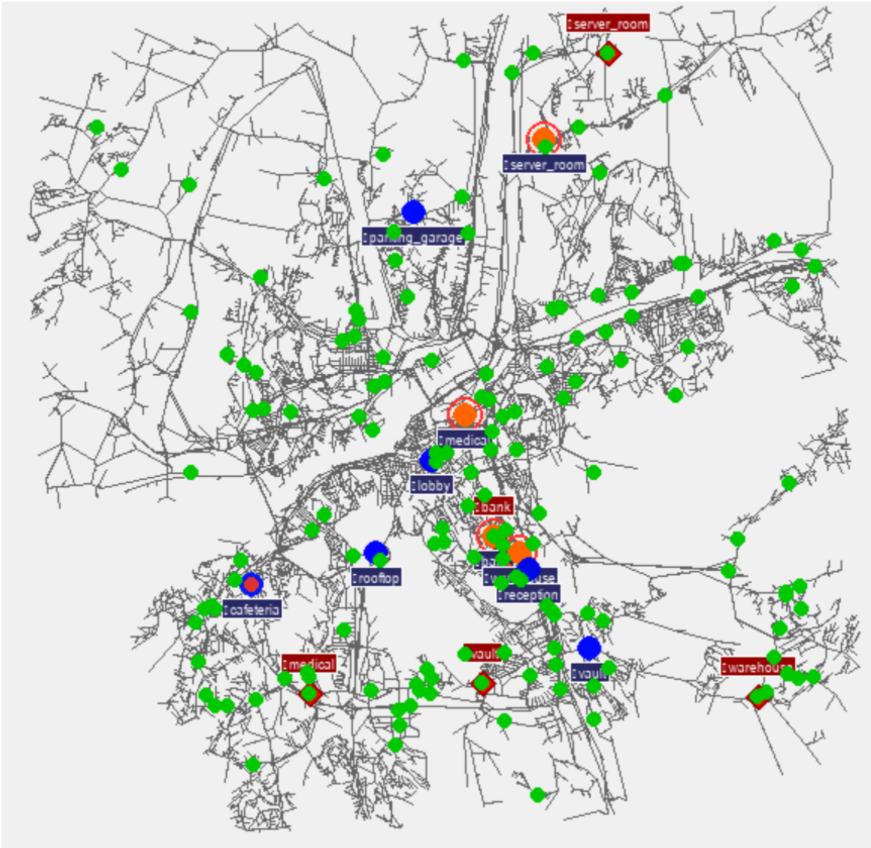
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Evaluating LLM Adaptation Strategies in Physical Security Operations

Reducing Cognitive Load and Alarm Fatigue in Control Rooms

Master's thesis 2026

Olof Lindberg, Casper Christiansson

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's thesis 2026

Evaluating LLM Adaptation Strategies in Physical Security Operations

Reducing Cognitive Load and Alarm Fatigue in Control Rooms

Olof Lindberg, Casper Christiansson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2026

Evaluating LLM Adaptation Strategies in Physical Security Operations
Reducing Cognitive Load and Alarm Fatigue in Control Rooms
Olof Lindberg, Casper Christiansson

Supervisor: Carl-Johan Seger, Department of Computer Science and Engineering
Supervisor: Peter Wahlgren, Blue Mobile Systems
Examiner: Aarne Ranta, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Division of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of a dispatch scenario being simulated

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Evaluating LLM Adaptation Strategies in Physical Security Operations
Reducing Cognitive Load and Alarm Fatigue in Control Rooms
Olof Lindberg, Casper Christiansson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Physical security control rooms demand rapid decisions under high alarm volumes, producing cognitive load and alarm fatigue that degrade operator performance. This thesis evaluates three LLM adaptation strategies (zero-shot, few-shot, and retrieval-augmented generation) against a rule-based baseline on two tasks: alarm dispatch and guard chat. Evaluation was conducted in a simulation environment modelling GuardTools Command and Control, a SaaS platform for manned guarding operations. For dispatch, the rule-based strategy was fastest and cheapest. Among LLM strategies, few-shot achieved the lowest execution failure rate (1.82%), while zero-shot degraded sharply under clustered alarms, reaching 26.62% failures. RAG consumed roughly twice the tokens of few-shot without improving reliability. For chat, the decisive factor was context management rather than prompting technique: filtered few-shot and RAG isolated the relevant location description before prompting, achieving the highest quality scores while minimizing token usage.

Acknowledgements

We would like to thank our supervisor Carl-Johan Seger for his guidance, feedback, and support throughout this thesis. His input helped us shape the direction of the work and the written report.
We would also like to thank the people at Blue Mobile Systems for their help during the knowledge acquisition phase and providing us with a good work environment. Their insight into the physical security domain, control-room workflows, alarm handling, and guard dispatching was valuable for understanding the real-world context behind the problem and for designing a more relevant simulation environment.

Olof Lindberg and Casper Christiansson, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
API	Application Programming Interface
BPE	Byte Pair Encoding
DSR	Design Science Research
ETA	Estimated Time of Arrival
FAE	False Alarm Experience
GPS	Global Positioning System
GPT	Generative Pre-trained Transformer
GTCC	GuardTools Command and Control
IQR	Interquartile Range
JSON	JavaScript Object Notation
LLM	Large Language Model
MDSP	Multi-stage Dialogue Prompting
ML	Machine Learning
MPP	Multi-Problem Prompting
NIAH	Needle in a Haystack
NLP	Natural Language Processing
P95	95th Percentile
PPS	Physical Protection System
RAG	Retrieval-Augmented Generation
RNN	Recurrent Neural Network
SaaS	Software as a Service
SOC	Security Operations Centre

Contents

List of Figures

List of Tables

1
Introduction

Physical security operations rely on human operators making rapid, high-stakes decisions in environments where information is fragmented across multiple interfaces and alarms can arrive faster than they can be processed. As AI capabilities grow, there is increasing interest in how large language models can support these operators, but it is not obvious how such systems should be built or which approach to adapting an LLM works best for this kind of environment. This thesis investigates that question.

1.1 Background

Modern manned guarding operations depend on efficient communication and coordination between field guards and control-center operators. To support this workflow, the company GuardTools developed GuardTools Command and Control, a Software as a Service platform designed specifically for security operations. The system functions as the operator's main workspace, providing real-time information about guards' locations, active alarms, patrol routes, lone-worker safety alerts, and key management. In essence, it is the operational backbone that keeps the entire guarding service running.

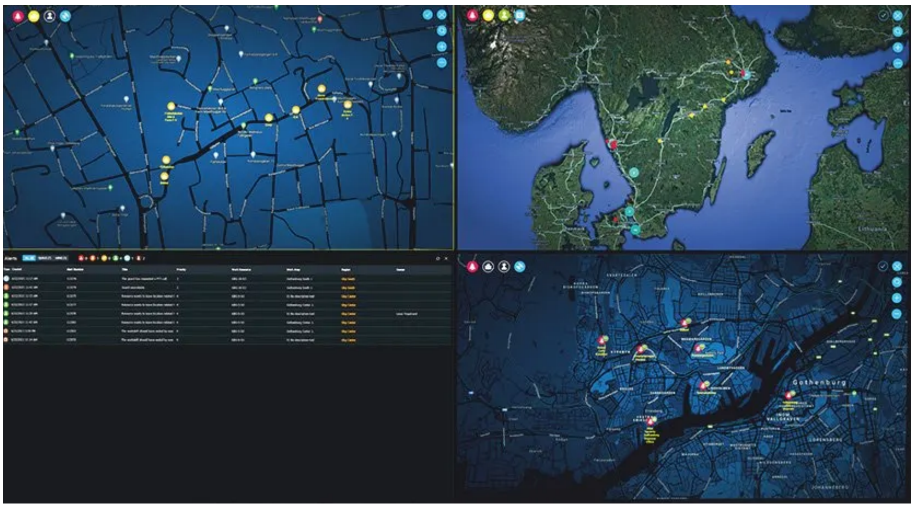


Figure 1.1: GuardTools Command and Control

Operators in GuardTools handle multiple simultaneous tasks. Monitoring alarms, coordinating guards, managing lone worker alerts, and ensuring that ongoing patrols and keyrings are properly executed. Although GuardTools Command and Control provides all necessary information, operators must manually search for relevant details and data across different interface views, and decide on the appropriate action under time pressure. This manual workflow increases cognitive load, slows response times, and can in worst case lead to inconsistent decision-making, especially during high-alert periods.

With the rise of large language models (LLMs) and agent like AI systems, there is an opportunity to explore how AI can support these operators by understanding the context, predicting their intent, and automating or assisting in complex workflows. However, it is not yet clear how different LLM adaptation methods affect the accuracy and efficiency of such AI-assisted decision-support in a real world alarm management system like GuardTools Command and Control.

1.2 Aim

The core problem this thesis addresses is understanding how different LLM adaptation strategies influence the accuracy and efficiency of AI-assisted decision-making in alarm management systems. The outcome should not simply demonstrate that AI can be applied in this context, but evaluate which adaptation approach best supports accurate, efficient, and reliable integration into alarm management.

The study is guided by the following research questions:

- **RQ1:** How do different LLM adaptation strategies compare to a rule-based baseline for alarm dispatch in terms of efficiency and reliability?
- **RQ2:** How do the strategies perform when answering guard chat questions that require context-specific knowledge, such as access procedures, alarm configurations, and site layouts?
- **RQ3:** What do the results imply for the practical use of LLM-based assistants in security control rooms, particularly in relation to cognitive load of the operator?

1.3 Related Work

Modern work in building AI systems for control-room operators and security teams has been growing fast. However, most of it focuses on either cybersecurity “security operations centres” (SOCs), industrial control rooms, or general research on how humans and AI teams work together. Few studies examine an AI assistant-type system that supports a guard control room in a field-based security operation.

LLMs in Security Operation Centres

Specifically, recent work from Microsoft Security Research has been done leveraging Machine Learning (ML) in assisting the work of SOCs. The research team developed a “copilot”-style assistant that helps analysts by guiding responses, triaging and investigating alerts. Their work produced excellent results and showed the potential for ML-based assistants to assist in complex tasks and speed up analysis. But one major difference between this project and ours has been the lack of LLM usage. Although LLMs was used in other parts of their system, such as incident summarization, script analyzer and incident report, it was not used in assisting with responses to alerts. Another noteable difference are that they aimed at cyber-defence use cases rather than real-time physical security operations (such as coordinating field guards).[1]

In a related study, Singh et al. [2] examined how SOC analysts choose to use LLMs in their work. The research found that LLMs were used primarily as cognitive support, and was overwhelmingly used in command interpretation and text processing. The SOC analysts rarely used the LLM to receive explicit recommendations on what actions to take, but rather preferring evidence and contextual information that preserved their decision-making authority. This suggests that current SOC analysts do not yet view the decision-making capabilities of LLMs as sufficiently reliable for high-stakes actions. While this could be interpreted as a limitation of LLMs in security contexts, it also implies that without a robust adaptation strategy, the full potential of LLMs remains untapped. This highlights the value of developing successful adaptation methods (such as those explored in this thesis) to bridge the trust gap in critical decision-making.

Human-AI Collaboration in Decision Support

While these SOC-focused systems demonstrate the technical potential of LLM-based assistants, understanding the human element is also important. Peng et al. [3] investigated the relationship between user proficiency and AI capability within intelligent decision-support systems. Their experiments with university students involved tasks such as memory matching and visual illusions. The study found that while the AI significantly boosted the performance of less skilled participants, the effect on the more competent users showed a small decrease from 74% to 73%. Interestingly, these users reported a significant increase in confidence (from 64% to 75%) despite the lack of objective improvement. This suggests that while users might not always gain accuracy from assistance of an AI, the system could still provide a 'perceived' support. For our work, this shows the need to evaluate whether adaptation methods provide actual operational value or simply increase an operator's confidence in their existing decisions.

Attempts are also being made to integrate LLMs into systems designed to help personnel interpret the broader context of security incidents in surveillance and security domains. For example, a recent study by Chen et al. [4] have demonstrated how LLMs can interpret live surveillance video streams and provide real-time summaries of suspicious activities. This capability of LLMs is particularly relevant for environments like GuardTools, where the value of an AI assistant lies in reducing the cognitive effort required to obtain an understanding of the situation from multiple data sources, such as patrol logs, GPS data, and alarm descriptions.

LLM Adaptation Methods

Kermani et al. [5] conducted a systematic comparison of three LLM adaptation strategies, where fine-tuning, prompt engineering, and retrieval-augmented generation (RAG) were used for mental health text analysis. Their work showed that fine-tuning had a substantially higher accuracy, with zero-shot prompting in second place, and RAG in last. While their work focused on static classification rather than real-time decision support, their observation that RAG performance depends heavily on retrieval quality raises important questions whether RAG can be used reliably on continously updating data. Additionally, their framework for comparison, where consistent metrics on the same base model were used, can be used as a foundation to our evalutation framework.

Soudani et al. [6] conducted a similar study where RAG and fine-tuning was compared in the context of less frequent topics and reached a different conclusion. When each method was used independently, it was found that RAG consistently outperforms fine-tuning in question answering tasks for domain-specific knowledge. Combining RAG and fine-tuning yields the optimal performance in small models, but for larger models it is not always an improvement compared to using only RAG. Their study also supports Kermani et al.'s observation about retrieval quality, showing that different methods of retrieval affected the overall performance.

Multi-stage prompting, proposed by Tan et al. [7], is another method of adapting an LLM. The concept of decomposing complex tasks into sequential prompts has shown promise in adapting general pre-trained models to domain-specific applications. Their experiments found that performance improvements were primarily attributed to the clear separation of steps rather than the increase model capacity, showing the potential of decomposition. While their study focused on machine translation, it is possible this generalizes to other areas because the core contribution addresses a fundamental challenge applicable to any complex task: a single prompt handling multiple subtasks simultaneously is likely suboptimal.

A similar study about decomposing prompts in more than one step was done by Liu et al. with their Multi-stage Dialogue Prompting framework (MDSP) [8]. They applied the concept to a dialogue system where two steps were performed: first, relevant knowledge was generated, and second, feeding it into a prompt to produce a response based on that context. A key difference from Tan et al.'s approach is that MDSP uses few-shot prompting with natural language examples rather than using prompt tuning, a method of fine-tuning without updating the model's internal weights [9].

1.4 Scope and Delimitations

For this thesis, not all LLM adaptation methods will be evaluated. The selection will be based on feasibility within the project's timeframe, the data available, and how fairly they can be compared. The assistant will focus on a subset of the possible operators tasks instead of the full range of GuardTools functionality. Evaluation will be performed in a simulated environment rather than a production environment. Due to potential privacy and security concerns, synthetic data may be used in some cases. The choice of LLM model can influence the results and may therefore not generalize as well for other models. Additionally, since LLM models are frequently updated, the evaluation reflects model versions available at the time of experimentation.

1.5 Contribution

This thesis provides a comparative evaluation of three LLM adaptation strategies (zero-shot, few-shot, and RAG) against a rule-based baseline for two core tasks in physical security operations: alarm dispatch and guard chat. The main contribution is empirical evidence on how the choice of adaptation strategy affects reliability, response time, and cost, interpreted in relation to cognitive load, alarm fatigue, and operator trust.

1.6 Disposition

The remainder of this thesis is structured as follows:

- **Chapter 2: Theory** provides background on physical protection systems, human factors in alarm operations such as vigilance decrement and alarm fatigue, large language models and their adaptation strategies, and rule-based expert systems.
- **Chapter 3: Methods** describes the knowledge acquisition process, the design and implementation of the simulation environment, the implementation of each LLM adaptation strategy and the rule-based baseline, and the evaluation framework including metrics for dispatch performance and chat response quality.
- **Chapter 4: Results** presents the empirical findings across four test categories: Hawkes workload tests, edge-case scenarios, context-window load tests, and guard chat evaluation.
- **Chapter 5: Discussion** interprets the results in terms of practical implications for dispatch and chat, examines the prompt injection risk introduced by the chat interface, discusses implications for cognitive load and operator trust, addresses threats to validity, and outlines directions for future work.
- **Chapter 6: Conclusion** summarises the main findings and revisits the research aim.

2

Theory

2.1 Security Alarm Systems

A security alarm system is a part of a Physical Protection System (PPS). A PPS is defined as an integrated collection of components or elements designed to prevent the success of malevolent actions, such as theft or sabotage, according to plan [10, 11].

In security theory, an alarm system is only effective if it fulfills three sequential functions:

1. Detection: The process of sensing an adversary action and assessing the event to verify its validity [12].
2. Delay: The use of physical barriers or obstacles to slow down an adversary's progress, often in the form of equipment and/or guards[10].
3. Response: The actions taken by a response force to interrupt the adversary [12].

For a system to be considered "secure" the time required for the adversary to complete their task (T_{attack}) must be greater than the time required for the system to detect and respond ($T_{\text{detect}}+T_{\text{respond}}$) [10].

2.2 Human Factors in Security Alarm Operations

2.2.1 Vigilance and the Vigilance Decrement

Vigilance, or sustained attention, refers to a neuropsychological state commonly explained by being "alertly watchful" to detect and respond to sudden signals over extended periods. A core challenge in security monitoring is the decline in the probability of detecting signals as time on the task increases, also known as vigilance decrement. Research has shown that this performance decay is robust and typically becomes prominent after the first 30 minutes. [13, 14]

Contrary to traditional theories suggesting that monitoring is "understimulating," research indicates that maintaining vigilance imposes a large amount of mental workload. Global workload scores for vigilance tasks are often higher than those for grammatical reasoning or simple tracking tasks [15]. Warm et al. [15] argues that this high cognitive workload is driven primarily by mental demand and frustration. As the operator must continuously make decisions under conditions of high uncertainty during long periods of time the decrement of vigilance will kick in, likely resulting in increased response times.

2.2.2 Alarm Fatigue and the "Cry Wolf" Effect

In security operations, the human operator is not a passive receiver, but rather an active interpreter of the alarm. When an operator is exposed to a high volume of alarms, it can cause alarm fatigue. This is a term used to describe the cognitive overload that an operator can experience where the distinction between a true threat and "noise" becomes blurred. This leads to the "Cry Wolf" effect, a psychological desensitization where the credibility of the alarm system is worsened by False Alarm Experiences (FAEs) [16, 17].

When the perceived reliability of a system is low, human factors degrade the PPS in critical ways. If an operator repeatedly encounters alarms that do not result in a visible threat, they might not take future warnings seriously. When studying nurses responses to alarms Cvach et al. explains that the perceived reliability of the alarm system would follow the response rate. So if an alarm system was perceived to be 90% reliable the response rate would be about 90% [17]. In security alarm systems things like these could lead to intentional delays or silencing of alerts without proper assessment, directly extending T_{detect}.

Recent empirical research by Zhu et al. provides a great understanding of **Alarm Fatigue** by studying it among nurses in intensive care units. It identifies alarm fatigue not as a static failure, but as a dynamic process between three cognitive states: cognitive reserve deficit, cognitive load balance, and cognitive overload. Nurses frequently enter high-stress environments with a limited cognitive reserve, and if they are able to maintain their performance or succumb to fatigue depends on the balance of "demands" such as alarm frequency and "resources" such as team support and experience. In this environment, cognitive overload was found to be the direct trigger for fatigue, which leads to an experience of "inattentional deafness" where individuals physically hear an alarm but fail to cognitively register its significance. These findings are transferable to the security domain, as they suggest that when the workload of complex monitoring and the frustration of persistent false alarms exceed an operator's cognitive capacity, mistakes will be made[18].

2.3 Large Language Models

Before Large Language Models (LLMs), Recurrent Neural Networks (RNNs) were the state of the art approaches to transduction problems such as language modeling and machine translation [19]. These were limited in their ability to capture long-range dependencies and were computationally inefficient on large datasets. A shift to parallelism began with the introduction of the Transformer architecture by Vaswani et al. [19], which removed the need for recurrence. The parallel processing that the Transformer allows, enabled the training of models on datasets of unprecedented size.

2.3.1 Tokenization

Before an LLM can work with a text, the text needs to be converted into a numerical representation that it can understand. Using a tokenizer, raw text is split into smaller segments called tokens. There are several methods of tokenization, with the most prominent being Byte Pair Encoding (BPE). While originally a compression algorithm, Sennrich et al. [20] showed how it can be used to segment rare and unknown words for neural machine translation. By iteratively merging the most frequent adjacent symbol pairs into a new symbol, a vocabulary of subword units that can represent any word is achieved. How text is tokenized vary depending on the training data. The tokenizer used by Llama 3 would divide the word "dispatched" into "dispatch" and "ed". Since "ed" is a common suffix in the English language and the tokenizer is frequency-driven, it ends up as its own token.

The context window is the hard limit on how many tokens a model can process in a single request. Research by Hong et al. [21] evaluated 18 LLMs using the Needle in a Haystack (NIAH) benchmark and found that their performance grows increasingly unreliable as input length grows. This tradeoff is relevant to this thesis, as the adaptation strategies differ in how much context is injected into the prompt.

2.3.2 Hallucinations

Hallucination in the context of LLMs refer to the generation of output that sounds coherent but is factually incorrect. This is a major limitation of autoregressive language models which simply predict the most probable next token rather than verify its correctness. The causes of hallucinations are present across the entire LLM lifecycle, from misinformation in the pre-training corpora to the way tokens are sampled during generation [22].

In high-stakes operational environments such as security alarm dispatch, hallucinations are an immediate threat. For example, an LLM might fabricate a guard identifier or invent a key requirement, which would break the system's response function. This risk is particularly relevant for this thesis, as the adaptation strategies differ in how they mitigate hallucinations: RAG grounds outputs in retrieved evidence, and few-shot prompting provides concrete examples to constrain behavior. Additionally, the sampling temperature can be adjusted during inference, where a higher value increases the probability of selecting low-frequency tokens, which raises hallucination risk [22].

2.3.3 In-Context Learning

A key finding in the development of LLMs is that sufficiently large models can adapt to new tasks at inference time without any modification to their parameters. Brown et al. [23] demonstrated this with GPT-3, a model with a staggering 175 billion parameters, and found that this significantly improved its ability to perform new tasks with little to no task-specific data. Their approach involved three in-context learning methods:

1. Few-Shot: As many demonstrations as the model's context window allows.
2. One-Shot: The model is provided with exactly one demonstration of the task in addition to a natural language description. This closely resembles how a human would be taught a task.
3. Zero-Shot: The model relies solely on a natural language instruction for the task.

These modes of in-context learning form the foundation of the prompt design strategies evaluated in this thesis, described in the following section.

2.4 Prompt Design

2.4.1 Zero-Shot

Zero-shot prompting is an adaptation method where a Large Language Model (LLM) is asked to perform a task without being provided with any specific examples of that task in the prompt. Instead of learning from a set of "demonstrations" provided by the user, the model relies entirely on its pre-trained internal knowledge to interpret and execute the instruction.[23]

In the context of GuardTools, a zero-shot prompt might consist of a task description and input data, such as: "An alarm with the following properties has gone off: <alarm_properties>. From the available guards: <guard_list>, dispatch the most appropriate guard". A key advantage of this approach is that it uses no additional knowledge or examples, which makes it very simple and flexible [23]. However, due to its simplicity, the performance can vary depending on the task complexity. Research by Wang et al. shows that while LLMs are competent in isolated classification or reasoning tasks, their performance declines sharply when multiple problems are fit into a single prompt, a phenomenon termed the "Multi-Problem Prompting" (MPP) bottleneck [24]. Notably, these performance drops can occur even when the number of subtasks is small (e.g., ≤ 5), suggesting a lack of true logical planning or "understanding" in complex zero-shot scenarios.

2.4.2 Few-Shot

Few-shot prompting is an adaptation technique where a small number of input-output examples are included within the prompt to demonstrate the desired task to the LLM. Unlike fine-tuning, which involves updating the model's internal weights through backpropagation, few-shot prompting utilizes in-context learning. [23] To build on the example from the zero-shot technique, a few-shot prompt could look something like: "An Alarm with these properties has gone off <alarm_properties>. Previous alarm similiar to this one was handled like this: <history of previous alarms with context and solution>. Please dispatch the alarm to the most appropriate guard".

In a specialized environment like GuardTools one advantage of few-shot prompting could be its ability to reduce ambiguity. For instance, the system's prompt might include multiple structured examples demonstrating different dispatch scenarios such as re-routing a guard to a closer alarm or dispatching a guard without the right key to perform a perimeter check. By embedding these input-output examples directly in the prompt, the model can align its responses with the system's specific action rules and decision logic, improving performance for domain-specific tasks it may not have encountered during pre-training [23, 25].

The efficiency of few-shot prompting is highly dependent on the quality of the provided examples. Research indicates that LLMs are susceptible to majority label bias (favoring an answer that appears more frequently in the examples) and recency bias (placing more weight on the examples closest to the final instruction) [26]. Furthermore, increasing the amount of examples can lead to "over-prompting", where excessive context consumes the model's limited token window and leads to degraded performance [27].

2.4.3 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is an architectural framework that enhances LLMs by integrating a retrieval mechanism that accesses external, non-parametric data. By dynamically retrieving relevant document snippets, for example from a vector database, it is able to better "ground" its responses to factual sources, reducing hallucinations and enabling more accurate, up-to-date, and verifiable outputs [28]. Compared to zero-shot and few-shot, this allows the LLM to use not only examples, but also domain-specific knowledge.

RAG typically operates in two stages: a retrieval phase and a generation phase. During the retrieval phase, the most relevant information from an external knowledge base is extracted. In the RAG framework introduced by Lewis et al. [28], this is achieved through a 'dense neural retriever' where the query and documents are converted into numerical vectors to perform a similarity search. Then, in the generation phase, the retrieved context is integrated with the original query and provided to the LLM.

A primary advantage of RAG, particularly important for an environment such as security operations, is that it aims to minimize hallucinations by providing evidence for the model to follow. By grounding the output in retrieved documents, the model is less likely to generate factually incorrect information compared to models relying solely on their internal parameters and a handful of examples. Additionally, the knowledge base can be updated without having to retrain or fine-tune the RAG in any way. If a security protocol changes, the system can simply index the new document, allowing the model to provide revised instructions immediately. This makes RAG particularly effective for handling domain-specific knowledge and less frequent topics where a general-purpose LLM might otherwise struggle due to a lack of relevant training data.[28]

However, the effectiveness of a RAG system is heavily contingent on its retrieval quality, as fetching irrelevant or "noisy" data can lead to incorrect or conflicting model outputs. If the retriever fails to identify the correct protocol or includes distracting information, the overall performance of the system may decrease. Additionally, the nature of RAG introduces higher latency compared to simpler zero-shot or few-shot prompting, which may be a critical consideration in time-sensitive contexts.[29]

2.5 Expert Systems

An expert system is a kind of AI program used to apply the decision-making knowledge of human specialists within a specific domain. Introduced in the 1970s, they were one of the earliest practical applications of AI. Expert systems consists of two core components: a knowledge base and an inference engine. The knowledge base stored the domain-specific knowledge and rules, while the inference engine applies those rules to the input in order to derive a conclusion [30].

Rules are typically created using explicit conditional "if-then" statements. For example, a simple dispatch rule written in pseudocode in the context of GTCC might state: "if a guard is available and holds the required keys and has the shortest ETA, then dispatch that guard". These rules can be applied in two directions: forward- and backwards chaining. Forward chaining begins with known facts and applies rules iteratively until a conclusion is reached. In contrast, backwards chaining starts from a desired goal (e.g. identifying the cause of a fault) and works backward to determine what conditions must be true to support that conclusion. This is more commonly used for diagnostic reasoning.

In the context of alarm management, early applications of expert systems in industrial process control demonstrated that they could assist operators by processing alarm floods and proposing corrective actions faster than an unaided human could. For example, Tsurushima, et al. [31] applied rule-based technology to generate security plans for alarm systems. Their findings suggest that the approach is reliable and computationally efficient under laboratory conditions, although with limitations when deployed in unconstrained real-world environments. In process control, nuclear plant fault diagnosis expert systems were shown to accurately identify failed subsystems and guide operator responses, directly reducing the cognitive burden on the operator during high-stress conditions [32].

The main strengths of rule-based expert systems are predictability, transparency, and speed [33]. Every decision can be traced directly to the rule that fired, which makes these systems fully explainable by a human, unlike responses from LLMs. This is an important property in safety-critical settings where it is crucial that an operator can trust a recommendation by the system. They are also incredibly computationally effective and therefore can be run locally which minimizes the latency from network calls.

However, expert systems are not perfect. Their knowledge must be explicitly encoded by a programmer, meaning the system cannot handle scenarios that fall outside the defined rules. They have no ability to learn or reason about ambiguous natural language input. Early efforts in the field demonstrated that methods used in narrow “blocks worlds” did not extend to more general tasks where ambiguity is introduced [34, p. 41].

As the operational domain grows in complexity, maintaining the rule base becomes increasingly laborious, and rules may interact in unanticipated ways. Compared to LLM-based approaches, expert systems are brittle. A guard question phrased in a slightly unexpected way, or an alarm scenario not anticipated by the designer, may produce a wrong or empty response. This brittleness stems from the lack of a general method for using world knowledge to handle the uncertainty inherent in real-world environments [34, p. 41].

3

Methods

3.1 Research Design

This thesis follows a Design Science Research (DSR) approach as described by Peffers et al. [35]. The designed artifacts are a simulation environment that recreates the relevant aspects of a security control-room workflow and a set of LLM adaptation strategies implemented within that environment. The primary contribution is the comparative evaluation of these strategies against a rule-based baseline. The research follows four stages that align with the general DSR process:

1. **Problem identification** through exploratory analysis of the GuardTools system and an operator interview.
2. **Artifact design and development**, covering both the simulation environment and the LLM strategies.
3. **Evaluation** through controlled experiments in which all strategies are compared against a rule-based baseline using a shared set of metrics.
4. **Communication** of the results through this thesis.

Each strategy is evaluated on identical alarm sequences under deterministic conditions to isolate the effect of the adaptation method itself.

3.2 Knowledge Acquisition

This section describes the two activities used to build an understanding of the operational domain: an exploratory analysis of the GuardTools system, and an interview with an experienced operator.

3.2.1 Exploratory System Analysis

The first step that was taken was to gather insight on the domain and problem. This was done with a technical examination of the GuardTools system. Access to a dedicated instance of the GuardTools system was given and populated with test data derived from the company's development environment. The analysis primarily focused on the Command and Control (GTCC) application. GTCC is the primary tool within the GuardTools ecosystem used by operators. By interacting with GTCC and its associated modules, the data flow from the moment an alarm is triggered to its final resolution could be mapped. This was useful in seeing which metadata was associated with different alarms and the specific information available to an operator during a dispatch.

To further explore the system, automated "bots" developed by the company were used. These bots simulated the work of guards in the field, allowing for observation of how the system handles geographical proximity and guard availability. This exploratory phase of the GuardTools software system was important in identifying the underlying logic and constraints of the existing workflow, which directly affected the design of the simulation environment that would later be developed for testing purposes. It also gave key details about what data should be provided to the LLM adaptation strategies.

3.2.2 Operator Interview

To better understand the decision-making of an actual operator in the field, an interview with an expert was conducted. The expert in question was an operator with extensive experience using platforms such as GuardTools for over 15 years and with a great understanding of the daily challenges of an operator. The questions were structured around three key objectives: understanding the operator's core responsibilities, understanding how operators make decisions in practice, and gathering realistic parameters for designing the simulation used in this thesis. The specific questions can be found in Appendix B.

3.2.2.1 Operational Responsibilities and Workflow

The first set of questions were aimed at the operator's tasks such as alarm dispatching and managing communications with guards. This provided an overview of the operator's role and helped identify which tasks were the most demanding cognitively and where LLM assistance could be of extra help. To help in prioritizing which areas should be focused on, the operator was asked explicitly what they consider an ideal virtual assistant.

3.2.2.2 Decision-Making Logic and Constraints

A substantial portion of the interview focused on unpacking the thought process behind guard selection for alarm dispatch. The operator was asked to explain the variables that are considered when choosing which guard to send to a specific alarm. Some examples of factors that were taken into account when dispatching an alarm was geographical proximity, current guard availability, and shift status. There were also questions asked about whether guards could be reassigned mid-task (e.g., pulled from a regular patrol to respond to a higher-priority alarm) and whether they would return to complete their original assignment afterward. The interview also investigated the balance between formal protocols and individual judgment.

3.2.2.3 Simulation Calibration Parameters

Finally, several questions were directed specifically at gathering parameters for the simulation environment. These included questions about the typical guard-to-operator ratio, the total number of guards in a given operational area (e.g., Gothenburg), and the frequency of alarm occurrences. This information was key to making sure that the simulation scenarios would reflect realistic conditions.

3.3 Simulation Environment Development

The simulation environment was developed to provide a controlled and reproducible setting in which the LLM adaptation strategies could be evaluated. The environment models the core workflow of the GTCC application, including guard patrolling, alarm dispatching, and guard-operator communication, while omitting aspects of the system that were deemed irrelevant to the research questions. This section describes the architecture of the simulation, the design and generation of the data used within it, and the structure of the test scenarios used for evaluation.

3.3.1 Simulation Architecture and Logic

3.3.1.1 Map and Pathfinding

The simulation renders the central Gothenburg road network, populates it with guard agents who patrol predefined routes, and injects alarm events that the AI copilot listens to and reacts to. The system is implemented in Python, using PyGame for real-time visualisation and the OSMnx library to retrieve and project actual street graphs from OpenStreetMap. Pathfinding uses the A* algorithm, through the OSMnx library, on this graph. Using an actual street network rather than a synthetic grid ensures that guard travel times and routing constraints reflects realistic urban geography.

3.3.1.2 Alarm and Dispatch Logic

Each simulation scenario is configured through three JSON files that define the available guards (with their access keys and patrol routes), the alarm events (with trigger times, types, required keys, and locations), and the patrol points as geographic coordinates. The data design is described further in Section 3.3.2.

When the simulation clock reaches an alarm's trigger time, the simulation pauses to allow the AI copilot to issue a dispatch decision before resuming. A dispatched guard then follows an A*-computed path to the alarm location. Upon arrival, the system checks whether the guard holds all keys required by the alarm. If validation fails, the simulation raises a *dispatch_failed* event, the dispatch is aborted, and the guard returns to its patrol. The copilot then receives this event and may attempt an alternative action, such as dispatching a different guard or ordering a perimeter check. If validation succeeds, the guard remains on site for a configurable time representing incident handling, after which the alarm is cleared and the guard navigates back to its patrol route. At that point the simulation raises a *guard_freed* event, giving the copilot the opportunity to immediately reassign the now-available guard to any remaining active alarm rather than letting it idle back on patrol.

The system supports several dispatch modes. A *full response* performs key validation and resolves the alarm. A *perimeter check* records a physical inspection but does not resolve the alarm, modelling situations where a guard lacks the authority to handle the incident. Guards can also receive *queued dispatches*, which trigger automatically once the guard's current task completes. Finally, *panic alarms* represent high-priority emergencies that take precedence over queued or lower-priority dispatches.

3.3.1.3 Simulation Time Management

The simulation decouples logical time progression from the rendering frame rate via a dedicated time manager that maintains the simulation clock, a configurable speed multiplier, and pause/resume state. When the simulation pauses during an LLM call, a time debt mechanism compensates for the processing latency. Elapsed wall-clock time during processing accumulates as debt and is fast-forwarded once the call completes. This ensures that logical time remains consistent across all experimental conditions regardless of variation in LLM response latency, so that measured response times accurately reflects the copilot's decision quality. This is also done to be able to speed up the simulation to allow for more rapid test runs.

3.3.2 Data Design and Generation

All simulation data was generated programmatically using custom Python scripts, with the goal of producing scenarios that closely mirror real operational conditions observed in the GuardTools system. The following data was generated:

1. **Alarm events** including their patrol schedules. The following is a representative example of a generated alarm event:


```
{
    "id": "alarm_005",
    "sender": "Bank Security",
    "type": "Silent Alarm",
    "info": "Panic button activated at branch office",
    "location_index": 120,
    "trigger_time": 1700.0,
    "required_keys": ["bank", "warehouse"],
    "question_index": 0
}
```


2. **Locations** containing information such as name, location coordinates, description. The following is a representative example of a generated location:


```
{
  "index": 147,
  "name": "Location 147",
  "latitude": 57.7014003,
  "longitude": 11.9758646,
```

```
    "description": "Veterinary Clinic located at Skolvägen 174B in Göteborg. Red brick building. Lobby entrance through the main building entrance. Use intercom system, press code 5076, door will buzz open. Push immediately as lock releases for only 5 seconds.\n\nAlarm control panel is mounted on the wall in the entrance hall, immediately to the right as you enter. White panel with digital display. Code must be entered within 45 seconds of entry to prevent full alarm activation.\n\nPerimeter alarm system on all exterior doors Card reader access system for employee areas 24/7 video surveillance covers all entry points and main areas Smoke and fire detection system integrated with security panel\n\nUse parking garage on sublevel 1, entrance from north side of building Split-level design: Main entrance on ground level, offices on level +1",
    "description_detail_level": "medium",
    "questions": [
      "Should I turn on the lights when I enter?",
      "Where's the main entrance?",
      "Am I being recorded?",
      "Where's the control panel?"
    ]
  }
```

3. **Guard profiles** with location indeces, event types, timestamps when the alarm will trigger. A representative example of a generated guard:

```
    {
        "id": "guard_003",
        "name": "Charlie Unit",
        "keys": ["master"],
        "patrol_location_indices": [8, 16, 24, 32, 40]
    }
```

Location descriptions were procedurally generated at varying detail levels, ranging from minimal entries consisting of only 3 to 4 rows of text covering only essential access information to comprehensive descriptions spanning approximately 1.5 A4 pages including operating hours and on-site hazards. This variability attempts to reflect the uneven data quality encountered in practice, and tests whether the LLM strategies can extract relevant information from sparse or noisy context.

Each location's four questions were generated by prompting Claude Sonnet 4.6 [36] with the location description. The prompt instructed the model to write questions as a guard might type them and covering distinct aspects such as entry codes, alarm panel position, parking, and floor layout. Questions were deduplicated across locations. It should be noted that both the questions and the reference answers are model-generated rather than verified by a human expert. Quantitative evaluation metrics based on similarity to these reference answers should therefore be interpreted with some caution.

Scripts was developed to generate alarm events. Each of the ten alarm types has a location-type preference weight that increases its selection probability when the type matches the location (e.g. intrusion alarms are weighted higher for offices and stores, medical alarms for clinics). The info text is built from templates whose placeholders are filled by extracting location type, and street address from the location description.

Some parameters, such as predetermined alarm trigger times, exist solely for simulation control and have no production counterpart. Others, such as estimated guard arrival times, are computed at runtime.

3.3.3 Test Scenario Design

To enable a reproducible evaluation of the LLM adaptation strategies, the tests were designed to be as deterministic as possible. Each test scenario was seeded with fixed values, ensuring that the same sequence of alarms and system states would occur identically across multiple runs. This determinism was essential for isolating performance differences between the zero-shot, few-shot, and RAG approaches, as well as the rule-based baseline.

The test scenarios fall into four categories. The first is a set of edge case scenarios, each designed to test a specific operational situation with a binary pass or fail outcome. For example, one scenario requires the model to recall an en-route guard to handle a new, closer alarm and then assign a different guard to the original alarm.

The second category uses a Hawkes process [37] to generate alarm sequences at varying intensities, producing workload conditions ranging from low to high pressure. Hawkes process is a self-exciting point process that is used to model clustered or cascading events. These scenarios are evaluated on average guard response time capturing how each strategy degrades under increasing load. The clustering is visualized in Figure 3.1.

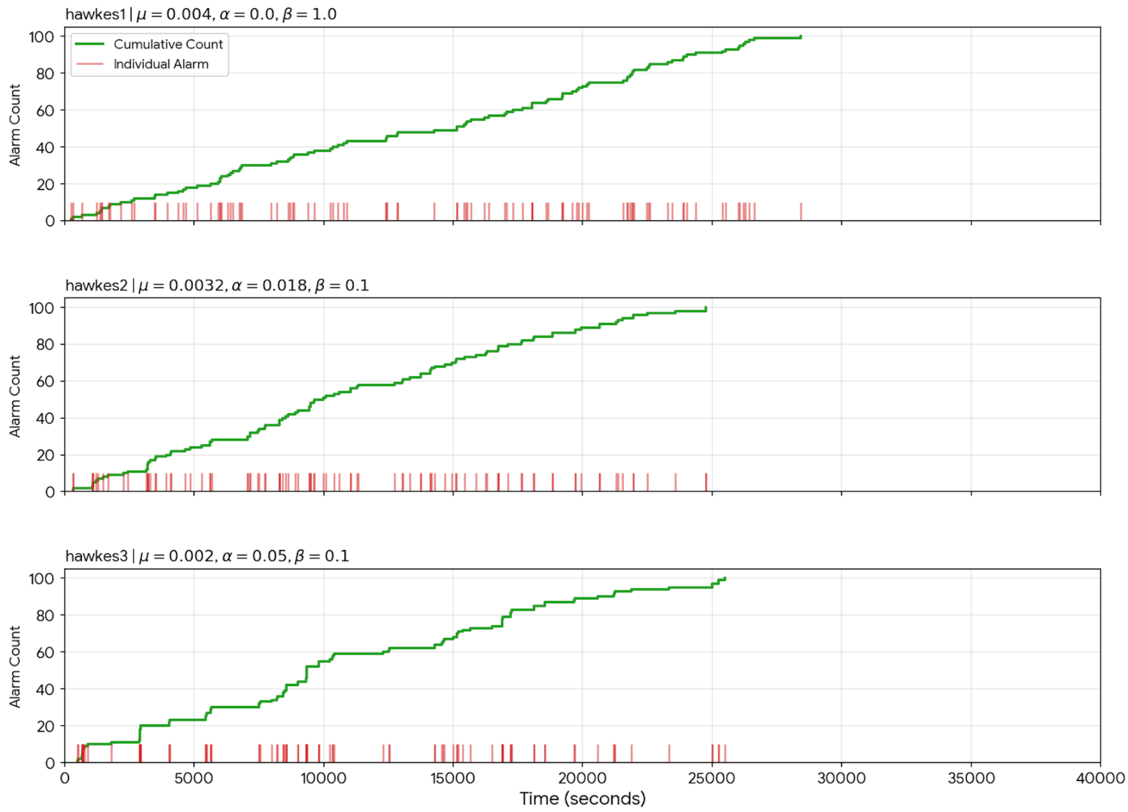


Figure 3.1: Clustering of the alarms in the Hawkes tests.

The third category tests how strategies scale as alarm volume increases against a fixed guard pool. Unlike the Hawkes tests, which vary clustering patterns at a fixed workload, these scenarios push the system toward its limit. The key metric is the open alarm backlog rather than response time. Since an increasing amount of alarms are expected to break the system, response time calculated only over resolved alarms would exclude the worst failures and skew results. Instead, these tests track how the backlog develops over time and how many alarms fail entirely.

The fourth category tests how each strategy handles increasing context window load. A baseline scenario presents five guards with clear alarm assignments, where the correct dispatch is unambiguous. Subsequent scenarios then progressively add larger numbers of irrelevant guard profiles that should never be selected. This tests whether the model can maintain correct decision-making as the input grows and contains more distracting information.

The fifth category evaluates the model's ability to handle guard chat messages. It consists of 18 scenarios containing between 1 and 6 alarms each, with locations and alarm types chosen to cover a broad range of operational settings (banks, hotels, hospitals, retail, industrial sites, residential buildings, and others). When an alarm activates, the corresponding pre-generated guard question is sent as a chat message, and the model's response is compared against the pre-generated reference answer using the chat response quality metrics described in 3.6.

3.4 Implementation of LLM Adaptation Strategies

All LLM strategies use the OpenAI-GPT-OSS-120b model for dispatching and the Llama-3.3-70b-versatile model for guard chats, both accessed through the Groq API [38]. A temperature of 0.1 was used across all strategies rather than 0 to avoid the degenerate output patterns, such as repetitive loops, that pure maximization-based decoding can produce [39]. Each strategy was constrained to return a valid JSON object containing an action, its parameters, and a short reasoning string, enabling direct integration with the simulation loop.

3.4.1 Zero-shot and Few-shot

The zero-shot prompt defined the LLM's role as a security operations dispatcher and dynamically injected JSON-encoded context containing the current event details, active alarms, real-time guard statuses with pre-computed ETAs, and the list of valid actions. The full template is provided in Appendix A.1.

The few-shot strategy extended this by prepending a set of input-output examples to the prompt. These examples covered recurring dispatch patterns identified during the knowledge acquisition phase, such as dispatching a keyholder versus sending the nearest guard for a perimeter check. The goal was to reduce ambiguity by showing the model concrete decision examples rather than relying solely on written instructions. The examples used are provided in Appendix A.3.

3.4.2 Filtered Few-shot

The filtered few-shot strategy is applied exclusively to guard chat responses. It extends the standard few-shot approach with a context reduction step applied before the prompt is constructed. Rather than passing the full location dictionary for all locations in the simulation, the filtered variant extracts only the single entry whose `location_index` matches the alarm associated with the incoming chat message. This ensures that the model receives only the description of the building it is being asked about, rather than the entire location knowledge base. The prompt structure, few-shot examples, and response rules are otherwise identical to the standard few-shot strategy. The motivation is to reduce the number of tokens processed per decision, which directly lowers API cost and latency, while also narrowing the model's attention to the single location that is actually relevant. This strategy was not extended to dispatch decisions because dispatching requires simultaneous reasoning over the full operational state, including all active alarms, guard positions, key inventories, and availability. Removing any subset of this context risks eliminating critical information for its operation.

3.4.3 Retrieval-Augmented Generation

The RAG strategy was developed to provide the LLM with relevant historical context to support decision-making. It implements two distinct retrieval paths, one for dispatch decisions and one for guard chat responses, each tailored to the type of information being retrieved.

Dispatch Retrieval

When an alarm fires, a feature vector is extracted from the current dispatch state. The vector consists of six components, listed in Table 3.1 alongside their similarity weights.

This feature vector is compared against pre-computed vectors stored alongside each entry in a historical dispatch log. Each feature component is scored individually for similarity against the corresponding component in the historical entry, and the total similarity is computed as a weighted sum of these scores. The weights reflect the relative importance each factor had in the dispatch logic identified during knowledge acquisition. Table 3.1 lists the components and their weights.

Table 3.1: Feature components and weights used in the RAG dispatch similarity scoring.

Component	**Description**	**Weight**
Key situation	Whether an available, busy, or no guard holds the required keys	0.35
Current state	Active alarms, available/busy guards, guard deficit, spatial spread	0.25
ETA profile	Sorted arrival times of available and busy guards	0.15
Alarm type	Category of the triggered alarm	0.10
Perimeter check	If there is no available guard with key	0.10
Panic flag	Whether the alarm is a panic alarm	0.05

Key situation receives the highest weight because key possession determines whether a guard can resolve an alarm or is limited to a perimeter check, making it the single most consequential constraint in dispatch. A relevance threshold of 0.4 is applied: if the highest-scoring historical entry falls below this value, no examples are injected into the prompt. This threshold was chosen empirically during development by observing that matches scoring below 0.4 consistently described situations too dissimilar to provide useful guidance. When the threshold is met, up to three historical entries are included.

Chat Retrieval

For guard chat responses, retrieval is based on semantic similarity rather than structured features, since chat messages are free-form text that cannot be reliably decomposed into discrete feature categories.

Each location in the simulation has a set of pre-written Q&A pairs covering procedural details such as access codes, entry procedures, and on-site instructions. These pairs form the retrieval pool. To prevent data leakage, the question currently being evaluated is always held out from the pool so that the model cannot simply retrieve its own ground-truth answer.

When a guard sends a message, the text is encoded into an embedding vector using all-MiniLM-L6-v2 [40] and compared against the pool via cosine similarity. This model was chosen for its low inference latency and strong performance on short-text semantic similarity tasks. The five most similar questions and their corresponding answers are retrieved and injected into the prompt as reference examples.

Prompt Construction

In both retrieval paths, the retrieved examples are appended to the same prompt template used in the zero-shot strategy. This design isolates the effect of retrieved context from other prompt engineering variables, ensuring that any performance difference between the RAG and zero-shot strategies can be attributed to the additional historical information rather than to changes in prompt structure.

3.5 Implementation of Expert System

To provide a deterministic reference point for the comparative evaluation, a rule-based dispatch and message strategy was implemented as a baseline alongside the LLM adaptation strategies. Unlike the LLM strategies, this implementation requires no model inference, produces no probabilistic output, and introduces no latency beyond basic Python execution. Its decisions are fully explainable and perfectly reproducible across simulation runs, making it a stable ground-truth reference for measuring the value added by LLM reasoning.

3.5.1 Alarm Dispatch Logic

The rule-based strategy handles the same events emitted by the simulation as the LLM strategies. It evaluates guards based on status, ETA, distance, and key inventory. A guard qualifies for a full dispatch only if the guard is on patrol and holds every key the alarm requires.

Decisions follow a fixed priority order. Panic alarms are handled first: if an en-route guard can reach the panic alarm faster than any patrol guard and is not already assigned to another panic alarm, the strategy recalls and redirects that guard, then attempts to cover the vacated alarm. Next, if no patrol guard holds the keys needed for a non-panic alarm, the strategy attempts to reroute an en-route perimeter-check guard who does hold them, again covering the vacated alarm where possible. The strategy then evaluates redispatch opportunities, recalling an en-route guard only when the swap improves the worst-case response time across both alarm. If none of these conditions apply, the fastest qualifying patrol guard is dispatched. A faster patrol guard may be sent ahead for a perimeter check.

3.5.2 Chat Handling Logic

When a chat message is received from a guard, the strategy classifies the question by scoring it against regular expression based categories such as entrance code, lockbox, alarm panel location, keys required, parking, and floor layout. The location description is parsed into labelled sections covering entrance information, alarm panel details, security systems, and spatial layout. The highest-scoring category determines which section or extracted detail is returned. If no location description is available, the strategy replies with an instruction to contact dispatch.

3.6 Evaluation Metrics

All strategies are evaluated on the metrics summarised in Table 3.2. Dispatch quality is measured by response time (the elapsed time from alarm activation to guard arrival), reported as mean, median, P95, and IQR to capture both typical and tail-case performance. Panic-response time and perimeter-check time are reported separately where the distinction is operationally relevant. Execution failures record the share of structurally valid actions rejected by the simulator. Token usage and LLM response latency are recorded per alarm for dispatch tests and per run for chat tests. Edge-case scenarios use binary pass/fail assertions rather than distributional metrics. For the capacity stress tests, resolution rate and open-alarm backlog are recorded to capture whether the system keeps up with incoming alarms or accumulates unresolved work. The context-window tests measure dispatch accuracy, defined as the share of scenarios in which the model selects the correct guard despite increasing numbers of irrelevant profiles. Chat response quality is assessed by embedding-based cosine similarity against the reference answer (using all-MiniLM-L6-v2) and by an LLM-as-a-Judge score (Claude Opus 4.6, 0–1 scale covering tone, clarity, and helpfulness).

Table 3.2: Summary of evaluation metrics.

Category	Metric	Measurement Goal
Efficiency	Response time (mean, median, P95, IQR)	Time from alarm activation to guard arrival
Throughput	Resolution rate (%)	Share of alarms resolved
Throughput	Open-alarm backlog	Unresolved alarms accumulated over time
Correctness	Edge-case assertions	Binary pass/fail on targeted dispatch rules
Correctness	Dispatch accuracy (%)	Correct guard selected
Reliability	Execution failures (%)	Share of returned actions rejected by simulator
NLP Quality	Cosine similarity (embedding)	Semantic alignment with reference answers
NLP Quality	LLM-as-a-Judge (0–1)	Qualitative score for tone, clarity, and utility
Cost	LLM latency	Time to generate a decision or reply
Cost	Token count	Tokens consumed per alarm or per run

4
Results

This chapter presents the main results from the simulation runs. The dispatch results cover three test families. The Hawkes tests measure how strategies handle clustered alarm arrivals at realistic scale. The edge-case tests are small targeted scenarios that check specific dispatch rules, such as panic priority, key constraints, and redispatch behaviour, through pass/fail assertions rather than statistics. The context-load tests examine how adding irrelevant guards to the prompt affects dispatch correctness, cost, and latency. The guard chat results are presented separately because they evaluate response quality rather than dispatch behaviour.

For the dispatch experiments, an execution failure means any action returned by a strategy rejected by the simulator, such as recalling a guard with no target alarm or dispatching a guard who is already busy.

4.1 Hawkes Tests

The Hawkes tests measure how the strategies handle longer runs with different alarm clustering. The three scenarios use the same workload size and guard setting, but differ in the clustering produced by the Hawkes process. Each scenario contains 100 alarms and 30 guards. Ten alarms in each scenario are panic alarms, and 80 alarms require one or more keys. The point of these tests is to see how the strategies manage resolving alarms when they arrive close together, when guards are already busy, and when the system has to recover after an overwhelming influx of alarms.

All alarms were resolved in all of the tests, so the differences are in response speed, token cost, and execution failures of the returned actions. The rule-based strategy is fastest on every latency measure, with a mean of 10.31 minutes, a P95 of 23.24 minutes, and the tightest IQR at 9.59 minutes. Among the LLM strategies, few-shot (13.26 min mean) and RAG (13.83 min mean) come closest to this. Execution failures vary widely between the LLM strategies. Few-shot (1.82%) is better than the rule-based baseline at 2.32%, but zero-shot (18.90%) and RAG (14.44%) reject a much larger share of their returned actions. RAG also costs roughly twice as much per alarm as the other LLM strategies, at 21,731 tokens versus the 9,300–11,000 range, due to the added context.

Table 4.1: Aggregate Hawkes dispatch summary across all three scenarios and both repetitions. Response latency values are minutes.

Strategy	Mean	Median	P95	IQR	Tokens/alarm	Execution failures (%)
Rule-based	10.31	9.15	23.24	9.59	0.00	2.32%
Zero-shot	15.70	10.42	46.22	13.82	9315.17	18.90%
Few-shot	13.26	10.16	37.20	13.44	11042.51	1.82%
RAG	13.83	11.36	34.16	11.45	21731.08	14.44%

Figure 4.2 separates panic response and perimeter check latency. For panic response, the rule-based strategy has both the lowest median and the tightest P95 among all strategies. Zero-shot show the longest panic response tail, with P95 values extending past 65 minutes. This shows that under heavy clustering this strategy leaves individual panic alarms waiting far longer than its medians suggest. For perimeter checks, RAG has the lowest median and P95 of the LLM strategies, while zero-shot again show the widest spread between median and tail. Note that perimeter checks are only recorded when a strategy chooses to dispatch a perimeter guard, so sample counts vary across strategies. The rule-based strategy triggered roughly 50–80% more perimeter checks than any LLM strategy, which should be considered when comparing absolute perimeter check times.

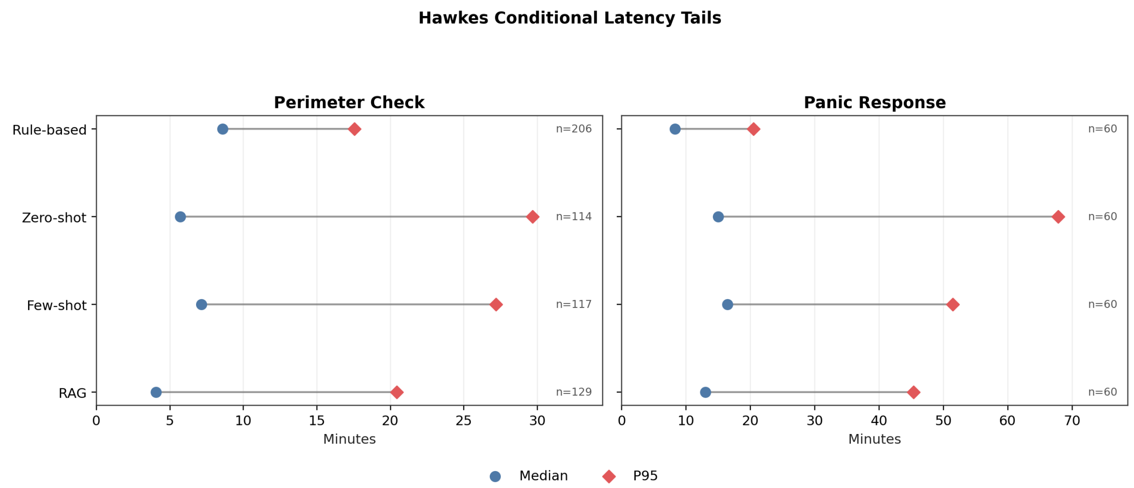


Figure 4.2: Panic-response and perimeter-check latency in the Hawkes runs.

Table 4.2 shows that the aggregate ranking does not hold across all three scenarios. Zero-shot has its largest tail in Hawkes 3, where the P95 jumps to 81.28, far above any other strategy in any scenario. In Hawkes 1, zero-shot is closer to rule-based, which suggests that the clustering pattern in Hawkes 3 triggers the tail rather than the strategy being uniformly slow. RAG is rarely the fastest LLM strategy in any single scenario, but it is comparatively consistent.

Table 4.2: Hawkes response latency and execution failures by scenario and strategy. Execution failures are rejected actions as a percentage of all returned actions.

Scenario	Strategy	Mean	Median	P95	IQR	Exec. fail. (%)
Hawkes 1	Rule-based	9.92	8.75	23.35	9.38	1.99%
Hawkes 1	Zero-shot	10.91	8.84	29.07	12.13	16.84%
Hawkes 1	Few-shot	12.57	9.32	33.11	11.47	2.65%
Hawkes 1	RAG	13.15	10.30	33.32	11.48	12.54%
Hawkes 2	Rule-based	10.51	9.64	21.46	8.90	3.21%
Hawkes 2	Zero-shot	15.28	11.57	39.72	13.51	11.48%
Hawkes 2	Few-shot	14.16	10.83	41.80	14.87	1.23%
Hawkes 2	RAG	15.04	12.53	34.71	12.02	17.07%
Hawkes 3	Rule-based	10.49	9.81	23.09	10.82	1.74%
Hawkes 3	Zero-shot	20.92	11.79	81.28	16.96	26.62%
Hawkes 3	Few-shot	13.05	11.17	31.92	13.99	1.70%
Hawkes 3	RAG	13.31	10.82	32.35	11.14	13.56%

Figure 4.3 show the mean response time of each individual run within a scenario. The label on each strategy reports the gap between the two run means in minutes, making it possible to see whether a scenario average is supported by both repetitions or pulled by a single run. A small number of very late alarms is operationally important, and the mean captures that tail effect more clearly than the median.

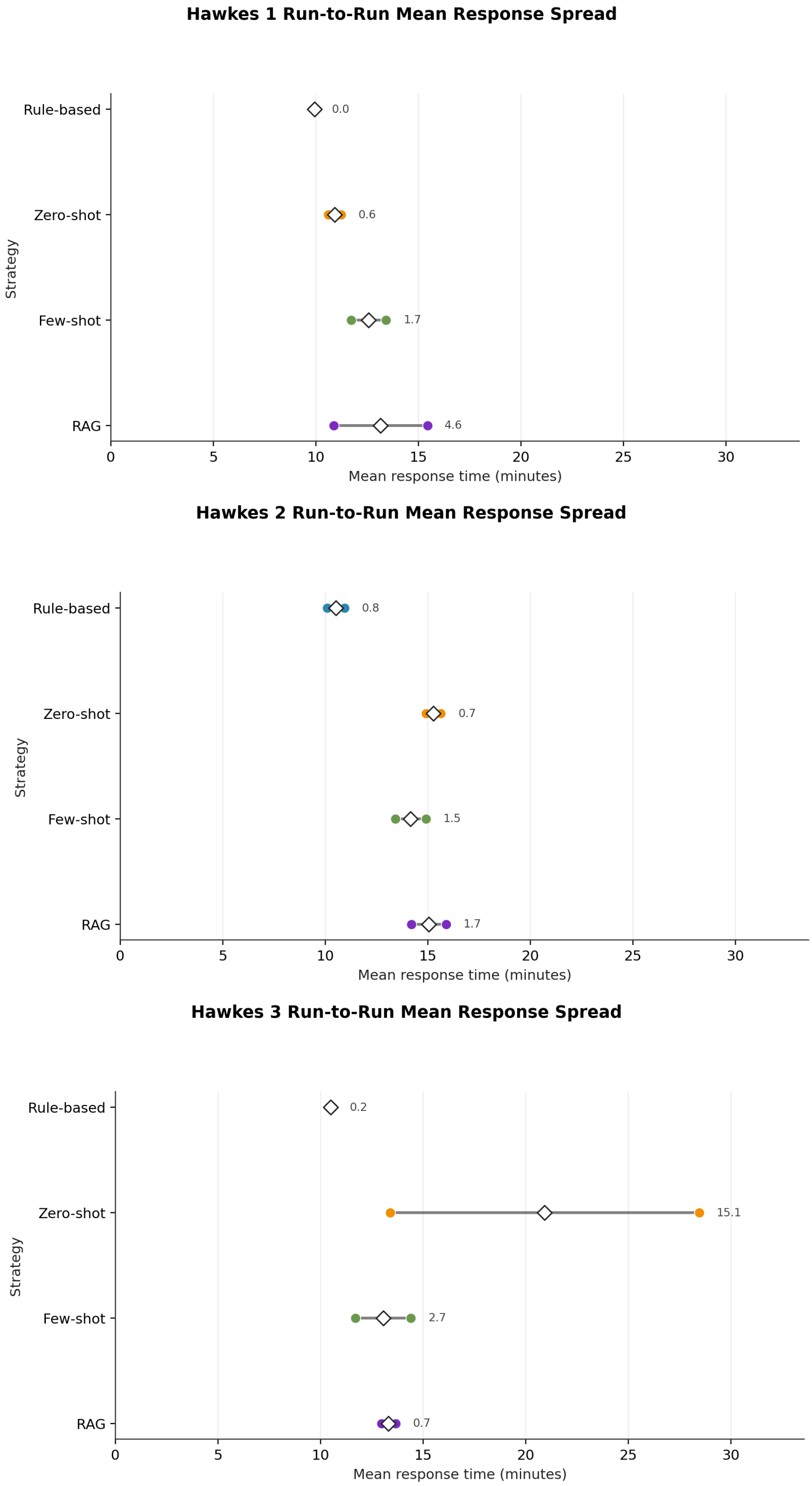


Figure 4.3: Run-to-run spread in mean response time across the three Hawkes scenarios. Colored circles show the two run means, white diamonds show their average, and labels show the run-to-run range in minutes.

Most run pairs cluster tightly across the three scenarios with some obvious exceptions. Hawkes 1 is the most compact with every strategy staying within a 4.6 minute run-to-run range, and the rule-based baseline returns identical means across both runs. In Hawkes 2, rule-based, zero-shot, few-shot, and RAG all stay within 1.7 minutes. Hawkes 3 shows the widest zero-shot spread of any scenario, 13.4 to 28.5 minutes. Few-shot, RAG, and the rule-based strategy remain comparatively stable.

4.2 Capacity Stress Tests

The capacity stress tests evaluate how the strategies behave when the total alarm volume is increased while the guard pool remains fixed. The scenarios contain 100, 200, and 300 alarms respectively, all with 30 guards. Unlike the Hawkes tests, which compare different clustering patterns at the same workload size, these tests are intended to push the simulated dispatch system toward its breaking point.

Since unresolved alarms appear in the larger stress tests, response time is less central here than in the Hawkes tests. Response time is only defined for alarms that are eventually resolved, which means it can hide the worst cases once a strategy leaves alarms open at the end of the run. The capacity results therefore focus on how many alarms were resolved, how large the open backlog became, and how many returned actions were rejected by the simulator.

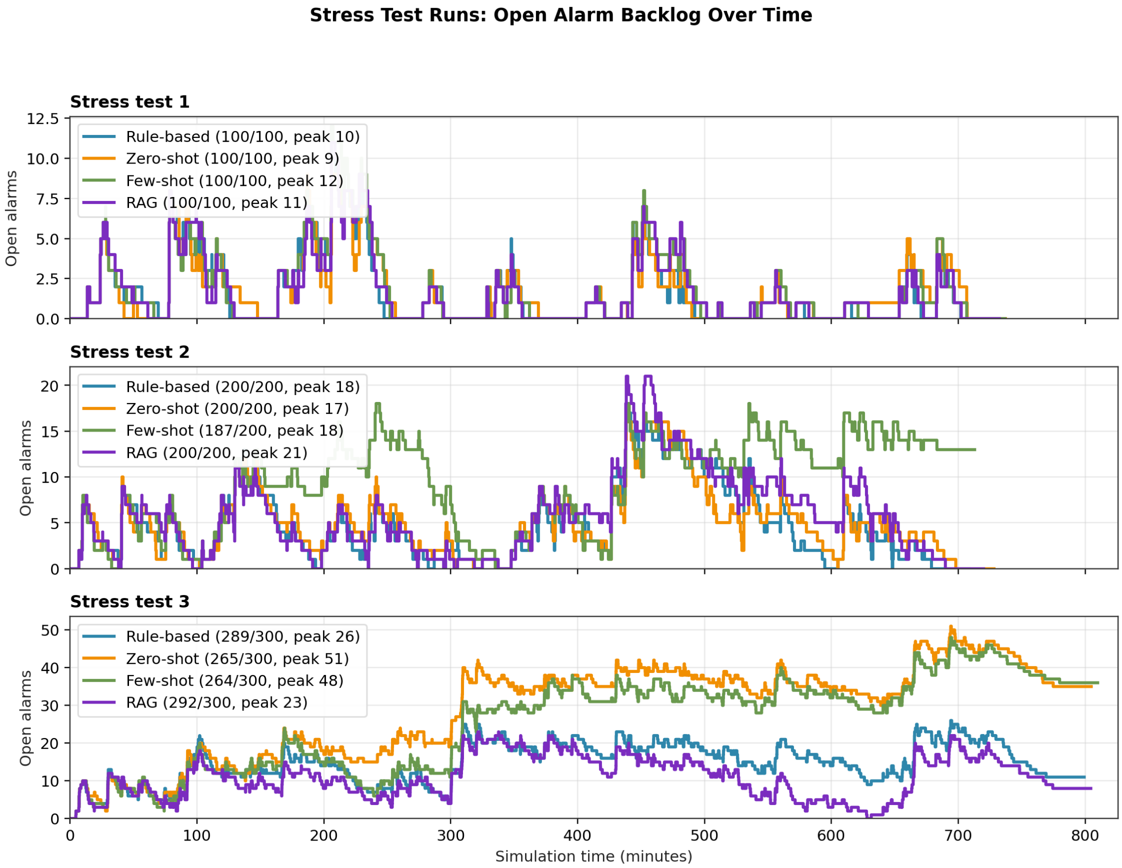


Figure 4.4: Open-alarm backlog over time in the capacity stress tests.

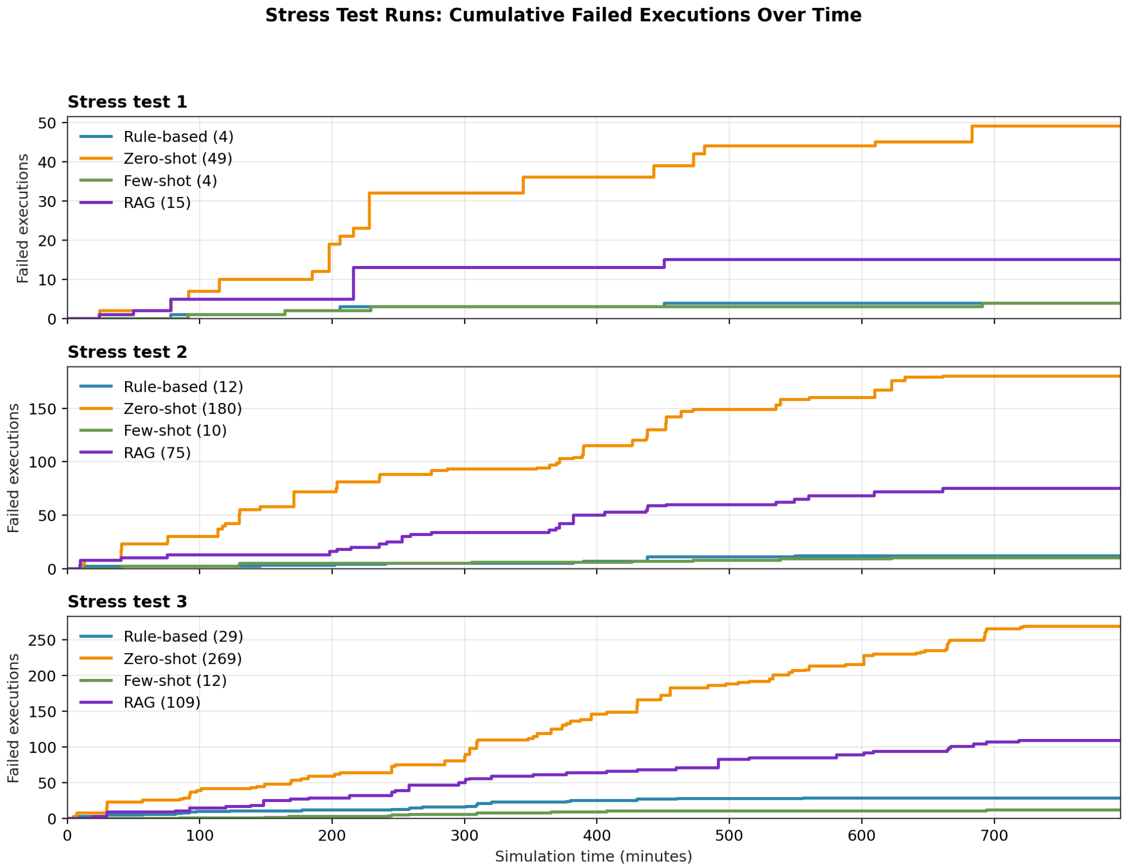


Figure 4.5: Cumulative failed executions over time in the capacity stress tests.

Stress test 1 is within capacity for all four strategies. Every strategy resolves all 100 alarms and the backlog curves in Figure 4.4 are visually indistinguishable. At this scale, the only important distinction is in their failed executions. As seen in Figure 4.5, zero-shot accumulates 49 failures compared to 4 each for few-shot and rule-based.

In Stress test 2, few-shot is the only strategy that leaves alarms unresolved, yet it has the fewest failed executions. The backlog plot shows that few-shot's open count rises steadily in the second half of the run and never recovers, while zero-shot clears its backlog entirely, despite accumulating 180 failed executions. The logs for few-shot strategy show several instances of alarms receiving a full dispatch but then redispatching the same guard to another alarm, leaving the first unresolved. This happened several times, sometimes moving around the same guard several times before it managed to reach and resolve an alarm. So, while zero-shot attempted many more invalid actions, it continued producing enough successful responses to eventually clear the backlog. This is likely an issue of the added examples in the few-shot prompt adding weight to the redispatch action.

In Stress test 3 the alarm frequency becomes unmanageable for all strategies. Rule-based and RAG keep the backlog moving up and down throughout the run, while zero-shot and few-shot both start falling behind around the midpoint and never recover. Zero-shot attempts far more invalid actions than any other strategy, wasting guard time on dispatches the simulator rejects. Few-shot seldom issues an invalid action but undermines itself in the same way as in Stress test 2. Rule-based ends with fewer unresolved alarms, but these should be interpreted with caution. Log inspection showed that the rule-based implementation could queue the same guard for two different alarms, causing the second queued dispatch to overwrite the first. The first alarm was then left without a pending response, and since strategies are only triggered by new alarms or freed guards, these forgotten alarms could only be recovered one at a time through later events. The unresolved rule-based alarms in Stress test 3 are therefore likely caused by an implementation bug in the queuing logic. The plateau seen in Stress test 2 and 3 is caused by the simulation exiting due to no new alarms being resolved for 30 minutes.

4.3 Edge-Case Tests

The edge-case tests cover different scenarios such as unavailable guards, cascading alarms, key constraints, panic priority, and redispatching. A scenario passes only if all predefined expected-outcome assertions pass. Each strategy was run 5 times for each scenario, producing the same result each time. Unlike the Hawkes tests, these scenarios are intended less of a performance benchmark and more as diagnostics. Each scenario isolates one dispatch rule and checks it with direct assertions on the resulting actions. The purpose is to confirm that each strategy handles the targeted behaviour correctly. Table 4.3 shows the result for each scenario and strategy. All four strategies pass all eight scenarios, which means that on this targeted suite no strategy violates the rules being checked.

Table 4.3: Edge Cases: Pass/Fail Detail

Scenario	**rule_based**	**zeroshot**	**fewshot**	**rag**
cascading-alarms	1/1	1/1	1/1	1/1
key-priority-over-speed	1/1	1/1	1/1	1/1
no-guard-has-required-keys	1/1	1/1	1/1	1/1
panic-alarm-immediate-response	1/1	1/1	1/1	1/1
partial-keys-insufficient	1/1	1/1	1/1	1/1
redispatch-beneficial	1/1	1/1	1/1	1/1
redispatch-not-triggered	1/1	1/1	1/1	1/1
three-keys-required	1/1	1/1	1/1	1/1

4.4 Context Window Load

The context load tests check whether adding irrelevant guard profiles to the prompt causes the strategies to make worse dispatch decisions. The base scenario contains five alarms and five eligible guards for dispatch. The scaled versions keep these fixed and pad the visible guard list to 10, 15, 30, 50, 100, and 250 guards.

Figure 4.6 shows how average guard response time changes as distractors are added. All strategies except for RAG remain flat across all context sizes. RAG has the same performance as the other LLM strategies in the base case, but worse in the others. This is because it prefers to send more perimeter checks which displaces the guards that need to be dispatched later, causing a higher response time. However, despite this it can be said that for all strategies the context size does not affect the performance.

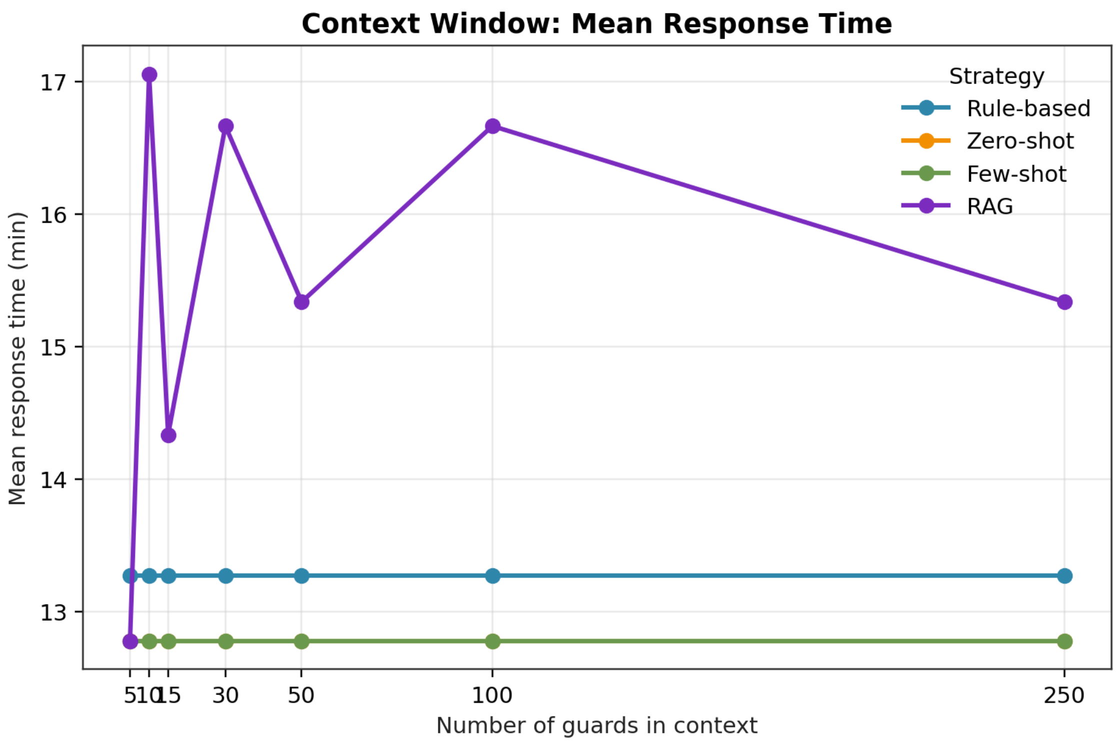


Figure 4.6: Average response time per scenario under context load scaling.

Since the purpose of this experiment is to detect whether increasing context size causes degradation in dispatch quality rather than to estimate precise performance figures, a single run per configuration is used. At an LLM temperature of 0.1 stochastic variation is low but not absent, so the reported values should be interpreted as indicative of whether context scaling introduces degradation, not as exact performance estimates.

Figure 4.7 shows that request latency does not track token count linearly. All LLM-based strategies show large increases in token count across the tested range, yet their latency grows only moderately. The rule-based strategy sits near zero on both axes, as expected.

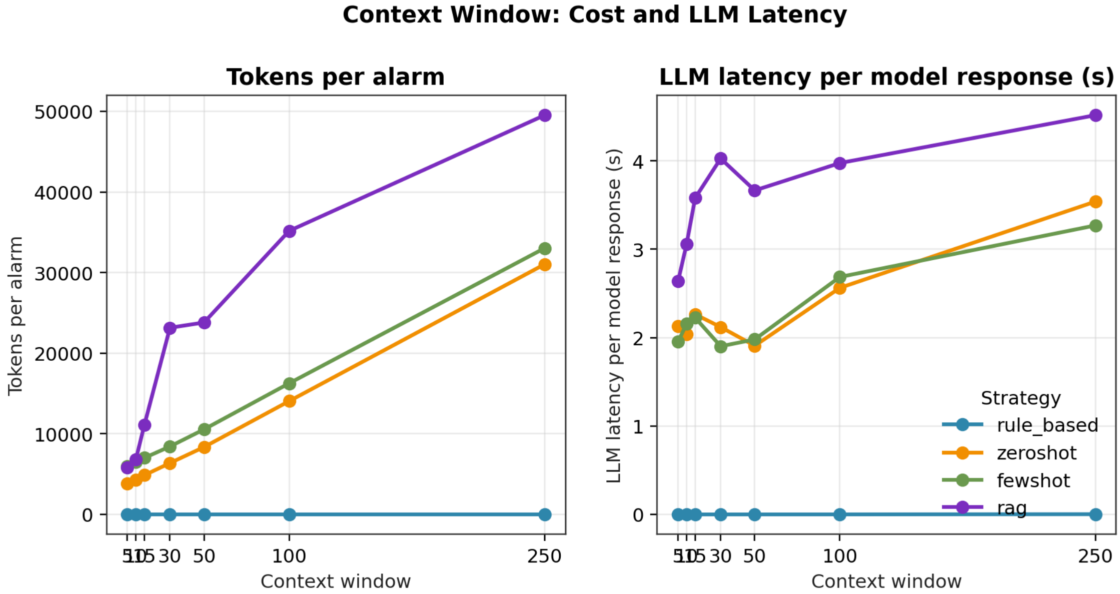


Figure 4.7: Token cost and LLM latency under context-load scaling.

4.5 Chat Response Quality

To evaluate how each adaptation strategy handles guard chat messages, the 18 chat scenarios described in section 3.3.3 were run across all four LLM strategies and the rule-based baseline. Each generated guard question was compared against the pre-generated reference answer using both semantic similarity and an LLM-as-a-Judge evaluation. Latency and token usage were recorded in parallel.

4.5.1 Semantic Similarity

Strategy	Avg Chat Similarity (cosine, embedding)
Few-shot	0,604
Zero-shot	0,604
Filtered Few-shot	0,696
RAG	0,686
Rule-based	0,648

Table 4.4: Average chat response similarity (cosine, embedding-based) across adaptation strategies.

Semantic similarity scores between each strategy's response and the reference answer are shown in Table 4.4. Filtered few-shot achieved the highest score (0.696), followed closely by RAG (0.686) and the rule-based baseline (0.648). Standard few-shot and zero-shot tied at the lowest position (0.604 each). The spread between the best and worst strategy is approximately 0.09, indicating that the choice of adaptation method has a measurable but moderate effect on semantic alignment with the reference answer.

4.5.2 Qualitative Scoring (LLM-as-a-Judge)

Strategy	Avg LLM Judge Score (0–1)
Filtered Few-shot	0.820
RAG	0.810
Rule-based	0.686
Zero-shot	0.661
Few-shot	0.656

Table 4.5: Average LLM-as-a-Judge score across adaptation strategies, evaluated by Claude Opus 4.6 on a 0–1 scale.

The LLM-as-a-Judge results, shown in Table 4.5, follow the same overall ranking but show a wider spread. Filtered few-shot (0.820) and RAG (0.810) lead by a clear margin, with the rule-based baseline (0.686), zero-shot (0.661), and few-shot (0.656) trailing behind. The gap between the top two strategies and the rest is approximately 0.13, substantially larger than the corresponding gap in the similarity scores. This suggests that the qualitative aspects (covering tone, clarity, and helpfulness) is more sensitive to differences between strategies than purely embedding-based semantic similarity.

4.5.3 Token Usage and Latency

Strategy	Avg Tokens per Run	Avg Tokens per Request
Few-shot	457,443	66,403
Zero-shot	454,132	65,922
Filtered Few-shot	10,240	1,486
RAG	10,412	1,511
Rule-based	—	—

Table 4.6: Average tokens used per chat run and per chat request across adaptation strategies. A run is one scenario containing multiple guard chat exchanges; a request is a single LLM call triggered by a guard message. The rule-based strategy consumes no tokens as it does not invoke an LLM.

Strategy	Avg Latency per Decision (s)
Few-shot	8.29
Zero-shot	8.19
Filtered Few-shot	0.41
RAG	0.40
Rule-based	0.00

Table 4.7: Average LLM latency per decision across adaptation strategies. The rule-based strategy operates in effectively zero time as it does not invoke an LLM.

Token usage and latency, shown in Table 4.6 and Table 4.7, reveal a great divide between strategies. Few-shot and zero-shot consume an average of 457,443 and 454,132 tokens per run respectively, more than an order of magnitude higher than filtered few-shot (10,240) and RAG (10,412).
Latency follows the same pattern: few-shot and zero-shot average over 8 seconds per decision, while filtered few-shot and RAG remain below half a second (0.41 and 0.40 seconds respectively). The rule-based strategy operates in effectively zero time. Notably, the two top-performing strategies on quality (filtered few-shot and RAG) are also the two most efficient on both cost and latency, while the two worst-performing strategies on quality (few-shot and zero-shot) are also the most expensive.

4.5.4 Example Responses

To show how the scores translate into actual responses, five example outputs are shown below. Each example shows the guard's question, the reference answer, the response from the strategy, and the similarity and judge scores for that response.

`fewshot_filtered / alarm_001` **judge 1.00 / sim 0.855**

Q whats the entry situation at Storavagen 191?

A The entrance is unlocked, pull the door firmly to open as it is heavy and self-closing. The front entrance faces Skolstigen 137.

Ref. Front entrance faces Skolstigen 137 - door is unlocked. Give it a firm pull, it is heavy and self-closing.

Judge The AI answer includes all critical details: the entrance is unlocked, the door requires a firm pull, it is heavy and self-closing, and the front entrance faces Skolstigen 137.

Figure 4.8: Example response generated by the filtered few-shot strategy.

Figure 4.8 shows a typical filtered few-shot response. The strategy correctly extracts the relevant procedural detail from the location description and presents it in the concise, operator-style tone matching the reference answer.

`zeroshot / alarm_004`	**judge 0.50 / sim 0.465**
Q	anything to watch out for on site?
A	The location index for alarm_004 is 50. For Location 50, the entrance code is 6612. Use the intercom system at the main building entrance, press code 6612, and push immediately as the lock releases for only 5 seconds. The alarm panel is behind the reception desk in the main lobby, concealed in a cabinet under the desk. It is a white rectangular panel with a backlit display.
Ref.	It is a daycare, two floors. Check lobby first, panel is your first stop under the reception desk. Building is all white so easy to spot from street.
Judge	The AI provided accurate details about the entry code, intercom procedure, and alarm panel location, but failed to mention critical site-specific context such as it being a daycare center, the two-story layout, and the white building exterior, and also included irrelevant internal index references that do not help the guard.

Figure 4.9: Example response generated by the zero-shot strategy.

Figure 4.9 shows a representative zero-shot response. While the answer contains accurate procedural detail such as the entrance code, intercom procedure, and alarm panel location, it also includes irrelevant internal artifacts like the location index reference, which are meaningless to a guard and not present in the reference answer. The response also omits site-context that the reference answer prioritises, such as the building being a two-floor daycare. This combination of over-informing on internal metadata while under-informing on operational context is consistent with the strategy's lower judge scores.

`rule_based / alarm_001` **judge 1.00 / sim 0.606**

Q	where's the alarm panel?
A	Alarm control panel is mounted on the wall in the entrance hall, immediately to the right as you enter. White panel with digital display. Code must be entered within 60 seconds of entry to prevent full alarm activation.
Ref.	Entrance hall, immediately to your right as you walk in. White panel with digital display. You have 60 seconds from entry to punch in the code.
Judge	The AI answer contains all critical details from the expected answer: location in the entrance hall, immediately to the right, description as a white panel with digital display, and the 60-second time constraint, with perfect factual accuracy.

Figure 4.10: Rule-based response handled correctly.

`rule_based / alarm_002` **judge 0.40 / sim 0.644**

Q	anything i should know before going in? parking etc
A	Side entrance accessible from the parking area.
Ref.	Parking is available at the side of the building near the entrance. Historic 1920s building so expect older layout. No other noted hazards on file.
Judge	The answer mentions the parking area near the side entrance, which is partially correct, but omits the historic 1920s building context and provides minimal actionable detail about parking or what to expect.

Figure 4.11: Rule-based response with a partial category match.

`rule_based / alarm_003`	**judge 0.30 / sim 0.543**
Q	anything specific to watch for on arrival?
A	Alarm control panel is behind the reception desk in the main lobby. Panel is concealed in cabinet under desk. Open cabinet door, panel is beige plastic with LED display. Storage Facility located at Industrivagen 41 in Goteborg. Red brick building. Main entrance on the west side. Main glass double doors, accessible during business hours. After hours, use access code 4535 on the electronic keypad.
Ref.	It is a storage facility so watch for unauthorized vehicles or open loading areas. Main entry is the glass double doors on the west.
Judge	The AI dumped the entire location description including sensitive information like the access code and alarm panel details without being asked, failed to provide arrival-specific watchpoints like unauthorized vehicles or open loading areas, and did not tailor the response to the guard's question about what to watch for on arrival.

Figure 4.12: Rule-based response that falls outside the defined extraction categories.

Figure 4.10, Figure 4.11, and Figure 4.12 show three responses from the rule-based strategy that together illustrate the full range of its behaviour. When a question maps cleanly to one of the seven pattern-matching categories, the strategy returns a focused and accurate response drawn directly from the location description. When a question partially matches a category, the response can be relevant but incomplete. When a question falls outside the defined categories, the strategy either returns a generic fallback response or fails to identify the relevant information entirely. This spread explains how the rule-based strategy can achieve a respectable aggregate score while still failing on a meaningful fraction of questions.

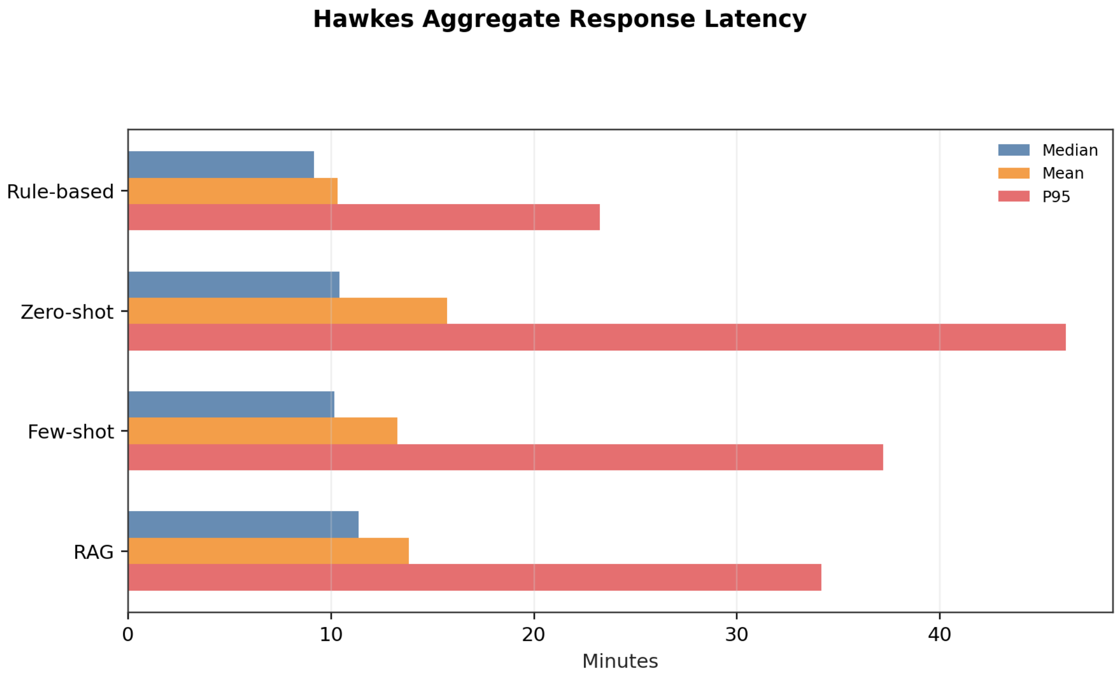


Figure 4.1: Aggregate Hawkes response latency across all runs.

5
Discussion

5.1 Dispatch Evaluation

All four strategies resolved every alarm in the Hawkes runs and passed every edge-case scenario, so the comparison rests on response-time tails, token cost, and execution failure rate. The capacity stress tests extend this picture by revealing how each strategy degrades with an overwhelming alarm volume.

The rule-based strategy was fastest on every latency measure (10.31 min mean, 23.24 min P95) and essentially flat across scenarios and runs. Under the highest capacity it was the second best, resolving 289/300 alarms in Stress test 3. Although, as noted in the results, some of the unresolved alarms appear to stem from a queueing bug rather than a fundamental limitation of the strategy.

Among the LLM strategies, few-shot and RAG trade off on different dimensions while zero-shot lags both. Neither dominates the other across the full set of metrics. Few-shot had the lowest execution failure rate of any strategy in the Hawkes runs (1.82%), roughly half the per-alarm token cost of RAG, and the best mean and median response times among the LLM strategies. The concrete demonstrations of valid dispatch actions in the prompt clearly grounded the model's outputs. However, few-shot was the only strategy that left alarms unresolved in Stress test 2, and in Stress test 3 it resolved the fewest alarms overall, despite having the fewest failed executions of any strategy. The logs show few-shot repeatedly redispatched guards away from alarms they were already en route to, shuffling the same guard across multiple alarms before any was resolved. While the added examples in the prompt helped produce more structurally valid responses, it appears to have given excessive weight to the redispatch action, causing the strategy to make individually valid but collectively bad decisions. Few-shot's low execution failure rate is therefore misleading as a standalone reliability metric.

Zero-shot performed well under light load (Hawkes 1 mean of 10.91 min, within one minute off the baseline) but degraded sharply as clustering and volume increased. In Hawkes 3 its P95 reached 81.28 min and its execution failure rate hit 26.62%, the worst of any strategy in any scenario. Under stress (Stress test 3) it accumulated 269 failed executions and left 35 alarms unresolved. Yet in Stress test 2, zero-shot cleared its entire backlog despite 180 failures, while few-shot with far fewer failures did not. Zero-shot wastes many attempts on invalid actions but keeps producing enough successful ones to make progress.

RAG had the second-worst execution failure rate at 14.44% and roughly doubled the per-alarm token cost. In exchange, it produced a tighter response time distribution than few-shot in the Hawkes runs, with lower P95 and IQR across all three scenarios, and resolved the most alarms of any strategy (292/300) in Stress test 3 with the lowest peak backlog. Unlike few-shot's valid but collectively worse decisions, RAG's execution failures can be caught before reaching the operator, so the raw rate overstates their operational impact.

The context-load tests showed no systematic degradation in dispatch quality as irrelevant guard profiles were added up to 250. All strategies continued selecting the correct key-holding guards at the largest context size, and three of the four kept response times flat. RAG was the exception, at several larger context sizes it issued more perimeter checks, which displaced guards needed for later alarms. However, this does not appear to be a result of the model being overwhelmed by context length because it was still able to identify the correct key-holding guards. The issue was not general decision failure, but an overuse of perimeter checks.

5.1.1 Practical Implications

The results suggest that for a dispatch domain with a stable and well-defined rule space, an LLM does not improve on a rule-based strategy. The rule-based approach was faster, cheaper, deterministic, and matched or exceeded the best LLM failure rate. The case for using an LLM strategy becomes stronger when the rule space is harder to maintain. For example, if dispatch logic must account for soft constraints that are difficult to encode, then the flexibility of a prompted model offers practical value. Similarly, when the system must handle new situations, adapting a prompt is likely easier than updating the copilot system itself. In this case, RAG is likely the most appropriate strategy because it has a shorter response time tail, degrades more gracefully under sustained load, and fails in ways that can be caught before reaching the operator. Few-shot should only be considered when high alarm clustering is rare and cost is an important constraint. This comparison describes the strategies as implemented, not their best under further tuning. Few-shot's redispatch behaviour may be solved through tweaking the prompt examples, and RAG's failure rate may similarly be reduced through changes to retrieval or base prompt.

5.2 Chat Evaluation

The chat results show a clear divide driven by how context is managed. Filtered few-shot and RAG led on both the LLM judge (0.820 and 0.810) and semantic similarity (0.696 and 0.686). The rule-based strategy sat in the middle (0.686 judge, 0.648 similarity), and zero-shot and few-shot were tied at the bottom (0.604 similarity, around 0.66 on the judge).

The two top strategies both narrow the prompt to the relevant location description before calling the model. They differ only in what accompanies that location, hard-coded Q&A examples for filtered few-shot and dynamically retrieved Q&A pairs for RAG, yet their scores are essentially tied. What matters is therefore the location filtering, not the choice of examples. These two strategies were also both the cheapest.

Zero-shot and few-shot show the opposite. Both include the entire location list for every question, so the relevant description is one entry among many. The gap between filtered few-shot (0.820) and few-shot (0.656) on the judge metric is larger than the gap between filtered few-shot and the rule-based baseline. Adding the full location list did not just cost more, it also degraded the answers. The rule-based strategy sits between these tiers because its pattern-matching cover the questions they were designed for and produce generic fallbacks otherwise.

What separates the strategies here is not whether they use an LLM but how much context they feed it. Filtered few-shot and RAG both isolate the relevant location description before generating a response and come out on top. Zero-shot and few-shot dump the entire list into the prompt and let the model sort it out, landing at the bottom. The rule-based approach falls in between, its pattern matching covers the questions it was built for and falls back to generic responses otherwise. At least when the answer is concentrated in a single entry, narrowing the context does more work than choosing the right prompting technique. The location descriptions in this study were self-contained, but in a settings where the relevant information is scattered across a longer document or split between multiple sources, isolating the right context becomes a harder problem in itself. This might require a substantially bigger context as is done with the zero-shot and few-shot strategies.

5.2.1 Prompt Injection Risk

The chat interface introduces a risk that the dispatch does not face. The guard's chat messages are the only untrusted input in the prompt. The model has no mechanism for distinguishing the guard's text from the surrounding instructions, so a crafted message can attempt to override or bypass them. This is a well-documented weakness of LLM systems [41]. In the context of a security assistant, successful prompt injection can be a severe security risk. An attacker could extract the system prompt itself, revealing the internal instructions which contains the full prompt of potentially sensitive information that it was given as context. Most consequential for the security domain, an attacker could prompt the assistant to reveal additional sensitive information such as guard schedules, response procedures, access codes, building layouts, or details about which guards hold which keys.

5.3 Cost and Practical Feasibility

Token use varies significantly between strategies and tasks. On dispatch, RAG averaged 21,731 tokens per alarm in the Hawkes runs, roughly twice as much as few-shot (11,043). On chat, the spread is much wider with zero-shot and few-shot averaging around 66,000 tokens per chat request, while filtered few-shot and RAG averaged roughly 1,500 each.

In a production environment with roughly 0.5 to 1 alarms per minute, as estimated from the operator interview, a system running an expensive LLM strategy on dispatch would accumulate millions of tokens per shift. If the same system also ran an unfiltered LLM strategy on chat, the chat requests would dominate the bill. At commercial API pricing, the choice between strategies becomes an important operational concern.

5.4 Implications for Cognitive Load and Alarm Fatigue

The theory chapter established that a physical protection system fails when $T_{\text{detect}} + T_{\text{respond}}$ exceeds T_{attack}, and that the human operator is the weakest link in that equation. Vigilance decrement, alarm fatigue, and the Cry Wolf effect all extend T_{respond} by degrading the operator's willingness or ability to act on an alarm. The results from this evaluation speak to whether an LLM-based assistant can counteract those failure modes or whether it risks making them worse.

5.4.1 Importance of Reliability

If the assistant's suggestions are frequently rejected, the operator will stop trusting them, and the assistant becomes additional noise rather than a cognitive resource. This makes the execution failure rate becomes an important metric for evaluating whether the assistant actually reduces workload.

The zero-shot strategy's 18.90% execution failure rate across the Hawkes runs, rising to 26.62% in the most clustered scenario, places it a position where desensitisation becomes likely. An operator who sees roughly one in five suggestions rejected by the system has strong reason to stop trusting the rest. Overload leads to inattentional deafness, where the operator physically registers the assistant's output but stops treating it as meaningful. At that point the assistant is not reducing response time but instead adds a filtering task on top of the operator's existing workload.

Few-shot's 1.82% failure rate sits on the other side of that threshold. It is low enough that the occasional rejected action looks like an exception rather than a pattern. The same can be said for the rule-based baseline at 2.32%. The practical difference between 1.82% and 2.32% is probably too small to affect perceived reliability, but both are categorically different from 18.90%.

However, a low rejection rate is necessary but not sufficient for perceived reliability. The capacity stress tests showed few-shot producing very few rejections while still possibly ruining trust through bad redispatch actions, so strategies with similar failure rates can differ substantially in how often their accepted actions produce useful outcomes.

5.4.2 Reducing Search Load

The literature establishes that sustained monitoring create high mental workload. In a typical alarm centre, answering a guard's question adds to that demand. The operator must find the right information among a potentially vast source of knowledge while simultaneously monitoring incoming alarms.

The chat results show that an LLM assistant can absorb this search cost. For cognitive load, the latency difference can matter as much as the quality difference. An eight-second delay on a chat response during a clustered alarm cascade forces the operator to either wait idle or switch back to dispatch monitoring and re-engage with the conversation later. Either outcome breaks the operator's focus, which worsens vigilance decrement.

5.4.3 The Risk of Misplaced Confidence

These benefits assume the assistant's output is correct. A chat assistant that is accurate most of the time encourages operators to stop verifying its answers, which means any error is more likely to go unnoticed. This creates an inversion of the Cry Wolf dynamic where instead of too many false alarms causing the operator to ignore real ones, too many correct answers cause the operator to accept the one that is wrong. How much workload the assistant actually saves therefore depends on whether the operator is expected to verify its output.

5.5 Threats to Validity

5.5.1 Internal Validity

Each strategy was tested on a single model (OpenAI-GPT-OSS-120b for dispatch, Llama-3.3-70b-versatile for chat), so it is hard to tell how much of the observed differences stem from the prompting technique or from the model itself. With more capable models, the added context-specific information might not be as needed. Zero-shot's high execution failure rate, for instance, could be a property of this particular model rather than a general weakness of zero-shot prompting. Running each strategy on different models would have been needed to separate the two.

5.5.2 External Validity

The simulation models a single city with a fixed guard count and a simplified version of the GuardTools workflow. Real environments involve shift changes, unreachable guards, and false alarms, none of which are modelled. All alarm logs and site descriptions were generated programmatically from templates. Real data may contain issues such as irregular fields, misleading sender information, and outdated or contradictory documentation. Testing on anonymised real data would have strengthened generalisability. The use of a single API provider also ties all latency figures to that platform's hardware. Whether the strategy rankings would hold on different infrastructure or in a live deployment is an open question.

5.5.3 Construct Validity

Execution failure rate measures the share of returned actions rejected by the simulator, not the quality of the accepted ones. Few-shot's low rate in the capacity stress tests masked repeated redispatch decisions that left alarms unresolved, so the metric should not be read as a direct measure of reliability. Chat quality depends on two metrics with known weaknesses. Cosine similarity penalises correct answers that are phrased differently from the reference. The LLM judge (Claude Opus) scores against references generated by Claude Sonnet, so the judge may reward stylistic traits of its own output rather than operational usefulness. The rule-based strategy is most exposed to this bias, since its template-driven replies are the furthest from LLM-style text. A follow-up with human operators would be needed to confirm whether the scores reflect actual operational usefulness.

5.5.4 Conclusion Validity

No statistical significance tests were applied to the results, so all reported differences between strategies should be treated as indicative rather than confirmed. The Hawkes runs used two repetitions per configuration, and the capacity stress tests and context-load tests used a single run. This is enough to detect large effects, such as zero-shot's 18.90% execution failure rate against few-shot's 1.82%, but not to separate small strategy effects from run-to-run variance.

5.6 Future Work

5.6.1 Short-Term

The most immediate next step is strengthening the statistical basis of the current findings. More runs per configuration would separate strategy effects from sampling noise, particularly for the high-variance cases like zero-shot under heavy clustering. Running each strategy on a second model would clarify how much of the observed behaviour is a property of the prompting technique versus the specific model.

Another direction is to test more structured reasoning strategies, such as chain-of-thought-style prompting or prompt chaining. The strategies evaluated in this thesis mostly ask the model to produce a final action from a single prompt. A future implementation could instead divide the decision into several explicit steps: first identify the relevant constraints, then rank eligible guards, then validate the planned action against key possession, guard status, and alarm priority, and finally produce the JSON action. For guard chat, a chained approach could first select the relevant location information, then draft an answer, and then check that the answer does not include irrelevant or sensitive details. This could reduce invalid actions and improve reliability, but it might also increase latency and token usage.

The chat evaluation would benefit from operators familiar with the domain to validate whether the strategy ranking holds under a judge independent of the models used to generate the reference answers. Testing with real operational data would also address the external validity issue, since real data contain noise that the synthetic generator does not produce.

Future work should also extend the simulation to include false alarms. In the current evaluation, every alarm represents an event that should be handled, while the operator interview suggested that real control rooms often face a high proportion of false alarms. This is especially relevant when evaluating how well the strategies scale. Increasing the number of alarms in the simulation only by adding genuine incidents may overstate the operational load compared to a real control room, where a larger alarm volume would likely include many alarms that can be dismissed. Modelling false alarms would therefore make scaling experiments more realistic.

5.6.2 Long-Term

Future studies should also include real operators. A controlled operator study could measure whether the assistant actually reduces cognitive load, improves response behaviour, and maintains appropriate trust. Such a study could compare operators working with and without an AI assistant. Measures such as response time, override rate, and subjective workload would test whether the simulation results translate into operational benefit.

Finally, future work should address the security risks of the chat system. The chat interface introduces prompt injection risk because guard messages are given directly to the model and should not always be trusted. A system in production would need safeguards such as strict separation between the system instructions and user input. This is especially important in physical security, where leaked access procedures, guard schedules, or key information could create real operational risk.

5.7 Use of Generative AI

Some generative AI tools were used for this thesis. The tools were used for improving the language and clarity of the written text, assisting with parts of the code implementation, data generation, and refining the LLM prompts used in the experimental work. All AI-generated output was critically reviewed before incorporation. The authors assume full responsibility for the content of this thesis.

6
Conclusion

This thesis evaluated three LLM adaptation strategies — zero-shot, few-shot, and RAG — against a rule-based baseline on two tasks central to physical security operations: alarm dispatch and guard chat. The evaluation was conducted in a simulation environment modelling GuardTools Command and Control, across scenarios at sustainable load, stress tests, targeted edge-case scenarios, context-load tests, and a chat response quality benchmarks. The contribution is empirical evidence on how the choice of adaptation strategy affects reliability, response time, and cost in this domain, and an interpretation of those results in relation to operator cognitive load.

RQ1: When the dispatch rule space is stable and well defined, an LLM does not improve on a rule-based strategy. The baseline was the fastest, the cheapest, deterministic, and matched the lowest execution failure rate observed. The motivation for using an LLM strategy is only clear when the rule space is hard to maintain or must adapt to new situations without engineering changes. Among the strategies evaluated, RAG is the most appropriate fit in that case. It degrades more gracefully under high load compared to few-shot, and its failures can be intercepted before reaching the operator. Few-shot's apparent advantage on the execution failure metric is misleading, since its low rejection rate masked structurally bad redispatch decisions. This behaviour may be addressable through prompt tuning but was not in the implementation tested. Zero-shot, as implemented, is not a viable choice for this domain.

RQ2: What separated the chat strategies was not the prompting technique but how much context the model received. Filtered few-shot and RAG, which both narrow the prompt to the relevant location description, scored highest on both the LLM judge and semantic similarity while consuming a small fraction of the tokens used by the unfiltered strategies. Zero-shot and few-shot, which kept the full location list in every prompt, scored below the rule-based baseline on the judge metric. The choice between hard-coded and retrieved examples mattered far less than the decision to isolate the relevant context.

RQ3: Whether an LLM assistant reduces operator cognitive load or adds to it depends on how reliably it produces useful outputs. The execution failures for the few-shot strategy and rule-based strategy (1.82% and 2.32% respectively) are low enough that rejected actions are exceptions, while zero-shot's 18.90% is high enough to push the operator toward desensitisation and inattentional deafness. At that point the assistant creates a filtering task layered on top of the existing workload rather than reducing it. Few-shot's behaviour in the capacity stress tests showed that a low failure rate is not sufficient on its own, since structurally valid but bad dispatch decisions by the LLM degrades the assistant's usefulness in practice. For guard chat, the results suggest that filtered few-shot and RAG are the most plausible candidates for reducing operator search load, since they produced higher quality answers with much lower latency than the unfiltered LLM strategies. The study did not directly measure operator cognitive load, so this remains an implication rather than a demonstrated effect. The practical implication is that an LLM assistant only acts as a cognitive offload when it is both reliable enough that the operator stops treating it as a source of noise and fast enough that it does not interrupt the operator's primary task.

Bibliography

[1] S. Freitas, J. Kalajdjieski, A. Gharib, and R. McCann, "Ai-driven guided response for security operation centers with microsoft copilot for security," in *Companion Proceedings of the ACM on Web Conference 2025*, ser. WWW '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 191–200. [Online]. Available: https://doi.org/10.1145/3701716.3715209

[2] R. Singh, S. Tariq, F. Jalalvand, M. B. Chhetri, S. Nepal, C. Paris, and M. Lochner, "Llms in the soc: An empirical study of human-ai collaboration in security operations centres," 2025. [Online]. Available: https://arxiv.org/abs/2508.18947

[3] L. Peng, D. Li, Z. Zhang, T. Zhang, A. Huang, S. Yang, and Y. Hu, "Human-ai collaboration: Unraveling the effects of user proficiency and ai agent capability in intelligent decision support systems," *International Journal of Industrial Ergonomics*, vol. 103, p. 103629, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169814124000854

[4] G. Chen, Y.-D. Zheng, J. Wang, J. Xu, Y. Huang, J. Pan, Y. Wang, Y. Wang, Y. Qiao, T. Lu, and L. Wang, "Videollm: Modeling video sequence with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2305.13292

[5] A. Kermani, V. Perez-Rosas, and V. Metsis, "A systematic evaluation of llm strategies for mental health text analysis: Fine-tuning vs. prompt engineering vs. rag," 2025. [Online]. Available: https://arxiv.org/abs/2503.24307

[6] H. Soudani, E. Kanoulas, and F. Hasibi, "Fine tuning vs. retrieval augmented generation for less popular knowledge," in *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*, ser. SIGIR-AP 2024. ACM, Dec. 2024, p. 12–22. [Online]. Available: http://dx.doi.org/10.1145/3673791.3698415

[7] Z. Tan, X. Zhang, S. Wang, and Y. Liu, "MSP: Multi-stage prompting for making pre-trained language models better translators," 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. 6131–6142. [Online]. Available: https://aclanthology.org/2022.acl-long.424/

[8] Z. Liu, M. Patwary, R. J. Prenger, S. Prabhumoye, W. Ping, M. Shoeybi, and B. Catanzaro, "Multi-stage prompting for knowledgeable dialogue generation," *ArXiv*, vol. abs/2203.08745, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247475908

[9] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 3045–3059. [Online]. Available: https://aclanthology.org/2021.emnlp-main.243/

[10] M. L. Garcia, *The design and evaluation of physical protection systems*. Oxford, England: Butterworth-Heinemann, May 2001.

[11] S.-S. Jang, S.-W. Kwan, H.-S. Yoo, J.-S. Kim, and W.-K. Yoon, “Development of a vulnerability assessment code for a physical protection system: Systematic analysis of physical protection effectiveness (sape),” *Nucl. Eng. Technol.*, vol. 41, no. 5, pp. 747–752, Jun. 2009.

[12] H. Lin, D. Burnett, D. Sheaffer, and E. Arnold, “Applying decision analysis process to exterior physical security system technology design and selection,” in *43rd Annual 2009 International Carnahan Conference on Security Technology*, 2009, pp. 312–316.

[13] R. M. Klein and B. B. T. Feltmate, “The vigilance decrement: its first 75 years,” *Frontiers in Cognition*, vol. Volume 4 - 2025, 2025. [Online]. Available: https://www.frontiersin.org/journals/cognition/articles/10.3389/fcogn.2025.1632885

[14] N. H. Mackworth, “The breakdown of vigilance during prolonged visual search 1,” *Quarterly Journal of Experimental Psychology*, vol. 1, pp. 21 – 6, 1948. [Online]. Available: https://api.semanticscholar.org/CorpusID:146511016

[15] J. S. Warm, W. N. Dember, and P. A. Hancock, “Workload and vigilance,” *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 42, no. 10, pp. 769–771, 1998. [Online]. Available: https://doi.org/10.1177/154193129804201025

[16] S. Breznitz, *Cry wolf*. Mahwah, NJ: Lawrence Erlbaum Associates, Jan. 1984.

[17] M. Cvach, “<i>monitor alarm fatigue</i>: An integrative review,” *Biomedical Instrumentation & Technology*, vol. 46, no. 4, pp. 268–277, 2012. [Online]. Available: https://array.aami.org/doi/abs/10.2345/0899-8205-46.4.268

[18] L. Zhu, S. Wei, Y. An, W. Hu, and X. Xie, “Mechanism, contributing factors, and coping strategies of alarm fatigue in intensive care nursing: a qualitative study,” *Front. Public Health*, vol. 13, no. 1654389, p. 1654389, Oct. 2025.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[20] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162/

[21] K. Hong, A. Troynikov, and J. Huber, “Context rot: How increasing input tokens impacts llm performance,” Chroma, Tech. Rep., July 2025. [Online]. Available: https://research.trychroma.com/context-rot

[22] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3703155

[23] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[24] Z. Wang, J. Kodner, and O. Rambow, "Evaluating llms with multiple problems at once," 2025. [Online]. Available: https://arxiv.org/abs/2406.10786

[25] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[26] Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, "Calibrate before use: Improving few-shot performance of language models," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 12 697–12 706. [Online]. Available: https://proceedings.mlr.press/v139/zhao21c.html

[27] Y. Tang, D. Tuncel, C. Koerner, and T. Runkler, "The few-shot dilemma: Over-prompting large language models," 2025. [Online]. Available: https://arxiv.org/abs/2509.13196

[28] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021. [Online]. Available: https://arxiv.org/abs/2005.11401

[29] S. Barnett, S. Kurniawan, S. Thudumu, Z. Brannelly, and M. Abdelrazek, "Seven failure points when engineering a retrieval augmented generation system," 2024. [Online]. Available: https://arxiv.org/abs/2401.05856

[30] D. A. Waterman, *A Guide to Expert Systems.* Reading, MA: Addison-Wesley, 1986.

[31] D. S. Akira Tsurushima, Kenji Urushima, "An expert system for alarm system planning," in *Proceedings of the Tenth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-98).* AAAI Press, 1998. [Online]. Available: https://cdn.aaai.org/IAAI/1998/IAAI98-021.pdf

[32] P. Legaud, "Processing of alarms by means of an expert system," *Reliability Engineering System Safety*, vol. 22, no. 1, pp. 401–409, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0951832088900865

[33] I. Mikulić, D. Lisjak, and N. Štefanić, "A rule-based system for human performance evaluation: A case study," *Applied Sciences*, vol. 11, no. 7, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/7/2904

[34] S. Russell and P. Norvig, *Artificial intelligence: A modern approach, global edition*, 4th ed. London, England: Pearson Education, May 2021.

[35] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, pp. 45–77, 01 2007.

[36] Anthropic, *Claude*, 2026, accessed: May 5, 2026. [Online]. Available: https://www.anthropic.com

[37] A. Hawkes, “Point spectra of some mutually exciting point processes,” *Journal of the Royal Statistical Society. Series B*, vol. 33, 07 1971.

[38] Groq, Inc., “Groq: Fast, low cost AI inference,” 2026, accessed: May 4, 2026. [Online]. Available: https://groq.com/

[39] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The curious case of neural text degeneration,” 2020. [Online]. Available: https://arxiv.org/abs/1904.09751

[40] Sentence Transformers, “sentence-transformers/all-minilm-l6-v2,” https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2021, accessed: May 4, 2026.

[41] F. Perez and I. Ribeiro, “Ignore previous prompt: Attack techniques for language models,” 2022, neurIPS 2022 ML Safety Workshop. [Online]. Available: https://arxiv.org/abs/2211.09527

A

Appendix 1

A.1 Zero-shot Strategy Dispatch Prompt

The following is the dispatch prompt used in the zero-shot strategy:

Listing A.1: Zero-shot dispatch Prompt

```
  prompt = f"""You are a Security Operations Dispatcher. Decide
     what actions to take based on the context below.

    ## Current Situation
    **Time:** {context.current_time}
    **New Event Type:** {context.event.type.value}
    **New Alarm Details:** {json.dumps(slim_event, indent=2)}

    ## Active Alarms
    {json.dumps(slim_alarms, indent=2)}

    ## Dispatched Guards (currently en route to other alarms)
    These guards CAN be recalled and redirected. Their ETA/distance
        is to THIS new alarm.
{dispatched_summary}

    ## Patrol Guard ETAs to Current Alarm (pre-computed, sorted
       fastest first)
    Use `eta_seconds` (NOT raw coordinates) to determine proximity.
        Already sorted ascending.
{eta_summary}

    ## Available Guards & Status
    {json.dumps(slim_guards, indent=2)}

    ## Available Actions
    {json.dumps(context.available_actions, indent=2)}

    ## Action Constraints
    | Action            | Valid guard statuses
                                                   |
    |------------------|-----------------------------------------------------

    | `dispatch`        | `patrol` only
                                                                  |
    | `queue_dispatch` | `busy`, `dispatched`, `solving_alarm`, `
       perimeter_check`, `checking_perimeter` |
    | `recall`          | `dispatched` or `perimeter_check` only (
```

```
NOT `solving_alarm` / `checking_perimeter`    already on-
    site) |
A `queued` guard already has a pending task for their `
    queued_alarm_id`. Do not re-queue unless the new alarm is
    higher priority.
**PERIMETER CHECK RULE: When dispatching a guard who lacks
    required keys, you MUST include `"is_perimeter_check": true`
     in `parameters`. Writing "perimeter check" in the reasoning
     is NOT enough    the parameter controls the action.
    Omitting it sends the guard for a full response which will
    fail.**
**PERIMETER CHECK LIMIT: A maximum of ONE perimeter check is
    allowed per alarm. If `perimeter_checked=true` in the alarm
    data, do NOT dispatch another guard for a perimeter check to
     that alarm. Ignore that option entirely.**

## Key Resolution Rules (HIGHEST PRIORITY    evaluate BEFORE
    speed)
A guard can only RESOLVE an alarm if they hold **every** key in
     `required_keys`.
Holding *some* keys is not enough. Never dispatch for a full
    response if the guard cannot resolve.
**Empty required_keys rule**: If `required_keys` is `[]` (empty
     list), the alarm requires NO keys at all. Every guard
    including `dispatched` guards en route    qualifies for a
    full response. Do NOT claim a guard "lacks keys" when `
    required_keys` is empty.

## Decision Logic by Event Type

### `alarm`    New alarm fired
For EVERY alarm, after finding the best patrol option, also
    check the Dispatched Guards section for speed re-routing (
    even when a patrol guard qualifies).

1. **Identify who can resolve:** Find all guards (patrol +
    dispatched) that hold all `required_keys` (or alarm has `
    required_keys: []`).
2. **Check for speed re-route:** Look at the Dispatched Guards
    section. If any dispatched guard (en route, not arrived) who
     can resolve has a **significantly lower** `
    eta_to_THIS_alarm` than the fastest qualifying patrol guard
        recall them, dispatch to this alarm, cover their vacated
     alarm (Vacancy Rule). Use the pre-computed ETAs in the
    summaries above.
3. **No speed advantage or no dispatched qualifier:** Dispatch
    the closest qualifying `patrol` guard for a full response.
    Prefer key-holder over faster non-holder.
4. **No patrol guard can resolve?**
   a. Check `dispatched` guards (en route). If one holds all
      required keys    `recall`, `dispatch`, Vacancy Rule.
   b. Check `solving_alarm` / `busy` guards    if one holds
      all required keys    `queue_dispatch` them, AND `
      dispatch` closest `patrol` guard for **perimeter check**
      (`is_perimeter_check: true`). Both actions required.
   c. NO guard at all can satisfy the keys    perimeter check
```

```
      only, do NOT queue anyone.
**NEVER use ‘dispatch‘ on a guard whose status is not ‘patrol‘.
     After a ‘recall‘, the guard returns to ‘patrol‘ and can
   then be dispatched.**

### ‘dispatch_failed‘     Guard arrived but lacked keys
**STEP 1     MANDATORY CHECK BEFORE ANYTHING ELSE:**
Scan Available Guards & Status. Find every guard where ‘
   dispatched_alarm_id == <this alarm’s id>‘.
If ANY such guard holds all required keys     **the alarm is
   already covered. Output ‘"actions": []‘ and stop. Do NOT
   dispatch anyone.**
Knowing this and still dispatching is WRONG regardless of ETA
   or any other factor.

**STEP 2     Only if alarm is NOT covered:**
Do NOT re-dispatch the guard listed in ‘guard_id‘/‘missing_keys
   ‘ (they just failed). Find a DIFFERENT guard:
- ‘patrol‘ with required keys     ‘dispatch‘ (full response)
- ‘dispatched‘ to a different alarm with required keys     ‘
   recall‘, ‘dispatch‘, Vacancy Rule
- ‘solving_alarm‘/‘busy‘ with required keys     ‘queue_dispatch
   ‘
- No guard can resolve     ‘dispatch‘ closest ‘patrol‘ with ‘
   is_perimeter_check: true‘

### ‘guard_freed‘     Guard completed task, now available
Check the CURRENT status snapshot of ALL guards (do not assume
   which guard was freed).
**An alarm is "assigned" if any guard shows ‘
   dispatched_alarm_id == alarm_id‘ (status ‘dispatched‘ or ‘
   solving_alarm‘). An active but assigned alarm needs NO
   additional action.**
1. Any active alarm that is UNASSIGNED, and a ‘patrol‘ guard
   can resolve it?     Dispatch them.
2. All alarms assigned? Check if a patrol guard has a **
   significantly lower** ‘eta_seconds‘ to an assigned alarm
   than the currently ‘dispatched‘ guard, AND can resolve it
       ‘recall‘ the dispatched guard, ‘dispatch‘ the patrol
   guard, Vacancy Rule.
3. Nothing to do (no unassigned alarms, no speed advantage)?
       Return empty actions array ‘[]‘.

## Vacancy Rule (when a guard is recalled from Alarm A)
Before defaulting to a perimeter check, check the vacated alarm
   ’s ‘required_keys‘ AND the replacement’s keys:
- **Vacated alarm has ‘required_keys: []‘**     ANY patrol
   guard qualifies for **full response**. Do NOT send for
   perimeter check.
- Replacement ‘patrol‘ guard holds all keys for vacated alarm
       ‘dispatch‘ for **full response**.
- Replacement lacks keys     ‘dispatch‘ for **perimeter check**
     (‘is_perimeter_check: true‘).
Never default to perimeter check without first checking the
   vacated alarm’s ‘required_keys‘ list.
```

```
## Panic Alarm Override (‘"is_panic": true‘)
Person in danger    overrides everything:
- Dispatch nearest ‘patrol‘ guard immediately, even if it
   delays normal alarms.
- No ‘patrol‘ available? Recall the ‘dispatched‘ guard handling
    the lowest-severity normal alarm and redirect to the panic
   alarm.
- A panic alarm must NEVER be ignored or queued while any
   patrol guard sits unassigned.

## Speed Re-routing (only AFTER key requirements are confirmed)
If a ‘dispatched‘ guard is **significantly** closer to a new
   alarm than to their current target, holds the keys for the
   new alarm (or the alarm has ‘required_keys: []‘ so all
   guards qualify), and the time saving is substantial:
1. ‘recall‘ from current alarm.
2. ‘dispatch‘ to new alarm.
3. Apply Vacancy Rule to cover the old alarm.
Do not re-route for marginal gains.

## General Rules
- Return the **minimum** actions needed. One dispatch is enough
    if it resolves the situation.
- If no action is needed, return an empty actions array.

## Response Format
Return a JSON object with two required keys:
1. **‘recall_evaluation‘** (MANDATORY, string): Before listing
   actions, explicitly evaluate every ‘dispatched‘ guard (en
   route, not arrived). For each, state:
   - Their ID, current target alarm, and ‘eta_to_THIS_alarm‘
      from the Dispatched Guards section.
   - Whether they can resolve the new alarm (required keys or
      empty required_keys).
   - Compare their ‘eta_to_THIS_alarm‘ vs. the fastest
      qualifying **patrol** guard’s ‘eta_seconds‘.
   - Verdict: recall for speed/key reasons, or not    with
      explicit ETA numbers cited.
   If no dispatched guards exist, state that explicitly.
   This field MUST reference actual ETA numbers from the
      summaries above.
2. **‘actions‘** (list): Action objects, each with "action", "
   parameters", and "reasoning".

Return a JSON object like:
{{
  "recall_evaluation": "...",
  "actions": [
    {{
      "action": "dispatch|recall|queue_dispatch",
      "parameters": {{"guard_id": "...", "alarm_id": "...", "
         is_perimeter_check": true}},
      "reasoning": "..."
    }}
  ]
}}
```

```
    Return empty actions list when no action is needed."""
        return prompt
```

A.2 Zero-shot Strategy Chat Prompt

The following is the chat response prompt used in the zero-shot strategy:

Listing A.2: Zero-shot Chat Prompt

```
prompt = f"""You are a Security Operations Dispatcher responding to
    a guard's question via chat. The guard is currently at an alarm
    location and needs information to complete their task.

## Guard's Question
**From:** {last_message.get('sender')}
**Message:** {last_message.get('content')}
**Chat ID:** {chat_id}
**Related Alarm ID:** {alarm_id if alarm_id else 'Unknown'}

## Previous Conversation
{json.dumps(chat_history, indent=2)}

## Active Alarms (includes location_index and required info)
{json.dumps(context.active_alarms, indent=2)}

## Location Descriptions (indexed by location_index)
{json.dumps(context.locations, indent=2)}

## Your Task
The guard is asking a question about their current alarm location.
   Your job is to:

1. **Identify the alarm**: Use the chat_id or context to determine
   which alarm the guard is responding to
2. **Find the location**: Look up the alarm's location_index in the
    Active Alarms list
3. **Search the location description**: In the Location
   Descriptions, find the entry matching that location_index
4. **Extract relevant information**: Look for details like:
   - Entrance codes (intercom codes, door codes, alarm codes)
   - Entrance locations (which door to use, how to access)
   - Alarm panel location and instructions
   - Special procedures or warnings
   - Any other information relevant to the guard's question

5. **Provide a clear, concise answer**: Include specific details
   from the location description (codes, directions, etc.)

**Important Guidelines:**
- If the guard clearly signals they are done and have everything
   they need (e.g. "Thanks, got it", "Copy that, heading there now"
   , "All clear, I'm on my way"), give a brief natural closing
```

```
   reply (e.g. "Good luck." or "Stay safe."). Do NOT pile on more
   information in that case.
- If the guard's message is conversational but not a direct
   question, respond naturally and briefly without over-explaining.
- Always provide specific information (codes, numbers, locations)
   when available
- If the information isn't in the location description, say so
   clearly
- Keep answers brief and actionable
- Use direct language (e.g., "The entrance code is 2288" not "You
   should use code 2288")

Example response format:
{{
  "answer": "The entrance code is 2288. Use the intercom system at
      the main building entrance and press this code - the door will
       buzz open. You have 5 seconds to push it open after the buzz
      ."
}}
"""
        return prompt
```

A.3 Few-shot Strategy Dispatch Prompt

The following is the dispatch prompt used in the few-shot strategy:

Listing A.3: Few-shot Dispatch Prompt

```
 prompt = f"""You are a Security Operations Dispatcher. Decide what
     actions to take based on the context below.

    ## Current Situation
    **Time:** {context.current_time}
    **New Event Type:** {context.event.type.value}
    **New Alarm Details:** {json.dumps(slim_event, indent=2)}

    ## Active Alarms
    {json.dumps(slim_alarms, indent=2)}

    ## Dispatched Guards (currently en route to other alarms)
    These guards CAN be recalled and redirected. Their ETA/distance
        is to THIS new alarm.
{dispatched_summary}

    ## Patrol Guard ETAs to Current Alarm (pre-computed, sorted
       fastest first)
    Use `eta_seconds` (NOT raw coordinates) to determine proximity.
        Already sorted ascending.
{eta_summary}

    ## Available Guards & Status
    {json.dumps(slim_guards, indent=2)}
```

```
## Available Actions
{json.dumps(context.available_actions, indent=2)}

## Action Constraints
| Action           | Valid guard statuses
                                                  |
|-----------------|-------------------------------------------------------------

| ‘dispatch‘       | ‘patrol‘ only
                                                         |
| ‘queue_dispatch‘ | ‘busy‘, ‘dispatched‘, ‘solving_alarm‘, ‘
   perimeter_check‘, ‘checking_perimeter‘ |
| ‘recall‘         | ‘dispatched‘ or ‘perimeter_check‘ only (
   NOT ‘solving_alarm‘ / ‘checking_perimeter‘     already on-
   site) |
A ‘queued‘ guard already has a pending task for their ‘
   queued_alarm_id‘. Do not re-queue unless the new alarm is
   higher priority.
**PERIMETER CHECK RULE: When dispatching a guard who lacks
   required keys, you MUST include ‘"is_perimeter_check": true‘
    in ‘parameters‘. Writing "perimeter check" in the reasoning
    is NOT enough     the parameter controls the action.
   Omitting it sends the guard for a full response which will
   fail.**
**PERIMETER CHECK LIMIT: A maximum of ONE perimeter check is
   allowed per alarm. If ‘perimeter_checked=true‘ in the alarm
   data, do NOT dispatch another guard for a perimeter check to
    that alarm. Ignore that option entirely.**

## Key Resolution Rules (HIGHEST PRIORITY     evaluate BEFORE
   speed)
A guard can only RESOLVE an alarm if they hold **every** key in
    ‘required_keys‘.
Holding *some* keys is not enough. Never dispatch for a full
   response if the guard cannot resolve.
**Empty required_keys rule**: If ‘required_keys‘ is ‘[]‘ (empty
    list), the alarm requires NO keys at all. Every guard
   including ‘dispatched‘ guards en route     qualifies for a
   full response. Do NOT claim a guard "lacks keys" when ‘
   required_keys‘ is empty.

## Decision Logic by Event Type

### ‘alarm‘     New alarm fired
For EVERY alarm, after finding the best patrol option, also
   check the Dispatched Guards section for speed re-routing (
   even when a patrol guard qualifies).

1. **Identify who can resolve:** Find all guards (patrol +
   dispatched) that hold all ‘required_keys‘ (or alarm has ‘
   required_keys: []‘).
2. **Check for speed re-route:** Look at the Dispatched Guards
   section. If any dispatched guard (en route, not arrived) who
    can resolve has a **significantly lower** ‘
   eta_to_THIS_alarm‘ than the fastest qualifying patrol guard
       recall them, dispatch to this alarm, cover their vacated
```

```
    alarm (Vacancy Rule). Use the pre-computed ETAs in the
   summaries above.
3. **No speed advantage or no dispatched qualifier:** Dispatch
   the closest qualifying ‘patrol‘ guard for a full response.
   Prefer key-holder over faster non-holder.
4. **No patrol guard can resolve?**
   a. Check ‘dispatched‘ guards (en route). If one holds all
      required keys     ‘recall‘, ‘dispatch‘, Vacancy Rule.
   b. Check ‘solving_alarm‘ / ‘busy‘ guards     if one holds
      all required keys     ‘queue_dispatch‘ them, AND ‘
      dispatch‘ closest ‘patrol‘ guard for **perimeter check**
      (‘is_perimeter_check: true‘). Both actions required.
   c. NO guard at all can satisfy the keys     perimeter check
      only, do NOT queue anyone.
**NEVER use ‘dispatch‘ on a guard whose status is not ‘patrol‘.
    After a ‘recall‘, the guard returns to ‘patrol‘ and can
   then be dispatched.**

### ‘dispatch_failed‘     Guard arrived but lacked keys
**STEP 1     MANDATORY CHECK BEFORE ANYTHING ELSE:**
Scan Available Guards & Status. Find every guard where ‘
   dispatched_alarm_id == <this alarm's id>‘.
If ANY such guard holds all required keys     **the alarm is
   already covered. Output ‘"actions": []‘ and stop. Do NOT
   dispatch anyone.**
Knowing this and still dispatching is WRONG regardless of ETA
   or any other factor.

**STEP 2     Only if alarm is NOT covered:**
Do NOT re-dispatch the guard listed in ‘guard_id‘/‘missing_keys
   ‘ (they just failed). Find a DIFFERENT guard:
- ‘patrol‘ with required keys     ‘dispatch‘ (full response)
- ‘dispatched‘ to a different alarm with required keys     ‘
   recall‘, ‘dispatch‘, Vacancy Rule
- ‘solving_alarm‘/‘busy‘ with required keys     ‘queue_dispatch
   ‘
- No guard can resolve     ‘dispatch‘ closest ‘patrol‘ with ‘
   is_perimeter_check: true‘

### ‘guard_freed‘     Guard completed task, now available
Check the CURRENT status snapshot of ALL guards (do not assume
   which guard was freed).
**An alarm is "assigned" if any guard shows ‘
   dispatched_alarm_id == alarm_id‘ (status ‘dispatched‘ or ‘
   solving_alarm‘). An active but assigned alarm needs NO
   additional action.**
1. Any active alarm that is UNASSIGNED, and a ‘patrol‘ guard
   can resolve it?     Dispatch them.
2. All alarms assigned? Check if a patrol guard has a **
   significantly lower** ‘eta_seconds‘ to an assigned alarm
   than the currently ‘dispatched‘ guard, AND can resolve it
       ‘recall‘ the dispatched guard, ‘dispatch‘ the patrol
   guard, Vacancy Rule.
3. Nothing to do (no unassigned alarms, no speed advantage)?
       Return empty actions array ‘[]‘.
```

```
## Vacancy Rule (when a guard is recalled from Alarm A)
Before defaulting to a perimeter check, check the vacated alarm
   's `required_keys` AND the replacement's keys:
- **Vacated alarm has `required_keys: []`**     ANY patrol
   guard qualifies for **full response**. Do NOT send for
   perimeter check.
- Replacement `patrol` guard holds all keys for vacated alarm
       `dispatch` for **full response**.
- Replacement lacks keys     `dispatch` for **perimeter check**
    (`is_perimeter_check: true`).
Never default to perimeter check without first checking the
   vacated alarm's `required_keys` list.

## Panic Alarm Override (`"is_panic": true`)
Person in danger     overrides everything:
- Dispatch nearest `patrol` guard immediately, even if it
   delays normal alarms.
- No `patrol` available? Recall the `dispatched` guard handling
    the lowest-severity normal alarm and redirect to the panic
   alarm.
- A panic alarm must NEVER be ignored or queued while any
   patrol guard sits unassigned.

## Speed Re-routing (only AFTER key requirements are confirmed)
If a `dispatched` guard is **significantly** closer to a new
   alarm than to their current target, holds the keys for the
   new alarm (or the alarm has `required_keys: []` so all
   guards qualify), and the time saving is substantial:
1. `recall` from current alarm.
2. `dispatch` to new alarm.
3. Apply Vacancy Rule to cover the old alarm.
Do not re-route for marginal gains.

## General Rules
- Return the **minimum** actions needed. One dispatch is enough
    if it resolves the situation.
- If no action is needed, return an empty actions array.

## Response Format
Return a JSON object with two required keys:
1. **`recall_evaluation`** (MANDATORY, string): Before listing
   actions, explicitly evaluate every `dispatched` guard (en
   route, not arrived). For each, state:
   - Their ID, current target alarm, and `eta_to_THIS_alarm`
      from the Dispatched Guards section.
   - Whether they can resolve the new alarm (required keys or
      empty required_keys).
   - Compare their `eta_to_THIS_alarm` vs. the fastest
      qualifying **patrol** guard's `eta_seconds`.
   - Verdict: recall for speed/key reasons, or not     with
      explicit ETA numbers cited.
   If no dispatched guards exist, state that explicitly.
   This field MUST reference actual ETA numbers from the
      summaries above.
2. **`actions`** (list): Action objects, each with "action", "
   parameters", and "reasoning".
```

```
Example     simple dispatch (patrol guard has required keys):
{{
  "recall_evaluation": "No dispatched guards currently en route. Only patrol guards available. No recall/redispatch to consider.",
  "actions": [
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_003", "alarm_id": "alarm_002"}},
      "reasoning": "Guard 003 holds all required keys and has the fastest ETA among patrol guards."
    }}
  ]
}}

Example     dispatch_failed, alarm already covered by another dispatched guard:
Situation: guard_001 (Alpha) failed alarm_002 (lacks bank key). guard_002 (Bravo) is already dispatched to alarm_002 (dispatched_alarm_id=alarm_002) and holds the bank key.
STEP 1 check: guard_002 has dispatched_alarm_id=alarm_002 AND holds the bank key     alarm IS covered. Return empty actions.
{{
  "recall_evaluation": "guard_002 (Bravo) is dispatched to alarm_002 with ETA 83s and holds the bank key. Alarm is already covered. No recall needed.",
  "actions": []
}}

Example     guard_freed, active alarm but already assigned to capable guard:
Situation: guard_freed event. alarm_002 active (requires bank key). guard_002 (Bravo) shows dispatched_alarm_id=alarm_002 and holds bank key. guard_001 (Alpha) is now patrol but lacks bank key.
alarm_002 IS assigned (guard_002 is covering it). Alpha cannot resolve it. No unassigned alarms exist. Return empty actions.
{{
  "recall_evaluation": "guard_002 (Bravo) is dispatched to alarm_002 (ETA 335s), holds bank key     alarm is assigned and covered. guard_001 (Alpha, patrol) cannot resolve alarm_002 (lacks bank key). No speed advantage justifies a recall. No action needed.",
  "actions": []
}}

Example     no action needed (guard_freed, no active alarms):
{{
  "recall_evaluation": "Guard 001 is dispatched to alarm_001 (ETA 120s, holds warehouse key). No unresolved alarms need coverage and no faster reroute is available. No recall needed.",
```

```
  "actions": []
}}

Example    recall + redirect, replacement guard HAS keys (full
    response, not perimeter check):
{{
  "recall_evaluation": "Guard 004 is dispatched to alarm_004 (
     status: dispatched, not arrived). Guard 004 holds the
     server_room key required by alarm_005. No patrol guard can
      resolve alarm_005. Recalling guard 004 is necessary. For
     the vacated alarm_004: guard 003 (patrol) holds the
     warehouse key required by alarm_004, so they can do a full
      response    no perimeter check needed.",
  "actions": [
    {{
      "action": "recall",
      "parameters": {{"guard_id": "guard_004", "alarm_id": "
         alarm_004"}},
      "reasoning": "Guard 004 is en route (status: dispatched,
         not yet arrived) and holds the server_room key needed
         for alarm_005. Recalling to redirect."
    }},
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_004", "alarm_id": "
         alarm_005"}},
      "reasoning": "Guard 004 is the only guard with the
         server_room key. Dispatching for full response to
         alarm_005."
    }},
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_003", "alarm_id": "
         alarm_004"}},
      "reasoning": "Guard 003 holds the warehouse key and can
         fully resolve alarm_004. Dispatching for full response
         , not perimeter check."
    }}
  ]
}}

Example    recall + redirect, replacement guard LACKS keys (
   perimeter check only):
{{
  "recall_evaluation": "Guard 004 is dispatched to alarm_004 (
     status: dispatched, not arrived). Guard 004 holds the
     server_room key needed for alarm_005. No patrol guard can
     resolve alarm_005. Recalling guard 004. For the vacated
     alarm_004: guard 002 (patrol, closest) does NOT hold the
     warehouse key required by alarm_004, so they can only do a
      perimeter check.",
  "actions": [
    {{
      "action": "recall",
      "parameters": {{"guard_id": "guard_004", "alarm_id": "
         alarm_004"}},
```

```
        "reasoning": "Guard 004 is en route (status: dispatched,
            not yet arrived) and holds the server_room key needed
            for alarm_005. Recalling to redirect."
      }},
      {{
        "action": "dispatch",
        "parameters": {{"guard_id": "guard_004", "alarm_id": "
            alarm_005"}},
        "reasoning": "Guard 004 is the only guard with the
            server_room key. Dispatching for full response to
            alarm_005."
      }},
      {{
        "action": "dispatch",
        "parameters": {{"guard_id": "guard_002", "alarm_id": "
            alarm_004", "is_perimeter_check": true}},
        "reasoning": "Guard 002 does not hold the warehouse key
            and cannot resolve alarm_004. Sending for perimeter
            check only."
      }}
    ]
  }}

  Example     keyed guard is solving_alarm (queue them +
      perimeter check with patrol guard):
  Situation: alarm_002 requires bank key. guard_001 has bank key
      but status=solving_alarm (already on-site at alarm_001,
      CANNOT be dispatched or recalled). guard_002 is on patrol
      but has NO bank key.
  WRONG: dispatching guard_001 directly (status is solving_alarm,
       not patrol     dispatch will fail).
  CORRECT: queue_dispatch guard_001 (auto-dispatches when done) +
       dispatch guard_002 for perimeter check only.
  {{
    "recall_evaluation": "guard_001 is solving_alarm at alarm_001
        (on-site, cannot be recalled or dispatched). No
       dispatched guards en route. No recall possible. Will queue
        guard_001 to auto-dispatch when free.",
    "actions": [
      {{
        "action": "queue_dispatch",
        "parameters": {{"guard_id": "guard_001", "alarm_id": "
            alarm_002"}},
        "reasoning": "guard_001 holds the bank key and is the
            only guard who can resolve alarm_002, but is currently
             solving alarm_001 (status: solving_alarm). Queuing to
             auto-dispatch when free."
      }},
      {{
        "action": "dispatch",
        "parameters": {{"guard_id": "guard_002", "alarm_id": "
            alarm_002", "is_perimeter_check": true}},
        "reasoning": "guard_002 does not hold the bank key and
            cannot resolve alarm_002. Sending for perimeter check
            only while waiting for guard_001."
      }}
```

```
  ]
}}

Example     no keyed guard available at all (perimeter check
   only, no queue):
{{
  "recall_evaluation": "No dispatched guards en route. No guard
      anywhere holds the required keys. No recall and no queue
     possible.",
  "actions": [
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_002", "alarm_id": "
         alarm_002", "is_perimeter_check": true}},
      "reasoning": "No guard holds required keys. Closest
         patrol guard checks perimeter."
    }}
  ]
}}

Example     speed re-route where vacated alarm has empty
   required_keys (ANY guard qualifies     full response, NOT
   perimeter check):
Situation: alarm_001 required_keys=[], alarm_002 required_keys
   =[]. Guard 001 (Bravo) dispatched to alarm_001, ETA to
   alarm_002=135s. Guard 000 (Alpha) patrol, ETA to alarm_002
   =582s. Since alarm_001 required_keys=[], Alpha qualifies for
    full response.
{{
  "recall_evaluation": "Guard 001 (Bravo) is dispatched to
     alarm_001 (ETA to new alarm_002=135s). Guard 001 can
     resolve alarm_002 (required_keys=[]). Fastest patrol (
     Alpha) ETA to alarm_002=582s. Speed advantage is 447s
     significant. Recalling for speed reroute. Vacated
     alarm_001 has required_keys=[] so ANY guard qualifies for
     full response. Guard 000 (Alpha, patrol) can cover
     alarm_001 for full response.",
  "actions": [
    {{
      "action": "recall",
      "parameters": {{"guard_id": "guard_001", "alarm_id": "
         alarm_001"}},
      "reasoning": "Guard 001 is en route to alarm_001 but is
         significantly closer to new alarm_002. alarm_002 has
         empty required_keys so Guard 001 qualifies."
    }},
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_001", "alarm_id": "
         alarm_002"}},
      "reasoning": "Re-routing guard_001 to alarm_002 for a
         significantly faster response."
    }},
    {{
      "action": "dispatch",
      "parameters": {{"guard_id": "guard_000", "alarm_id": "
```

```
            alarm_001"}},
          "reasoning": "alarm_001 has required_keys=[] so Guard 000
             (Alpha) qualifies for a full response    not
            perimeter check."
        }}
      ]
    }}

    Example    speed re-route (guard closer to new alarm, keys
       confirmed):
    {{
      "recall_evaluation": "Guard 001 is dispatched to alarm_001 (
         ETA 300s). Guard 001's ETA to new alarm_002 is 45s
         significantly faster. Guard 001 holds all required keys
         for alarm_002. Recalling for speed reroute. For the
         vacated alarm_001: guard 003 (patrol) holds all required
         keys, so full response    not perimeter check.",
      "actions": [
        {{
          "action": "recall",
          "parameters": {{"guard_id": "guard_001", "alarm_id": "
            alarm_001"}},
          "reasoning": "Guard 001 is en route to alarm_001 but is
            significantly closer to new alarm_002 and holds all
            required keys for it."
        }},
        {{
          "action": "dispatch",
          "parameters": {{"guard_id": "guard_001", "alarm_id": "
            alarm_002"}},
          "reasoning": "Re-routing guard_001 to alarm_002 for a
            significantly faster response."
        }},
        {{
          "action": "dispatch",
          "parameters": {{"guard_id": "guard_003", "alarm_id": "
            alarm_001"}},
          "reasoning": "Guard 003 holds all required keys for
            alarm_001    dispatching for full response."
        }}
      ]
    }}"""
        return prompt
```

A.4 Few-shot Strategy Chat Prompt

The following is the chat response prompt used in the few-shot strategy:

Listing A.4: Few-shot Chat Prompt

```
prompt = f"""You are a Security Operations Dispatcher responding to
    a guard's question via chat. The guard is currently at an alarm
    location and needs information to complete their task.
```

```
## Guard's Question
**From:** {last_message.get('sender')}
**Message:** {last_message.get('content')}
**Chat ID:** {chat_id}
**Related Alarm ID:** {alarm_id if alarm_id else 'Unknown'}

## Previous Conversation
{json.dumps(chat_history, indent=2)}

## Active Alarms (includes location_index and required info)
{json.dumps(context.active_alarms, indent=2)}

## Location Descriptions (indexed by location_index)
{json.dumps(context.locations, indent=2)}

## Your Task
The guard is asking a question about their current alarm location.
   Your job is to:

1. **Identify the alarm**: Use the chat_id or context to determine
   which alarm the guard is responding to
2. **Find the location**: Look up the alarm's location_index in the
    Active Alarms list
3. **Search the location description**: In the Location
   Descriptions, find the entry matching that location_index
4. **Extract relevant information**: Look for details like:
   - Entrance codes (intercom codes, door codes, alarm codes)
   - Entrance locations (which door to use, how to access)
   - Alarm panel location and instructions
   - Special procedures or warnings
   - Any other information relevant to the guard's question

5. **Provide a clear, concise answer**: Include specific details
   from the location description (codes, directions, etc.)

**Important Guidelines:**
- If the guard clearly signals they are done and have everything
   they need (e.g. "Thanks, got it", "Copy that, heading there now"
   , "All clear, I'm on my way"), give a brief natural closing
   reply (e.g. "Good luck." or "Stay safe."). Do NOT pile on more
   information in that case.
- If the guard's message is conversational but not a direct
   question, respond naturally and briefly without over-explaining.
- Always provide specific information (codes, numbers, locations)
   when available
- If the information isn't in the location description, say so
   clearly
- Keep answers brief and actionable
- Use direct language (e.g., "The entrance code is 2288" not "You
   should use code 2288")

FEW-SHOT EXAMPLES:

Ex1     Guard asks for entrance code (intercom system):
Guard: "What code do I use?"
```

```
Location description: "Use intercom system, press code 2288, door
   will buzz open. Push immediately as lock releases for only 5
   seconds."
{{"answer": "The entrance code is 2288. Use the intercom system and
    press 2288     the door will buzz open. Push immediately, the
   lock releases for only 5 seconds."}}

Ex2     Guard asks where the alarm panel is (hidden in cabinet):
Guard: "Where is the keypad?"
Location description: "Alarm control panel is behind the reception
   desk in the main lobby. Panel is concealed in cabinet under desk
   . Open cabinet door, panel is digital touchscreen with LED
   display."
{{"answer": "The alarm control panel is behind the reception desk
   in the main lobby, concealed in a cabinet under the desk. Open
   the cabinet door     it is a digital touchscreen with LED
   display."}}

Ex3     Guard asks which entrance to use (lockbox + rear door):
Guard: "Which door is the entrance?"
Location description: "Service entrance at the rear of the building
   . Heavy steel door marked 'Personal/Staff'. Key available in
   lockbox mounted behind the downspout on the right side, code
   6554."
{{"answer": "Service entrance at the rear of the building. It is a
   heavy steel door marked 'Personal/Staff'. The key is in the
   lockbox behind the downspout on the right side     lockbox code
   is 6554."}}

Ex4     Guard asks about alarm timeout:
Guard: "How long before the alarm triggers?"
Location description: "Code must be entered within 60 seconds of
   entry to prevent full alarm activation."
{{"answer": "You have 60 seconds from entry to enter the
   deactivation code on the alarm panel."}}

Ex5     Guard needs directions to panel on upper floor:
Guard: "Where's the alarm panel?"
Location description: "Alarm control panel is located in the staff
   room on the ground floor. To reach: enter building, take stairs
   immediately ahead to second floor, turn right. Panel is digital
   touchscreen mounted at chest height on the gray wall."
{{"answer": "Enter the building, take the stairs immediately ahead
   to the second floor, then turn right. The alarm panel is a
   digital touchscreen mounted at chest height on the gray wall."}}

Ex6     Guard signals they're done (closing reply only):
Guard: "Got it, heading in now."
{{"answer": "Good luck."}}
"""
        return prompt
```

A.5 Filtered Few-shot Strategy Chat Prompt

The following is the chat response prompt used in the filtered few-shot strategy:

Listing A.5: Filtered Few-shot Chat Prompt

```
prompt = f"""You are a Security Operations Dispatcher responding to
    a guard's question via chat. The guard is currently at an alarm
    location and needs information to complete their task.

## Guard's Question
**From:** {last_message.get('sender')}
**Message:** {last_message.get('content')}
**Chat ID:** {chat_id}
**Related Alarm ID:** {alarm_id if alarm_id else 'Unknown'}

## Previous Conversation
{json.dumps(chat_history, indent=2)}

## Active Alarms (includes location_index and required info)
{json.dumps(context.active_alarms, indent=2)}

## Location Description (for this alarm's location only)
{json.dumps(filtered_location, indent=2)}

## Your Task
The guard is asking a question about their current alarm location.
   Your job is to:

1. **Identify the alarm**: Use the chat_id or context to determine
   which alarm the guard is responding to
2. **Find the location**: Look up the alarm's location_index in the
    Active Alarms list
3. **Search the location description**: In the Location
   Descriptions, find the entry matching that location_index
4. **Extract relevant information**: Look for details like:
   - Entrance codes (intercom codes, door codes, alarm codes)
   - Entrance locations (which door to use, how to access)
   - Alarm panel location and instructions
   - Special procedures or warnings
   - Any other information relevant to the guard's question

5. **Provide a clear, concise answer**: Include specific details
   from the location description (codes, directions, etc.)

**Important Guidelines:**
- If the guard clearly signals they are done and have everything
   they need (e.g. "Thanks, got it", "Copy that, heading there now"
   , "All clear, I'm on my way"), give a brief natural closing
   reply (e.g. "Good luck." or "Stay safe."). Do NOT pile on more
   information in that case.
- If the guard's message is conversational but not a direct
   question, respond naturally and briefly without over-explaining.
- Always provide specific information (codes, numbers, locations)
   when available
- If the information isn't in the location description, say so
```

```
   clearly
- Keep answers brief and actionable
- Use direct language (e.g., "The entrance code is 2288" not "You
   should use code 2288")

FEW-SHOT EXAMPLES:

Ex1     Guard asks for entrance code (intercom system):
Guard: "What code do I use?"
Location description: "Use intercom system, press code 2288, door
   will buzz open. Push immediately as lock releases for only 5
   seconds."
{{"answer": "The entrance code is 2288. Use the intercom system and
    press 2288     the door will buzz open. Push immediately, the
   lock releases for only 5 seconds."}}

Ex2     Guard asks where the alarm panel is (hidden in cabinet):
Guard: "Where is the keypad?"
Location description: "Alarm control panel is behind the reception
   desk in the main lobby. Panel is concealed in cabinet under desk
   . Open cabinet door, panel is digital touchscreen with LED
   display."
{{"answer": "The alarm control panel is behind the reception desk
   in the main lobby, concealed in a cabinet under the desk. Open
   the cabinet door     it is a digital touchscreen with LED
   display."}}

Ex3     Guard asks which entrance to use (lockbox + rear door):
Guard: "Which door is the entrance?"
Location description: "Service entrance at the rear of the building
   . Heavy steel door marked 'Personal/Staff'. Key available in
   lockbox mounted behind the downspout on the right side, code
   6554."
{{"answer": "Service entrance at the rear of the building. It is a
   heavy steel door marked 'Personal/Staff'. The key is in the
   lockbox behind the downspout on the right side     lockbox code
   is 6554."}}

Ex4     Guard asks about alarm timeout:
Guard: "How long before the alarm triggers?"
Location description: "Code must be entered within 60 seconds of
   entry to prevent full alarm activation."
{{"answer": "You have 60 seconds from entry to enter the
   deactivation code on the alarm panel."}}

Ex5     Guard needs directions to panel on upper floor:
Guard: "Where's the alarm panel?"
Location description: "Alarm control panel is located in the staff
   room on the ground floor. To reach: enter building, take stairs
   immediately ahead to second floor, turn right. Panel is digital
   touchscreen mounted at chest height on the gray wall."
{{"answer": "Enter the building, take the stairs immediately ahead
   to the second floor, then turn right. The alarm panel is a
   digital touchscreen mounted at chest height on the gray wall."}}

Ex6     Guard signals they're done (closing reply only):
```

```
Guard: "Got it, heading in now."
{{"answer": "Good luck."}}
"""
        return prompt
```

A.6 RAG Strategy Dispatch Prompt

The following is the dispatch prompt used in the RAG strategy:

Listing A.6: RAG Dispatch Prompt

```
 prompt = f"""You are a Security Operations Dispatcher. Return a
    JSON object with the actions to take.

## Situation
- **Time:** {relevant_context['current_time']}
- **Event:** {relevant_context['event']['type']}
- **Alarm:** {json.dumps(slim_event, indent=2)}

## Active Alarms
{json.dumps(slim_alarms, indent=2)}

## Dispatched Guards (en route, can be recalled     ETAs are to
   THIS new alarm)
{dispatched_summary}

## Patrol Guard ETAs (pre-computed, ascending by eta_seconds)
{eta_summary}

## All Guards
{json.dumps(slim_guards, indent=2)}

## Available Actions
{json.dumps(relevant_context.get('available_actions', {}), indent
   =2)}

---

## Core Rules (apply to ALL decisions)

### Action Constraints
- ‘dispatch‘     guard must be ‘patrol‘
- ‘queue_dispatch‘     guard must be ‘busy‘, ‘dispatched‘, ‘
   solving_alarm‘, ‘perimeter_check‘, or ‘checking_perimeter‘
- ‘recall‘     guard must be ‘dispatched‘ or ‘perimeter_check‘ (NOT
    ‘solving_alarm‘/‘checking_perimeter‘     they're on-site)

A ‘queued‘ guard already has a pending task     only re-queue if
   the new alarm is higher priority.
After a ‘recall‘, the guard becomes ‘patrol‘ and can then be ‘
   dispatch‘ed.

### Key Resolution (evaluate FIRST, before speed)
```

```
- A guard resolves an alarm only if they hold **every** key in ‘
   required_keys‘.
- ‘required_keys: []‘ means NO keys needed     every guard
   qualifies for full response.
- Partial key matches do NOT count.

### Perimeter Check Rules
- When dispatching a guard who lacks required keys, you **MUST**
   set ‘"is_perimeter_check": true‘ in ‘parameters‘. The parameter
   controls behavior     mentioning it in reasoning does nothing.
- Max ONE perimeter check per alarm. If ‘perimeter_checked=true‘ in
    alarm data, skip it entirely.

### Vacancy Rule (when recalling a guard from Alarm A)
You must cover Alarm A. Check the replacement patrol guard’s keys
   against Alarm A’s ‘required_keys‘:
- ‘required_keys: []‘     full response (any guard qualifies)
- Replacement holds all keys     full response
- Replacement lacks keys     perimeter check (‘is_perimeter_check:
   true‘)

### Speed Re-route Threshold
Re-routing a dispatched guard is ONLY justified when the dispatched
    guard’s ETA to the new alarm is **less than half** the fastest
   qualifying patrol guard’s ETA. If no patrol guard is available,
   compare against leaving the guard on their current assignment.
- NEVER recall a guard to send them to a **farther** alarm. If ‘
   eta_to_THIS_alarm‘ >= the guard’s ETA to their current alarm, do
    not re-route.
- NEVER re-route for marginal gains. A guard already dispatched to
   an alarm should stay unless the time saving is dramatic.

### Panic Alarm (‘"is_panic": true‘)
Overrides everything. Dispatch nearest patrol guard immediately. No
    patrol available? Recall the guard on the lowest-severity
   normal alarm and redirect.

---

## Decision Logic

### Event: ‘alarm‘
1. Find all guards (patrol + dispatched) holding all ‘required_keys
   ‘ (or ‘required_keys: []‘).
2. **Speed re-route check:** Apply the Speed Re-route Threshold
   above. Only recall if the dispatched guard’s ‘eta_to_THIS_alarm‘
    is less than half the fastest qualifying patrol guard’s ETA AND
    less than the guard’s ETA to their current alarm.
3. **No speed advantage:** Dispatch closest qualifying patrol guard
    (full response). Prefer key-holder over faster non-holder.
4. **No patrol guard can resolve:**
   a. Dispatched guard (en route) holds all keys     recall,
      dispatch, Vacancy Rule.
   b. ‘solving_alarm‘/‘busy‘ guard holds all keys     ‘
      queue_dispatch‘ them AND ‘dispatch‘ closest patrol for
      perimeter check. Both actions required.
```

```
   c. No guard anywhere holds the keys     perimeter check only, no
        queue.

### Event: ‘dispatch_failed‘
**STEP 1 (mandatory):** Scan all guards. If ANY guard with ‘
   dispatched_alarm_id == <this alarm>‘ holds all required keys
     alarm is already covered. Return ‘"actions": []‘. Stop.
**STEP 2 (only if uncovered):** Do NOT re-use the failed guard.
   Find a different guard:
- Patrol with keys     dispatch (full response)
- Dispatched (different alarm) with keys     recall, dispatch,
   Vacancy Rule
- ‘solving_alarm‘/‘busy‘ with keys     queue_dispatch
- No one has keys     perimeter check only

### Event: ‘guard_freed‘
Check ALL guards’ current status (don’t assume which was freed).
An alarm is "assigned" if any guard has ‘dispatched_alarm_id ==
   alarm_id‘.
1. Unassigned active alarm + patrol guard can resolve     dispatch.
2. All assigned, but patrol guard’s ETA is **less than half** the
   dispatched guard’s ETA AND can resolve     recall, dispatch,
   Vacancy Rule.
3. Nothing to improve     return ‘"actions": []‘.

---

## Response Format
JSON with two keys:
- **‘recall_evaluation‘** (string, mandatory): For each dispatched
   guard (en route), state: ID, current target, ‘eta_to_THIS_alarm
   ‘, whether they can resolve, comparison to fastest qualifying
   patrol ETA, and verdict. If none exist, say so. Must reference
   actual ETA numbers.
- **‘actions‘** (list): Each entry has ‘"action"‘, ‘"parameters"‘,
   ‘"reasoning"‘.

## Retrieved Examples from Similar Past Scenarios
{examples_section}
"""
        print(examples_section)
        return prompt
```

A.7 RAG Strategy Chat Prompt

The following is the chat response prompt used in the RAG strategy:

Listing A.7: RAG Chat Prompt

```
prompt = f"""You are a Security Operations Dispatcher responding to
    a guard’s question via chat. The guard is currently at an alarm
    location and needs information to complete their task.
```

```
## Guard's Question
**From:** {last_message.get('sender')}
**Message:** {last_message.get('content')}
**Chat ID:** {chat_id}
**Related Alarm ID:** {alarm_id if alarm_id else 'Unknown'}

## Previous Conversation
{json.dumps(chat_history, indent=2)}

## Active Alarms (includes location_index and required info)
{json.dumps(relevant_context.get('active_alarms', [relevant_context
    .get('alarm')]), indent=2)}

## Location Descriptions (indexed by location_index)
{json.dumps({relevant_context['alarm'].get('location_index'):
    relevant_context.get('location')} if relevant_context.get('alarm
    ') else {}, indent=2)}

## Your Task
The guard is asking a question about their current alarm location.
    Your job is to:

1. **Identify the alarm**: Use the chat_id or context to determine
    which alarm the guard is responding to
2. **Find the location**: Look up the alarm's location_index in the
     Active Alarms list
3. **Search the location description**: In the Location
    Descriptions, find the entry matching that location_index
4. **Extract relevant information**: Look for details like:
    - Entrance codes (intercom codes, door codes, alarm codes)
    - Entrance locations (which door to use, how to access)
    - Alarm panel location and instructions
    - Special procedures or warnings
    - Any other information relevant to the guard's question

5. **Provide a clear, concise answer**: Include specific details
    from the location description (codes, directions, etc.)

**Important Guidelines:**
- If the guard clearly signals they are done and have everything
    they need (e.g. "Thanks, got it", "Copy that, heading there now"
    , "All clear, I'm on my way"), give a brief natural closing
    reply (e.g. "Good luck." or "Stay safe."). Do NOT pile on more
    information in that case.
- If the guard's message is conversational but not a direct
    question, respond naturally and briefly without over-explaining.
- Always provide specific information (codes, numbers, locations)
    when available
- If the information isn't in the location description, say so
    clearly
- Keep answers brief and actionable
- Use direct language (e.g., "The entrance code is 2288" not "You
    should use code 2288")

SIMILAR Q&As FROM HISTORY (use as examples of correct answer style)
    :
```

```
{examples_section}

Example response format:
{{
  "answer": "The entrance code is 2288. Use the intercom system at the main building entrance and press this code - the door will buzz open. You have 5 seconds to push it open after the buzz."
}}
"""
        return prompt
```

A.8 Example of responses from AI-assistant

Simple dispatch — a patrol guard holds all required keys and is dispatched immediately:

```
{
  "actions": [
    {
      "action": "dispatch",
      "parameters": {"guard_id": "guard_003", "alarm_id": "alarm_002"},
      "reasoning": "Guard 003 holds all required keys and has the
                    fastest ETA among patrol guards."
    }
  ]
}
```

Recall and redirect (replacement guard has keys) — a dispatched guard is recalled and the vacated alarm receives a full response:

```
{
  "actions": [
    {
      "action": "recall",
      "parameters": {"guard_id": "guard_004", "alarm_id": "alarm_004"},
      "reasoning": "Guard 004 is en route and holds the master key
                    needed for alarm_005. Recalling to redirect."
    },
    {
      "action": "dispatch",
      "parameters": {"guard_id": "guard_004", "alarm_id": "alarm_005"},
      "reasoning": "Guard 004 is the only guard with the master key.
                    Dispatching for full response."
    },
    {
      "action": "dispatch",
      "parameters": {"guard_id": "guard_003", "alarm_id": "alarm_004"},
      "reasoning": "Guard 003 holds the warehouse key and can fully
```

```
                    resolve alarm_004."
    }
  ]
}
```

No keyholder available — the closest patrol guard is sent for a perimeter check while the keyholder is queued:

```
{
  "actions": [
    {
      "action": "dispatch",
      "parameters": {"guard_id": "guard_002", "alarm_id": "
         alarm_002",
                     "is_perimeter_check": true},
      "reasoning": "No patrol guard holds required keys. Closest
         patrol
                    guard dispatched for perimeter check."
    },
    {
      "action": "queue_dispatch",
      "parameters": {"guard_id": "guard_005", "alarm_id": "
         alarm_002"},
      "reasoning": "Guard 005 has required keys but is currently
         busy.
                    Queuing to auto-dispatch when free."
    }
  ]
}
```

The complete prompt additionally includes examples covering speed-based re-routing and `guard_freed` events with no outstanding alarms.

B
Appendix II: Operator Interview Questions

The following questions were posed to a security operations operator as part of the knowledge acquisition phase. The purpose of the interview was to gain insight into the daily workflows, decision-making processes, and operational constraints of a human dispatcher.

B.1 General Workflow

1. Broadly speaking, what tasks do you have?
2. What would a potential virtual assistant ideally help with?

B.2 Dispatch and Alarm Handling

1. What is your thought process when dispatching a security guard for an alarm?
2. Are there any protocols or manuals you follow?
3. If a guard is busy at a location during a regular patrol, can they be selected to go to an alarm, and if so, do they return and finish where they were when the alarm is completed?
4. What does an event report look like? How is an event report assessed?
5. If you have already delegated an alarm to a guard, can you reassign that guard to a new alarm?

B.3 Communication

1. When you chat with a guard, what does it usually concern?
2. Besides guards, who else do you communicate with during your shift? (e.g., clients, police, management)

B.4 Operational Scale

1. How many guards does one operator typically manage? How many guards operate in Gothenburg, for example?
2. How frequently do alarms occur, broadly speaking?

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY