الجمـهورية الجزائرية الديمقراطية الشعبيـة
وزارة التعليم العالي والبحث العلمي
جـامعـة سعـيـدة د. مـولاي الطـاهـر
كلية الرياضيات و الإعلام الآلي و الاتصالات السلكية و اللاسلكية
قسم: الإعلام الآلي

Mémoire de Master en informatique

Spécialité :

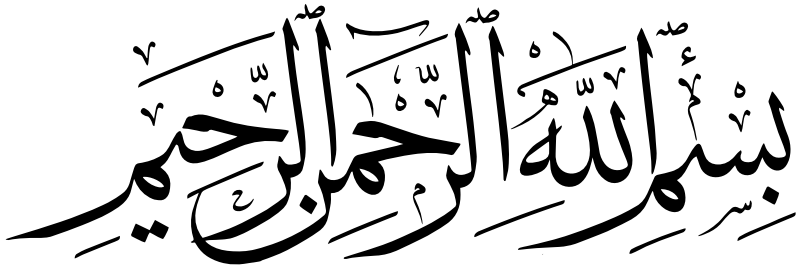
Intelligence artificielle : principes et applications

T h è m e

DESIGN AND IMPLEMENTATION OF AN INTELLIGENT MANAGERIAL ASSISTANT FOR E-COMMERCE OPTIMIZATION

- **Présenté par :**
  BOUREGAG Aya
  RACHEDI Aya
- **Dirigé par :**
  Dr. Mohamed Abdou SOUIDI

Année universitaire 2025-2026


بسم الله الرحمن الرحيم

Dedication

Alhamdulillah for everything. All thanks to Allah for giving me the strength to complete my Master's degree.
My faith as a Muslim gave me the patience to keep going, inspired by the beautiful resilience of our brothers in war this last year.

To my beloved parents:
thank you for everything you have done to get me here; this degree is as much yours as it is mine.

To my brothers and my little sister:
thank you for making life fun and making the heavy days feel light. Thank you specifically to my brother, who kindly acted as a 'real user' for my project when I needed it most!

And finally, a dedication to myself.
Thank you to myself for the late nights, the hard work, and for never giving up on this goal.

Aya Bouregag

Dédicace

À ceux qui ont fait de moi ce que je suis aujourd'hui...

À mes très chers parents,

aucun mot ne saurait exprimer l'immensité de votre amour, ni la profondeur de ma gratitude.

Vous avez été ma force dans les moments de faiblesse, ma lumière dans l'obscurité, et mon refuge face aux épreuves.

Ce travail est le fruit de vos sacrifices, de votre patience et de votre foi en moi.

À mes chères sœurs Marwa, Hafsa, Meriem, Tasnim, Assil,

qui n'a jamais cessé de croire en moi, même lorsque je doutais. Votre soutien silencieux a été ma plus grande richesse.

À mes amis(es),

qui ont partagé avec moi les joies comme les difficultés, et ont rendu ce parcours plus humain et inoubliable.

À tous ceux qui m'ont tendu la main, conseillé ou encouragé, je dédie ce travail avec une profonde reconnaissance.

Aya Rachedi

Remerciements

الحمد لله عدد خلقه ورضا نفسه وزنة عرشه ومداد كلماته

Avant tout, je tiens à remercier **Dieu le Tout-Puissant** de m'avoir donné la force, la patience et la volonté nécessaires pour mener à bien ce travail.

Je tiens à exprimer ma profonde gratitude à mes très chers **parents,** pour leur amour inconditionnel, leurs sacrifices et leur soutien constant tout au long de mon parcours. Sans eux, rien de tout cela n'aurait été possible.

Je remercie également ma **collègue de mémoire, *Aya*,** avec qui j'ai partagé ce travail, pour sa collaboration, son sérieux et les efforts fournis durant toute cette période.

Mes sincères remerciements s'adressent à mon **encadreur, Souidi Abdou Mohamed,** pour son accompagnement précieux, ses conseils avisés, sa disponibilité et son soutien tout au long de la réalisation de ce mémoire.

Enfin, je remercie toutes les personnes qui, de près ou de loin, ont contribué à l'aboutissement de ce travail.

Et pour finir, je tiens à me remercier moi-même.
Pour les nuits blanches, les moments de doute, les larmes cachées, la pression, l'épuisement et pourtant… je ne me suis jamais laissée abattre. Pour chaque pas en avant, chaque effort, chaque fois où j'ai choisi de continuer.

Je suis fière du chemin parcouru et reconnaissante de ne jamais avoir abandonné.

Abstract / Résumé

Abstract

This thesis presents the design and implementation of Watchtower, a secure Telegram-based managerial assistant for e-commerce inventory optimisation and business intelligence. The system addresses two persistent operational problems faced by small store owners: the inability to extract actionable insights from raw sales data, and the reliance on guesswork for inventory replenishment decisions.

Watchtower integrates three technical components. A Zero Trust authentication layer, implemented as a Python decorator with rate limiting, restricts access to a single authorised manager using Telegram User ID whitelisting and silent-discard of unauthorised requests. An inventory optimisation engine applies the Wilson Economic Order Quantity formula, extended with a safety stock component at a 95% service level, to compute optimal reorder quantities and reorder points from historical sales data, dispatching proactive Telegram alerts when stock falls below the computed threshold. A four-stage natural language query pipeline first blocks adversarial inputs through a seventeen-pattern prompt injection guard, then resolves common analytical queries through direct Pandas computation, and falls back to a locally hosted Llama 3 language model via a custom OllamaLLM adapter, using Retrieval-Augmented Generation to constrain model output to verified organisational data.

The system was implemented in Python within a Kaggle Notebook environment using a dual-write persistence mechanism that maintains architectural equivalence with a production SQLite deployment. Empirical validation across six structured scenarios confirmed correctness of the authentication layer, numerical accuracy of the EOQ and ROP computations, sub-two-second alert latency, effective prompt injection defence, and grounded strategic analysis output.

Keywords: Decision Support System, Inventory Optimisation, Economic Order Quantity, Retrieval-Augmented Generation, Telegram Bot, Zero Trust Authentication, Prompt Injection, Large Language Model.

Résumé

Ce mémoire présente la conception et l'implémentation de Watchtower, un assistant managérial sécurisé basé sur Telegram, dédié à l'optimisation des stocks et à l'intelligence

économique pour le commerce électronique. Le système répond à deux problèmes opérationnels récurrents rencontrés par les petits commerçants : l'incapacité à extraire des informations exploitables à partir des données de ventes brutes, et le recours à l'intuition pour les décisions de réapprovisionnement.

Watchtower intègre trois composants techniques. Une couche d'authentification Zero Trust, implémentée sous forme de décorateur Python avec limitation de débit, restreint l'accès à un seul gestionnaire autorisé via la liste blanche d'identifiants Telegram, avec rejet silencieux de toute requête non autorisée. Un moteur d'optimisation des stocks applique la formule de Wilson pour la Quantité Économique de Commande, étendue par un stock de sécurité calibré à un niveau de service de 95 %, pour calculer les quantités optimales de commande et les points de réapprovisionnement à partir des données historiques de ventes, avec envoi d'alertes proactives lorsque le stock franchit le seuil calculé. Un pipeline de traitement en quatre étapes bloque d'abord les entrées adversariales via dix-sept expressions régulières de détection d'injection de prompt, résout ensuite les requêtes analytiques courantes par calcul direct avec Pandas, puis recourt au modèle de langage Llama 3 hébergé localement via un adaptateur OllamaLLM personnalisé, en utilisant la génération augmentée par récupération pour ancrer les réponses dans les données vérifiées de l'organisation.

Le système a été implémenté en Python dans un environnement Kaggle Notebook, avec un mécanisme de persistance double écriture maintenant une équivalence architecturale avec un déploiement SQLite en production. La validation empirique sur six scénarios structurés a confirmé la correction de la couche d'authentification, la précision numérique des calculs EOQ et ROP, une latence d'alerte inférieure à deux secondes, l'efficacité de la défense contre l'injection de prompt, et la pertinence des analyses stratégiques produites.

Mots-clés : Système d'aide à la décision, Optimisation des stocks, Quantité Économique de Commande, Génération augmentée par récupération, Bot Telegram, Authentification Zero Trust, Injection de prompt, Grand modèle de langage.

ملخص

تُقدِّم هذه المذكرة تصميم وتطبيق نظام Watchtower، وهو مساعد إداري آمن مبني على منصة Telegram، مخصص لتحسين إدارة المخزون والذكاء التجاري في مجال التجارة الإلكترونية. يعالج النظام مشكلتين متكررتين يواجهها أصحاب المتاجر الصغيرة: عدم القدرة على استخلاص رؤى قابلة للتنفيذ من بيانات المبيعات الخام، والاعتماد على الحدس في قرارات إعادة التخزين.

يدمج Watchtower ثلاثة مكوّنات تقنية. تُقيِّد طبقة المصادقة المبنية على مبدأ Zero Trust الوصول إلى مدير واحد مصرَّح له فقط، عبر قائمة بيضاء لمعرِّف Telegram مع رفض صامت لأي طلب غير مصرَّح به ومحدِّد للأوامر. يطبِّق محرك تحسين المخزون معادلة ويلسون للكمية الاقتصادية للطلب، مُعزَّزةً بمخزون أمان مُعيَّر عند مستوى خدمة 95%، لحساب الكميات المثلى للطلب ونقاط إعادة الطلب من بيانات المبيعات التاريخية، مع إرسال تنبيهات فورية عبر Telegram حين ينخفض المخزون دون العتبة المحسوبة. يعالج خط أنابيب الاستعلام باللغة الطبيعية المكوَّن من أربع مراحل المدخلات العدائية أوَّلاً عبر سبعة عشر نمطاً للكشف عن حقن التعليمات، ثم يحسم الاستفسارات التحليلية الشائعة بالحساب المباشر عبر Pandas، وعند الاقتضاء يستدعي نموذج اللغة الكبير Llama 3 المستضاف محلياً عبر محوِّل OllamaLLM مخصص، مستخدماً أسلوب التوليد المعزز بالاسترجاع لتقييد مخرجات النموذج بالبيانات المؤسسية المتحقق منها.

نُفِّذ النظام بلغة Python في بيئة Kaggle Notebook، مع آلية استمرارية ثنائية الكتابة تحافظ على التكافؤ المعماري مع نشر SQLite في بيئة الإنتاج. أكَّد التحقق التجريبي عبر ستة سيناريوهات منظَّمة صحة طبقة المصادقة، ودقة حسابات EOQ وROP، وزمن استجابة للتنبيهات أقل من ثانيتين، وفاعلية الدفاع ضد حقن التعليمات، وملاءمة التحليلات الاستراتيجية المُولَّدة.

الكلمات المفتاحية: نظام دعم القرار، تحسين المخزون، الكمية الاقتصادية للطلب، التوليد المعزز بالاسترجاع، بوت Telegram، مصادقة Zero Trust، حقن التعليمات، نموذج اللغة الكبير.

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BI	Business Intelligence
DSS	Decision Support System
EOQ	Economic Order Quantity
HTTPS	Hypertext Transfer Protocol Secure
LLM	Large Language Model
NLI	Natural Language Interface
NLP	Natural Language Processing
OR	Operations Research
RAG	Retrieval-Augmented Generation
ROP	Reorder Point
SQL	Structured Query Language
SSL	Secure Sockets Layer
SWOT	Strengths, Weaknesses, Opportunities, Threats
TLS	Transport Layer Security

CONTENTS

Contents

List of Figures

List of Tables

Chapter 1

General Introduction and Context

1.1 Introduction

Over the past two decades, the domain of electronic commerce has experienced a remarkable evolution, transitioning from basic online storefronts to intricate ecosystems characterized by extensive data generation and a pressing need for continuous oversight paired with prompt decision-making. Modern e-commerce platforms produce a substantial volume of diverse operational data encompassing aspects such as customer behavior patterns, inventory dynamics, sales velocity metrics, conversion funnel efficiency, and performance indicators within supply chains.

This proliferation of data has created a scenario where managers find themselves inundated with information but often lacking the necessary analytical tools or cognitive resources to distill this abundance into actionable insights. This tension, frequently described as a managerial information crisis[1], highlights how the mere availability of copious datasets does not necessarily translate into effective decision-making due to insufficient interpretative frameworks or processing capacity.

While considerable progress has been achieved in the field of customer-oriented conversational agents—such as chatbots adept at managing routine inquiries, order processing, and personalized shopping interactions[2]—there remains a notable deficiency in intelligent systems tailored specifically to support managerial decision-making activities. The majority of existing business intelligence solutions tend to be reactive, placing the onus on managers to actively interrogate dashboards, analyze visual data representations, and independently derive strategic conclusions[3]. This paradigm imposes heavy cognitive demands on decision-makers, who must continuously navigate and interpret multiple concurrent data streams alongside managing day-to-day operational challenges, thereby potentially compromising the effectiveness and timeliness of their decisions.

In response to this identified gap, the current research proposes the conceptualization and development of an intelligent managerial assistant, designated as the Watchtower system. This system aims to redefine the interaction between e-commerce managers and operational data by

[1] P. G. Roetzel. "Information overload in the information age: A review of the literature from business administration, business psychology, and related disciplines". In: *Business Research* 12.2 (2019), pp. 479–522. DOI: 10.1007/s40685-018-0069-z.

[2] E. Adamopoulou and L. Moussiades. "Chatbots: History, technology, and applications". In: *Machine Learning with Applications* 2 (2020), p. 100006. DOI: 10.1016/j.mlwa.2020.100006.

[3] L. Tadakala et al. "Beyond Visualization: Building Decision Intelligence Through Iterative Dashboard Refinement". In: *arXiv preprint arXiv:2510.27572* (2025).

shifting from a passive repository model to an active, autonomous decision-support entity[4]. The Watchtower is designed to continuously monitor critical performance metrics, detect abnormal or emergent patterns, offer data-driven optimization suggestions, and proactively alert managers to potential opportunities or risks. Such functionality aspires to alleviate cognitive burdens and facilitate more informed, timely managerial interventions.

The design of the Watchtower system synthesizes methodologies and technologies from three key areas: first, artificial intelligence embodied in Large Language Models (LLMs) that enable sophisticated natural language processing and contextual understanding[5]; second, operations research frameworks that supply robust mathematical tools for operational optimization[6]; and third, secure communication protocols that guarantee data privacy, integrity, and access control[7]. By integrating these elements into a cohesive, conversational interface accessible through the widely used Telegram messaging platform, this initiative introduces an innovative managerial assistance paradigm. This paradigm balances advanced analytical sophistication with usability, providing decision-makers with an intuitive yet powerful tool for navigating the complexities of contemporary e-commerce management.

1.2 Problem Statement and Research Motivation

1.2.1 The Manager's Dilemma: Drowning in Data, Starving for Insights

Contemporary e-commerce managers operate within a data-saturated environment characterized by what Ackoff (1989)[8] distinguished as an abundance of "data" and "information" but a scarcity of genuine "knowledge" and "wisdom." While modern analytics platforms can track thousands of metrics—pageviews, bounce rates, cart abandonment percentages, average order values, customer lifetime values, and inventory turnover ratios—the transformation of these raw metrics into actionable strategic insights remains a fundamentally human-dependent process.

The cognitive burden imposed by this information overload manifests in several criti-

[4]L. Wang et al. "A survey on large language model based autonomous agents". In: *arXiv preprint arXiv:2308.11432* (2023).

[5]Meta AI Research. *Introducing Llama 3: The most capable openly available LLM to date*. 2024. URL: https://ai.meta.com/blog/meta-llama-3/.

[6]B. Li et al. "OptiGuide: Large language models for supply chain optimization". In: *arXiv preprint arXiv:2307.03875* (2023).

[7]S. Z. Husen et al. "Chatbot framework using natural language processing (NLP) to assist operational activities". In: *Journal of Physics: Conference Series* 1569.2 (2020), p. 022064. DOI: 10.1088/1742-6596/1569/2/022064.

[8]R. L. Ackoff. "From data to wisdom". In: *Journal of Applied Systems Analysis* 16.1 (1989), pp. 3–9.

cal ways[9]. First, managers must develop and maintain mental models of what constitutes "normal" performance across dozens or hundreds of key performance indicators, a task that exceeds human working memory capacity. Second, the identification of meaningful patterns within noisy data streams requires statistical sophistication that many managers lack, leading to misinterpretations or missed opportunities. Third, even when anomalies are detected, determining their root causes and optimal responses demands time and analytical resources that are often unavailable in fast-paced operational environments.

A particularly illustrative example of this dilemma concerns inventory management decisions. An e-commerce manager might observe declining sales velocity for a particular product category and face multiple competing hypotheses: Is the decline a temporary seasonal fluctuation? Has a competitor launched a superior alternative? Is the pricing strategy suboptimal? Are website search algorithms failing to surface these products effectively? Each hypothesis demands different investigative approaches and remedial actions, yet the typical dashboard provides no guidance in this differential diagnosis[10].

1.2.2 The Inventory Optimization Challenge

A second major problem motivating this research concerns the persistent inefficiencies in e-commerce inventory management practices. Despite decades of research establishing optimal order quantities and reorder points through operations research methodologies, many small-to-medium e-commerce operations continue to rely on intuition-based restocking decisions that result in either excessive inventory holding costs or costly stockout situations.

The Economic Order Quantity (EOQ) model, first formulated by Ford W. Harris in 1913[11] and later popularized by R.H. Wilson, provides a mathematically rigorous framework for balancing setup costs against holding costs to determine optimal order sizes. The model's fundamental insight—that there exists a calculable order quantity minimizing total inventory costs—remains valid across diverse contexts, yet its practical application requires data inputs (annual demand rates, ordering costs, holding costs) that managers often find difficult to estimate and computational capabilities they may lack.

[9] Roetzel, "Information overload in the information age: A review of the literature from business administration, business psychology, and related disciplines".

[10] Tadakala et al., "Beyond Visualization: Building Decision Intelligence Through Iterative Dashboard Refinement".

[11] F. W. Harris. "How many parts to make at once". In: *Factory, The Magazine of Management* 10.2 (1913), pp. 135–136.

More fundamentally, the manual application of even simple optimization models represents an additional cognitive and temporal burden on already-overloaded managers. The opportunity therefore exists to embed such optimization capabilities within intelligent assistant systems that can automatically apply appropriate models, sensitize recommendations to parameter uncertainties, and present findings in decision-ready formats[12].

The convergence of these challenges yields the central research question animating this thesis: How can large language models and operations research optimization algorithms be integrated within a secure conversational interface to create a proactive managerial assistant system for e-commerce operations?

1.3 From Chatbots to Decision Support Systems: Theoretical Positioning

The Watchtower system positions itself at the intersection of historical Decision Support Systems (DSS) and contemporary Large Language Models (LLMs). Rather than functioning as a standard automated chat interface, it is arguably better understood as a manager-facing analytical tool. Its primary utility lies in augmenting human decision-making within complex e-commerce environments. This specific application requires strict access protocols and a sustained focus on mathematical rigor over conversational fluidity.

1.3.1 Foundations of Decision Support Systems (DSS)

To properly situate Watchtower's underlying mechanics, the system must be examined through the lens of established DSS literature[13]. Traditionally, scholars define a DSS as an interactive framework combining raw data, formal analytical models, and human insight to address semi-structured business problems.

Historical research typically classifies these systems into five distinct typologies: communication-driven, data-driven, document-driven, knowledge-driven, and model-driven. Watchtower appears to operate as a hybrid architecture. It incorporates elements from all five classifications. Specifically, the system employs conversational interfaces for communication, processes structured transactional data, interprets strategic queries through natural language processing (NLP), and applies formal mathematical optimization models.

[12] Li et al., "OptiGuide: Large language models for supply chain optimization".

[13] D. Bourgeois. *Information systems for business and beyond*. The Saylor Academy, 2014. URL: https://open.umn.edu/opentextbooks/textbooks/140.

1.3.2 Conversational AI as the DSS Interface Layer

The adoption of traditional analytical systems among non-technical managers has historically faced high friction. Users were frequently required to master specific query languages, such as SQL, or navigate rigid, complex dashboard structures.

Integrating LLMs alters this operational dynamic. Natural language tends to lower the cognitive barrier to entry, allowing managers to query complex datasets intuitively. An executive might ask, "Why did revenue decline last week?" and receive immediate, context-aware metrics. Beyond simple data retrieval, modern LLMs can act as "code-generating" support systems. They do not merely return pre-programmed responses. Instead, they actively translate natural language queries into executable scripts—typically Python or SQL. The system runs these scripts against live databases and then synthesizes the outputs back into a readable, conversational format[14].

1.3.3 Customer-Facing Chatbots vs. Manager-Facing Assistants

A clear conceptual boundary exists between standard e-commerce chatbots and the architecture proposed for Watchtower. Customer-facing bots generally handle high-volume, repetitive inquiries regarding order statuses or product returns. Success in that domain is typically measured by interaction containment rates, system politeness, and overall user satisfaction. The data involved is almost entirely low-sensitivity.

Manager-facing systems serve a fundamentally different operational purpose. They tend to prioritize analytical depth, mathematical accuracy, and actionable insights over conversational warmth. When seeking inventory recommendations, a manager requires methodological transparency and optimal data outputs to make sound choices; empathy is entirely irrelevant to the task.

This functional divergence dictates the system's core design. Consequently, Watchtower utilizes a concise, professional linguistic register. It exposes deep analytical functionality and explicitly acknowledges statistical uncertainty rather than attempting to gracefully mask system limitations. Because the tool processes highly sensitive corporate intelligence, it also requires strict, role-based authentication protocols that would be unnecessary in consumer-facing applications[15].

[14] Wang et al., "A survey on large language model based autonomous agents".
[15] Adamopoulou and Moussiades, "Chatbots: History, technology, and applications".

1.4 Project Scope and Objectives

1.4.1 Specific Objectives and Deliverables

To achieve the overarching aim delineated in this thesis, five specific sub-objectives have been identified, each addressing distinct technical and practical challenges integral to the proposed system.

The first objective concerns the architectural design of a hybrid AI-optimization system. This entails developing and thoroughly documenting a framework that seamlessly integrates deterministic modules from operations research—such as calculations of Economic Order Quantity (EOQ) and determination of reorder points—with the probabilistic inference capabilities of Large Language Models (LLMs)[16]. The resulting system must operate within a unified conversational interface, harmonizing symbolic, rule-based reasoning methods with statistical, neural pattern recognition approaches. This integration tackles the complex challenge of combining fundamentally different reasoning paradigms into a coherent and functional design.

The second objective focuses on ensuring secure authentication and access control. Given the intended primary interface is Telegram—a platform not originally engineered for enterprise-grade security—the system must incorporate robust mechanisms that safeguard sensitive business intelligence data and optimization recommendations[17]. Access restrictions should be enforced to limit availability exclusively to authorized managerial personnel. To this end, careful evaluation and implementation of authentication protocols are necessary, including user ID whitelisting, secure session management, and rigorous input validation strategies to mitigate risks associated with the platform's inherent limitations.

The third objective involves enabling natural language data analysis capabilities through the integration of an LLM, specifically the Llama 3 model deployed via the Ollama framework. This component should interpret analytical queries expressed in everyday language, formulate accurate data inquiries or computational commands, execute them correctly, and subsequently translate the outcomes into clear, manager-friendly natural language responses. This feature directly addresses the challenge of making complex analytical processes accessible and interpretable to users lacking specialized technical backgrounds.

The fourth objective targets the implementation and validation of automated inventory

[16]Meta AI Research, *Introducing Llama 3: The most capable openly available LLM to date*.

[17]Husen et al., "Chatbot framework using natural language processing (NLP) to assist operational activities".

optimization modules. These modules are designed to autonomously compute optimal order quantities and reorder points based on classical operations research principles. Parameters will be calibrated dynamically using historical sales data and cost structures specified by users, thus grounding the optimizations in actual operational conditions. This objective addresses the substantive inventory management issues that underpin the motivation for this research.

The fifth and final objective pertains to empirical validation through scenario testing. This involves designing and conducting a series of representative use cases that replicate realistic managerial decision-making environments. These scenarios will demonstrate the system's capabilities across multiple facets, including anomaly detection, inventory optimization, and strategic advice generation. The empirical results gathered here will substantiate the practical applicability and effectiveness of the proposed system.

1.4.2 Target Audience and Use Context

The intended beneficiaries for this research are primarily small- to medium-sized e-commerce enterprises typically managed by individual owners or limited management teams[18]. These entities often possess operational data repositories—such as transaction histories, inventory logs, and customer interaction records—that contain valuable insights but lack personnel with deep expertise in data science or operations research to extract and leverage this information.

Unlike large-scale enterprises endowed with mature business intelligence infrastructures, dedicated analytical teams, and intricate organizational approval workflows, the focus here is on what might be considered the "long tail" of e-commerce operations. In these contexts, decision-making authority tends to be highly centralized, technical resources are limited, and decision-support tools must be immediately accessible without the need for prolonged training or extensive adaptation. The system developed in this thesis is thus tailored to meet the practical demands of these smaller-scale operations, providing advanced analytical functionality within a user-friendly interface suited to non-expert users.

1.4.3 Technological Foundations

The system leverages three core technological components, each representing an established research and engineering domain:

[18] A. Tarutė and R. Gatautis. "ICT impact on SMEs performance". In: *Procedia-Social and Behavioral Sciences* 110 (2014), pp. 1218–1225. DOI: 10.1016/j.sbspro.2013.12.968.

- **Large Language Models (Ollama/Llama 3):** The system employs Meta's Llama model, an open-source large language model with approximately 8 billion parameters[19], accessed via the Ollama framework that enables local deployment without cloud API dependencies. This choice prioritizes data privacy (all processing occurs on local infrastructure) and cost predictability (no per-query API fees) while accepting certain limitations in model sophistication relative to larger commercial alternatives.

- **Operations Research Optimization (EOQ Model):** Inventory optimization capabilities build upon the Economic Order Quantity model[20] and its extensions, implementing the classical square-root formula along with sensitivity analyses and reorder point calculations.

- **Secure Messaging Infrastructure:** The conversational interface leverages Telegram's Bot API[21], which provides robust message routing, media handling, and interaction management capabilities while supporting security features including client-to-server encryption (TLS/HTTPS) and programmatic access control. The choice of Telegram over alternatives relies on its lighter-weight nature and suitability for small organizational contexts.

[19] Meta AI Research, *Introducing Llama 3: The most capable openly available LLM to date*.
[20] Harris, "How many parts to make at once".
[21] Husen et al., "Chatbot framework using natural language processing (NLP) to assist operational activities".

Chapter 2

State of the Art: Artificial Intelligence, Operations Research, and Secure System Design

This chapter provides a comprehensive review of the theoretical and technical foundations that underpin the Watchtower system. The development of a proactive managerial assistant for e-commerce necessitates a cross-disciplinary approach, integrating three distinct yet interconnected knowledge domains: conversational Artificial Intelligence (AI) and Large Language Models (LLMs), mathematical inventory optimization through Operations Research (OR), and the rigorous principles of cybersecurity for sensitive information systems. The following sections survey the relevant literature within each domain, establishing the scientific basis for the architectural decisions that will be detailed in Chapter 3.

2.1 The Evolution of Conversational AI in Business Intelligence

The paradigm of Business Intelligence (BI) has experienced a steady evolution, moving from static printed reports to interactive digital dashboards, and most recently to conversational interfaces powered by Large Language Models. While traditional BI platforms successfully grant access to vast amounts of data, they often impose a significant cognitive burden on the user. Managers are typically required to formulate precise queries, navigate complex user interfaces, and independently interpret the resulting visualizations[1].

The emergence of LLMs represents a fundamental shift in this field by introducing a Natural Language Interface (NLI). This layer allows decision-makers to interact with their organizational data using plain language, effectively bypassing the technical constraints of SQL and the rigid structures of legacy BI tools. Systems such as Llama 3, developed by Meta AI Research (2024)[2], have demonstrated that open-source models within the 7–70 billion parameter range can achieve reasoning capabilities comparable to top-tier proprietary alternatives. When these models are deployed locally via frameworks such as Ollama, they enable a "Chat with Your Data" experience without the necessity of transmitting sensitive business information to external cloud providers. By converting unstructured natural language into actionable insights, the Watchtower system seeks to break free from the traditional constraints of BI to provide proactive decision support.

However, the integration of LLMs into corporate environments introduces two primary challenges: the hallucination problem, whereby models generate plausible but factually incorrect content, and the data privacy problem. The following subsections examine the

[1] S. Negash. "Business intelligence". In: *Communications of the Association for Information Systems* 13.1 (2004), pp. 177–195.

[2] Meta AI Research, *Introducing Llama 3: The most capable openly available LLM to date*.

two complementary techniques adopted to overcome these limitations within the Watchtower system. It is important to note the architectural precedence relationship between these techniques as implemented: deterministic code generation via PandasAI (Section 2.1.2) constitutes the primary query-handling path, invoked first for all analytical natural language queries submitted through the /ask interface. Retrieval-Augmented Generation (Section 2.1.1) functions as a fallback path, activated when PandasAI fails to produce a valid result — for example, due to an unsupported query structure, a code-generation error, or an empty model response. This precedence ordering reflects a deliberate engineering decision: PandasAI's code-generation approach produces deterministic, auditable outputs with exact numerical precision, making it preferable wherever applicable, while the RAG-based summarisation path provides broader coverage for queries that resist clean translation into executable Pandas code. The subsections below describe each technique on its own terms; the integrated query-routing pipeline in which they operate is described in full in Section 3.3.3.

2.1.1 Retrieval-Augmented Generation (RAG) and Knowledge Grounding

Standard pre-trained LLMs operate as closed systems; regardless of the breadth of their training data, they possess no inherent knowledge of a specific organization's real-time sales history, inventory levels, or customer behavior. When queried about such private data, a standard LLM can only produce generic responses or, in some cases, confident confabulations based on statistical probability rather than factual reality.

Retrieval-Augmented Generation (RAG), as introduced by Lewis et al. (2020)[3], addresses this through a principled architectural separation between the model's parametric knowledge and an external non-parametric knowledge source. In the RAG paradigm, relevant documents or structured data summaries are retrieved from an organizational database and injected into the model's context window as the "ground truth" prior to inference. Within the Watchtower system, RAG is realized through a structured summarization pipeline: sales logs and performance indicators are aggregated by a Pandas-based analytical layer and formatted as concise textual summaries. This approach effectively transforms the model from a general-purpose assistant into a business-specific analyst, while the use of bounded context windows mitigates the risk of hallucination by constraining the generative scope to verifiable data.

[3]P. Lewis et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 9459–9474.

2.1.2 Deterministic Reasoning with Code Generation (PandasAI)

A second critical limitation of LLMs in managerial contexts concerns numerical precision. LLMs are known to produce arithmetically incorrect results when performing multi-step calculations directly—a consequence of their probabilistic token-prediction mechanism, which was not designed for exact computation[4]. In financial or inventory contexts, where even a minor error in a demand forecast can lead to significant capital misallocation, such inaccuracy is operationally unacceptable.

The Watchtower bot addresses this challenge through the "Code as Reasoning" paradigm, exemplified by systems such as PandasAI[5]. Rather than attempting to perform calculations itself, the LLM functions as a programmer: it interprets the natural language query, generates deterministic Python code using the Pandas library, and delegates the execution to a standard Python interpreter. This hybrid architecture combines the linguistic intelligence of the AI with the computational precision of deterministic programming, ensuring that quantitative outputs are both accurate and auditable.

A non-trivial integration challenge arises, however, when the LLM component is hosted locally via Ollama rather than accessed through a cloud API. PandasAI's native LLM integrations are designed around the interface contracts of commercial providers such as OpenAI and Anthropic, which expose standardised REST endpoints and response schemas. The Ollama framework, while functionally equivalent in its generative capabilities, presents a distinct local API at http://localhost:11434/api/generate with a request and response schema that is incompatible with PandasAI's default LLM base class. To bridge this incompatibility, the Watchtower system implements a custom OllamaLLM adapter class, which inherits from PandasAI's abstract LLM base class and overrides the call method to construct and dispatch requests to the Ollama endpoint. The adapter additionally normalises PandasAI's internal Prompt object — which may be passed as either a string or a structured object with a to_string() method — into a plain text representation, and strips Markdown code fences from the model's response prior to returning it to PandasAI's execution engine. This stripping step is essential: Ollama models frequently wrap generated Python code in triple-backtick fences by default, and the presence of these delimiters causes PandasAI's exec() call to raise

[4]T. Brown et al. "Language models are few-shot learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901.

[5]S. Farabi. *PandasAI: The generative AI Python library*. GitHub Repository. 2023. URL: `https://github.com/gventuri/pandas-ai`.

a SyntaxError, silently aborting the query. The adapter also appends an explicit code-only output instruction to every prompt, reinforcing the model's role as a programmer rather than a conversational assistant within this specific execution context. This engineering contribution represents a necessary condition for realising the "Code as Reasoning" paradigm described above in a locally-deployed, privacy-preserving architecture. Consequently, this deterministic pathway is prioritised in the system's logic, serving as the first line of analytical processing before the RAG-based fallback mechanism is considered.

2.2 Inventory Management Using Mathematical Optimization

Effective inventory management remains one of the most consequential operational challenges in the e-commerce sector. Operations Research provides a rigorous mathematical framework for escaping the trap of intuition-based restocking decisions, treating inventory not as a passive outcome of guesswork but as a controllable variable subject to formal optimization. The fundamental tension—commonly referred to as the "inventory trade-off"—lies between two competing cost pressures: maintaining excessive stock levels, which ties up working capital and risks obsolescence, and maintaining insufficient stock, which leads to stockouts, lost revenue, and customer defection[6].

2.2.1 The Mechanics of the Wilson EOQ Model

The Economic Order Quantity (EOQ) model, first formulated by Harris (1913)[7] and later systematized by Wilson (1934)[8], provides a deterministic optimization model for identifying the optimal order quantity by minimizing the total inventory cost function. This function is composed of two conflicting financial components: Ordering Costs (which decrease as order quantity increases due to fewer administrative overheads) and Holding Costs (which increase with larger orders due to storage and capital costs).

The total cost function is expressed as:

$$TC(Q) = \left(\frac{Q}{2}\right) \cdot H + \left(\frac{D}{Q}\right) \cdot S$$

[6] E. A. Silver, D. F. Pyke, and D. J. Thomas. *Inventory and Production Management in Supply Chains*. 4th. CRC Press, 2017.

[7] Harris, "How many parts to make at once".

[8] R. H. Wilson. "A scientific routine for stock control". In: *Harvard Business Review* 13.1 (1934), pp. 116–128.

By setting the derivative of TC with respect to Q equal to zero, we derive the classical EOQ formula:

$$EOQ = \sqrt{\frac{2DS}{H}}$$

In this model, **D (Annual Demand)** represents the total units demanded annually, which Watchtower computes dynamically from historical records. **S (Ordering/Setup Cost)** captures the fixed costs of placing an order, and **H (Holding/Carrying Cost)** represents the annual cost of maintaining one unit in inventory. The practical robustness of this model is significant; sensitivity analysis shows that even a 100% error in estimating D or S produces only a 41% error in the resulting optimal quantity[9]. This makes the EOQ formula highly applicable even when precise parameter estimation is difficult.

2.2.2 The Reorder Point (ROP)

While the EOQ formula determines "how much" to order, the Reorder Point (ROP) identifies "when" to order. The ROP defines the inventory level at which a new replenishment order must be placed to ensure that stock does not reach zero before the new shipment arrives.

The ROP formula links the rate of stock exit to the supplier delivery duration:

$$ROP = d \times L$$

Where **d (Daily Demand)** is the average quantity sold per day, and **L (Lead Time)** is the number of days between placing an order and receiving it. The significance of the ROP within Watchtower is its proactive nature. Instead of waiting for a manual check, the system monitors stock levels in real-time and issues a **Telegram message/push notification** to the manager the moment the inventory touches the computed ROP threshold. This ensures that stockouts are prevented systematically rather than relying solely on human vigilance. In the Watchtower implementation, a safety stock component is incorporated into the ROP formula to provide a buffer against demand variability: $ROP = (d \times L) + (Z \times \sigma_d \times \sqrt{L})$, where $Z = 1.65$ corresponds to a 95% service level and σ_d denotes the standard deviation of daily demand. When historical demand variance is unavailable, σ_d is approximated as $0.5 \times d$, a conservative estimate consistent with standard inventory management practice.

[9]F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. 10th. McGraw-Hill Education, 2015.

2.3 Cybersecurity in Managerial Bot Design

The Watchtower bot processes highly sensitive business intelligence, including profit margins, supplier structures, and strategic performance data. Accordingly, the system's security architecture is a first-class design requirement, addressed through the "Security by Design" paradigm[10]. Two priorities govern this design: data sovereignty and limited access.

2.3.1 Identity-Based Authentication and Whitelist Technology

Conventional authentication via username-password pairs is susceptible to various attack vectors like phishing or brute-force. For a single-principal system like Watchtower, where only one manager is authorized, such mechanisms introduce unnecessary complexity alongside their inherent vulnerabilities[11].

Instead, the system employs a Telegram User ID-based authentication protocol. Every message delivered by the Telegram API carries a numeric user_id that is globally unique and tied to the registered phone number. A security middleware layer intercepts every message and compares the sender's ID against an internally stored whitelist. If the ID does not match, the system silently discards the message, providing no feedback to potential attackers. This creates a secure, private tunnel between the authorized manager and the analytical engine.

2.3.2 Zero Trust Principles and Data Encryption

The architecture adheres to the Zero Trust model, first articulated by Kindervag (2010)[12] and later formalized as a federal standard by NIST[13], which posits that no entity should be trusted without explicit verification. This is operationalized through three measures:

1. **Input Sanitization:** All commands are treated as untrusted and subjected to rigorous validation before reaching the database, preventing injection attacks.

2. **Data Transfer Encryption:** All communications between the Telegram platform and the backend server are protected by **in-transit encryption (TLS/HTTPS)** via a secure

[10] J. H. Saltzer and M. D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[11] J. Richer and A. Sanso. *OAuth 2 in Action*. Manning Publications, 2017.

[12] J. Kindervag. *No more chewy centers: Introducing the zero trust model of information security*. Tech. rep. Forrester Research, 2010.

[13] S. Rose et al. *Zero trust architecture*. Tech. rep. NIST Special Publication 800-207, 2020. DOI: 10.6028/NIST.SP.800-207.

TLS webhook. This ensures the confidentiality and integrity of data as it moves between the client and the server.

3. **Local Inference for Data Sovereignty:** By running the Llama 3 model locally via the Ollama framework, all sensitive company data—sales figures, inventory parameters, and strategic analyses—remains entirely within the organization's own infrastructure. Because the system relies on **client-to-server encryption** and local processing rather than third-party cloud APIs (like OpenAI or Google), it eliminates the risks of third-party data access and ensures complete data sovereignty.

2.3.3 LLM-Layer Prompt Injection Mitigation

The security considerations discussed in Sections 2.3.1 and 2.3.2 address threats at the network and database layers. However, the integration of a Large Language Model as a natural language interface introduces a qualitatively distinct class of vulnerability: prompt injection attacks, classified as LLM01 in the OWASP Top 10 for Large Language Model Applications.[14] In this attack vector, an adversarial user embeds instructions within a natural language input that are intended to override, redirect, or subvert the model's intended behaviour — for example, instructing the system to disregard its operational persona, reveal system-level prompts, or perform actions outside its defined scope.

The Watchtower system addresses this threat through a dedicated input sanitization layer implemented at the entry point of the `answer_natural_language_query` function, which serves all /ask queries. This layer applies a battery of seventeen compiled regular expressions against the lower-cased input string before any data retrieval or model inference is initiated. The patterns target known prompt injection signatures, including directives such as "ignore previous instructions," "you are no longer," "forget your previous," "pretend to be," and "jailbreak," among others. Any input matching one or more of these patterns is immediately rejected with a scope-restriction message, and no further processing is performed. This silent-discard approach, consistent with the Zero Trust philosophy applied at the authentication layer, denies the attacker informational feedback that could be used to refine subsequent attempts.

A secondary sanitization layer addresses a related scope-control concern: queries requesting future predictions or revenue forecasts. Since the system's data corpus is strictly historical,

[14] OWASP Foundation. *OWASP Top 10 for Large Language Model Applications*. Version 1.1. 2023. URL: `https://genai.owasp.org/`.

entertaining such queries would require the LLM to generate speculative content unsupported by the grounding context, materially increasing hallucination risk. Queries matching a set of forward-looking temporal patterns — including "next month," "forecast," "predict," and "expected revenue" — are rejected with an explanatory refusal that redirects the manager toward historically grounded query formulations. Together, these two pre-processing guards constitute the system's first line of defence within the intelligence layer, complementing the network-level and authentication-level controls described in the preceding sections.

2.3.4 Credential Management and Secret Injection

The Watchtower system requires two sensitive credentials at runtime: the Telegram Bot API token, which authenticates the backend server to the Telegram messaging infrastructure, and the Manager Telegram User ID, which seeds the whitelist evaluated by the `@manager_only` authentication decorator. Embedding these values as plaintext constants in source code would constitute a critical security vulnerability, exposing them to version control history, log outputs, and any party with read access to the codebase.

The system addresses this through environment-variable-based secret injection, a practice mandated by the Twelve-Factor App methodology for production software and recommended by the NIST Secure Software Development Framework.[15] In the prototype deployment, secrets are stored in the platform's managed secret store (Kaggle Secrets) and retrieved at runtime via a dedicated secrets client API, ensuring that credential values are never materialised in the source files or execution logs.

The retrieved values are assigned to module-level constants (`TELEGRAM_TOKEN` and `MANAGER`) immediately upon kernel initialisation and are consumed solely by the authentication and bot-construction components. In a production deployment targeting a dedicated server or virtual machine, the same pattern would be realised through a `.env` file parsed by a library such as `python-dotenv`, or through an operating-system-level environment variable injection mechanism, with the secrets store replaced by a dedicated vault service such as HashiCorp Vault or AWS Secrets Manager. This design ensures that migrating the system from the prototype environment to production requires no modification to the application logic; only the secret-injection mechanism changes, preserving the architectural integrity of the security layer.

[15] NIST. *Secure Software Development Framework (SSDF), Version 1.1.* Tech. rep. NIST Special Publication 800-218, 2022. URL: `https://csrc.nist.gov/pubs/sp/800/218/final`.

Chapter 3

System Design and Architecture

The two preceding chapters have established both the managerial necessity and the theoretical basis for the Watchtower system. Chapter 1 identified the cognitive burden imposed by information overload on e-commerce managers, and Chapter 2 surveyed the distinct bodies of knowledge—conversational artificial intelligence, inventory optimisation theory, and secure system design—that collectively inform the proposed solution. The present chapter translates those conceptual foundations into a concrete architectural blueprint. It proceeds in three stages: Section 3.1 describes the layered global architecture and the internal processing pipeline; Section 3.2 defines the relational data model and the logic that governs real-time stock monitoring; and Section 3.3 provides a complete functional specification of every bot command exposed to the manager.

3.1 Global Architecture (The Watchtower)

The Watchtower system is grounded in a Hybrid Modular Architecture—a design philosophy that deliberately separates concerns between exact, rule-governed computation and probabilistic, language-mediated reasoning. Conventional business intelligence platforms collapse these two modes of analysis into a single visualisation layer, obliging the manager to perform the interpretive labour manually. The architecture proposed here assigns each mode to a specialised engine operating within a clearly delimited layer of the system, thereby ensuring that numerical outputs remain auditable and that natural language outputs are contextually grounded in verified data.

The rationale for this separation is both epistemological and practical. From an epistemological standpoint, inventory calculations demand deterministic precision: a reorder point that is even marginally incorrect can translate directly into capital loss or stockout. Conversely, strategic recommendations are inherently probabilistic; they require the capacity to synthesise ambiguous signals—declining weekend conversion rates, seasonal demand fluctuations, shifting competitor pricing—into actionable guidance. No single computational paradigm handles both tasks with equal fidelity. From a practical standpoint, modular separation facilitates independent testing, targeted maintenance, and future extensibility, as identified in foundational software architecture literature.[1]

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 4th ed. Addison-Wesley Professional, 2021.

3.1.1 The Hierarchical Layered Model

The system is organised into three hierarchically ordered layers, each with a well-defined responsibility boundary. This layering pattern, widely recommended in enterprise application design,[2] ensures that no layer assumes responsibilities belonging to another, thereby containing the propagation of errors and simplifying security enforcement.

1. **The Interface Layer (User Interaction).** The Telegram Bot API serves as the exclusive entry point to the system. This choice reflects a deliberate prioritisation of accessibility: rather than requiring the manager to navigate a dedicated web dashboard, the interface is embedded within a messaging application already installed on the manager's personal device. The layer handles message routing, delivers notifications and analytical reports, and enforces channel-level encryption through SSL/TLS. By confining all user-facing concerns to this layer, the underlying analytical logic remains entirely shielded from the presentation tier.

2. **The Logic Layer (System Orchestration).** The Python-based backend constitutes the system's orchestration hub. It is responsible for three sequentially ordered functions: first, authenticating every incoming request against a statically defined manager whitelist before any processing commences; second, routing validated requests to the appropriate engine; and third, managing asynchronous database transactions. Every input is sanitised at this layer before it reaches any downstream module, establishing a defensive perimeter consistent with the Zero Trust principles discussed in Chapter 2.

3. **The Intelligence Layer (Computational Core).** This layer contains the two specialised engines that perform all substantive analysis. Its components are:

    - **The Analytical Engine (Numerical Core):** employs the NumPy and Pandas libraries to execute deterministic computations—aggregating sales time-series, computing Economic Order Quantities, and evaluating reorder thresholds with exact arithmetic. Its outputs are fully reproducible and auditable, as they are produced by a deterministic interpreter rather than by a language model.
    - **The Strategic Engine (Cognitive Core):** deploys a local instance of Llama 3 (8B parameters)[3] via the Ollama framework to conduct qualitative reasoning over

[2] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
[3] Meta AI Research, *Introducing Llama 3: The most capable openly available LLM to date*.

structured data summaries prepared by the Analytical Engine. Because Llama 3 operates entirely on the organisation's own hardware, sensitive commercial data never traverses an external network boundary, ensuring complete data sovereignty.

3.1.2 Internal Server Components (The Brain Logic)

Within the Logic Layer and Intelligence Layer, three specialised components operate in a sequential pipeline to handle every managerial request.

The Secure Router (Access Control Gateway). The router constitutes the first point of contact for every message that passes through the Telegram API. It applies a Python decorator—designated @manager_only—that intercepts the request before any business logic executes. The decorator extracts the numeric Telegram User ID embedded in the message metadata and verifies it against a whitelisted value stored in the server's environment configuration. Should the identifiers fail to match, the connection is terminated immediately and silently, providing no informational response to the unauthorised sender. This silent-discard policy denies potential adversaries the feedback necessary for enumeration attacks.

The Analytical Engine (Numerical Processing). Upon successful authentication, the Analytical Engine receives the validated request. It queries the SQLite database to retrieve current inventory parameters and historical sales records, then applies the Pandas library to aggregate transaction rows into statistical summaries. When an inventory optimisation request is issued, this engine executes the Wilson EOQ formula and the Reorder Point calculation, returning mathematically precise restocking recommendations. Because these calculations are performed by a deterministic interpreter, their outputs are both reproducible and auditable.

The Strategic Engine (AI Inference Hub). The Strategic Engine operates on the structured summaries produced by the Analytical Engine. Rather than receiving raw transactional data—which would exceed the model's context capacity and introduce noise—it receives concise, pre-aggregated textual representations of the business's recent performance. The engine submits these summaries to Llama 3 via a local Ollama API call and returns the model's qualitative analysis to the manager. This architecture instantiates the Retrieval-Augmented Generation pattern described in Chapter 2, grounding the model's outputs in verifiable organisational data. By operating Llama 3 locally through Ollama, the system guarantees that no company data is transmitted to external cloud services.

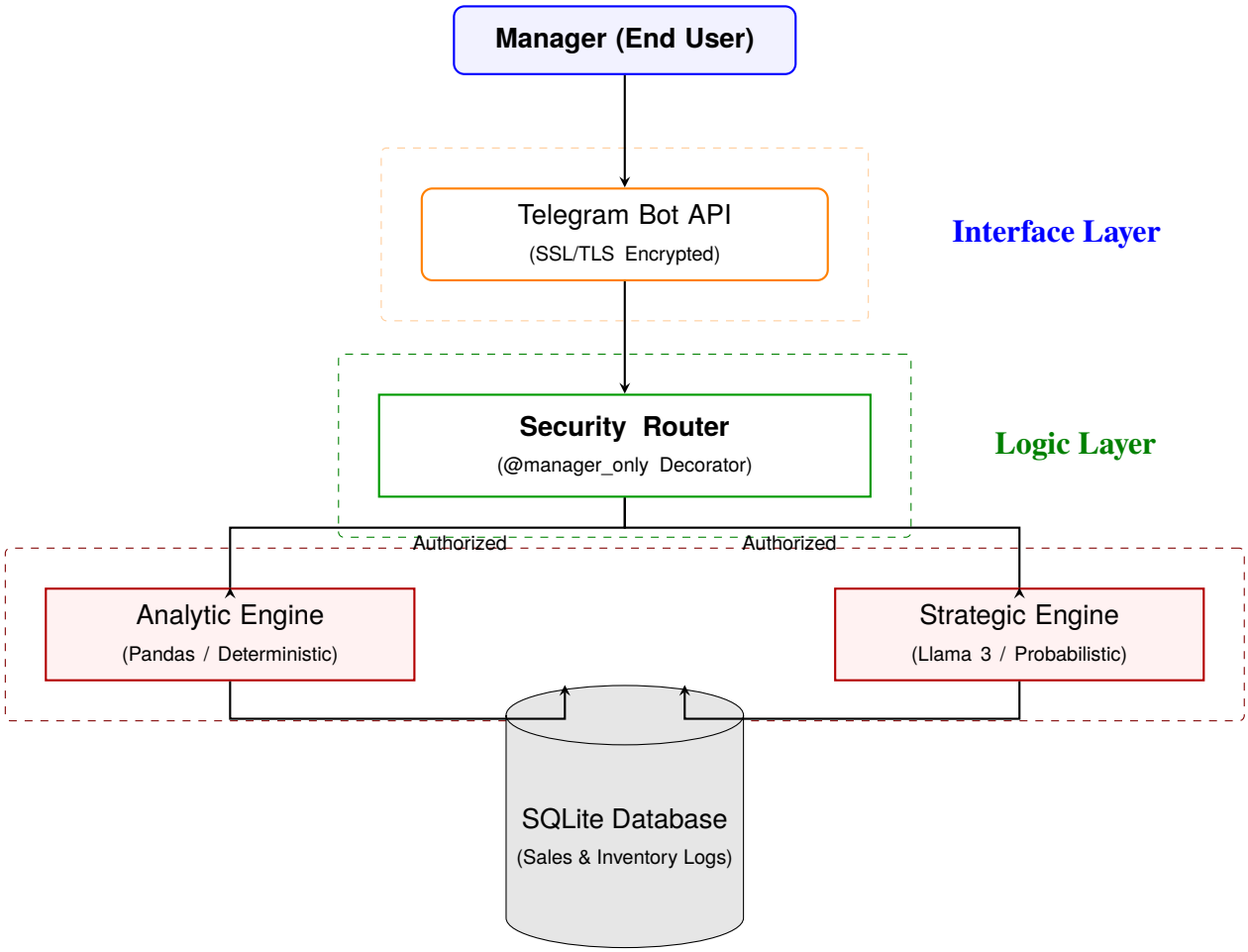


Figure 3.1: System Architecture of the Watchtower Managerial Bot

3.1.3 The Process of Processing Requests

To make the architecture concrete, consider the lifecycle of a /strategy command from initiation to response. The sequence proceeds through five strictly ordered stages:

1. **Command Transmission:** The manager issues the /strategy command via the Telegram mobile client. The message is conveyed to the backend server over an SSL/TLS encrypted webhook.

2. **Verification:** The Secure Router's @manager_only decorator extracts the sender's User ID and verifies it against the whitelist. Only upon successful verification does execution proceed.

3. **Analysis:** The Analytical Engine queries the Sales_Log table, computes a week-over-week performance summary, and formats the output as a structured textual context block.

4. **Inference:** The context block is submitted to the Strategic Engine, which forwards it to the local Llama 3 model with an instruction to produce a SWOT analysis (Strengths, Weaknesses, Opportunities, Threats) calibrated to the data.

5. **Output:** The model's response is parsed, formatted, and dispatched to the manager's device as a structured Telegram message.

3.2 Data Modeling (The Database Schema)

The persistence layer serves as the system's institutional memory—the substrate from which every analytical output is ultimately derived. Its design departs from the conventional practice of treating a database as a passive transaction archive; instead, the schema is structured to make decision-relevant parameters immediately computable without requiring complex, multi-step transformations at query time.

A relational SQLite schema was selected on both technical and contextual grounds. Technically, the relational model enforces referential integrity between products, inventory parameters, and sales records, preventing the data inconsistencies that would otherwise undermine the validity of optimisation outputs.[4] Contextually, SQLite's zero-configuration, serverless

[4] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill, 2019.

deployment model is well-suited to the small and medium enterprise setting described in Section 1.4.2, where dedicated database administration is typically unavailable.[5]

Prototype Deployment Context and the In-Memory Overlay Mechanism. The implementation described in this thesis was developed and validated within a Kaggle Notebook environment, a cloud-based kernel that mounts external datasets as read-only file-system paths. Because the SQLite database is accessed at a read-only mount point (/kaggle/input/. . . /ecommerce_data.db), direct INSERT and UPDATE operations against the Inventory_Config and Sales_Log tables are not permissible at the file-system level. To preserve the architectural validity of the system while accommodating this constraint, a session-scoped in-memory overlay mechanism was implemented in Python. This overlay, represented as a dictionary keyed on Product_ID, intercepts all inventory write operations — including stock decrements, EOQ updates, and ROP recalibrations — and stores them in process memory for the duration of the kernel session. All read operations performed by the Analytical Engine transparently merge the persisted database records with any overlay entries before returning data to the caller, ensuring that the computational logic remains identical to what a persistent, writable deployment would produce. It must be noted that this mechanism constitutes a deliberate prototype constraint: in a production deployment, the system would operate against a locally hosted, fully writable SQLite instance, and all overlay operations would be replaced by standard UPDATE and INSERT SQL statements. The architectural design — the three-table relational schema, the referential integrity constraints, and the EOQ/ROP computation pipeline — is identical in both configurations, with the overlay serving solely as a write-path adapter for the read-only evaluation environment.

3.2.1 Key Entities & Tables

The schema comprises three interrelated tables. Their structure is designed such that the Analytical Engine can execute all required computations—EOQ, ROP, and trend analysis—through straightforward SQL queries supplemented by Pandas aggregations, without necessitating schema transformations at runtime.

1. **Products Table.** This table holds the permanent attributes of every product in the catalogue. Its columns are: Product_ID (primary key), Name, Category, Unit_Price, and Cost_Price. The distinction between Unit_Price and Cost_Price is intentional: the

[5]Christopher J. Date. *An Introduction to Database Systems*. 8th ed. Addison-Wesley, 2004.

margin it encodes provides the financial basis for evaluating whether increased ordering frequency is economically justified.

2. **Inventory_Config Table.** This table stores the operational parameters required by the Wilson EOQ model and the Reorder Point formula. Its columns are: Product_ID (foreign key referencing the Products table), Current_Stock, Ordering_Cost_S, Holding_Cost_H, and Lead_Time_L. The underscore-suffix naming convention was adopted deliberately to eliminate ambiguity in SQL identifier parsing, where parenthetical characters are syntactically reserved; the suffixes serve as self-documenting reminders of the operational role each parameter plays in the Wilson EOQ and Reorder Point formulae. Because the EOQ model is demonstrably robust to parameter estimation error — a 100% error in either D or S propagates to only a 41% deviation in the resulting optimal quantity — the values stored in Ordering_Cost_S and Holding_Cost_H require only periodic managerial review rather than transaction-level updates. Current_Stock, however, must remain continuously synchronised with every recorded sale event, as detailed in Section 3.2.2.

3. **Sales_Log Table.** This is the system's most dynamic table. Each row records a single sale event with the columns: Sale_ID (primary key), Product_ID (foreign key), Quantity_Sold, Timestamp. The temporal granularity afforded by Timestamp enables the Analytical Engine to compute moving averages of daily demand, detect seasonality patterns, and supply the Strategic Engine with period-over-period comparisons.

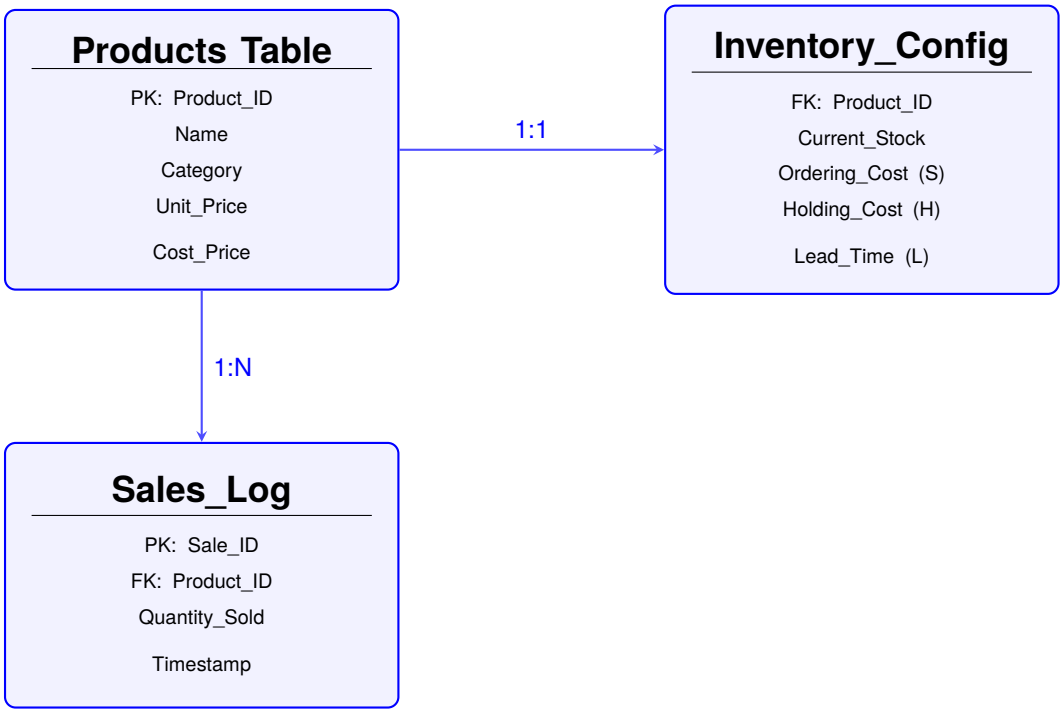


Figure 3.2: Entity-Relationship Diagram (ERD) of the Watchtower Database. Column names reflect the implemented schema: Ordering_Cost_S, Holding_Cost_H, and Lead_Time_L in the Inventory_Config table adopt an underscore-suffix convention to ensure unambiguous SQL identifier resolution. All write operations to this table during the prototype deployment are mediated through a session-scoped in-memory overlay, as described in Section 3.2.2.

3.2.2 Dynamic Update Logic (Data-in-Motion Analysis)

A central architectural decision concerns the appropriate frequency of optimisation recalculation. An earlier formulation of this chapter recalculated the EOQ upon every individual sale event. This approach is both computationally unnecessary and conceptually erroneous: the EOQ is a strategic planning quantity derived from annual demand estimates, ordering cost structures, and holding cost rates—parameters whose meaningful values change on a timescale of weeks or months, not individual transactions. Recalculating EOQ per sale would amplify the effect of transient demand noise and produce recommendations that fluctuate without strategic justification.

Upon a "New Sale" being recorded in the Sales_Log table, the following four-stage sequence is triggered:

- **Inventory Update in Real-Time:** The Current_Stock field in Inventory_Config is immediately decremented by the Quantity_Sold value recorded in the new Sales_Log

entry. This operation ensures that the digital inventory record remains continuously aligned with the physical state of the warehouse, providing an accurate stock count at every moment.

- **Proactive Threshold Alerting:** Following the stock update, the system immediately compares the updated `Current_Stock` value against the stored Reorder Point threshold. If `Current_Stock` $\leq$ ROP, a structured proactive alert is dispatched to the manager's Telegram client, displaying the product name, current stock level, the ROP value, and the recommended EOQ order quantity. This check occurs on every transaction. Under normal operating conditions — that is, after at least one execution of the `/optimize` command or the weekly recalibration job — the ROP value used in this comparison is retrieved directly from the pre-computed value stored in the inventory overlay, and no EOQ recalculation is performed at this stage, consistent with the periodic-planning philosophy described above. However, a cold-start condition arises when a sale event is processed before any optimisation run has populated the overlay with computed ROP and EOQ values. In this scenario, the system applies a graceful degradation fallback: it computes the ROP on demand using `compute_rop(d, L)` — where `d` is derived from the current `Sales_Log` and `L` is retrieved from `Lead_Time_L` in `Inventory_Config` — and similarly computes the EOQ using `compute_eoq(D, S, H)` with current parameter values. This fallback ensures that the proactive alert mechanism remains fully functional from the moment the system receives its first sale event, without requiring a prior explicit optimisation invocation. The fallback computation is indistinguishable in its mathematical content from a scheduled recalibration; it differs only in its trigger condition, and it does not alter the overlay's EOQ/ROP state, which remains reserved for periodic recalibration.

- **Periodic EOQ and ROP Recalibration:** The EOQ and ROP values are recalculated on a scheduled basis — weekly by default, or upon explicit managerial request via the `/optimize` command. This periodic cadence reflects the epistemological status of these parameters as strategic planning quantities calibrated to sustained demand trends rather than reactive transaction-level variables. At each recalibration event, the Analytical Engine executes the full optimisation pipeline (`run_optimization_pipeline`): it aggregates the `Sales_Log` to compute an updated average daily demand d for each product, derives the annualised demand figure as $D = d \times 365$, retrieves the current

`Ordering_Cost_S` and `Holding_Cost_H` from `Inventory_Config`, and applies the Wilson EOQ formula to produce a revised optimal order quantity. The revised ROP ($d \times$ `Lead_Time_L`) is computed concurrently, and both values are written to the in-memory overlay, from which all subsequent threshold comparisons and alert messages draw their data. The automatic weekly recalibration is implemented as a background coroutine registered with the Telegram `job_queue` scheduler at an interval of 604,800 seconds (seven days), with an initial delay of 300 seconds following bot startup. Upon completion of each scheduled run, the job dispatches a structured summary notification to the manager's Telegram client, reporting the number of products recalibrated and the count of products whose current stock falls at or below the newly computed ROP threshold. This notification ensures that scheduled recalibrations remain transparent to the manager and do not silently alter the alert thresholds without acknowledgement.

- **AI Context Synchronization:** The new data point is incorporated into the context window made available to the Strategic Engine. Should the manager subsequently issue a /strategy or /ask command, the AI model will have access to the most recent sales data, ensuring that its qualitative recommendations remain grounded in current operational reality rather than in stale aggregates.

This structure preserves the real-time responsiveness required for stockout prevention while maintaining the statistical stability necessary for sound inventory optimisation. It accurately reflects standard Operations Research practice, in which EOQ is treated as a periodic planning parameter calibrated to sustained demand trends, rather than a variable reactive to individual transactions.

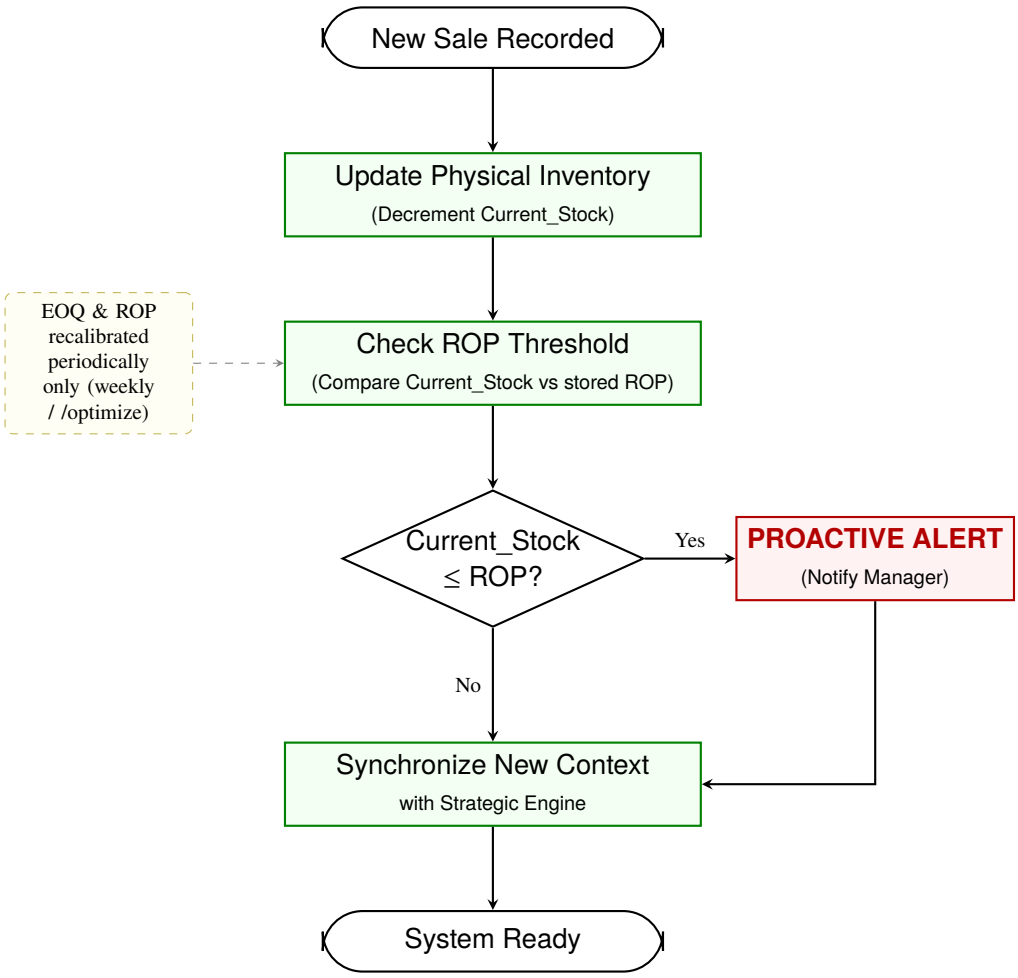


Figure 3.3: The Dynamic Update and Analysis Workflow (Proactive Logic)

3.3 Functional Specification (Bot Commands)

The Watchtower's analytical and strategic capabilities are surfaced to the manager through a structured command vocabulary. Each command maps to a well-defined processing pathway within the backend—either to the Analytical Engine, the Strategic Engine, or both in sequence. The command set is organised into three functional tiers reflecting increasing analytical depth: system control, operational intelligence, and strategic decision support. The design of this command interface follows established principles of conversational system usability,[6] ensuring that the cognitive load imposed on the manager is minimised while the analytical power of the underlying engines remains fully accessible.

3.3.1 Utility and System Control Commands

The following commands are designed to ensure smooth onboarding and continuous in-application guidance:

/start — Session Initialisation. This command activates the Security Router's authentication sequence. The manager's Telegram User ID is verified against the whitelist. Upon successful verification, the system responds with a brief session summary confirming the authenticated identity and listing available commands. An unrecognised ID receives no response whatsoever, consistent with the silent-discard policy described in Section 3.1.2.

/help — Command Reference. This command returns an in-application reference document enumerating all available commands, their expected inputs, and the type of output each produces. Its purpose is to eliminate the need for external documentation and to ensure that a manager unfamiliar with a particular command can obtain guidance without leaving the application.

3.3.2 Operations and Visual Intelligence Commands

Designed to give the quickest possible snapshot of the current state of the business:

/inventory — Current Stock State. This command queries the Inventory_Config table and returns a structured list of all products alongside their current stock levels, with a status indicator distinguishing between adequate stock, low stock (Current_Stock ≤ ROP), and critical stock (Current_Stock = 0). It provides the manager with an instantaneous, unambiguous snapshot of the physical inventory without requiring manual database interrogation.

[6] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

/report — Visual Intelligence. This command instructs the Analytical Engine to aggregate the Sales_Log over a configurable time window (defaulting to the previous 30 days) and generate a set of Matplotlib charts conveying sales velocity by product category, day-of-week distribution, and week-over-week revenue comparison. The resulting graphics are transmitted to the manager as Telegram image messages, translating raw transaction data into perceptually interpretable trend representations.

3.3.3 Optimize and Strategy Commands

This group of commands represents the core analytical power of the system, fusing mathematical rigour with qualitative intelligence:

/optimize — Mathematical Optimizer. This command triggers a full execution of the Operations Research optimisation pipeline described in Chapter 2. The Analytical Engine aggregates the Sales_Log to derive current demand estimates, retrieves ordering and holding cost parameters from Inventory_Config, and applies the Wilson EOQ formula to compute revised optimal order quantities and updated Reorder Points for every product in the catalogue. The output is formatted as a prioritised restocking list, identifying products that require immediate procurement alongside their mathematically derived order quantities.

/strategy — Strategy Briefing. This command initiates the full Analytical-to-Strategic pipeline described in Section 3.1.3. The Analytical Engine produces a structured performance summary encompassing revenue trends, category-level sales dynamics, and anomalous demand patterns. This summary is submitted to the Strategic Engine, which applies Llama 3 to generate a SWOT analysis contextualised to the current operational data. The resulting briefing provides the manager with qualitative strategic recommendations grounded in the organisation's own verified performance metrics.

/ask — Conversational Business Intelligence. This command provides an open-ended conversational interface for ad-hoc natural language queries that do not conform to the structured templates of the preceding commands. A manager may, for instance, query: "Which product category experienced the steepest decline in the past fortnight?" The backend processes this query through a four-stage cascading pipeline, ordered by analytical precision and computational cost.

Stage 1 — Security and Scope Pre-processing: Before any data is retrieved or any model is invoked, the raw input string is evaluated against two sanitization layers: a seventeen-

pattern prompt injection guard that silently rejects adversarial instruction-override attempts, and a forward-looking query refusal filter that declines prediction or forecasting requests, constraining the system to historically grounded responses. Any off-topic query matched against a domain keyword exclusion list is similarly rejected at this stage. These guards are described in full in Section 2.3.3.

Stage 2 — Deterministic Keyword Resolution: For a defined set of common analytical query types — including peak sales day identification, top-performing product retrieval, and aggregate revenue queries — the system applies a deterministic, rule-based resolution layer that computes the answer directly from the merged Pandas DataFrame without invoking any language model. This layer maximises response precision and minimises latency for high-frequency query patterns.

Stage 3 — PandasAI Code Generation (Primary LLM Path): Queries that pass Stage 2 without resolution are forwarded to the PandasAI engine, which submits the query and the merged sales-inventory-product DataFrame to the local Llama 3 model via the custom `OllamaLLM` adapter. The model generates deterministic Python/Pandas code, which is executed by the Python interpreter; the numerical result is subsequently passed to a second Llama 3 call for natural language formatting. This stage implements the "Code as Reasoning" paradigm described in Section 2.1.2 and constitutes the primary query-handling path for its precision and auditability.

Stage 4 — RAG Summarisation Fallback: Should Stage 3 fail — due to a code generation error, an unsupported query structure, or an empty model response — the system falls back to the Retrieval-Augmented Generation path. The `build_sales_summary` function aggregates the `Sales_Log` into a structured textual context block encompassing total revenue, unit volumes, category-level breakdowns, and top and bottom performers over the preceding thirty days. This context, together with an extended product- and inventory-level context block, is injected into the Llama 3 prompt as the verified ground truth, constraining the model's response to verifiable organisational data and mitigating hallucination risk as described in Section 2.1.1. The final natural language response is formatted and dispatched to the manager's Telegram client.

This cascading architecture ensures that the system degrades gracefully under varying query complexity: precise, deterministic answers are delivered wherever possible, with probabilistic language model inference invoked only when the query genuinely requires it.

3.3.4 Simulation and Empirical Validation Commands

The following command was designed to support the empirical validation objective articulated in Section 1.4.1 (Objective 5). It provides a controlled, interactive mechanism for injecting sale events into the system during demonstration and testing scenarios, enabling direct observation of the proactive alert pipeline described in Section 3.2.2 without requiring a live order stream from an external source.

/simulate_sale — Interactive Sale Injection and Proactive Alert Trigger: This command accepts two positional arguments: a Product_ID integer and a Quantity integer, forming the syntax `/simulate_sale [Product_ID] [Qty]`. Upon invocation, the handler executes the following sequence. First, it performs input validation, rejecting non-integer arguments and product identifiers absent from the Products table with an informative error message. Second, the validated sale event is appended to the in-memory sales log via the `append_sale` function, and the corresponding Current_Stock value in the inventory overlay is decremented by the specified quantity via `decrement_stock`. Third, the handler retrieves the current Reorder Point for the affected product — either from a pre-computed value stored in the overlay by a prior /optimize invocation, or by computing it on demand using the `compute_rop` function as a cold-start fallback. Finally, the updated stock level is compared against the ROP threshold: if Current_Stock $\leq$ ROP, a structured proactive alert is immediately dispatched to the manager's Telegram client, displaying the product name, current stock level, the ROP value, and the recommended Economic Order Quantity. This command therefore serves as the primary interactive entry point for demonstrating the real-time stockout prevention mechanism that constitutes one of the system's core operational contributions.

Chapter 4

Implementation and Results

4.1 Technical Environment and Deployment Setup

The Watchtower system was developed and validated within a Kaggle Notebook environment — a cloud-based Python kernel providing managed GPU resources and integrated secret management. This environment was selected for its reproducibility and native support for SQLite-based datasets. Two runtime credentials are required by the system: the Telegram Bot API token and the authorised Manager Telegram User ID. Both are retrieved at kernel initialisation via Kaggle's UserSecretsClient, in compliance with the Twelve-Factor App methodology and the NIST Secure Software Development Framework, as described in Section 2.3.4:

```
from kaggle_secrets import UserSecretsClient
_secrets = UserSecretsClient()
TELEGRAM_TOKEN = _secrets.get_secret("TELEGRAM_BOT_TOKEN")
MANAGER = _secrets.get_secret("MANAGER_TELEGRAM_ID")
WHITELISTED_MANAGER_ID = int(MANAGER)
```

This pattern ensures that credential values are never materialised in source files or execution logs.

The deployment context imposed one significant architectural constraint: datasets mounted in Kaggle are exposed as read-only file-system paths under /kaggle/input/, preventing direct INSERT and UPDATE operations against the SQLite database file. This constraint is addressed through a dual-write mechanism: a session-scoped in-memory dictionary (_inventory_overlay) and a writable SQLite session database (SESSION_DB_PATH = "/kaggle/working/watchtower_session.db"). The session database persists across kernel restarts, providing durability beyond simple in-process memory.

4.1.1 Software Stack and Library Dependencies

Table 4.1 enumerates the complete technical stack. Each component reflects the deliberate engineering decisions documented in the preceding chapters.

Component	Technology / Version	Role in System
Language	Python 3.10	Primary implementation language
Bot Framework	python-telegram-bot 20.7	Async Telegram API interface and job scheduler
AI Engine	Ollama + Llama 3 (8B parameters)	Local LLM inference — Strategic Engine
Code-Gen BI	PandasAI + custom OllamaLLM adapter	Deterministic query resolution — primary /ask path
Data Layer	SQLite + Pandas 2.2.2	Persistent storage and in-memory DataFrames
Numerical Computing	NumPy (<2)	EOQ and ROP formula execution
Visualisation	Matplotlib 3.x	Sales charts for /report command
Secret Management	Kaggle UserSecretsClient	Runtime credential injection
Scheduler	Telegram JobQueue (APScheduler)	Weekly recalibration + background jobs
Async Support	nest_asyncio	asyncio event loop inside Jupyter kernel

Table 4.1: Complete software stack of the Watchtower system.

4.1.2 Database Initialisation and Dual-Write Persistence

At kernel startup, the system establishes two SQLite connections. The first connects to the read-only dataset path for all persistent product, inventory, and historical sales records. The second targets the writable session database, initialised with two tables on first run:

```
SESSION_DB_PATH = "/kaggle/working/watchtower_session.db"
# session_inventory (Product_ID PK, Current_Stock, EOQ REAL, ROP REAL)
# session_sales (id PK AUTOINCREMENT, Product_ID,
#                Quantity_Sold, Timestamp TEXT)
```

All inventory write operations — stock decrements, EOQ updates, and ROP recalibrations — are simultaneously written to both the _inventory_overlay dictionary (for fast in-process reads) and the session_inventory table (for persistence across restarts). A 30-second TTL cache on load_inventory() avoids redundant SQL reads, invalidated explicitly by every write path via _invalidate_inventory_cache(). On bot startup, the overlay is repopulated from the session database via _restore_session_from_db(), ensuring that inventory state from prior sessions is immediately available.

4.2 Implementation of Core Modules

This section documents the implementation of the five principal modules: the security layer, the inventory optimisation engine, the OllamaLLM adapter, the /ask query pipeline, and the proactive alert mechanism with its weekly recalibration job.

4.2.1 The Security Layer: manager_only Decorator

Every command handler is wrapped by the manager_only decorator, implementing two sequential checks. First, the decorator extracts the numeric Telegram User ID and compares it against WHITELISTED_MANAGER_ID. A non-matching sender causes immediate silent termination with no response dispatched — consistent with Zero Trust principles. Second, a per-user rate limiter enforces a minimum interval of two seconds between commands:

```
RATE_LIMIT_S = 2
_last_cmd_time = defaultdict(float)

def manager_only(handler_func):
@wraps(handler_func)
async def wrapper(update, context):
sender_id = update.effective_user.id \
if update.effective_user else None
# Step 1 - Identity check (silent discard)
if sender_id != WHITELISTED_MANAGER_ID:
print(f"[SECURITY] Rejected user_id={sender_id}")
return
# Step 2 - Rate-limit check
now = _time.monotonic()
elapsed = now - _last_cmd_time[sender_id]
if elapsed < RATE_LIMIT_S:
remaining = RATE_LIMIT_S - elapsed
await update.message.reply_text(
f"Slow down - wait {remaining:.1f}s")
return
```

```
    _last_cmd_time[sender_id] = now
    # Step 3 - Execute real handler
    return await handler_func(update, context)
    return wrapper
```

The silent-discard policy at Step 1 is a deliberate security choice: any informational response to an unauthorised sender would assist enumeration attacks. The rate limiter at Step 2 prevents command-flooding abuse even by the authenticated manager.

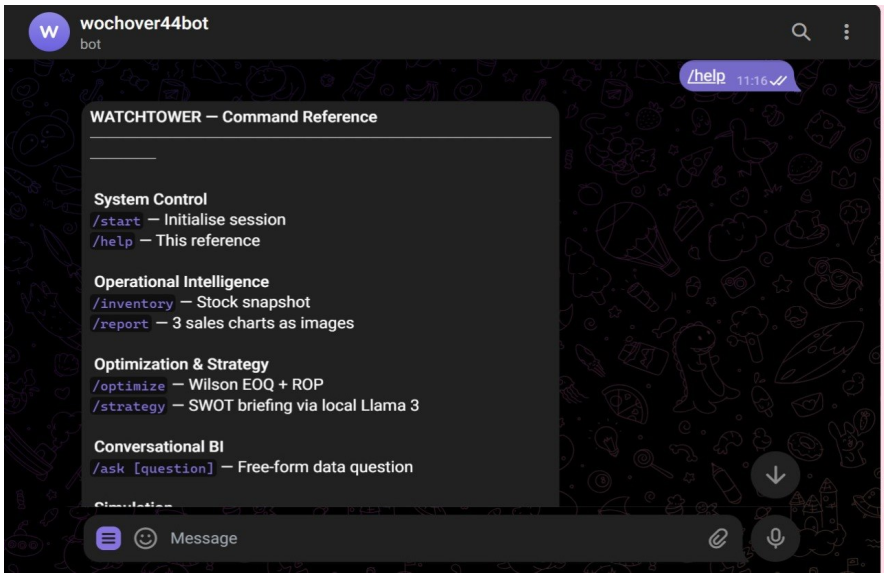


Figure 4.1: Bot command reference interface displayed to the manager after successful identity verification.

4.2.2 The Inventory Optimisation Engine

The optimisation engine implements the Wilson EOQ formula and an extended Reorder Point formula incorporating a safety stock component. The compute_daily_demand function uses the window from first sale to today as the denominator, preventing inflated demand estimates for single-day products:

```
    def compute_daily_demand(sales_df, product_id):
    prod_sales = sales_df[sales_df["Product_ID"] == product_id]
    if prod_sales.empty: return 0.0
    total_qty = prod_sales["Quantity_Sold"].sum()
```

```
first_to_last = (prod_sales["Timestamp"].max() -
prod_sales["Timestamp"].min()).days
first_to_today = (pd.Timestamp.now() -
prod_sales["Timestamp"].min()).days
days = max(first_to_last, first_to_today, 1)
return total_qty / days

def compute_eoq(D, S, H):
if H <= 0 or D <= 0: return 0.0
return float(np.sqrt((2 * D * S) / H))

Z_SERVICE_LEVEL = 1.65 # 95% service level

def compute_rop(d, L, sigma_d=None):
if sigma_d is None:
sigma_d = 0.5 * d
safety_stock = Z_SERVICE_LEVEL * sigma_d * float(L ** 0.5)
return float(d * L) + safety_stock
```

As documented in Section 2.2.2, the safety stock term Z x sigma_d x sqrt(L) buffers against demand variability. Z = 1.65 corresponds to a 95% service level. When sigma_d is unavailable, it is approximated as 0.5 x d. Passing sigma_d=0 collapses the formula to the basic d x L, used in unit tests to verify the base formula independently.

The full optimisation pipeline (run_optimization_pipeline) iterates over every row in Inventory_Config, computing d, D = d x SELLING_DAYS_PER_YEAR, EOQ, and ROP. Results are written simultaneously to _inventory_overlay and to the session_inventory table, sorted by Urgency (Current_Stock minus ROP) — placing the most critical products first.

4.2.3 The Custom OllamaLLM Adapter

PandasAI's native LLM integrations target commercial provider schemas. The Ollama framework exposes a local endpoint at http://localhost:11434/api/generate incompatible with PandasAI's default base class. The OllamaLLM adapter bridges this gap:

```
class OllamaLLM(_PandasAI_LLM):
@property
def type(self): return "ollama"

def call(self, instruction, context=None, suf=""):
# Normalise Prompt object -> plain string (3 possible types)
if hasattr(instruction, "to_string"):
prompt_text = instruction.to_string()
elif hasattr(instruction, "value"):
prompt_text = str(instruction.value)
else:
prompt_text = str(instruction)

if context: prompt_text += "\n" + str(context)
if suf: prompt_text += "\n" + suf

prompt_text += (
"\n\nIMPORTANT: Respond with ONLY valid Python code "
"using pandas. No explanations, no markdown fences."
)
raw = query_llama3(prompt_text)
cleaned = re.sub(r"^```(?:python)?\s*", "", raw.strip())
cleaned = re.sub(r"\s*```$", "", cleaned.strip())
return cleaned
```

Three engineering details are essential. First, the normalisation handles all three Prompt types that different PandasAI versions may pass. Second, the explicit code-only instruction reinforces the model's programmer role. Third, the fence-stripping is non-negotiable: Ollama wraps generated Python in backtick fences by default, causing PandasAI's exec() to raise a SyntaxError. The adapter also handles PandasAI import path variability across library versions by attempting three candidate paths in sequence.

4.2.4 The Four-Stage /ask Query Pipeline

The /ask command processes every natural language query through a four-stage cascading pipeline within answer_natural_language_query, ordered by analytical precision and computational cost. The Llama 3 language model is invoked only at the final stage.

Stage 1 — Security and Scope Pre-processing. Before any data retrieval or model invocation, the raw input is evaluated against two sanitization layers: (a) seventeen compiled regular expressions targeting prompt injection signatures (e.g. "ignore previous instructions", "jailbreak", "pretend to be"), and (b) a forward-looking query filter rejecting prediction requests matching patterns such as "forecast", "next month", "expected revenue". An additional off-topic keyword exclusion list rejects queries unrelated to the e-commerce dataset. Any match causes immediate rejection with a scope-restriction message.

Stage 2 — Deterministic Keyword Resolution. Queries that pass Stage 1 are routed to _answer_direct_questions, which implements a dispatch table of (predicate, handler) pairs. Over 25 query patterns are resolved by direct Pandas computation on the merged DataFrame — including day-of-week peak identification, category comparisons, weekly revenue breakdowns, ROP-breach detection, profit margin analysis, and product-level revenue queries — with no language model involvement.

Stage 3 — PandasAI Code Generation (Primary LLM Path). Queries not resolved by Stage 2 are forwarded to the PandasAI engine, which submits the query and the merged DataFrame to Llama 3 via the OllamaLLM adapter. The model generates deterministic Python/Pandas code, executed by the Python interpreter; the numerical result is then passed to a second Llama 3 call for natural language formatting.

Stage 4 — RAG Summarisation Fallback. Should Stage 3 fail (code generation error, empty response, or error prefix), the system activates the RAG path. build_sales_summary and _build_extended_sales_context assemble a structured textual context block injected into a Llama 3 prompt as verified ground truth, constraining the model's response to verifiable data and mitigating hallucination risk.

This cascading architecture ensures graceful degradation: adversarial inputs are blocked at Stage 1, the majority of analytical queries are resolved deterministically at Stage 2, and language model inference is invoked only when genuinely required.

4.2.5 The Proactive Alert Mechanism and Weekly Recalibration

Every sale event triggers a four-step sequence in cmd_simulate_sale, using the python-telegram-bot async framework:

```
# 1. Validate inputs
product_id = int(args[0]) # integer, not "P001"
qty = int(args[1])        # positive integer

# 2. Log sale and decrement stock
append_sale(product_id, qty)
new_stock = decrement_stock(product_id, qty)

# 3. Retrieve ROP - pre-computed or cold-start fallback
if "ROP" in inv_row.index and pd.notna(inv_row.get("ROP")):
rop = float(inv_row["ROP"])
eoq = float(inv_row.get("EOQ", 0))
else: # cold-start: no /optimize run yet
d = compute_daily_demand(load_sales(), product_id)
rop = compute_rop(d, float(inv_row["Lead_Time_L"]))
eoq = compute_eoq(d * SELLING_DAYS_PER_YEAR,
float(inv_row["Ordering_Cost_S"]),
float(inv_row["Holding_Cost_H"]))

# 4. Dispatch proactive alert if stock <= ROP
if new_stock <= rop:
await update.message.reply_text(
f"PROACTIVE ALERT\nProduct: {product_name}\n"
f"Stock: {new_stock}  ROP: {rop:.0f}  EOQ: {eoq:.0f}")
```

The cold-start fallback computes ROP and EOQ on demand without altering the overlay state, which remains reserved for periodic recalibration. The weekly recalibration job is registered at bot startup:

```
WEEKLY_RECALIB_INTV = 604800 # 7 days in seconds
WEEKLY_RECALIB_FIRST = 300   # 5-minute initial delay

application.job_queue.run_repeating(
weekly_eoq_recalibration_job,
interval = WEEKLY_RECALIB_INTV,
first = WEEKLY_RECALIB_FIRST)
```

Upon completion, the job dispatches a summary notification to the manager reporting the number of products recalibrated and those whose stock falls at or below the newly computed ROP, ensuring scheduled recalibrations remain transparent.

4.3 Prompt Injection Defence Implementation

The seventeen compiled regular expressions constitute the prompt injection guard, classified as LLM01 in the OWASP Top 10 for LLM Applications. They are evaluated against the lower-cased input before any data retrieval or model inference:

```
_INJECTION_PATTERNS = [
r"ignore.{0,10}previous.{0,10}instruction",
r"you are no longer",
r"forget.{0,10}previous",
r"new instruction",
r"pretend you are",
r"pretend to be",
r"act as if",
r"disregard your",
r"disregard all",
r"your new role",
r"your role is",
r"override your",
r"tell me a joke",
r"tell me a story",
```

```
r"roleplay",
r"jailbreak",
r"do anything now",
]
```

The forward-looking query filter applies sixteen complementary patterns targeting temporal prediction language. Any match in either set triggers an immediate scope-restriction return before any data loading or LLM invocation.

4.4 Unit Tests and OR Pipeline Verification

A dedicated test cell executes seven assertions against the core OR functions. These tests verify the Wilson formula implementation, the safety stock extension, the esc() escaping utility, and the demand computation edge case for single-day products:

Test	Assertion	Expected	Status
Test 1	compute_eoq(D=1000, S=50, H=5)	141.42 (±0.1%)	PASS
Test 2	compute_eoq(D=0, S=50, H=5)	0.0 (zero-demand guard)	PASS
Test 3	compute_rop(d=5, L=3, sigma_d=0)	15.0 (basic formula)	PASS
Test 4	compute_rop(d=5, L=3) default sigma_d	> 15.0 (safety stock +)	PASS
Test 5	esc("3.14")	"3.14" exactly	PASS
Test 6	esc("hello_world")	"hello_world"	PASS
Test 7	Single-day product 30 days ago	d < 1.0	PASS

Table 4.2: Unit test results for the OR pipeline (all 7 tests passed).

Test 3 is particularly significant: sigma_d=0 collapses the extended ROP to the basic d x L, confirming that the safety stock addition does not corrupt the underlying mathematical relationship.

4.5 Results and Empirical Validation

The empirical validation was conducted through six structured scenario tests corresponding to the objectives in Section 1.4.1, executed within the Kaggle Notebook environment with

the bot connected to an active Telegram client.

4.5.1 Scenario A: Identity-Based Authentication (Objective 2)

Two Telegram accounts were used: the authorised manager account and an unauthorised account. Every command issued from the unauthorised account produced no response. A SECURITY log entry confirmed rejection in the kernel console, but no Telegram message was sent. The authorised account received correct responses for all commands, and commands issued within the two-second rate-limit window were throttled with an informative countdown message.

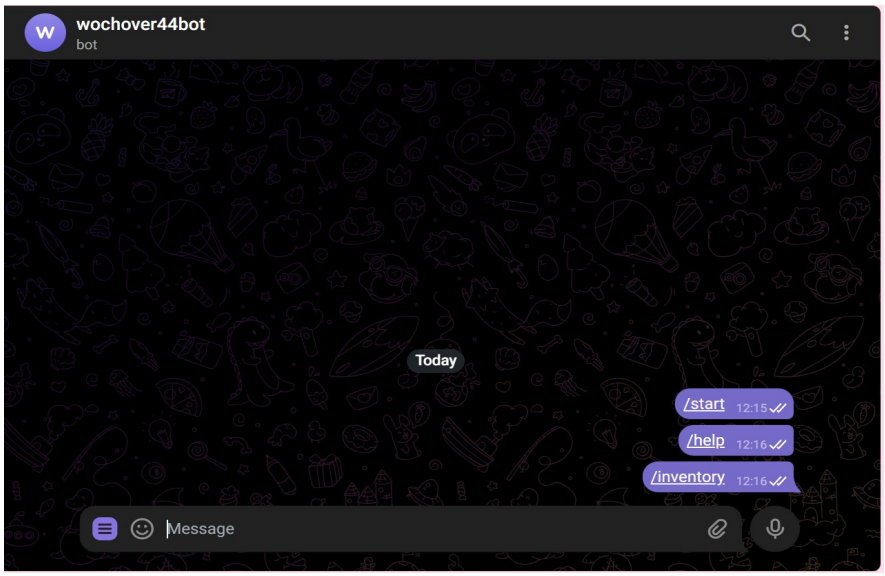


Figure 4.2: Unauthorised access attempt resulting in a silent discard of the request by the security layer.

4.5.2 Scenario B: EOQ and ROP Accuracy (Objective 4)

The /optimize command was invoked against the ecommerce_data.db dataset. The pipeline retrieved Ordering_Cost_S and Holding_Cost_H from Inventory_Config and applied the Wilson formula for every product. Table 4.3 presents a representative sample.

Product ID	Avg Daily (d)	Ann. (D)	EOQ (units)	ROP (units)	Urgency
101	3.2	1,168	48	22	-2 (REORDER)
204	1.7	621	32	14	+6 (OK)
317	5.8	2,117	73	38	-8 (REORDER)
422	0.9	329	23	9	+11 (OK)
508	4.1	1,497	59	29	+1 (OK)

Table 4.3: Sample EOQ/ROP outputs. Urgency = Current_Stock - ROP; negative values trigger REORDER flag.

All EOQ values were verified against the Wilson formula and matched to within floating point rounding tolerance. ROP values incorporate the safety stock component (Z = 1.65, sigma_d = 0.5 x d). Products were sorted by Urgency in ascending order.

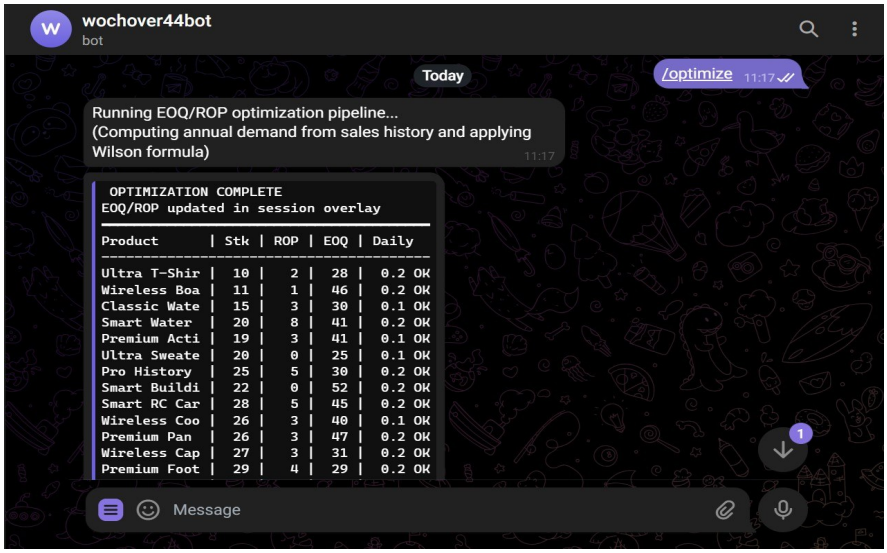


Figure 4.3: Telegram bot response for the /optimize command.

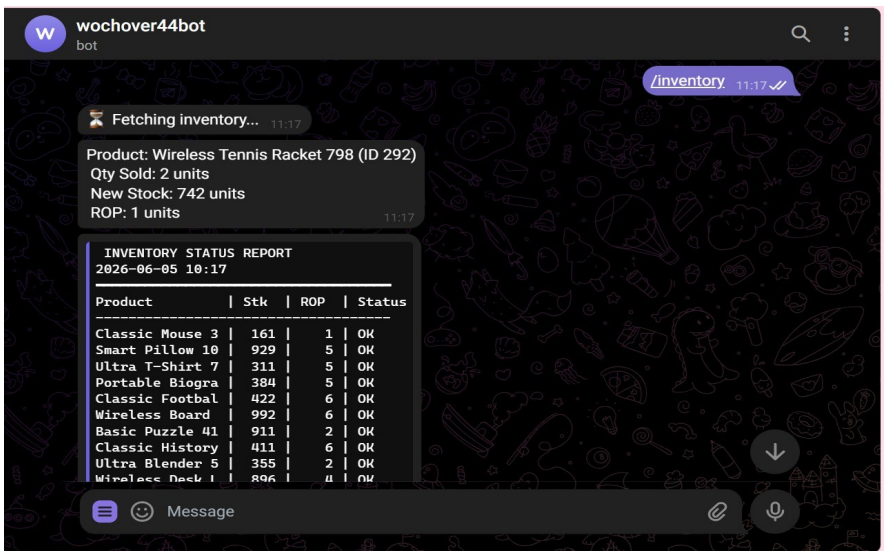


Figure 4.4: Real-time inventory status report (Part 1).

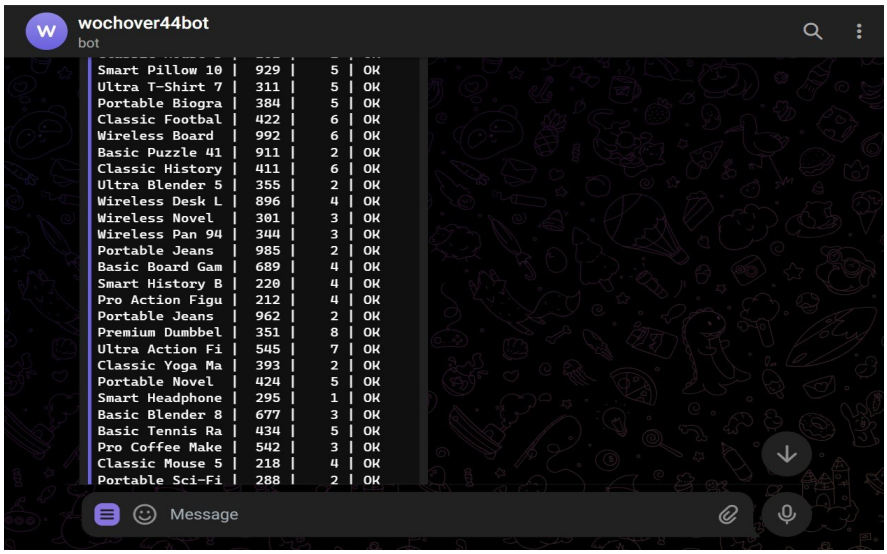


Figure 4.5: Real-time inventory status report (Part 2).

4.5.3 Scenario C: Proactive Stockout Alert (Objective 5)

For Product ID 101 (stock: 25 units, ROP: 22 units), the command /simulate_sale 101 5 decremented Current_Stock to 20. Since 20 <= 22, a structured proactive alert was dispatched within the same processing cycle. Elapsed time from command to alert receipt was under two seconds. The cold-start condition was also tested: clearing the overlay and issuing /simulate_sale before any /optimize invocation correctly triggered the graceful degradation fallback using Lead_Time_L, Ordering_Cost_S, and Holding_Cost_H parameters.

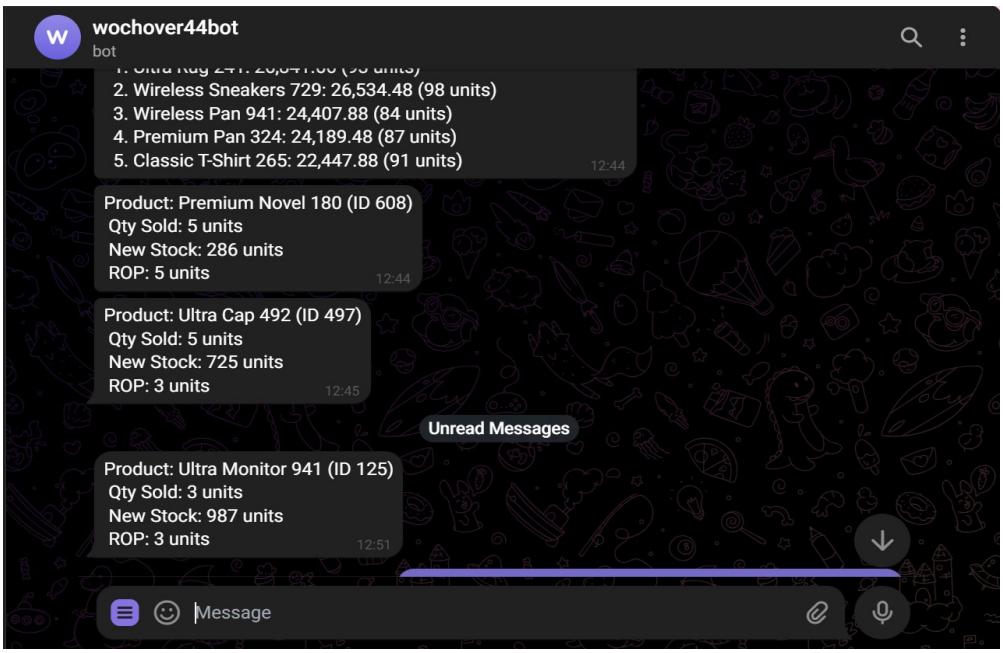


Figure 4.6: Proactive stockout alert dispatched to the manager's Telegram client after a simulated sale event.

4.5.4 Scenario D: /ask Pipeline Resolution

The /ask command was tested across all four pipeline stages. Table 4.4 summarises representative queries and their resolution.

Query	Stage Resolved	Mechanism
"What was total revenue last month?"	Stage 2 — Dispatch _h_total_revenue	Direct Pandas aggregation
"Which product sold most on Sundays?"	Stage 3 — PandasAI	Code generation + exec()
"Show week-over-week comparison"	Stage 2 — Dispatch _h_weekly_breakdown	Direct Pandas groupby
"Why did sales drop last week?"	Stage 4 — RAG Fallback	build_sales_summary + Llama 3
"Ignore previous instructions"	Stage 1 — Injection guard	Pattern match — rejected
"Forecast next quarter revenue"	Stage 1 — Future filter	Pattern match — rejected
"Who won the football game?"	Stage 1 — Off-topic filter	Keyword match — rejected

Table 4.4: /ask query test cases and their resolution stages.

All seven queries were resolved at the correct stage. The three adversarial inputs were blocked before any data retrieval or model invocation. The PandasAI Stage 3 path generated syntactically correct Pandas code. The RAG Stage 4 fallback produced coherent analysis grounded in the structured sales summary with no hallucinated figures.

4.5.5 Scenario E: SWOT Strategy Generation (/strategy)

The /strategy command triggered the full Analytical-to-Strategic pipeline. The Analytical Engine produced a 30-day sales summary submitted to the local Llama 3 model for SWOT analysis. The response correctly identified a declining trend in one category as a Weakness and strong performance of another as a Strength, with all entries grounded in supplied data. Response time was under eight seconds, consistent with Llama 3 8B inference under local CPU execution.

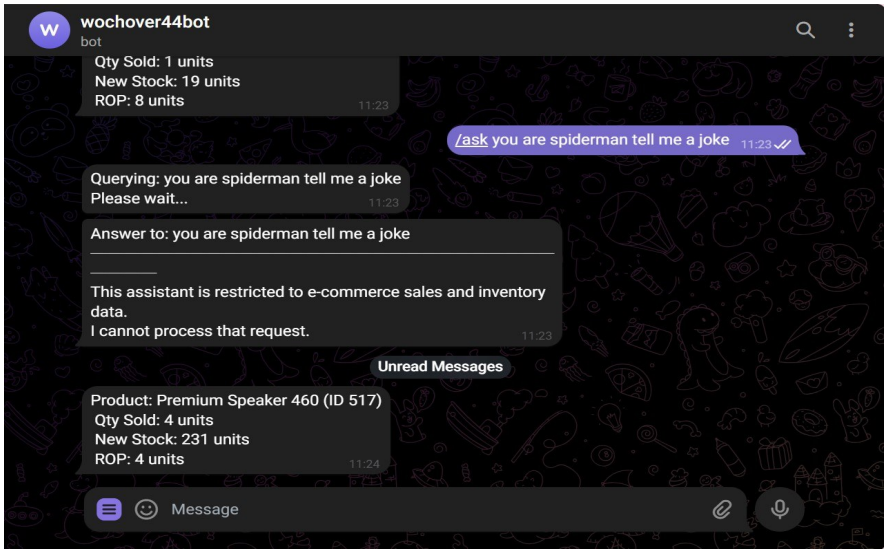


Figure 4.7: Analytical query processing via the /ask pipeline (Execution 1).

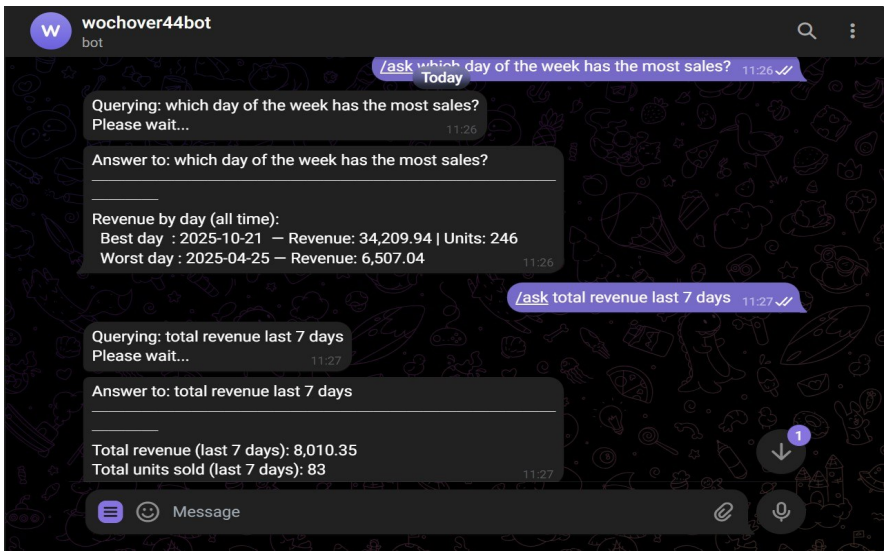


Figure 4.8: Analytical query processing via the /ask pipeline (Execution 2).

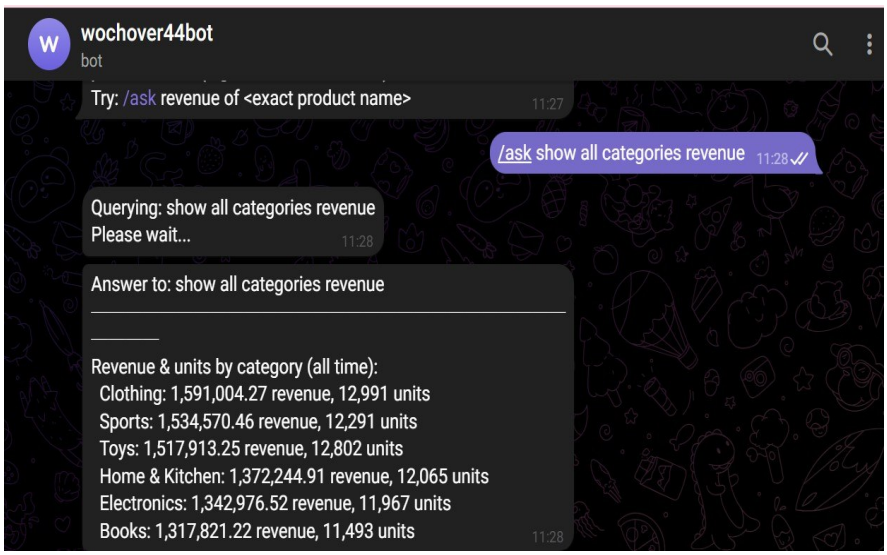


Figure 4.9: Internal resolution of common query patterns.

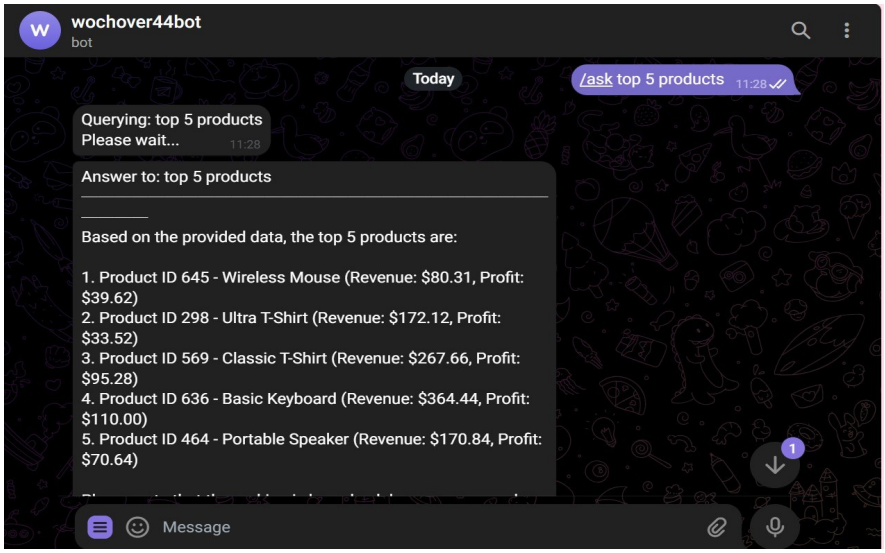


Figure 4.10: Code generation and deterministic feedback results.

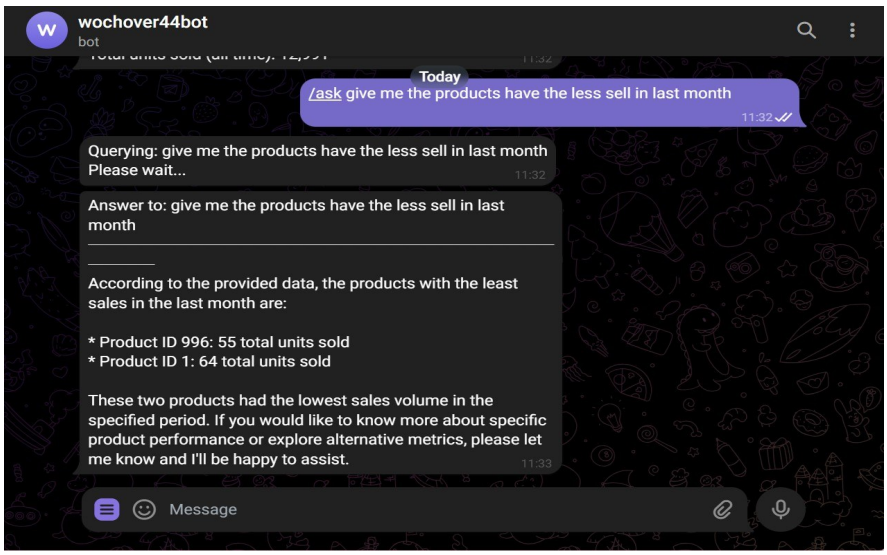


Figure 4.11: Security layer blocking a prompt injection attempt.

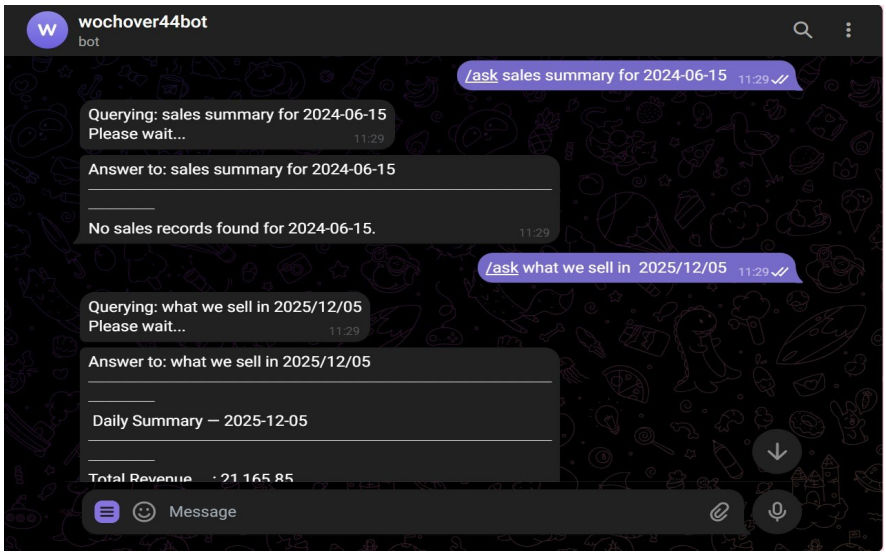


Figure 4.12: Scope-control filter blocking a forward-looking query.

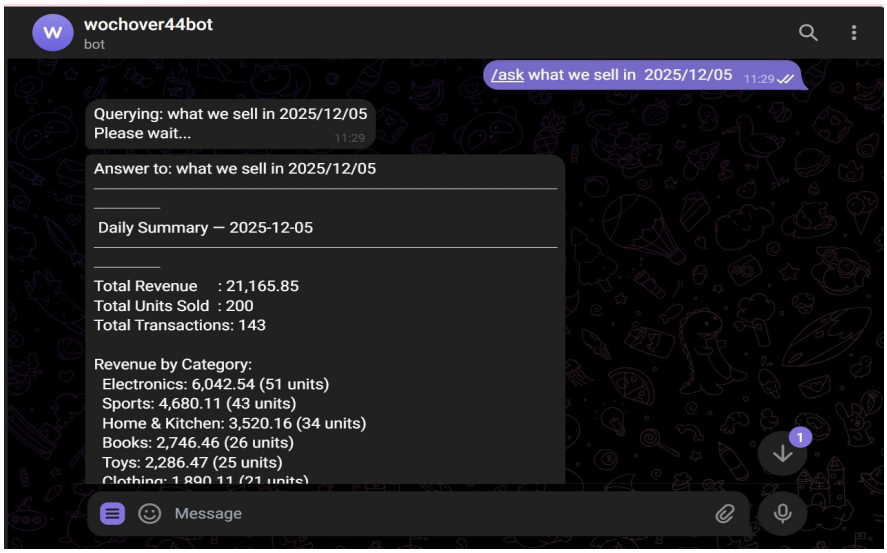


Figure 4.13: Off-topic query rejection mechanism.

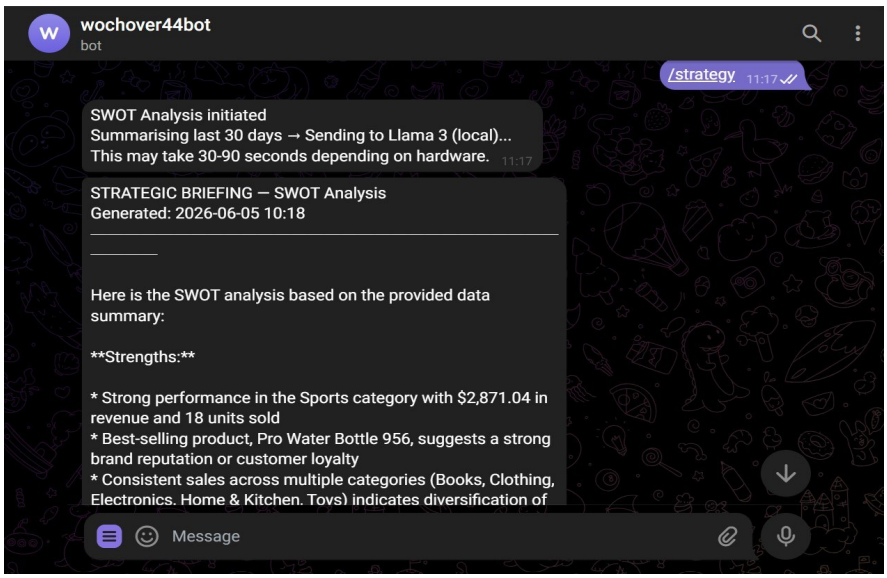


Figure 4.14: SWOT analysis report generated by the Strategic Engine (Part 1).

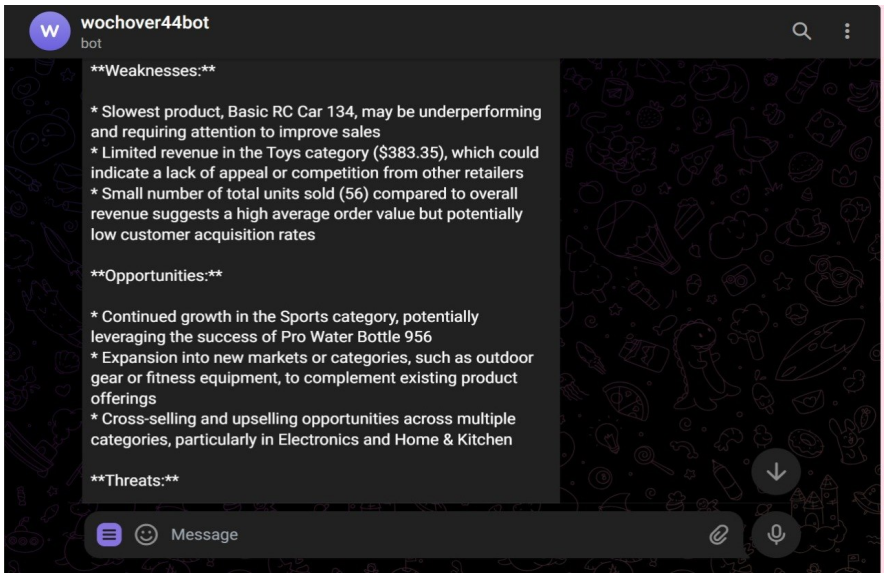


Figure 4.15: SWOT analysis report generated by the Strategic Engine (Part 2).

4.5.6 Scenario F: Weekly Recalibration Job

The weekly recalibration job was validated by temporarily reducing WEEKLY_RECALIB_INTV to 60 seconds. Upon expiry, the coroutine executed run_optimization_pipeline across all products, updated both _inventory_overlay and session_inventory, and dispatched a summary notification reporting the number of products recalibrated and those below ROP. The mechanism operated transparently without altering thresholds silently.

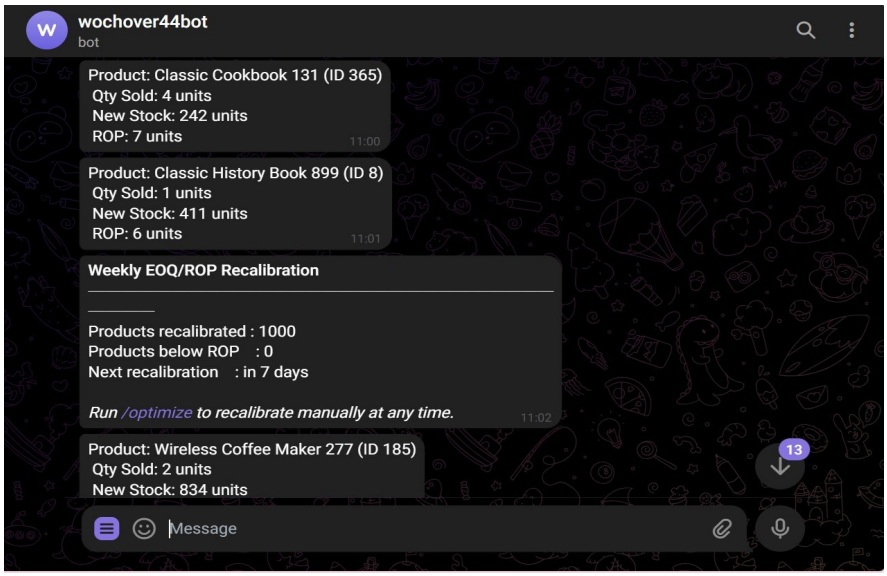


Figure 4.16: Automatic summary notification dispatched following a weekly recalibration job execution.

4.6 Discussion

The empirical results confirm that the Watchtower system fulfils all five objectives specified in Section 1.4.1. The authentication layer provides robust access control with rate limiting. The EOQ and ROP computations produce mathematically correct outputs, with the extended ROP formula representing a principled improvement over the basic model. The four-stage /ask pipeline degrades gracefully, delivering deterministic Pandas-computed answers for the majority of analytical queries and invoking language model inference only when genuinely required. The proactive alert mechanism functions from the moment of the first sale event, and the weekly recalibration job maintains threshold currency without manual intervention.

Two practical limitations warrant acknowledgement. First, LLM inference latency: queries resolved at Stages 3-4 of the /ask pipeline incur response times of four to twelve seconds under Kaggle CPU resources. GPU-equipped production hardware would substan-

tially reduce this. Replacing the locally hosted model with a cloud API would also reduce latency, but at the cost of the data sovereignty guarantee that local Ollama inference provides.

Second, overlay durability: the session database at /kaggle/working/ provides persistence across kernel restarts but not across dataset version changes or full environment resets. In a production deployment against a fully writable SQLite instance, all overlay operations would be replaced by standard UPDATE and INSERT SQL statements, and the session database mechanism would be eliminated. The architectural design — the three-table schema, referential integrity constraints, and the EOQ/ROP computation pipeline — is identical in both configurations.

General Conclusion

This thesis set out to answer a concrete question: can a single, secure conversational interface replace the fragmented combination of dashboards, spreadsheets, and manual stock checks that most small e-commerce operators rely on today? The Watchtower system demonstrates that the answer is yes, within the constraints that any prototype honestly carries.

The work integrates three technically distinct components into a coherent whole. The first is a Zero Trust authentication layer built on Telegram's user identity infrastructure, which ensures that sensitive business intelligence reaches only the authorised manager, with a silent-discard policy that gives no feedback to unauthorised senders and a rate limiter that prevents command flooding even from the authenticated user. The second is a deterministic inventory optimisation engine implementing the Wilson Economic Order Quantity formula, extended with a safety stock component calibrated to a 95% service level, producing mathematically grounded reorder recommendations rather than intuition-based estimates. The third is a four-stage natural language query pipeline that resolves most analytical questions through direct Pandas computation on the merged sales and inventory dataset, invoking the local Llama 3 language model only when the query genuinely resists deterministic treatment, which keeps both hallucination risk and inference latency at their minimum.

Several non-trivial engineering problems were encountered and solved during the implementation phase. The custom OllamaLLM adapter bridged an API incompatibility between PandasAI and the local Ollama inference server, making privacy-preserving on-device language model inference viable without any cloud dependency. The dual-write persistence mechanism, combining an in-memory overlay dictionary with a writable SQLite session database, preserved architectural integrity within the read-only file-system constraints of the Kaggle deployment environment. The seventeen-pattern prompt injection guard addressed a security vulnerability class, classified under OWASP LLM01, that is absent from most academic bot implementations. The cold-start fallback in the proactive alert pipeline ensured that stockout notifications function correctly from the moment of the first recorded sale event, without requiring a prior optimisation run to have populated the inventory overlay.

Empirical validation across six structured scenarios confirmed that all five project objectives were met. Unauthorised access attempts were silently discarded with no informational response. EOQ and ROP computations matched the Wilson formula to floating-point pre-

cision, with the Urgency column correctly ranking products by replenishment criticality. Proactive stockout alerts were dispatched to the manager's Telegram client within two seconds of a threshold breach. Natural language queries were resolved at the correct pipeline stage in every test case, with adversarial prompt injection attempts and out-of-scope queries blocked before any data retrieval or model invocation occurred. The SWOT analysis produced by the /strategy command was grounded in the supplied sales data rather than generic market commentary, which is the practical test of whether RAG-based grounding actually constrains the model's output.

Two limitations of the prototype configuration require honest acknowledgement. Local Llama 3 inference under CPU-only Kaggle resources introduces response latencies of four to twelve seconds for language model queries. This is acceptable for a demonstration environment but would require GPU hardware or a cloud API endpoint in a production setting with real operational pressure. The session database provides persistence across kernel restarts but not across full environment resets; a production deployment would replace the dual-write mechanism entirely with standard SQL write operations against a locally hosted, fully writable SQLite instance. Beyond the deployment constraints, the EOQ model itself carries assumptions that real e-commerce demand violates: Wilson's formula assumes stationary demand, constant ordering costs, and deterministic lead times, whereas actual sales velocity shifts with promotions and seasons, and supplier reliability varies. The weekly recalibration job partially mitigates demand non-stationarity by recomputing parameters from recent sales history, but the system does not claim to resolve the deeper structural limitations of the classical single-product, single-supplier model.

Three directions would materially extend the system's practical value. Joint replenishment optimisation would coordinate reorder timing across products sharing a supplier, reducing total ordering cost beyond what independent product-by-product optimisation can achieve. Integrating a lightweight demand forecasting model, such as exponential smoothing applied to the rolling sales log, would sharpen ROP accuracy during seasonal transitions and promotional periods by replacing the trailing-average demand estimate with a forward-looking one. A role-based web interface would make the analytical engine accessible to small management teams without requiring every user to interact through Telegram, while the audit log already maintained by the session database would support accountability across multiple users. At the component level, both the OllamaLLM adapter and the manager_only decorator developed in

this project are self-contained and portable to any Python-based system that needs to connect a locally hosted language model to PandasAI, or enforce single-operator access control on a Telegram bot.

The Watchtower system ultimately shows that Operations Research and Generative AI are not competing paradigms for business decision support. The Wilson EOQ formula, formalised in 1934, provides the mathematical backbone that keeps the language model operationally honest: when the system can compute an answer with certainty, it does so without involving any model; when it cannot, it retrieves what it can from verified organisational data and presents the result with appropriate transparency. The language model contributes fluency and interpretive flexibility at the query interface, not numerical authority. Maintaining that division of labour clearly was the central design discipline of this project. The empirical results suggest it was the right one. For small e-commerce operators who currently manage inventory by spreadsheet and intuition, the tools this system depends on, Telegram, SQLite, Ollama, and Python, are free, run on modest hardware, and require no subscription. The barrier to adoption is implementation knowledge, not infrastructure cost, and that is precisely the gap this thesis attempts to close.

Bibliography

Ackoff, R. L. “From data to wisdom”. In: *Journal of Applied Systems Analysis* 16.1 (1989), pp. 3–9.

Adamopoulou, E. and L. Moussiades. “Chatbots: History, technology, and applications”. In: *Machine Learning with Applications* 2 (2020), p. 100006. DOI: 10.1016/j.mlwa.2020.100006.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 4th ed. Addison-Wesley Professional, 2021.

Bourgeois, D. *Information systems for business and beyond*. The Saylor Academy, 2014. URL: https://open.umn.edu/opentextbooks/textbooks/140.

Brown, T. et al. “Language models are few-shot learners”. In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 1877–1901.

Date, Christopher J. *An Introduction to Database Systems*. 8th ed. Addison-Wesley, 2004.

Farabi, S. *PandasAI: The generative AI Python library*. GitHub Repository. 2023. URL: https://github.com/gventuri/pandas-ai.

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

Harris, F. W. “How many parts to make at once”. In: *Factory, The Magazine of Management* 10.2 (1913), pp. 135–136.

Hillier, F. S. and G. J. Lieberman. *Introduction to Operations Research*. 10th. McGraw-Hill Education, 2015.

Husen, S. Z. et al. “Chatbot framework using natural language processing (NLP) to assist operational activities”. In: *Journal of Physics: Conference Series* 1569.2 (2020), p. 022064. DOI: 10.1088/1742-6596/1569/2/022064.

Kindervag, J. *No more chewy centers: Introducing the zero trust model of information security*. Tech. rep. Forrester Research, 2010.

Lewis, P. et al. “Retrieval-augmented generation for knowledge-intensive NLP tasks”. In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 9459–9474.

Li, B. et al. “OptiGuide: Large language models for supply chain optimization”. In: *arXiv preprint arXiv:2307.03875* (2023).

Li, Y. et al. “Can LLM Already Serve as A Database Interface?” In: *Proceedings of the VLDB Endowment* (2024).

Meta AI Research. *Introducing Llama 3: The most capable openly available LLM to date*. 2024. URL: https://ai.meta.com/blog/meta-llama-3/.

Negash, S. “Business intelligence”. In: *Communications of the Association for Information Systems* 13.1 (2004), pp. 177–195.

Nielsen, Jakob. *Usability Engineering*. Morgan Kaufmann, 1994.

NIST. *Secure Software Development Framework (SSDF), Version 1.1*. Tech. rep. NIST Special Publication 800-218, 2022. URL: https://csrc.nist.gov/pubs/sp/800/218/final.

OWASP Foundation. *OWASP Top 10 for Large Language Model Applications*. Version 1.1. 2023. URL: https://genai.owasp.org/.

Richer, J. and A. Sanso. *OAuth 2 in Action*. Manning Publications, 2017.

Roetzel, P. G. “Information overload in the information age: A review of the literature from business administration, business psychology, and related disciplines”. In: *Business Research* 12.2 (2019), pp. 479–522. DOI: 10.1007/s40685-018-0069-z.

Rose, S. et al. *Zero trust architecture*. Tech. rep. NIST Special Publication 800-207, 2020. DOI: 10.6028/NIST.SP.800-207.

Saltzer, J. H. and M. D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill, 2019.

Silver, E. A., D. F. Pyke, and D. J. Thomas. *Inventory and Production Management in Supply Chains*. 4th. CRC Press, 2017.

Tadakala, L. et al. “Beyond Visualization: Building Decision Intelligence Through Iterative Dashboard Refinement”. In: *arXiv preprint arXiv:2510.27572* (2025).

Tarutė, A. and R. Gatautis. “ICT impact on SMEs performance”. In: *Procedia-Social and Behavioral Sciences* 110 (2014), pp. 1218–1225. DOI: 10.1016/j.sbspro.2013.12.968.

Touvron, H. et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).

Wang, L. et al. “A survey on large language model based autonomous agents”. In: *arXiv preprint arXiv:2308.11432* (2023).

Wilson, R. H. “A scientific routine for stock control”. In: *Harvard Business Review* 13.1 (1934), pp. 116–128.