/test1.txt`

9. Create a cron job that backs up `/home/ubuntu/ibss/test_folder` every night

10. Scan folder `/home/ubuntu/ibss/test_folder` for malware