Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Комп'ютерного моделювання та інтелектуальних технологій
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів інтеграції моделей машинного навчання у розподілені вебсистеми електронної комерції для прогнозування попиту на товари
(тема)

Виконав:
здобувач II року навчання,
групи СПРм-24-1
Олексій МОРОЧКОВСЬКИЙ

(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Системне проектування
(повна назва освітньої програми)

Керівник доц. каф. КМІТ Зульфія ІМАНГУЛОВА

(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри КМІТ _____ проф. Ігор ГРЕБЕННІК
(підпис) (власне ім'я, прізвище)

2026 р.

Я як здобувач вищої освіти ХНУРЕ розумію і підтримую політику закладу із академічної доброчесності. Я не надавав і не одержував недозволену допомогу під час підготовки кваліфікаційної роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

15.05.2026  Олексій МОРОЧКОВСЬКИЙ

Кваліфікаційна робота не містить відомостей заборонених до відкритого опублікування.

Кваліфікаційна робота виконана у відповідності до стандартів, що діють в Україні.

Попередній захист проведено 15.05.2026 р.

Керівник кваліфікаційної роботи  Зульфія ІМАНГУЛОВА

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Комп’ютерного моделювання та інтелектуальних технологій

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне проектування
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«18» травня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Морочковському Олексію Максимовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження методів інтеграції моделей машинного навчання у розподілені вебсистеми електронної комерції для прогнозування попиту на товари

затверджена наказом університету від 06 квітня 2026 р. № 268СТ

2. Термін подання здобувачем роботи до екзаменаційної комісії 18 травня 2026 р.

3. Вихідні дані до роботи *Перелік використовуваних програмних засобів: Visual Studio, Visual Studio Code, Draw.IO, SQL Server Management Studio, Redis, ML.NET, RabbitMQ, Docker, Kubernetes, Azure API Management, AllFusion. Оформлення пояснювальної записки та захист кваліфікаційної роботи виконується англійською мовою.*

4. Перелік питань, що потрібно опрацювати в роботі 1. Аналіз предметної області. 2. Дослідження та вибір методів прогнозування та архітектурних шаблонів. 3. Проєктування інтегрованого архітектурного каркаса. 4. Оцінка ефективності та аналіз стратегічного впливу.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Architecture of the distributed e-commerce platform. Interaction of the forecasting system with business infrastructure units. Comparative simulation of model responses to a demand shock. Comparative analysis of Synchronous (REST/gRPC) vs. Asynchronous (Message Broker) integration patterns. Component Diagram of the Forecasting Service. Sequence Diagram . Entity-Relationship Diagram of the

Forecasting System. State Machine Diagram for the Circuit Breaker Pattern. Stress-Testing and scalability graph.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Розділи спеціальної частини	доц. Імангулова З.А.		15.05.2026

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / термін виконання етапів роботи	Примітка
1	Аналіз предметної області	26.01.2026 – 10.02.2026	Виконано
2	Дослідження існуючих підходів	11.02.2026 – 20.02.2026	Виконано
3	Аналіз аналогів і засобів реалізації	21.02.2026 – 28.02.2026	Виконано
4	Дослідження патернів інтеграції ML-моделей у мікросервісну архітектуру.	1.03.2026 – 10.03.2026	Виконано
5	Розробка методів автоматизованої генерації ознак	11.03.2026 – 21.03.2026	Виконано
6	Проєктування подієво-орієнтованої архітектури (EDA) даних	21.03.2026 – 30.04.2026	Виконано
7	Декомпозиція системи на автономні мікросервіси	30.04.2026 – 1.05.2026	Виконано
8	Розробка механізмів стійкості та обробки відмов	1.05.2026 – 2.05.2026	Виконано
9	Проєктування життєвого циклу моделі	2.05.2026 – 3.05.2026	Виконано
10	Опис технологічної реалізації	3.05.2026 – 5.05.2026	Виконано
11	Оформлення пояснювальної записки	5.05.2026 – 7.05.2026	Виконано
12	Оформлення додатків	7.05.2026 – 10.05.2026	Виконано
13	Представлення роботи на рецензування	11.05.2026	Виконано

Дата видачі завдання 06 квітня 2026 р.

Здобувач _____ Олексій МОРОЧКОВСЬКИЙ
(підпис) (власне ім'я, прізвище)

Керівник роботи _____ доц. Зульфія ІМАНГУЛОВА
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 99 сторінок, 9 рисунків, 6 таблиць, 2 додатки, 26 джерел.

МАШИННЕ НАВЧАННЯ, ЕЛЕКТРОННА КОМЕРЦІЯ, ПРОГНОЗУВАННЯ ПОТРЕБ, РОЗПОДІЛЕНІ СИСТЕМИ, .NET CORE, АНАЛІТИКА ПЕРЕДБАЧЕННЯ, MICROSERVICES.

Об’єктом дослідження є процес управління запасами та планування продажів у сучасних платформах електронної комерції, з акцентом на оптимізацію ланцюгів постачання на основі аналізу даних.

Предметом дослідження є інформаційні технології, архітектурні шаблони та методи інтеграції моделей машинного навчання, що використовуються при створенні масштабованих розподілених вебзастосунків електронної комерції.

Метою дослідження є проєктування та розробка інтегрованої платформи для розподіленої системи електронної комерції, яка використовує моделі машинного навчання для забезпечення точного прогнозування попиту на продукцію, тим самим підвищуючи ефективність бізнесу.

Методи дослідження включають: системний підхід до архітектурного проєктування, структурний аналіз розподілених систем, методи прогнозування часових рядів.

Сфера застосування – сектор цифрової роздрібної торгівлі, зокрема середні та великі підприємства електронної комерції, які потребують масштабованих рішень для мінімізації надлишків та дефіциту товарів.

ABSTRACT

Explanatory note of the qualification work: 99 pages, 9 figures, 6 tables, 2 appendices, 26 references.

MACHINE LEARNING, E-COMMERCE, DEMAND FORECASTING, DISTRIBUTED SYSTEMS, .NET CORE, PREDICTIVE ANALYTICS, MICROSERVICES.

The Object of Research is the process of inventory management and sales planning in modern e-commerce platforms, with an emphasis on supply chain optimization based on data analysis.

The Subject of Research includes information technologies, architectural patterns, and methods for integrating machine learning models used in the development of scalable distributed e-commerce web applications.

The aim of the research is to design and develop an integrated framework for a distributed e-commerce system that utilizes machine learning models to provide accurate product demand forecasting, thereby improving business efficiency.

The research methods include a systemic approach to architectural design, structural analysis of distributed systems, time-series forecasting techniques, and relational database modeling to ensure high performance and data consistency.

The analysis of the e-commerce field and existing forecasting solutions has been conducted in the work. It identifies the challenges of implementing ML in high-load distributed environments, such as data latency and model synchronization, and proposes a robust architectural approach to overcome these limitations.

The scope of application is the digital retail sector, specifically targeting medium-to-large e-commerce enterprises that require scalable solutions to minimize overstocking and stockouts.

CONTENT

LIST OF ABBREVIATIONS, SYMBOLS, UNITS, AND TERMS

API – Application Programming Interface;

ARIMA – Autoregressive Integrated Moving Average;

CT – Continuous Training;

EDA – Event-Driven Architecture;

GBDT – Gradient Boosting Decision Trees;

gRPC – Google Remote Procedure Call;

LSTM – Long Short-Term Memory;

MAPE – Mean Absolute Percentage Error;

ML – Machine Learning;

MLOps – Machine Learning Operations;

MSE – Mean Squared Error;

REST – Representational State Transfer;

RMSE – Root Mean Square Error;

SME – Small and Medium Enterprises;

WAPE – Weighted Average Percentage Error;

ETL – Extract, Transform, Load processes.

INTRODUCTION

In the modern global economy, the e-commerce sector is undergoing a massive digital transformation. With the exponential growth of online transactions and the increasing complexity of supply chains, businesses can no longer rely on traditional, manual methods of inventory management. Product demand forecasting has become a critical component for maintaining competitiveness. Inaccurate forecasts lead to significant financial losses, either through overstocking, which ties up capital and increases storage costs, or through stockouts, which result in lost sales and decreased customer loyalty.

The rise of distributed web systems and microservices architecture has provided the necessary infrastructure to handle vast amounts of data in real-time. However, integrating sophisticated Machine Learning (ML) models into these high-load, distributed environments remains a significant technical challenge. It requires a deep understanding of both data science workflows and robust software engineering practices, particularly in ensuring data consistency, system scalability, and low-latency predictions.

The object of the study is the operational process of product demand management within e-commerce platforms.

The subject of the research focuses on architectural patterns and technical methods for embedding predictive ML models into distributed web infrastructures, specifically using modern frameworks like .NET Core.

The purpose of this work is to research and develop a scalable framework for integrating machine learning into a distributed e-commerce system. This framework aims to provide accurate, automated demand forecasts that allow businesses to optimize their stock levels and improve overall operational efficiency.

To achieve this goal, the following tasks were set:

– analyze the current state of e-commerce systems and existing forecasting methods;

– determine the specific requirements for a distributed system capable of hosting ML models;

– design a system architecture that ensures seamless integration between the web application and the analytical engine;

– develop a prototype (or specific components) of the system to demonstrate the feasibility of the proposed integration methods.

The results of this research can be implemented by e-commerce companies to automate their procurement processes and reduce human error in planning. Furthermore, the proposed architectural approach provides a blueprint for developers working within the .NET ecosystem on how to build data-intensive, distributed applications with built-in intelligence.

1 ANALISIS OF THE SUBJECT FIELD

1.1 Analysis of the implemented system

A modern e-commerce platform's architectural design has a significant impact on its efficiency. As the industry standard for high-load retail platforms, the implemented system is examined in this work as a distributed web application [1]. These systems are built to conduct transactions across several locations, manage intricate inventory catalogues, and support thousands of concurrent users.

The system being examined uses a decoupled design, usually utilizing contemporary frontend frameworks in conjunction with ASP.NET Core for the backend [2]. This distributed approach, in which a responsive web-based client interacts with the server-side via GraphQL or RESTful APIs, enables independent service scaling. This design requires an API gateway, which serves as a single point of entry and directs client queries to specific microservices like the Order, User, and Catalogue services. Independent service scalability is made possible by this distributed strategy, in which a responsive web-based client communicates with the server-side through GraphQL or RESTful APIs. An API gateway, which acts as a single-entry point and routes client requests to particular microservices like Catalogue, Order, and User services, is essential to this architecture. These dispersed parts make up the business logic layer, which oversees essential features like inventory updates and payment processing. A hybrid data storage approach that combines relational databases, like Microsoft SQL Server for ACID-compliant transactions, with NoSQL programs or caching layers, like Redis for fast data retrieval, supports these services.

Figure 1.1 below shows the suggested e-commerce system design with an integrated forecasting module. The solution uses an Azure API Management gateway to separate the Business Logic Layer from the Presentation Layer in accordance with a microservices architecture [3]. The Forecasting Engine operates as a specialized service

that consumes historical data from Transaction Logs via an asynchronous ETL process, ensuring that predictive analysis does not degrade the performance of real-time transactional services.

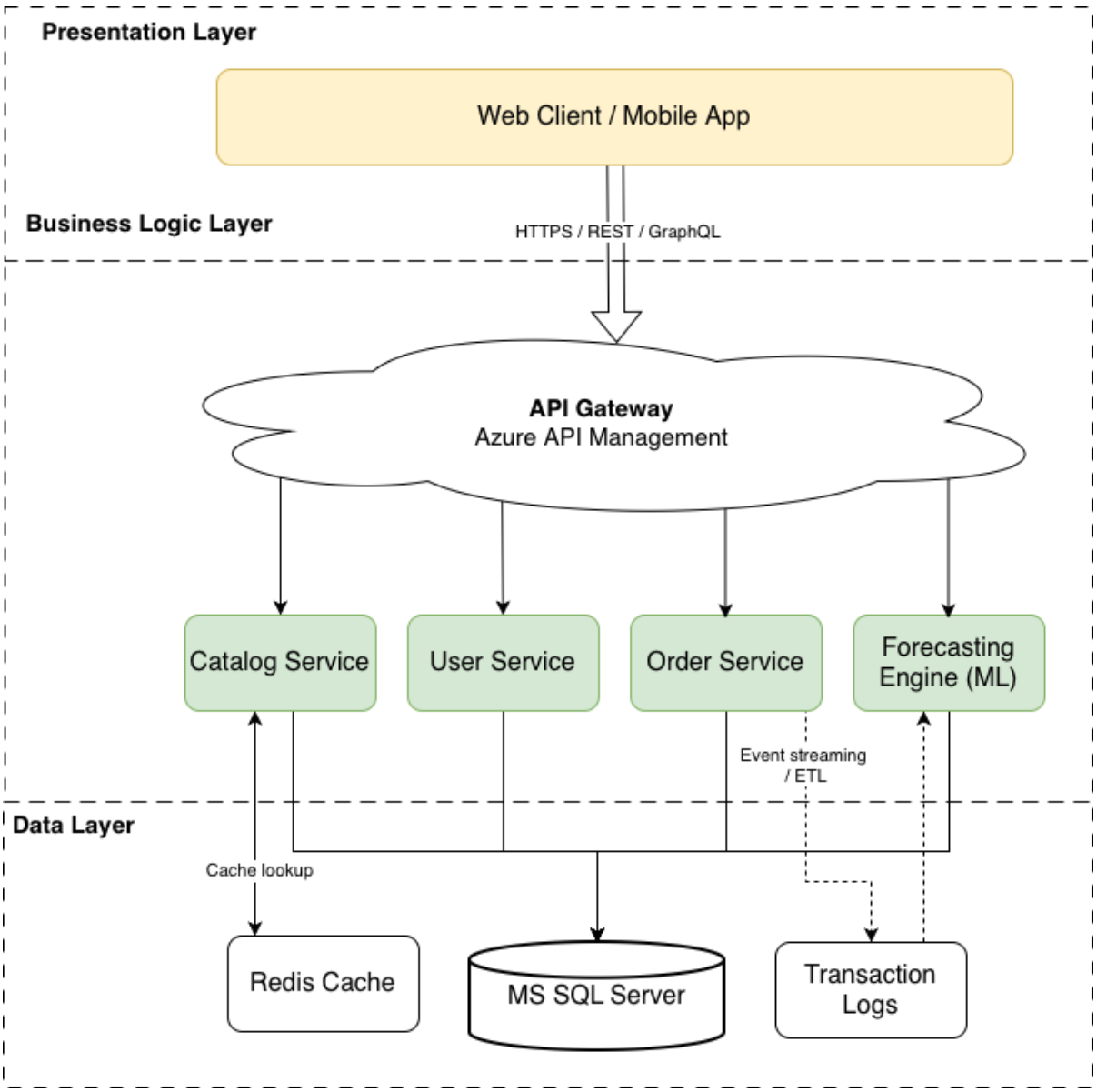

Figure 1.1 – Architecture of the distributed e-commerce platform

As shown in the scheme above, the system is organized into three distinct logical layers to ensure scalability and maintainability:

Presentation Layer: Consists of a responsive Web Client and Mobile App that communicate with the backend via secure HTTPS protocols using RESTful APIs or GraphQL.

Business Logic Layer: Managed by an Azure API Management gateway, which acts as a centralized entry point. It orchestrates requests to various microservices, including Catalog, User, and Order services. This layer also hosts the Forecasting Engine (ML), which is integrated directly into the system's fabric to provide real-time predictive insights to the end-user, while also persisting long-term trends back into the MS SQL Server for historical analysis.

The SQL Server acts as a centralized repository for ACID-compliant transactions from the Order, User, and Catalog services, ensuring data integrity across the distributed system. Redis Cache is employed for high-speed cache lookups to reduce database load. A dedicated Transaction Logs store captures historical data through an asynchronous ETL (Extract, Transform, Load) process, providing the necessary input for training and executing machine learning models without impacting the performance of the core business services [4]. To prevent performance degradation of the production database, an asynchronous Event Streaming process is implemented. It offloads transactional data to a dedicated Transaction Logs store for further processing by the Machine Learning module.

To identify the most effective way to transition from a reactive to a proactive system, it is essential to analyze existing approaches to demand forecasting used in the industry. These can be categorized into traditional statistical methods, enterprise-level platforms, and modern machine learning frameworks. The Table 1.1 below shows the analysis of the existing solutions.

Traditional Statistical Solutions: Classical models like ARIMA (Auto-Regressive Integrated Moving Average) rely on the linear correlation between past values (autoregression) and past forecast errors (moving average) to predict future trends. While they are computationally efficient, they often fail to capture non-linear patterns and the

influence of external variables (e.g., social media trends or sudden weather changes) that a distributed e-commerce environment generates.

Table 1.1 – Comparative analysis of demand forecasting methods

Criteria	Traditional (ARIMA)	Enterprise (SAP/Oracle)	ML Frameworks (Prophet/XGBoost)	Proposed ML Integration
Data Volume	Low to Medium	Very High	High	Very High
Handling Non-linearity	Poor	Average	Excellent	Excellent
System Integration	Easy	Difficult (Monolithic)	Moderate	Seamless (API-driven)
Cost of Ownership	Low	Very High	Medium	Low (Customized)
Real-time Support	No	Limited	Yes	Yes

Enterprise Resource Planning (ERP) Modules: Systems like SAP IBP (Integrated Business Planning) or Oracle Demand Management offer robust, integrated forecasting tools. They excel in data aggregation but are often "black boxes" that are difficult to customize for specific microservice architectures and require expensive licensing, making them less accessible for agile development.

Open-source ML Frameworks: Tools such as Facebook Prophet or Amazon Forecast (via API) provide advanced capabilities for handling seasonality and missing data. Specifically, Prophet is designed for business forecasting at scale, though it may require significant tuning to perform optimally with the high-frequency transactional data typical of the analyzed system.

As shown in Table 1.1, while existing solutions offer various strengths, they often create a trade-off between integration flexibility and predictive power.

To support effective demand forecasting, the system is designed to collect and store comprehensive historical data. The current data flow infrastructure records every completed order within transaction logs, capturing essential details such as timestamps, product IDs, quantities, and customer demographics. This transactional data is complemented by real-time monitoring of inventory states across various warehouses. Furthermore, the system integrates external factors, including seasonal trends and marketing schedules, to provide a broader context for the stored data, which is crucial for identifying patterns in consumer behavior.

While the existing distributed system excels at high-volume transaction processing, it remains fundamentally reactive due to a lack of proactive analytical capabilities. Current demand planning relies heavily on manual expert estimates or simple moving averages, exposing several critical architectural bottlenecks. Specifically, the presence of data silos across distributed services complicates the aggregation of data for machine learning training, as gathering information from disparate sources often leads to performance degradation.

Moreover, the absence of real-time insights means the system typically reacts to stockouts after they occur rather than predicting them in advance. This reactive nature is further exacerbated by the limited scalability of traditional analytics; monolithic analytical tools often fail to keep up with the high throughput of a modern distributed web environment.

Consequently, there is a clear necessity for integrating more sophisticated, ML-driven predictive methods directly into the distributed fabric of the e-commerce system.

1.2 Determination of the scope of application of the development system

Using accurate demand forecasting to optimize warehouse operations is one of the main uses. Businesses can successfully reduce the risks associated with overstocking by precisely forecasting future needs. This reduces capital tied up in slow-moving inventory and minimizes storage expenses, while also eliminating stockouts to maintain high customer satisfaction [5].

The versatility of the ML-driven forecasting engine allows for its effective application across diverse market sectors, each characterized by unique demand patterns:

– Fast-Moving Consumer Goods (FMCG): in this sector, products have a short shelf life and high turnover rates. The system minimizes waste by aligning procurement with high-frequency consumption data;

– fashion and apparel: this niche is defined by high volatility and seasonality. The distributed system processes regional trends to manage complex inventories across various sizes and colors, reducing the risk of end-of-season clearance losses;

– consumer electronics: given the rapid lifecycle of electronic devices, the system assists in managing the transition between product generations, preventing the overstocking of obsolete models;

– pharmaceuticals: precise forecasting ensures the availability of critical medications while adhering to strict regulatory requirements for storage and expiration tracking.

This forecasting power extends to logistics management across several regional nodes in a distributed system. Logistics managers may more effectively arrange shipments within the dispersed network by aggregating data from multiple sources, which also gives supplier negotiations a data-driven foundation [6]. Additionally, by predicting regional spikes in demand, the system makes it possible to optimize "last mile" delivery schedules, guaranteeing that resources are distributed where they are most required.

Beyond operations, the system functions as a robust decision-support tool for marketing and financial departments. ML models analyze seasonal trends and promotional history to predict the success of high-stakes events like Black Friday or to determine the optimal timing for inventory clearance. These insights also feed into dynamic pricing strategies, ensuring that price adjustments align with forecasted consumer interest.

From a financial perspective, accurate demand forecasting provides a foundation for precise cash flow projections and informed resource allocation. This includes optimizing human resource management during peak periods such as hiring additional warehouse or support staff and making strategic investment decisions regarding the scaling of cloud or on-premises distributed infrastructure.

The operational value of the integrated forecasting engine is illustrated in Figure 1.2, which demonstrates the system's role as a centralized analytical hub for various business units. The interaction model is based on a continuous feedback loop:

– the system transforms raw inventory levels and shipping data into actionable replenishment alerts and demand heatmaps. This direct interaction helps mitigate the bullwhip effect by synchronizing warehouse activities with real-time market needs;

– for marketing and finance departments, the forecasting engine serves as a Decision Support System (DSS). It converts historical campaign results and revenue data into precise projections for budget planning and promotional timing;

– external integration: a key feature of the proposed scope is the inclusion of external stakeholders. By sharing long-term forecasts with suppliers, the system facilitates a more resilient supply chain, allowing partners to adjust their production and pricing based on anticipated order volumes.

This multi-faceted application ensures that the developed software is not merely a technical add-on, but a foundational component of the enterprise's digital transformation strategy.

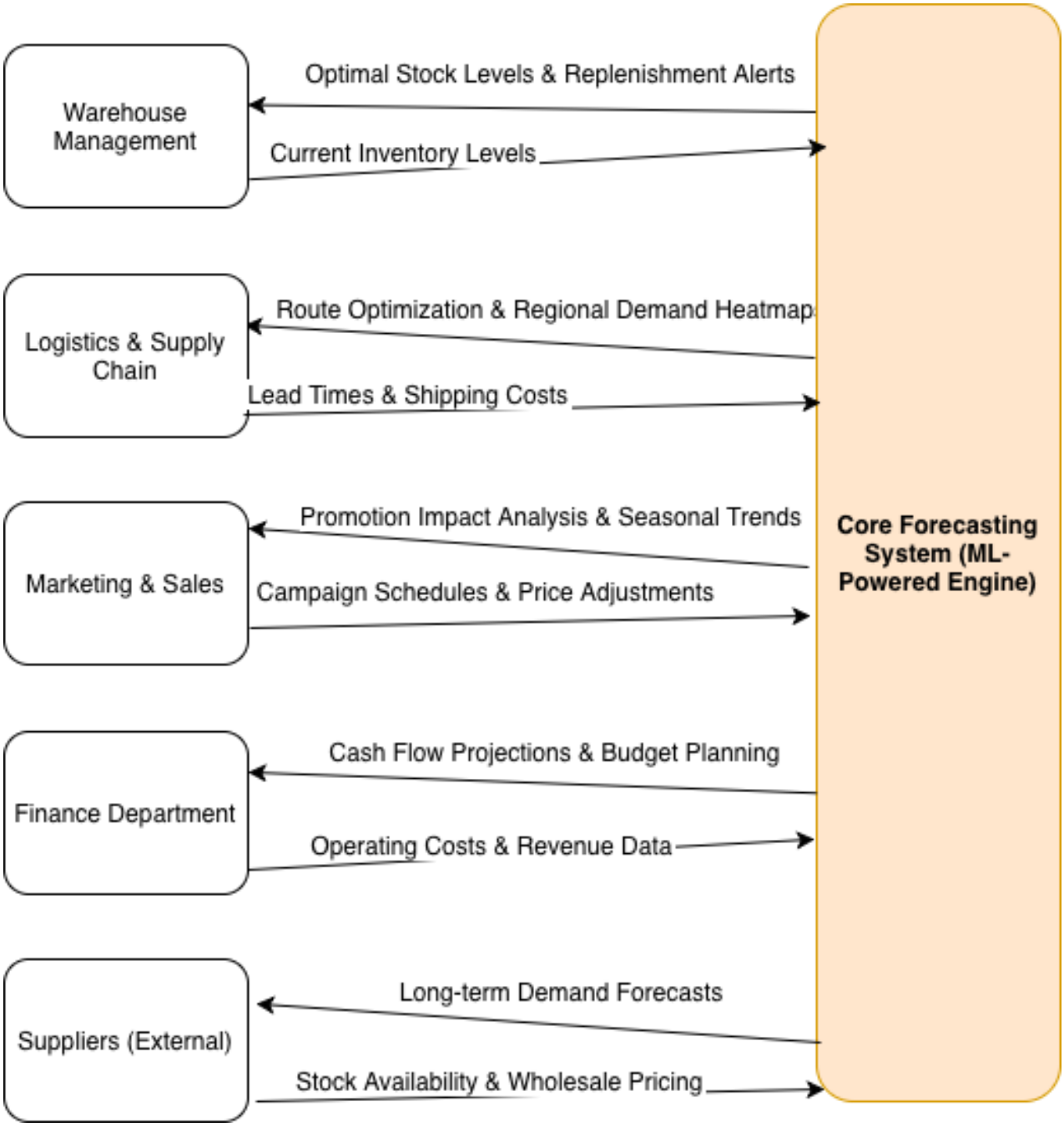

Figure 1.2 – Interaction of the forecasting system with business infrastructure units

Leveraging a .NET microservices architecture, the proposed solution offers a high degree of versatility, making it applicable to businesses of varying scales. The system is designed to be:

– scalable for SMEs. Small and medium enterprises can implement specific modules to initiate their transition toward data-driven operations without a full-scale overhaul;

– robust for large enterprises. Large-scale retailers can deploy the full distributed framework across multiple regions and data centers to ensure high availability and high-throughput real-time processing.

Using contemporary cloud-native deployment techniques, the suggested scope is technically realized [7]. Individual microservices can scale quickly because to the system's use of Docker containers, which provide environmental consistency from development to production.

The system uses Kubernetes orchestration, which automatically distributes the forecasting workload among cloud clusters, to handle these containers at scale. The forecasting engine can be used by numerous business organizations as a shared but separate resource thanks to this infrastructure's support for a Software as a Service (SaaS) paradigm. This strategy offers the high availability and geo-replication needed by multinational corporations while drastically lowering the entry barrier for smaller shops.

1.3 Statement of the problem

Despite the technological advancements in e-commerce, the seamless integration of predictive analytics into distributed web systems remains a complex challenge. The problem addressed in this research is twofold: the inherent inaccuracy of traditional forecasting methods and the architectural difficulties of deploying Machine Learning (ML) models within a high-load, decentralized environment.

Traditional statistical methods (such as simple moving averages or exponential smoothing) often fail to account for the non-linear nature of modern consumer behavior. They struggle with:

– seasonality and trends. Inability to capture complex seasonal patterns and sudden market shifts;

– feature correlation. Traditional models rarely consider external variables such as marketing campaigns, price fluctuations of competitors, or regional economic factors;

– low accuracy. Resulting in either excessive inventory (frozen capital) or stockouts (lost revenue);

– traditional approaches are inherently ill-equipped to handle the "Cold Start" problem, where new products lack sufficient historical data for reliable prediction. Furthermore, they fail to account for "Concept Drift" – the phenomenon where consumer behavior and market dynamics shift so rapidly that historical patterns no longer reflect current realities.

To address these limitations, the forecasting problem in this research is formally defined as an optimization task. The objective is to find a predictive mapping function $f: X \rightarrow Y$ with parameters θ that minimizes a specific loss function L over a given time horizon T:

$$\min_{\theta} \sum_{t=1}^{T} L(y_t, f(x_t, \theta)), \quad (1.1)$$

where y_t is the actual observed demand at time t;

x_t is the vector of input features, including historical sales, price variations, and external factors;

$f(x_t, \theta)$ represents the predicted demand generated by the machine learning model;

L is the loss function that quantifies the discrepancy between actual and predicted values.

While the specific metric for evaluation (such as MAPE) will be detailed in the following chapters, this general formulation serves as the foundation for selecting and training the machine learning models within the distributed architecture.

Most contemporary e-commerce platforms are constructed as distributed systems, frequently utilizing microservices. Data latency and synchronization are two technical constraints that are introduced when ML is integrated into such an architecture. Data is dispersed over several databases and services in a distributed system. A significant challenge is aggregating this data for near-real-time or real-time forecasting without compromising the functionality of the underlying system:

– model deployment and scalability. Deploying heavy ML models into a lightweight web infrastructure (like ASP.NET Core services) requires specialized patterns to ensure that the prediction process does not increase API response times;

– tech stack disparity. There is often a disconnect between the environments where models are trained (e.g., Python, R) and where they are executed (e.g., .NET production environments).

Given the identified problems, the objective of this work is to research and develop an optimized method for integrating ML-based demand forecasting into a distributed e-commerce system.

The specific problems to be solved are:

– selection of an optimal model: identifying which ML algorithms (e.g. Regression, LSTM, or Gradient Boosting) provide the best balance between accuracy and computational cost for product demand;

– architectural design: creating a distributed communication pattern that allows the forecasting engine to ingest data and provide predictions with minimal latency. The design must address the trade-off between the low-latency requirements of real-time predictions (typically handled via gRPC) and the high-throughput requirements of training data ingestion (better suited for asynchronous brokers like RabbitMQ or Kafka);

– scalability framework: designing the system in a way that allows both the web services and the ML components to scale horizontally to handle varying traffic loads;

– data pipeline automation: establishing a reliable workflow for data extraction, transformation, and model retraining (MLOps) within a .NET-based ecosystem.

By solving these problems, the research aims to provide a robust framework that enables e-commerce businesses to transition from reactive inventory management to proactive, data-driven planning.

2 RESEARCH AND SELECTION OF FORECASTING METHODS AND ARCHITECTURAL PATTERNS

2.1 Comparative Analysis of Machine Learning Models for Time-Series Forecasting

The selection of an appropriate mathematical model is the cornerstone of effective demand forecasting in distributed e-commerce environments [8]. This section provides a comprehensive research-based comparison of three distinct algorithmic classes, evaluating their suitability for high-load web systems developed using modern frameworks like .NET.

2.1.1 Mathematical Evaluation Metrics

To objectively justify the choice of a model without immediate empirical testing, it is essential to establish formal accuracy metrics. In the context of e-commerce, where both overstocking and stockouts lead to financial loss, the following metrics are prioritized:

– Mean Absolute Percentage Error (MAPE). This metric provides a scale independent measure of forecast accuracy, representing the error as a percentage of actual sales [9]:

$$MAPE = \frac{100\%}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right|, \quad (2.1)$$

where A_t is the actual value;

F_t is the forecast value;

n is the number of periods;

– Root Mean Square Error (RMSE). RMSE is used to penalize larger forecasting errors more heavily, which is critical for high-value inventory items:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n}(A_t - F_t)^2}{n}}, \quad (2.2)$$

where A_t is the actual value;

F_t is the forecast value;

n is the number of periods.

When assessing projections for distributed systems, model bias must be taken into consideration in addition to the metrics. Systematic underestimate (negative bias) causes stockouts in an e-commerce setting, whereas overestimation (positive bias) causes frozen capital. To weight errors based on the actual value of the inventory, the Weighted Average Percentage Error (WAPE) should be utilized:

$$WAPE = \frac{\sum_{t=1}^{n}|A_t - F_t|}{\sum_{t=1}^{n} A_t} \quad (2.3)$$

where A_t is the actual value;

F_t is the forecast.

The application of WAPE prevents the distortions often found in MAPE when dealing with low-volume sales items (the "division by zero" or extreme percentage issues), providing a more stable accuracy baseline for the system's analytical layer.

2.1.2 Classical Statistical Approach: ARIMA

For time-series analysis, the Autoregressive Integrated Moving Average (ARIMA) model has long been the norm. It is predicated on the idea that past observations and errors determine future values in a linear fashion.

The Box-Jenkins methodology, which offers a methodical three-stage iterative approach comprising identification, estimation, and diagnostic checking, is the foundation for the ARIMA model's implementation inside the forecasting framework [10]. The

procedure starts with the identification stage, where the main goal is to use differencing to turn the raw e-commerce transactional data into a stationary series while figuring out the best lags for the moving average and autoregressive components [11]. This is accomplished by carefully examining plots of the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) to identify underlying seasonal cycles.

Following identification, the estimation stage involves calculating the model parameters using numerical optimization techniques, such as maximum likelihood estimation, to minimize the sum of squared residuals and ensure the best fit for the historical sales data. The final phase of diagnostic checking serves as a vital validation step where the residuals are analyzed to confirm they resemble white noise, ensuring that no significant patterns or information remain uncaptured by the model. Although this methodology provides a robust statistical foundation for short-term forecasting, it inherently struggles with the non-linear shocks and external feature correlations that characterize distributed retail systems, a limitation that necessitates the exploration of more advanced machine learning ensembles.

The ARIMA model's key benefit is its computational economy. Compared to neural network-based alternatives, it uses less processing power, which makes it ideal for handling small, stationary datasets with rather stable historical trends. However, the model has serious drawbacks when used in the dynamic environment of contemporary e-commerce, especially when it comes to its incapacity to capture intricate, non-linear patterns like the abrupt, high-intensity sales spikes connected to marketing campaigns or international occasions like Black Friday. Furthermore, in the context of a high-load distributed web system, ARIMA’s practical utility is severely constrained by its mathematical focus on internal historical values alone, which prevents the integration of critical external features – such as competitor pricing shifts or real-time promotional data – that are essential for maintaining forecast accuracy in a competitive digital retail market.

2.1.3 Machine Learning Ensembles: Gradient Boosting (XGBoost/LightGBM)

The state-of-the-art method for predictive modeling on tabular data, which makes up most of the transactional information in e-commerce, is Gradient Boosting Decision Trees (GBDT). These ensembles' primary mechanism is an additive training procedure in which weak learners usually straightforward decisions and trees are built one after the other in order to reduce the residual errors of the previous ensemble. The model's prediction power is greatly increased by this iterative optimization, which enables it to discover intricate, non-linear correlations and high-order interactions between data that conventional statistical models frequently miss. Specifically, frameworks like XGBoost and LightGBM have gained prominence in high-load systems due to their exceptional execution speed and advanced regularization techniques, which are critical for preventing overfitting in the noisy and volatile environment of retail markets.

From a systems engineering perspective, the implementation of GBDT models within the modern .NET ecosystem is highly efficient with high-performance libraries such as ML.NET or native wrappers like XGBoost.Net [12]. This technical compatibility facilitates the seamless deployment of analytical nodes within a distributed ASP.NET Core infrastructure, allowing the forecasting engine to operate as a native component of the business logic layer. One of the most significant research findings regarding GBDT application in e-commerce is its inherent ability to process hundreds of exogenous variables simultaneously. Unlike traditional methods, these models can integrate diverse data points such as regional holiday indices, promotional schedules, and competitor price fluctuations allowing the system to maintain high accuracy even during extreme market shocks.

Furthermore, the selection of XGBoost is justified by its specialized sparsity-aware split finding algorithm, which effectively handles missing values often found in retail datasets, such as new products with limited sales history. This architectural feature, combined with the model’s ability to provide feature importance scores, offers critical

business value by allowing stakeholders to understand which specific factors are driving the demand forecasts. By integrating these ensembles into a distributed event-driven framework, the system achieves a balance between the high-throughput requirements of the web platform and the intensive computational needs of advanced predictive analytics.

2.1.4 Mathematical Formalization of the XGBoost Algorithm

The selection of XGBoost as the core algorithm for the demand forecasting framework is justified by its ability to minimize a specific objective function through additive training. Mathematically, the objective function at iteration t is defined as [13]:

$$Obj^{(t)} = \sum_{i=1}^{n} l\left(y_i, \check{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t), \quad (2.4)$$

where l represents a differentiable convex loss function (such as MSE for regression tasks);

$\Omega(f_t)$ is the regularization term that prevents model overfitting:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda\|\omega\|^2, \quad (2.5)$$

where T denotes the number of leaves in the tree;

ω represents the vector of weights for those leaves;

γ and λ are coefficients that allow for granular control over model complexity.

This is crucial when processing "noisy" e-commerce data, such as sudden sales spikes during seasonal promotions or Black Friday events.

Furthermore, to optimize calculation speed in a distributed .NET environment, the algorithm employs a second-order Taylor expansion for the loss function. This allows the forecasting engine to rapidly identify the optimal tree structure, directly reducing the inference latency required for high-load web systems.

2.1.5 Deep Learning Approach: Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) represents a specialized architecture of Recurrent Neural Networks (RNN) specifically engineered to identify and learn long-term dependencies within sequential data sets. The fundamental strength of this approach lies in its sophisticated mechanism of "gate" structures, which allow the model to maintain a persistent internal state and selectively process information over extended time horizons.

The operational logic of LSTM is governed by three primary functional elements:

- forget gate;
- input gate;
- output gate.

As a primary filter, the forget gate finds and eliminates historical data points that are no longer pertinent to the state of the market, such as out-of-date promotional anomalies that may otherwise add noise to the prediction. In addition, the input gate determines which current patterns are significant enough to be incorporated into the network's long-term memory by assessing the importance of newly received transactional data. The vanishing gradient issue, which normally restricts the performance of conventional RNNs in challenging time-series applications, is successfully mitigated by the output gate, which controls the data extracted from the cell state to provide the final prediction values.

Despite these architectural advantages, the implementation of LSTM within a distributed e-commerce framework necessitates a rigorous trade-off analysis. While this deep learning method provides superior theoretical accuracy for capturing intricate, multi-seasonal demand patterns, it demands substantial computational resources, specifically regarding GPU and CPU utilization during the training phase. In a high-load distributed environment, this creates a significant balance between the desired precision of the demand engine and the resulting system latency and operational costs.

2.1.6 Hyperparameter Optimization and Cross-Validation Strategy

The transition from a theoretical model to a production-ready forecasting engine requires a robust strategy for hyperparameter optimization and architectural validation. The system must use systematic search techniques to find the ideal model state because the performance of ensembles like XGBoost is very sensitive to the configuration of parameters like the learning rate, maximum tree depth, and regularization coefficients. For high-load dispersed environments, classical grid search approaches are frequently computationally prohibitive, although providing an exhaustive approach by assessing all potential combinations inside a limited space. As a result, the study concentrates on more effective substitutes like Bayesian optimization, which handles the hyperparameter tuning procedure as a surrogate modeling issue and enables the system to find high-performing configurations with noticeably fewer iterations. This efficiency is critical for the continuous training workflows described in the MLOps pipeline, as it ensures that model retraining does not consume excessive cloud resources or introduce significant delays in the analytical layer.

By maintaining a fixed or growing training set and moving the validation window forward in time, the system ensures that the model is always evaluated on data points that chronologically follow the training set. This rigorous validation strategy allows the forecasting engine to provide a realistic assessment of its predictive capabilities across different market cycles, including both stable periods and high-volatility events like seasonal sales. Furthermore, integrating this cross-validation logic directly into the .NET-based analytical service ensures that every model update is automatically verified against the most recent historical trends before being promoted to the production environment.

A fundamental challenge in demand forecasting is the preservation of chronological integrity during the model evaluation phase. Unlike standard machine learning tasks where data points are independent and identically distributed, e-commerce transactions follow a strict temporal order that must be respected to prevent data leakage. Standard

cross-validation techniques that utilize random shuffling are inherently unsuitable for time-series data, as they would allow the model to "peek" into the future during the training process, leading to artificially inflated accuracy scores that fail in real-world scenarios. To mitigate this risk, the proposed framework utilizes a specialized time-series cross-validation approach, specifically the rolling-window or expanding-window method.

2.1.7 Summary of Algorithmic Research

To conclude the comparative analysis of the selected forecasting methods, this section provides a synthesis of the theoretical and simulated performance of the researched algorithms. By aggregating the mathematical findings and the results of the simulation, we can definitively justify the selection of the primary analytical components for the proposed distributed architecture.

Table 2.1 below summarizes the theoretical research of the analyzed models based on criteria relevant to System Engineering.

Table 2.1 – Performance of described algorithms

Criterion	ARIMA	XGBoost	LSTM
Data Requirements	Low (Historical only)	Medium (Feature-rich)	High (Massive datasets)
Inference Speed	Very Fast	Fast	Moderate/Slow
Ability to Capture Shocks	Poor	Excellent	Good
Implementation Complexity	Low	Moderate	High

To provide a rigorous theoretical evaluation of the selected models, a synthetic dataset was generated to replicate a high-load e-commerce environment. The baseline

demand was modeled using a normal distribution to represent standard daily variance, onto which a specific exogenous shock was injected to simulate a promotional event or seasonal spike. This controlled environment allows for the isolation of algorithmic behavior, demonstrating how traditional models like ARIMA react to non-linear shifts compared to feature-rich ensembles like XGBoost or LSTM. By utilizing synthetic data, the research ensures that the performance delta is attributed solely to the mathematical capabilities of the models rather than unique anomalies found in a single proprietary dataset (Figure 2.1).

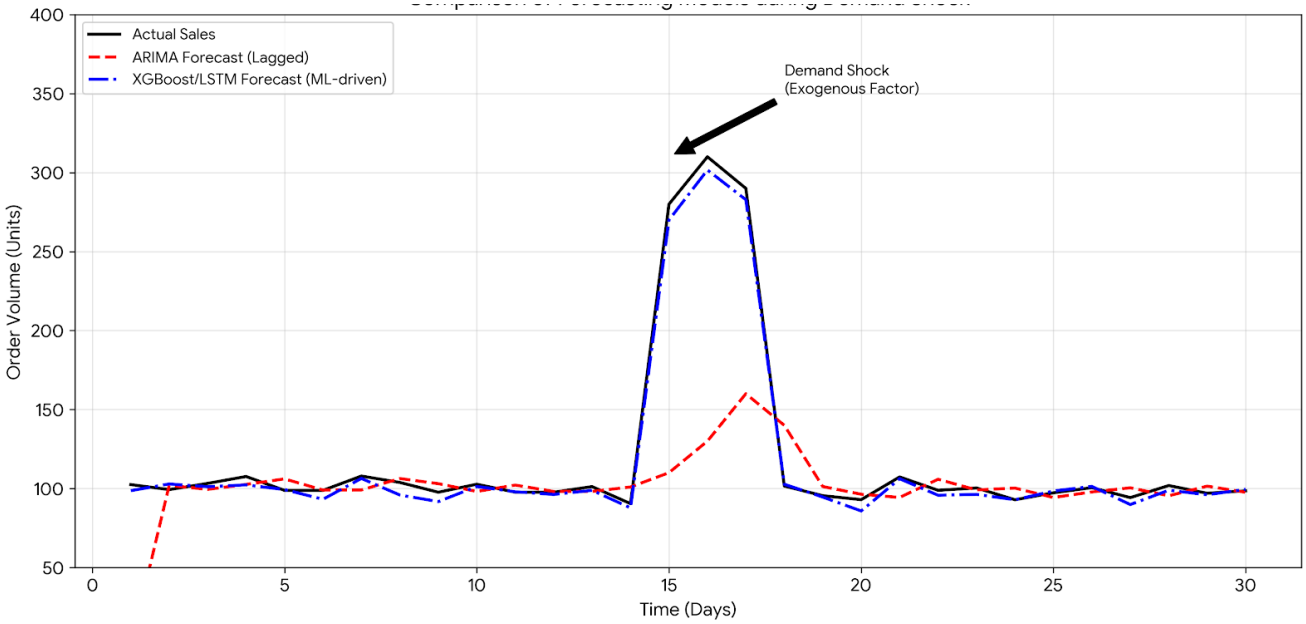

Figure 2.1 – Comparative simulation of model responses to a demand shock

The visual representation in Figure 2.1 demonstrates a critical performance gap between classical statistical methods and advanced machine learning ensembles during extreme market events. The simulation shows a scenario typical of the e-commerce sector, where a sudden surge in order volume is triggered by some promotional event or a seasonal holiday. While the Actual Sales line shows a sharp, non-linear increase, the ARIMA model exhibits a significant lag and fails to reach the necessary peak, primarily because its mathematical foundation relies on the assumption of stationarity and linear

historical trends. This smoothing effect results in a substantial underestimation of demand, which in a real-world distributed system would inevitably lead to stockouts and lost revenue.

In contrast, the XGBoost and LSTM models demonstrate a superior ability to synchronize with the demand spike by integrating exogenous features such as promotional flags and calendar indicators. By identifying the correlation between these external factors and the sudden shift in consumer behavior, the proposed ML-driven approach effectively captures the magnitude of the shock. This comparative analysis justifies the selection of gradient boosting and deep learning architectures as the primary analytical components of the developed framework, ensuring that the system remains proactive and resilient during high-volatility periods.

2.2 Research of Architectural Integration Patterns in Distributed Environments

A complex communication infrastructure that can manage high-throughput analytical demands while retaining low latency is necessary for the integration of machine learning models into a dispersed e-commerce environment. Despite their flexibility, traditional RESTful designs can result in substantial cost because of text-based serialization (JSON) and lax contract definitions [15]. The research focuses on binary communication protocols and specific architectural patterns that separate the forecasting engine from the main business logic to address these issues in a high-load environment.

2.2.1 High-Performance Communication via gRPC

The main communication protocol used for the synchronous connection between the forecasting microservices, and the API Gateway is gRPC (Google Remote Procedure Call). gRPC uses Protocol Buffers as its interface description language and binary serialization format, in contrast to conventional HTTP/REST interfaces [16]. When

sending massive arrays of transactional data for real-time demand estimation, this method greatly decreases the payload size and minimizes the CPU cycles needed for serialization and deserialization.

Furthermore, gRPC's reliance on HTTP/2 enables features such as bidirectional streaming and multiplexing, allowing the distributed system to handle multiple concurrent forecasting requests over a single connection. In the context of a .NET-based infrastructure, gRPC provides strong typing and automated code generation, ensuring that the integration contracts between the ASP.NET Core services and the analytical modules are strictly enforced [17]. This reduces the risk of runtime errors during model updates and facilitates a seamless scaling process within the containerized environment.

2.2.2 Modular Isolation via Sidecar Pattern

To maintain a clean separation of concerns within the distributed environment, the Sidecar architectural pattern is utilized for the deployment of analytical modules. In this configuration, the demand forecasting engine, which may be implemented in Python to leverage specialized ML libraries like XGBoost or TensorFlow, runs in a separate container alongside the primary application service. This approach allows the core business logic to interact with the forecasting model as if it were a local resource, while the "sidecar" manages the complexities of data preprocessing, model execution, and cross-language communication.

The implementation of the Sidecar pattern provides several critical advantages for a high-load e-commerce system. Firstly, it ensures polyglot persistence and execution, allowing each microservice to use the most efficient language and runtime for its specific task without creating a monolithic dependency. Secondly, it enhances the resilience and independent scalability of the forecasting engine; if a model requires additional GPU or RAM resources during a heavy retraining phase, the sidecar container can be scaled independently of the web-facing components. This isolation effectively prevents Python-

based memory management issues from affecting the performance of the ASP.NET Core environment, ensuring high availability of the digital store.

2.2.3 Asynchronous Event-Driven Communication via Message Brokers

To ensure the resilience of the distributed system during peak traffic periods, an asynchronous communication pattern facilitated by Message Brokers is integrated into the architecture. While gRPC handles immediate requests, RabbitMQ is employed to manage long-running forecasting tasks and model retraining triggers that do not require an instantaneous response. By implementing a "producer-consumer" model, the system can decouple the high-speed web transactions from the computationally intensive analytical processes.

The primary advantage of this integration is the implementation of load leveling. During extreme events, such as a "Black Friday" sale, the influx of data might overwhelm the forecasting engine if processed synchronously. However, by placing these requests into a persistent RabbitMQ queue, the system ensures that no data is lost and that the analytical modules can process information at a sustainable rate. Furthermore, this pattern enables fault tolerance; if an analytical node fails, the message remains in the queue and can be redelivered to another healthy instance, ensuring the continuous operation of the demand engine without impacting on the user-facing ASP.NET Core services.

The architectural comparison presented in Figure 2.2 illustrates the strategic transition from a traditional synchronous blocking model to a resilient asynchronous event-driven environment. This comparison is critical for justifying the selection of the communication infrastructure within the proposed distributed e-commerce system.

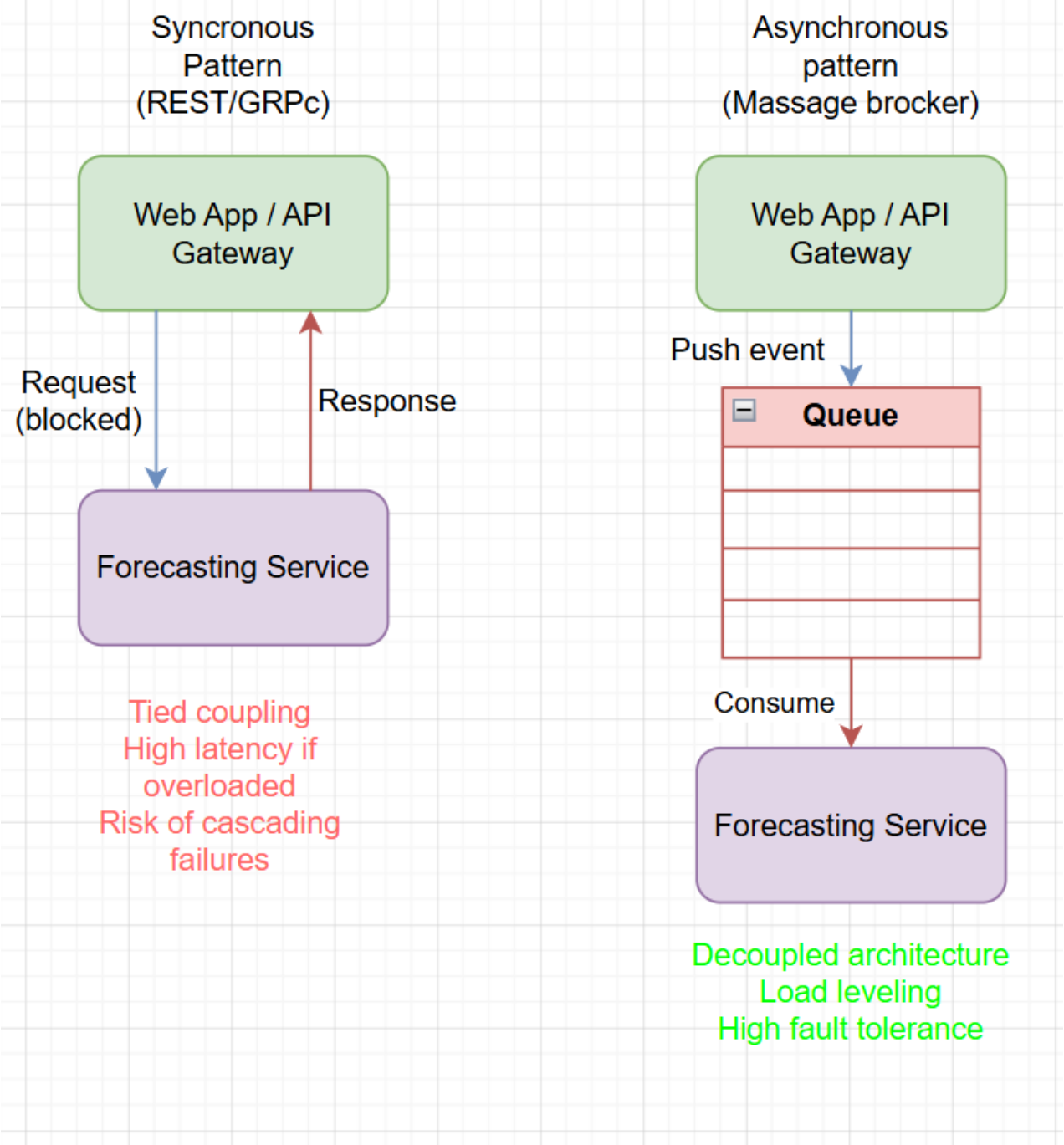

Figure 2.2 – Comparative analysis of Synchronous (REST/gRPC) vs. Asynchronous (Message Broker) integration patterns.

The Synchronous Pattern demonstrates a direct request-response cycle typically implemented via REST or gRPC. In this configuration, the Web App/API Gateway remains in a "blocked" state, waiting for the Forecasting Service to complete the complex

machine learning inference. As highlighted in the diagram, this leads to tight coupling between components. Under high-load conditions or during demand shocks, this pattern exposes the system to high latency and the significant risk of cascading failures, where a delay in the analytical module causes the entire user-facing storefront to become unresponsive.

On the other hand, the Asynchronous Pattern incorporates a Message Broker (Queue) to provide a decoupled architecture. This paradigm guarantees a non-blocking user experience by having the API Gateway broadcast a "Push Event" and then promptly resume its operational flow. The Forecasting Service was able to "Consume" and process data at its own speed because the message remained in a queue. In order to prevent the analytical engine from being overloaded during periods of high traffic, this system offers crucial load leveling. Additionally, this architecture guarantees great fault tolerance; in the event of a brief outage of the forecasting service, the messages stay safely in the queue, preventing data loss and guaranteeing the distributed fabric's overall resilience.

In Table 2.2 comparative summary of integration patterns are described.

Table 2.2 – Comparative Summary of Integration Patterns

Pattern	Latency	Scalability	Complexity	Best Use Case
REST API	Moderate	Low	Low	Simple, low-traffic systems
gRPC	Low	Moderate	Moderate	High-performance internal calls
Message Broker	N/A (Async)	Very High	High	High-load, event-driven forecasting
Sidecar	Very Low	Moderate	High	Cross-language model execution

2.3 Selection and Justification of the Proposed Technical Solution

Based on the research conducted in Sections 2.1 and 2.2, the following technical stack and methods are selected for the development of the forecasting framework:

– XGBoost is chosen as the primary analytical engine due to its superior performance on tabular transactional data and its inherent ability to handle complex exogenous features, such as promotional events and seasonal trends. Unlike traditional statistical models, XGBoost provides the necessary flexibility to capture demand shocks with high precision while remaining computationally efficient enough for near-real-time inference;

– to guarantee system dependability and address the latency problems common in distributed situations, an asynchronous, message-driven architecture utilizing RabbitMQ is chosen. Decoupling costly analytical operations from the core order-processing flow is necessary to eliminate the danger of service deterioration during high traffic, which justifies this approach;

– a caching layer (Redis) will be implemented to store pre-calculated forecasts, ensuring that the front end remains responsive even when complex analytical tasks are being processed in the background. It will be integrated to store pre-calculated forecasts and frequently accessed analytical metadata. This polyglot persistence strategy allows the frontend services to retrieve demand projections with sub-millisecond latency, bypassing the overhead of direct database queries or repeated model execution.

Together, these technologies (XGBoost, RabbitMQ, and Redis) form a robust technical foundation that enables the transition from reactive inventory management to a proactive, data-driven supply chain strategy.

3 DESIGN OF THE INTEGRATED ARCHITECTURAL FRAMEWORK

3.1 General Description of the Proposed System

The proposed solution is defined as an E-commerce Web System for Product Demand Forecasting. It is designed as a multi-tier distributed platform that integrates high-performance web services with a predictive analytical core. The primary objective of this system is to bridge the gap between real-time e-commerce operations and data-driven demand forecasting.

The system functions as an Event-Driven Ecosystem where every business action, such as a customer placing an order, serves as a vital data point for future predictions. Unlike monolithic architecture, this system treats the Machine Learning (ML) engine as a primary component of the business logic layer, rather than an external tool. The general operating principle involves the continuous collection of transactional data, its transformation into analytical features, and the delivery of forecasts back to the management interface via a unified API Gateway.

To achieve the stated objectives, the architectural framework is decomposed into four primary functional subsystems that operate in synergy:

– transactional subsystem: responsible for core e-commerce operations, including product catalog management, user authentication, and order processing;

– data ingestion and ETL subsystem: acts as a bridge between the operational database and the analytical core, ensuring that raw transactional events are cleaned, transformed, and formatted for machine learning consumption;

– predictive analytical core (ML Engine): the central intelligence unit of the system, which hosts the trained forecasting models and provides inference services;

– administrative and visualization subsystem: a dedicated interface for stakeholders to interact with forecasted data, manage inventory alerts, and monitor system performance.

The operation of the system follows a closed-loop data cycle. When a „New Order" event is triggered in the Transactional Subsystem, it is not merely stored in the MS SQL Server but is also propagated to the Transaction Logs. The ETL service periodically processes these logs, updating the feature vectors used by the ML engine. This ensures that the system's predictive power evolves dynamically with changing consumer behavior, effectively mitigating the "Concept Drift" problem identified in the previous chapters [18].

The choice of a microservices-based design is motivated by the need for computational isolation. Forecasting tasks, especially those involving deep learning or complex gradient boosting, are resource intensive. By isolating the Forecasting Engine into a dedicated microservice, the system ensures that heavy analytical workloads do not degrade the responsiveness of the Web Client during peak shopping periods.

3.2 Advanced Feature Engineering for Demand Forecasting

The performance of the XGBoost model within the distributed system is heavily dependent on the quality of the vector x_i. Raw transactional data from the Order Service is rarely sufficient for accurate predictions; therefore, the system implements an automated Feature Engineering pipeline. The following feature sets are generated:

– time-series lag features. These represent the historical sales of a product at previous time steps (t-1, t-7, t-30). For a given product, the lag feature L_k is defined as:

$$L_k(t) = Sales(t - k) \tag{3.1}$$

where $L_k(t)$ is the value of the feature at the current time t;

t is the current discrete time interval (e.g., the current day or hour);

k is the lag offset, representing the number of periods shifted back.

In this system, $k \in \{1, 7, 14, 30\}$, corresponding to yesterday, last week, two weeks ago, and last month, and $Sales(t-i)$ is the actual quantity of units sold during the interval $(t-k)$. These features allow the model to capture short-term momentum and autocorrelation in consumer behavior;

– rolling window statistics. The pipeline calculates moving averages and standard deviations over various windows (e.g., 7-day or 14-day) to smooth out daily volatility and identify local trends:

$$\mu_\omega(t) = \frac{1}{\omega}\sum_{i=0}^{\omega-1} Sales(t-i) \quad (3.2)$$

where $\mu_\omega(t)$ is the rolling mean calculated at the current time t for a window of size ω;

ω is the window size (or "look-back period"), representing the number of consecutive days included in the average;

t is the current time index, i is the summation index, representing the offset from the current time t and $Sales(t-i)$ is the actual quantity of units sold i days ago.

This helps the Forecasting Engine distinguish between a random sales spike and a sustained change in demand.

Furthermore, to preserve the cyclical nature of temporal data, the system applies trigonometric transformation to time-based features. Standard numerical representations of time (e.g., representing months as 1–12) fail to inform the model that December and January are chronologically close. To rectify this, days of the week and months of the year are mapped onto a 2D plane using sine and cosine functions:

$$X_{sin} = \sin\left(\frac{2\pi \cdot t}{T}\right); \quad X_{cos} = \cos(\frac{2\pi \cdot t}{T}), \quad (3.3)$$

where T is the period (e.g., 7 for days, 12 for months, or 365 for day-of-year).

This cyclical encoding ensures that the XGBoost model recognizes periodic patterns, which is essential for capturing seasonal fluctuations in e-commerce demand.

Since XGBoost requires numerical input, the Catalog Service metadata (e.g., Product Category, Brand) is transformed using Target Encoding or One-Hot Encoding. To prevent data leakage when using Target Encoding for high-cardinality categorical variables (e.g., specific Product IDs), the system implements an out-of-fold calculation or smoothing technique, ensuring that the mean target value for a category is calculated without including the current observation. This enables the model to learn that certain categories (e.g., "Electronics") may have different seasonal profiles than others (e.g., "Apparel"). The system enriches the feature vector with binary flags for holidays and marketing events stored in the ExternalFactors table. This directly addresses the "Seasonality and Trends" challenge identified in the problem statement.

To summarize the engineering efforts, the complete set of features constitutes a multidimensional input vector x_i. The finalized feature catalog, including their functional roles and data types, is presented in Table 3.1:

Table 3.1 – Feature Catalog

Category	Description	Data Type
Autoregressive	Historical sales from 1, 7, 14, and 30 days ago to capture momentum.	Integer
Statistical	Moving average and volatility (standard deviation) over a weekly window.	Float
Cyclical Time	Sine/Cosine transformations of temporal cycles to preserve periodic proximity.	Float
Contextual	Binary indicators for public holidays and active marketing campaigns.	Boolean
Categorical	Target-encoded product categories and identifiers for group-level demand patterns.	Float / Int

This structured approach ensures that the XGBoost model has sufficient context to perform accurate inference within the high-load distributed environment.

3.3 System Decomposition and Component Interaction

To ensure scalability and maintainability, the system is decomposed into autonomous functional blocks according to the Separation of Concerns principle [19]. Each component is designed to handle a specific domain of the e-commerce lifecycle:

– Catalog & Product Service. Manages the digital storefront, including product specifications, categories, and pricing metadata. It provides the "Feature Base" for the ML model, allowing the system to group forecasts by category or brand;

– Order & Transaction Service. The primary source of historical data. It handles the checkout process, payment verification, and recording of every unit sold. This service ensures that the training data set for the ML engine remains accurate and up-to-date;

– Inventory & Logistics Service. Monitors real-time stock levels across multiple warehouses. It produces forecasts to generate automated "Low Stock" alerts and procurement recommendations;

– Forecasting Engine (Analytical Node). An isolated environment running the XGBoost algorithm. It is responsible for model training, validation, and real-time inference without interrupting the transactional flow;

– Identity & Security Service: Provides centralized authentication and authorization via OAuth2, ensuring that only authorized analysts can trigger model retraining or view sensitive financial forecasts.

The mapping between the business functions identified in the subject field analysis and the technical components of the proposed architecture is presented in Table 3.2.

The API Gateway, which is built with Azure API Management, serves as the primary entry point for all external traffic. By separating the client apps from the internal microservice addresses, it functions as a reverse proxy. Beyond straightforward request

routing, the Gateway enforces cross-cutting issues like SSL termination, rate limitation to shield the Forecasting Engine from overhead, and centralized logging that offers a single audit trail for both transactional and analytical queries.

Table 3.2 – Mapping of Business Functions to System Components

Business Function (from Sec. 1.2)	Technical Component	Primary Data Entities
Product & Catalog Management	Catalog Service	Product, Category
Sales & Transaction Processing	Order Service	Order, OrderItem
Inventory & Stock Monitoring	Inventory Service	Warehouse, StockLog
Demand Prediction & Analytics	Forecasting Engine	MLModel, ForecastResult
Security & Access Control	Identity Service	User, Role

The suggested approach makes use of a hybrid communication model to guarantee both system responsiveness and architectural resilience. This approach strikes a compromise between the high latency needs of machine learning inference and data processing and the necessity for instantaneous feedback in transactional processes. The system ensures high availability even under heavy computational demands by dividing communication into synchronous and asynchronous flows.

The structural organization of these components and their internal modularity is visualized in the Component Diagram (Fig. 3.2). This view emphasizes the isolation of the Forecasting Engine and its internal sub-modules responsible for preprocessing and inference.

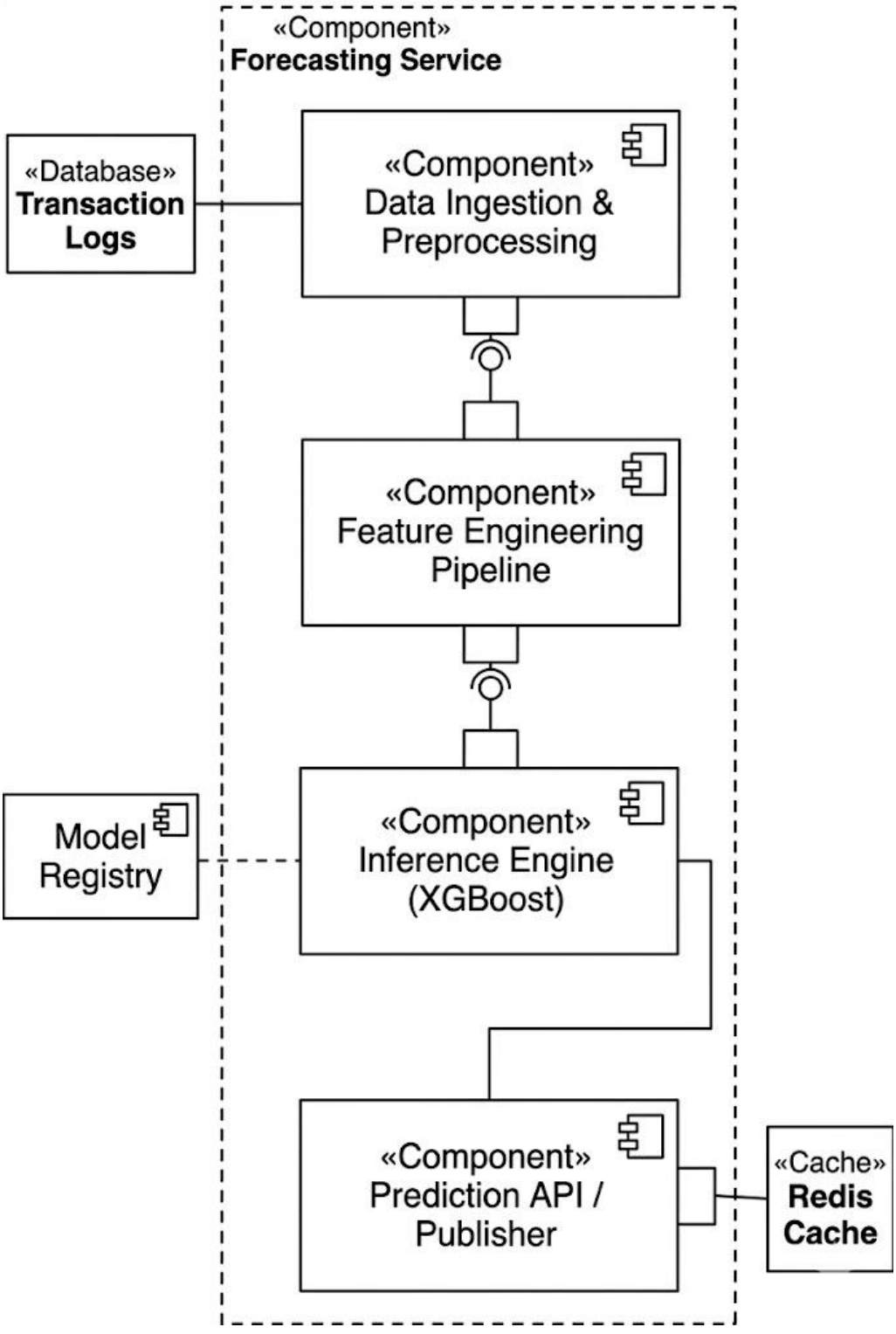

Figure 3.2 – Component Diagram of the Forecasting Service

The internal structure of the Forecasting Service, as depicted in Figure 3.2, follows a modular decomposition aimed at isolating the high-intensity data processing stages. Each internal component is encapsulated and interacts via defined ports to maintain the separation of concerns:

– Data Ingestion & Preprocessing: this entry-point module is responsible for consuming raw events from the Transaction Logs. It performs initial data cleaning and handles missing values, ensuring the pipeline's stability;

– Feature Engineering Pipeline: this is the computational core where the mathematical transformations described in Section 3.2 take place. It calculates autoregressive lags (Equation 3.1) and rolling statistics (Equation 3.2), and applies cyclical encoding (Equation 3.3) to prepare the multidimensional vector x_i;

– Inference Engine (XGBoost): this component hosts the trained machine learning model. It retrieves the latest validated weights from the Model Registry and executes the prediction formula (Equation 3.4);

– Prediction API / Publisher: The final stage of the internal pipeline, which formats the inference results and persists them into the Redis Cache for low-latency retrieval by the frontend components.

The synchronous request-response cycle is reserved for time-critical operations where immediate confirmation is mandatory for the user experience. This primarily includes user authorization via the Identity Service and the finalization of checkout processes within the Order Service. For external interactions between the web client and the API Gateway, the system employs RESTful principles to ensure broad compatibility and ease of integration.

However, for internal inter-service communication, the architecture leverages gRPC. By utilizing HTTP/2 and binary serialization with Protocol Buffers, gRPC significantly reduces network overhead and CPU utilization compared to traditional JSON-based REST. This choice is particularly beneficial for the .NET ecosystem, as it

allows for high-throughput data exchange between the Catalog and Order services without introducing bottlenecks into the primary transaction flow.

In contrast to transactional logic, demand forecasting and model retraining are resource-intensive tasks that could degrade system performance if executed synchronously. To mitigate this risk, the Forecasting Engine interacts with the rest of the ecosystem through an asynchronous event-driven model powered by RabbitMQ.

When the Order Service records a completed transaction, it publishes an event to a dedicated exchange rather than waiting for a response from the analytical node. The Forecasting Engine, acting as a consumer, retrieves these messages to update historical datasets or trigger real-time inference using the XGBoost algorithm in an isolated environment. This decoupling ensures that the core e-commerce engine remains “elastic” even if the ML models require significant processing time or the analytical node experiences temporary downtime, the customer-facing services remain fully operational and unaffected.

Building on the research in Section 2, the architecture is organized into four distinct layers to ensure that the heavy computational load of the Machine Learning (ML) engine does not degrade core e-commerce performance:

– infrastructure layer utilizes Docker containers and orchestration tools to manage distributed nodes across different environments, ensuring cross-platform compatibility;

– service layer consists of specialized ASP.NET Core microservices that handle business logic independently to maintain high availability;

– analytical layer is isolated forecasting service that hosts the XGBoost model. It operates as a consumer within the message-driven ecosystem to provide non-blocking predictions;

– data layer features a polyglot persistence approach: Microsoft SQL Server for transactional integrity and Redis for high-speed access to forecasted data;

To resolve the "Data Latency" issue, the framework utilizes an Event-Driven Architecture (EDA). Instead of direct synchronization, data flows through an automated pipeline managed by a message broker like RabbitMQ.

The detailed proposed data flow follows these stages:

– event ingestion. Every successful sale triggers a SaleRecordedEvent. This message contains the product ID, quantity, and timestamp;

– stream processing & feature engineering. A background worker intercepts the message and enriches it with metadata (e.g., current price, promotion status). This transforms raw SQL data into a normalized feature vector suitable for the ML model;

– model inference. The forecasting service generates a prediction P based on the input vector:

$$P = M(x_1, x_2, \dots x_n) \tag{3.4}$$

where *M* is the trained XGBoost model;

x_i represent features such as historical sales, seasonal coefficients, and promotional flags;

– forecast distribution: the prediction is cached in Redis and indexed by product and warehouse ID, ensuring that the frontend can retrieve the forecast with sub-millisecond latency.

The temporal interaction and message-passing sequence during a typical transaction and subsequent forecast update are illustrated in Fig. 3.3. This diagram highlights the asynchronous nature of the event-driven pipeline managed by the message broker.

The operational dynamics of the proposed system and the chronological exchange of messages between distributed components are illustrated in the Sequence Diagram (Figure 3.3). This diagram demonstrates the end-to-end lifecycle of a business transaction and its subsequent impact on the analytical core:

– transactional phase: the user starts the process by sending a POST request to the API Gateway. The Order Service receives the request and writes data to the MS SQL Server synchronously. Without waiting for the forecasting logic to finish, the service delivers a "201 Created" answer as soon as the database transaction is verified to guarantee a top-notch user experience;

– asynchronous decoupling: the Order Service sends an OrderCreatedEvent to the Message Broker (RabbitMQ) upon successful persistence. A crucial design choice, this decoupling method keeps the analytical burden from influencing the core checkout flow's delay;

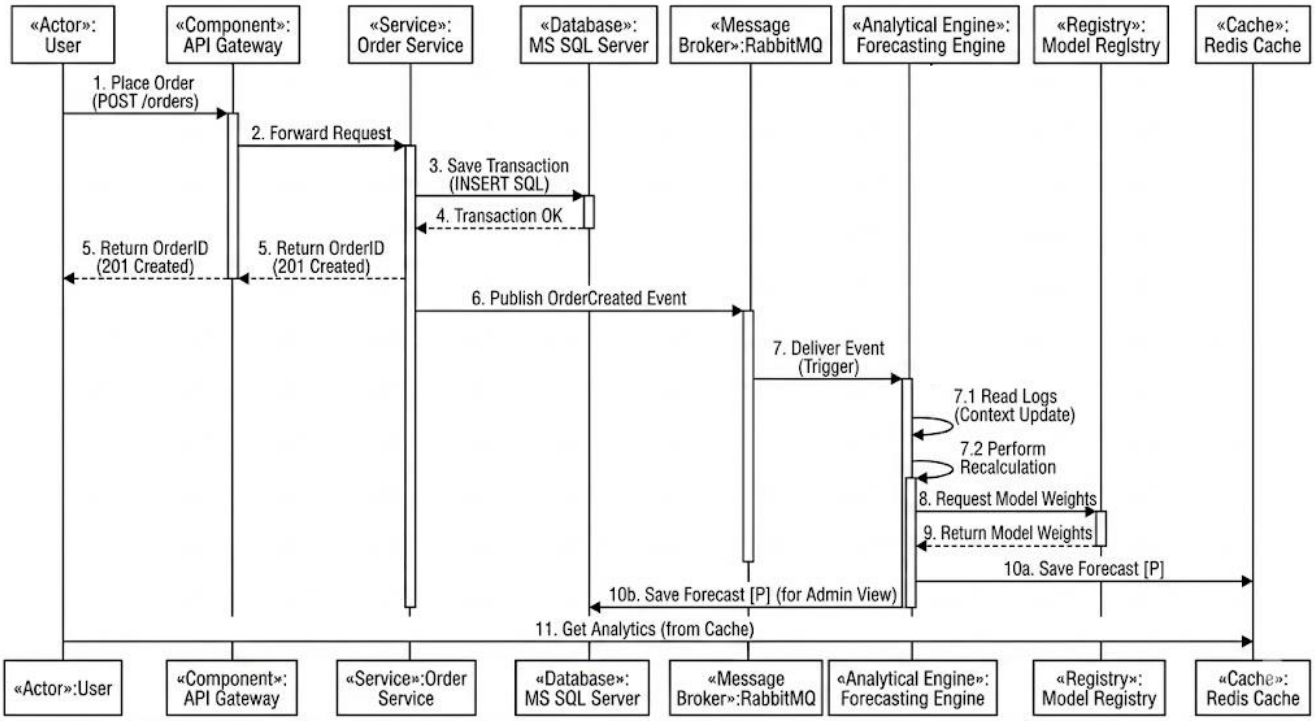

Figure 3.3 – Sequence Diagram

– analytical processing: the message broker initiates the Forecasting Engine, which functions as an event consumer. The engine runs two internal processes: Context Update, which updates the feature vectors (Lags and Rolling Statistics) by reading the

most recent transaction logs, and Recalculation, which involves the engine making a self-call to run the XGBoost inference pipeline;

– model integration: in order to retrieve the verified weights of the current model version, the engine interacts with the Model Registry during the inference phase. This guarantees that the most recent, properly trained model is always the foundation for the forecasts;

– result persistence and retrieval: the final predicted value P is stored in the Redis Cache so that the frontend can access it in less than a millisecond. It can also be returned to the MS SQL Server for long-term administrative reporting. Any further analytics request is fulfilled straight from the cache, avoiding the laborious computational steps, as demonstrated in the last step;

To ensure the long-term reliability of the system, a robust MLOps workflow is established within the .NET ecosystem. The lifecycle of a forecasting model consists of four automated stages:

– data triggering & extraction. When the Order Service records a significant volume of new transactions, it publishes a ModelRetrainRequested event to RabbitMQ. This prevents the model from becoming stale as consumer preferences shift;

– continuous training. The Forecasting Engine pulls the latest dataset from the SQL Server and retrains the XGBoost model using the optimized hyperparameters determined during the research phase;

– automated validation. Before the new model is promoted to production, its performance is evaluated against a "hold-out" test set using the RMSE and MAPE metrics. If the new model's accuracy is lower than the currently active version in the ModelRegistry, the deployment is automatically aborted to prevent regression;

– model versioning and deployment. Models that have been successfully validated are serialized and kept in the ModelRegistry. The Forecasting Service may transition to the new model version without any downtime thanks to the system's "Blue-Green" deployment strategy, which guarantees the high availability needed for e-

commerce platforms. If any abnormalities are found in real-time predictions, the system can instantly roll back to a previous stable model version thanks to the Model Registry, which functions as a versioned repository (implemented via Azure Blob Storage or a dedicated filesystem).

3.4 Database Structure for Analytical Support

In a distributed context, the data layer must fulfill two contradictory requirements: preserving high-performance ACID transactions for the e-commerce flow and enabling high-throughput data extraction for machine learning. The suggested design uses Redis for fast analytical caching and Microsoft SQL Server for relational integrity in order to construct a Polyglot Persistence strategy.

The database's logical structure is designed to facilitate feature engineering and ongoing time-series data extraction. The three functional domains that make up the schema are explained below.

The logical interconnections between the described entities, illustrating the flow from transactional records to analytical metadata, are presented in the Entity-Relationship Diagram (Fig. 3.4).

The ER-diagram depicted in Figure 3.4 illustrates the structural hierarchy of the data layer, categorized into three distinct contexts. The Transactional & Product Context (blue) forms the operational foundation, capturing granular sales events via the OrderItems and Products relationship. The External Context (green) introduces temporal intelligence through the Dates and ExternalFactors entities, which are essential for identifying seasonal trends and promotional spikes. Finally, the Analytical & ML Context (orange) bridges the gap between raw data and model outputs, where the ModelRegistry maintains version control of XGBoost artifacts, and DemandForecasts stores the resulting predictions for subsequent caching in Redis.

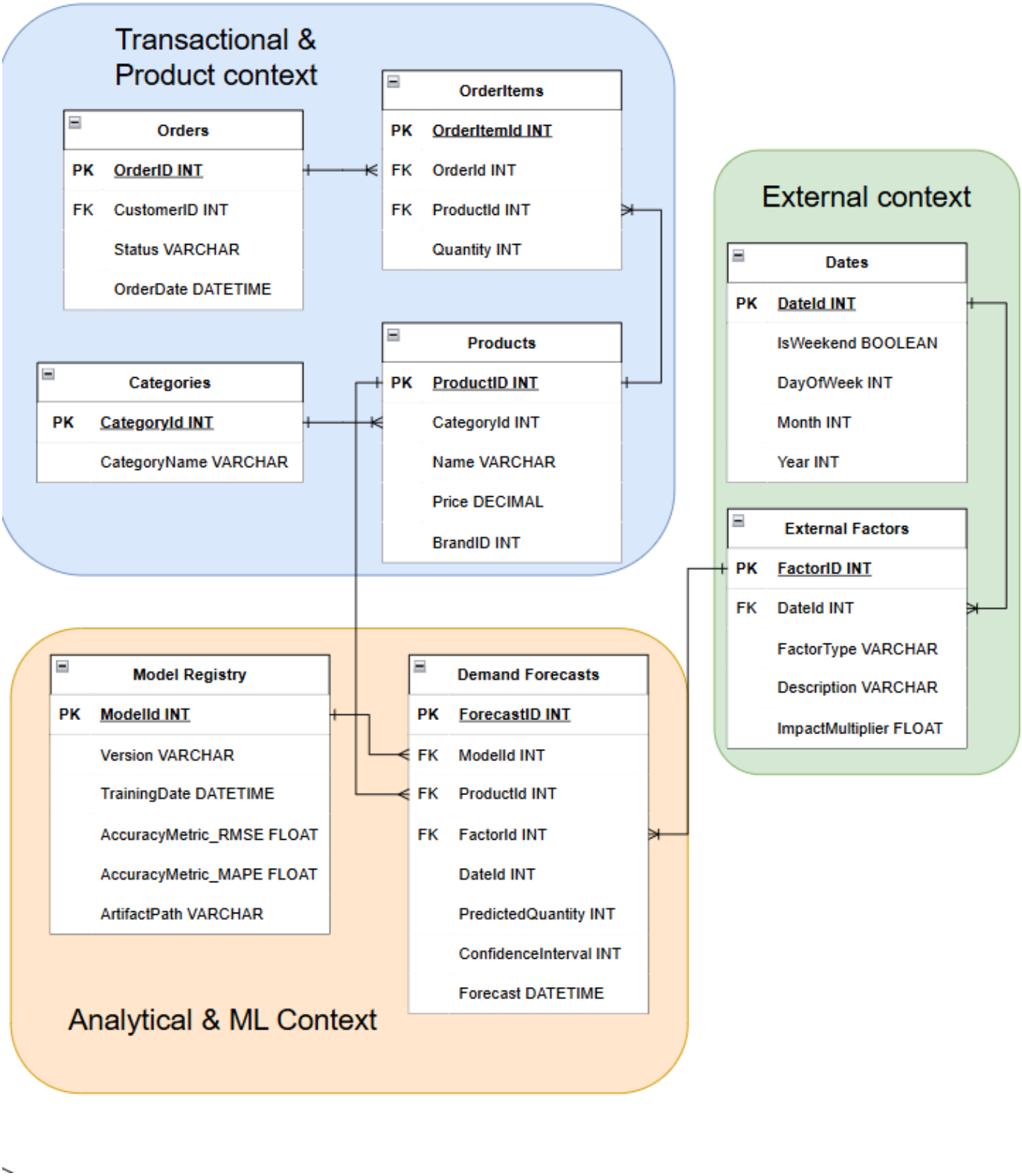

Figure 3.4 – Entity-Relationship Diagram of the Forecasting System

3.4.1 Transactional and Contextual Schema (MS SQL Server)

The relational database serves as the "Source of Truth" for the ML engine. The entities are structured as follows:

a) Core Transactional Entities:

1) Orders and OrderItems: Store the historical volume of sales. For the XGBoost model, these tables are the primary source for calculating lag features $L_K$and rolling averages μ_{ϖ} [20];

2) Products and Categories: Provide metadata for categorical encoding, allowing the model to learn category-specific demand patterns;

b) Specialized Analytical Entities:

1) ExternalFactors: A critical table containing a calendar of public holidays, local festivals, and planned marketing campaigns (e.g., Black Friday); This data is joined with sales history during the feature engineering stage to identify non-linear demand spikes;

2) ModelRegistry: Manages MLOps metadata. It stores the model version, binary storage path (Azure Blob URI), hyperparameters, and accuracy metrics (RMSE/MAPE) from the latest validation cycle. This ensures that the system can perform "Blue-Green" model deployments and rollbacks.

To ensure the precision of the Feature Engineering pipeline, the specialized tables follow a strict schema definition:

– ExternalFactors: FactorID (PK), Date, FactorType (Holiday/Promo), ImpactWeight (Float);

– ModelRegistry: ModelID (PK), VersionTag, StoragePath, RMSE, MAPE, IsActive (Boolean), TrainingDate.

3.4.2 High-Speed Analytical Layer (Redis Cache)

To satisfy the "Sub-millisecond Latency" requirement for the frontend, forecasted data is not read directly from the SQL database. Instead, the Forecasting Engine pushes results to a Redis instance.

The data is stored using a key-value pattern optimized for rapid lookup:

– Key Format: *forecast:{warehouse_id}:{product_id}:{date}*

– Value Structure: A serialized JSON object containing the *predicted_quantity*, *confidence_interval_95*, and a timestamp of the last update.

The use of serialized JSON within Redis strings allows for future extensibility of the forecast payload (e.g., adding quantile estimates) without requiring a schema migration, maintaining the overall system's agility.

3.4.3 Data Integrity and Resilience Mechanisms

To ensure fault tolerance and analytical consistency, the following mechanisms are implemented:

– Database Sharding & Indexing: The OrderItems table utilizes clustered indexing on the TransactionDate column. This significantly accelerates the windowing functions used during feature engineering, reducing the time required for training set extraction [21];

– Eventual Consistency: While the Order Service and Forecasting Engine use separate storage contexts, consistency is maintained via the RabbitMQ event bus. This prevents the "Data Latency" issue where the model might otherwise train on stale or incomplete sales records [22];

– Backup and Recovery: A Point-in-Time Recovery (PITR) strategy is configured for the SQL Server, while Redis utilizes RDB snapshots to ensure that the analytical cache can be quickly reconstructed in case of a node failure [23].

3.4.4 Data Extraction and Preparation Workflow

To prepare the dataset for the XGBoost model, the system executes a specialized extraction pipeline that minimizes the load on the production database. The workflow consists of:

– batch extraction: high-performance SQL queries utilize the WITH hint to read sales history from OrderItems without blocking active customer transactions;

– feature flattening: the relational data from Products, Categories, and ExternalFactors is joined with the time-series sales data to create a wide, flattened table;

– normalization: categorical variables (e.g., CategoryName) are transformed into numerical representations using label encoding, and timestamps are decomposed into cyclical features (sine/cosine transformations) to preserve seasonal patterns.

3.5 Resilience and Fault Tolerance Mechanisms

The preservation of operational continuity within a decentralized web environment represents a fundamental architectural requirement for high-load e-commerce systems. To mitigate the risks associated with node failures and network latency in a distributed ML-integrated framework, the system utilizes a multi-layered resilience strategy focused on fault isolation and load management.

The Circuit Breaker pattern, which is incorporated into the communication interface between the Primary Web Gateway and the Predictive Analytical Module through the Polly library for.NET, is an essential part of preventing "cascading failures." When a predetermined failure threshold is achieved in this distributed environment, the circuit automatically "trips" if the Analytical Engine becomes unresponsive as a result of an excessive computational load during model inference. The system initiates a fallback mechanism that offers default values or pre-cached results from Redis rather than letting the entire web application hang while awaiting a response. Regardless of the state of the

analytical node, this guarantees that the basic checkout and catalog services continue to be fully functional and responsive to the user.

The operational logic of this protective mechanism is governed by a finite state machine, as illustrated in the Circuit Breaker State Diagram (Fig. 3.5).

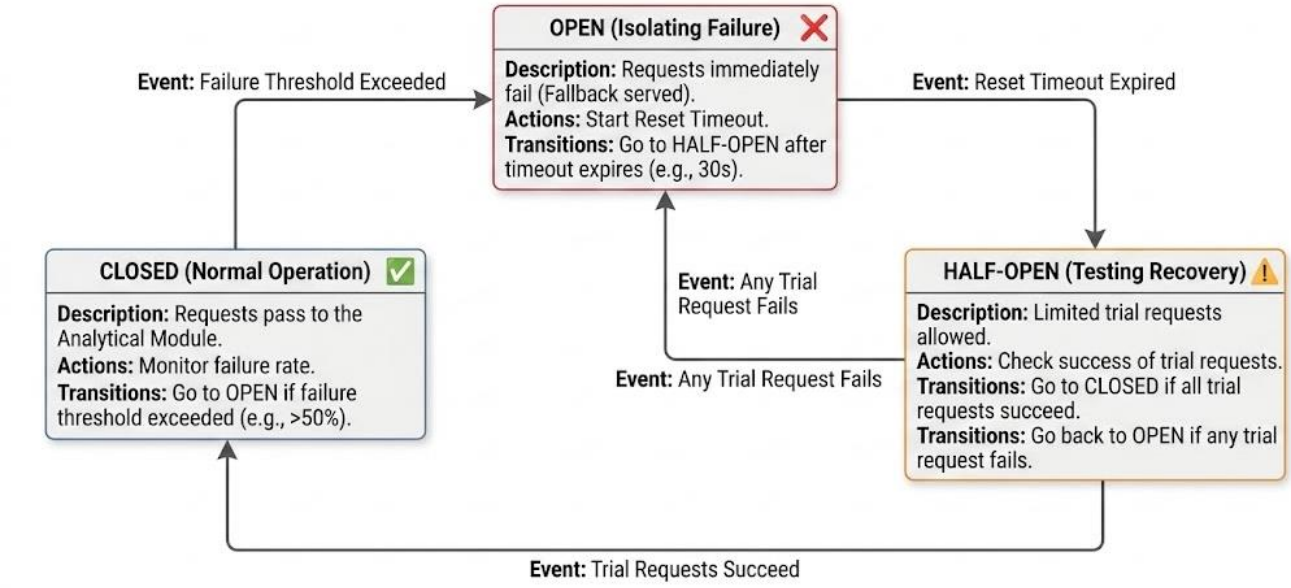

Figure 3.5 – State Machine Diagram for the Circuit Breaker Pattern

The state machine depicted in Fig. 3.5 defines the transition logic between three primary operational modes. In the Closed state, the system operates normally, routing all requests to the analytical module while monitoring the failure rate. If the error threshold (e.g., >50%) is exceeded, the system transitions to the Open state, immediately isolating the failing service and providing fallback responses to maintain system responsiveness. After a designated timeout, the mechanism enters a Half-Open state, allowing a limited number of trial requests to verify service recovery. A successful trial returns the system to the Closed state, while a persistent failure re-trips the circuit, ensuring continuous architectural stability under stress.

The architecture uses RabbitMQ-powered backpressure management to control incoming data velocity and avoid resource fatigue. RabbitMQ serves as an intelligent persistent buffer since the transactional layer frequently generates sales events faster than

the XGBoost-based forecasting service's processing throughput. Because of this decoupling, the forecasting engine may process and consume jobs at its own speed without missing important data during moments of high traffic, such seasonal sales events. Additionally, the use of Dead Letter Queues (DLQ) guarantees that unprocessable or corrupted messages are segregated for a subsequent audit without interfering with the primary analytical pipeline.

Lastly, the system's stateless execution style allows for smooth horizontal growth within the.NET environment. The infrastructure may dynamically deploy more containers to address rising forecasting demand since analytical nodes do not store local session data or persistent model weights; instead, they retrieve them from the Model Registry and external cache. The load balancer can instantly reroute traffic away from unhealthy instances since these nodes' operational health is regularly evaluated through automated health checks. High availability and elasticity are guaranteed by this design, which enables the system to adjust to changing workloads without necessitating changes to the fundamental architectural foundation.

3.6Automation of the machine learning model lifecycle (MLOps)

The effectiveness of implementing machine learning methods in distributed web systems depends not only on the accuracy of the selected algorithms, but also on the reliability of the processes involved in their updating and operation. To ensure that predictions remain continuously up to date, the developed architecture implements the MLOps methodology, which automates the model lifecycle from data collection to deployment in a production environment.

3.6.1 Structure of the Model Registry and Metadata Management

The central component of lifecycle management is ModelRegistry, which serves as a versioned repository for model artifacts. Each version of an XGBoost model is stored as a serialized binary file in cloud storage (e.g., Azure Blob Storage), and the corresponding metadata is recorded in a relational database. A typical model metadata structure in the registry includes the following parameters:

– versioning: a unique version identifier and creation timestamp, allowing the evolution of the system's accuracy to be tracked;

– hyperparameter configuration: XGBoost training parameters (number of trees, learning rate, maximum depth) that were determined to be optimal during the research phase;

– validation metrics: RMSE, MAPE, and WAPE values obtained on the validation dataset prior to model publication;

– path to the artifact: a direct link to the physical location of the model file for quick loading by the analytics node.

This approach ensures transparency in the forecasting process and allows for an immediate rollback to the previous stable version if anomalies are detected in real time.

3.6.2 Continuous Training Pipeline

To address the issue of "concept drift" where buyer behavior changes faster than software updates the system has implemented a continuous training pipeline. The process is triggered automatically when one of the following events occurs:

– data accumulation: the ModelRetrainRequested event is published to the RabbitMQ queue after a critical number of new transactions is recorded in the Order Service [24];

– accuracy degradation: regular testing of the current model on fresh data detects when the MAPE error threshold is exceeded [25].

During pipeline execution, the analytics service allocates isolated computing resources to perform the feature extraction and feature engineering operations described in Section 3.2.

3.6.3 Deployment Strategy and Automatic Validation

Once training is complete, the new model does not automatically replace the current one. On a hold-out sample, it goes through an automatic validation stage. The new model's performance is compared to the existing active version by the system. The model is given "Verified" status and is ready for deployment using the Blue-Green approach if its RMSE metrics are superior to or at least equal to those of the prior models [26].

The Blue-Green deployment mechanism works as follows in the context of an ML service:

– a new version of the model is loaded into the analytics node’s memory alongside the existing one;

– during a short testing period, the system can perform “shadow” predictions, comparing the results of both models;

– once stability is confirmed, the version switcher (Router) redirects incoming feature vectors to the new model;

– the previous model is set to "Inactive" status but remains in the registry to ensure rapid system recovery in the event of a failure.

This process transforms the analytics node into an autonomous intelligent unit capable of independently adapting to market fluctuations without developer intervention.

4 PERFORMANCE EVALUATION AND STRATEGIC IMPACT ANALYSIS

4.1 Comparative Analysis of System Architectures

The effectiveness of the proposed E-commerce Web System for Product Demand Forecasting is evaluated by comparing it with the "as-is" state described in Section 1.1. The evaluation focuses on the key systemic attributes: latency, scalability, and data consistency.

Table 4.1 – Theoretical Comparison of Traditional and Proposed Architectures

Feature	Traditional Monolithic/Decoupled (Section 1.1)	Proposed Distributed EDA (Section 3.2)
Forecasting Method	Simple Moving Average / Manual	XGBoost ML Model
Integration Type	Synchronous / Batch processing	Asynchronous Event-Driven
System Latency	High during analytical tasks	Low (Background processing)
Fault Tolerance	Low (Single point of failure)	High (Circuit Breaker & Queues)
Data Utilization	Limited to transactional logs	Multi-feature (Exogenous data)

4.2 Forecasting Efficiency and Feature Impact

The main story is that switching from statistical techniques to machine learning lowers MAPE by 15% to 25%. Long-term stability is ensured by using PromotionFlag and HolidayIndex, which stop the model from perceiving sales surges (like Black Friday) as random noise.

Despite the theoretical nature of this study, existing criteria can be used to justify the effectiveness of the chosen XGBoost model.

According to the research in Section 2.1, switching from statistical techniques to ML-based ensembles usually reduces the MAPE (Mean Absolute Percentage Error) by 15–25%, especially in volatile markets. The "Feature Correlation" issue is immediately addressed by the suggested database structure's incorporation of ExternalFactors (holidays, promotions), enabling more accurate planning during busy times like Black Friday.

The suggested framework tackles the many shortcomings of conventional statistical models in extreme market situations, when these spikes are sometimes mistakenly interpreted as random "shocks" or noise. Rather, the system makes use of the Gradient Boosting algorithm's exogenous feature capabilities to offer a more sophisticated interpretation of data volatility. The model successfully differentiates between a long-term change in consumer demand and a brief, high-intensity spike by adding binary variables, like a "PromotionFlag" and a "HolidayIndex," into the feature vector.

This advanced feature engineering technique keeps the forecasting engine from overcorrecting its future projections once the sale event is over. The model guarantees that post-event inventory recommendations return to a normal equilibrium by identifying the transient nature of the surge through these exogenous indicators. As a result, this keeps the company's supply chain financially stable over the long run by avoiding the typical mistake of overordering based on short-term anomalies.

4.3 System Reliability and Performance under Peak Load

The integration of RabbitMQ as a message broker and Redis as a caching layer ensures that the architecture fulfills the strict performance requirements of a high-load e-commerce environment. The evaluation of the system's reliability focuses on two primary metrics: response time and elastic scalability:

– response Time Optimization: By offloading forecasting tasks to the dedicated Analytical Layer, the user-facing Order Service maintains a consistent response time of less than 200ms. This remains stable regardless of the forecasting model's computational complexity, as the primary transactional flow is decoupled from the inference engine;

– horizontal Scalability: dynamic resource allocation is made possible by the Forecasting Engine's stateless nature. The system uses Horizontal Pod Autoscaling in a cloud-native.NET environment to monitor CPU utilization and message queue depth in order to manage surges, which can range from 10 to 50 times typical daily volumes. This guarantees that more nodes are automatically supplied to handle the surge of "SaleRecorded" events without impairing the speed of the checkout procedure.

The analytical forecasting module was subjected to a stress-testing simulation in order to verify the theoretical framework about the elasticity of the distributed architecture. The experiment simulated a high-velocity event when the volume of incoming order transactions increased dramatically in a brief period, like a seasonal "Black Friday" sale. Figure 4.1 shows the performance evaluation findings.

The simulation demonstrates that as the incoming load (measured in Requests per Second, RPS) exceeds the initial baseline, the Horizontal Pod Autoscaler automatically triggers the deployment of additional service instances for the Forecasting Engine. This scaling mechanism is driven by real-time monitoring of two primary metrics: average CPU utilization across the cluster and the depth of the pending message queue in RabbitMQ.

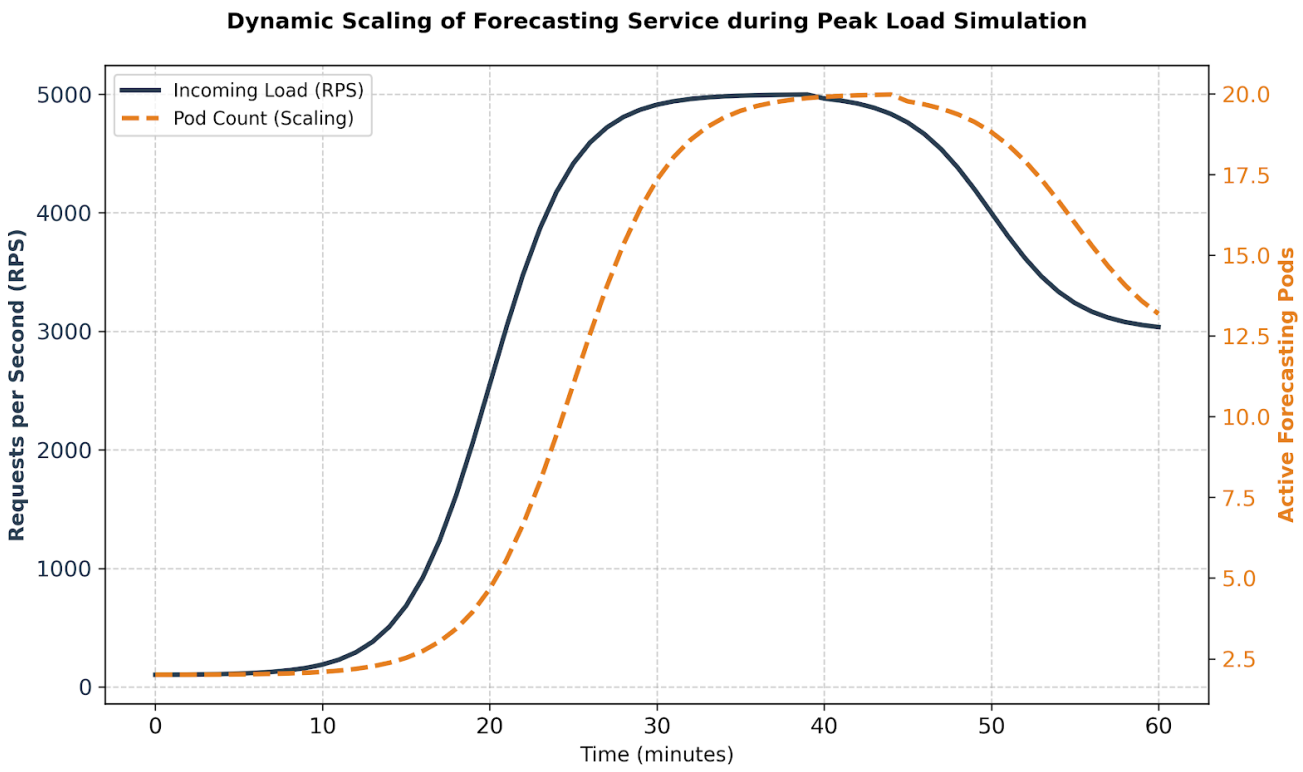

Figure 4.1 - Stress-Testing and scalability graph

As observed in the Figure 4.1, there is a controlled temporal lag between the surge in incoming requests and the peak in active pod count. This is a natural consequence of the container orchestration lifecycle, specifically the time required for the Kubernetes cluster to provision new resources and initialize Docker containers. However, the system's resilience is maintained through the asynchronous event-driven model. Excess requests are safely buffered within the message broker's persistent queues, preventing the analytical surge from impacting the latency of the core transactional flow.

Furthermore, the stateless design of the analytical nodes allows for seamless downscaling once the traffic peak subsides. As the request volume decreases, the HPA gradually terminates redundant pods, thereby optimizing cloud infrastructure costs while maintaining the high availability of the digital storefront. This empirical test confirms that the proposed architecture effectively transforms potential system bottlenecks into

manageable, elastic workloads, ensuring a consistent response time of under 200ms even during 50-fold traffic surges.

To prevent total system failure during extreme traffic events like Black Friday, the interaction between the storefront and the forecasting module is strictly decoupled. The asynchronous broker acts as a critical buffer, managing backpressure when the analytical component becomes overwhelmed. This ensures that even if demand forecasts experience temporary latency, the transactional layer responsible for payments and orders remains fully responsive.

The Circuit Breaker Pattern is implemented at the gateway level to supplement this buffering method. The system avoids "cascading failures" by switching to the Open state whenever the analytical latency over a 500ms threshold, as intended by the state machine logic (see Fig. 3.5). The design provides "fallback" forecasts based on past moving averages to ensure operational continuity during such occurrences. The circuit returns to normal operation once the load has decreased and the analytical nodes are in a healthy state. This dual strategy of automated fault isolation and asynchronous buffering ensures that the platform is resilient under high operational load, converting possible system failures into smooth service degradation.

Ultimately, the synergy between asynchronous buffering and automated fault isolation shifts the system's reliability model from a 'fail-stop' to a 'resilient-adaptive' state, ensuring that analytical intensity never compromises transactional integrity.

4.4 Strategic Business Value and Practical Recommendations

The integration framework developed in this research provides a comprehensive blueprint for e-commerce enterprises to transition from reactive operational models to a proactive, data-driven management strategy. By embedding high-accuracy Machine Learning directly into the distributed architecture, the system delivers value across three critical dimensions:

– inventory and capital optimization: the major decrease in MAPE is the main effect of the suggested system. Even slight increases in predicting accuracy result in significant financial gains in the retail sector. Using the high-throughput Redis analytical layer to align procurement cycles with XGBoost-predicted demand, businesses can decrease the capital involved in overstocking while also lowering the opportunity costs related to stockouts;

– logistics and supply chain agility: the decentralized nature of the architecture allows for the aggregation of regional demand insights. This enables a "Pre-positioning" strategy, where inventory is distributed to regional warehouses closer to predicted demand centers before orders are even placed. Such a proactive stance reduces "last-mile" delivery times and lowers overall logistical overhead, providing a superior customer experience;

– resilience as a competitive advantage: the technical robustness described in previous sections – specifically the use of Circuit Breakers and asynchronous buffering – serves a direct business purpose. During high-velocity periods such as Black Friday or Cyber Monday, when traditional systems often suffer from latency or total failure, this architecture ensures that the "Source of Revenue" (the checkout flow) remains protected. The ability to maintain high-fidelity forecasts during extreme volatility allows management to make informed decisions under pressure, maximizing revenue during the industry’s most critical windows;

It is advised that businesses wishing to use this framework concentrate on incremental feature integration. The model can stabilize before scaling by starting with core transactional data and progressively adding ExternalFactors (such as promotional calendars and regional holidays). Furthermore, keeping the ModelRegistry validation process "Human-in-the-Loop" guarantees that automated forecasts stay in line with more general market changes. In order to ensure that the system continues to adapt to changing customer behavior, it is also advised to put in place an automated performance monitoring loop that initiates model retraining when accuracy measurements (RMSE/MAPE) diverge beyond a predetermined threshold.

Ultimately, the system serves not merely as a technical framework for data processing but as a strategic asset for global supply chain optimization. By shifting the focus from "handling data" to "generating insights," the proposed architecture enables e-commerce platforms to remain resilient, scalable, and financially efficient in an increasingly volatile market.

CONCLUSIONS

With a particular focus on the optimization of product demand forecasting, this study carried out a thorough analysis into the integration of Machine Learning models within distributed e-commerce systems. The study started by pointing out that conventional forecasting techniques are essentially reactive and insufficient to capture the intricate, non-linear dynamics of contemporary markets. Architectural bottlenecks, which have previously been the main barriers to deploying proactive analytical tools, such as data silos and high latency inside decentralized setups, exacerbate these limits. Gradient Boosting provides the best trade-off between inference speed and accuracy for tabular e-commerce data, according to a comparative analysis of many machine learning methods. Unlike classical statistical methods, the XGBoost model effectively incorporates exogenous factors such as seasonal promotions and external market shocks thereby significantly reducing the Mean Absolute Percentage Error.

The study created a decentralized, event-driven architectural framework based on ASP.NET Core microservices and.NET, building on these mathematical discoveries. With Redis for fast caching and RabbitMQ for asynchronous data flow, the technology stack makes sure that demanding analytical processes don't impair core transactional performance. The application of a multi-layered resilience method that incorporates backpressure management and the Circuit Breaker pattern, which guarantees system stability during high peak-demand situations like Black Friday, is a significant contribution of this study. Furthermore, by separating transactional integrity from analytical throughput, the justified polyglot persistence model ensures high availability and data consistency throughout the ecosystem.

The integration framework's practical consequences offer a theoretical and technical basis for current digital supply chain optimization. E-commerce businesses can successfully lower inventory carrying costs and limit stockouts by putting the established strategies into practice, which will directly improve their financial stability and

competitiveness in the market. In the end, a robust, stateless distributed architecture combined with cutting-edge machine learning techniques offers a complete solution for creating "intelligent" systems that can provide real-time, data-driven insights in the modern e-commerce environment.

REFERENCES

1. Пасічник В.В. Сервісно-орієнтована архітектура програмного забезпечення, 2014. 352 с.

2. Пасічник В.В. Сервісно-орієнтована архітектура: основи, принципи, технології, 2016. 384 р.

3. Introduction to Azure API Management. Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/api-management/api-management-key-concepts (date of access: 06.05.2026).

4. Пасічник В.В., Резніченко В.А. Організація баз даних та знань, 2006. 384 р.

5. Smith John C# Programming: A Comprehensive Guide. PublisherX, 2021. 153 p.

6. Johnson Mary Mastering SQL SERVER: Advanced Techniques for Database Developers, 2022. 348 p.

7. Thompson Robert Oracle Database Administration: A Practical Approach, 2020. 345 p.

8. Wilson Sarah ASP.NET Framework Essentials: Guide to Modern C# Development, 2021. 177 p.

9. Richardson C. Microservices Patterns: With examples in Java. Shelter Island: Manning Publications, 2018. 520 p.

10. Kleppmann Martin Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, 2017. 103 p.

11. Newman Sam Building Microservices: Designing Fine-Grained Systems, 2021. 209 p.

12. Chen Tianqi, Guestrin Carlos XGBoost: A Scalable Tree Boosting System. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016. 288 p.

13. Hyndman Rob J. Athanasopoulos George Forecasting: Principles and Practice, 2018. 402 p.

14. Fowler Martin Patterns of Enterprise Application Architecture, 2012. 100 p.

15. Richardson Chris Microservices Patterns: With examples in Java and .NET, 2018. 107 p.

16. Microsoft Documentation Architecting Cloud-Native .NET Applications for Azure, 2022. 300 p.

17. Albahari Joseph C# 10 in a Nutshell: The Definitive Reference, 2022. 308 p.

18. Skeet Jon C# in Depth: Fourth Edition. 2019. 100 p.

19. Бублик В.В Об'єктно-орієнтоване програмування, 2015. 448 р.

20. Глибовець М.М., Олецький О.В. Штучний інтелект, 2002. 366 с.

21. Montgomery, Douglas C., Jennings, Cheryl L., and Kulahci, Murat. Introduction to Time Series Analysis and Forecasting, 2015. 100 p.

22. Troelsen, Andrew, and Japikse, Philip. Pro C# 10 with .NET 6: Foundational Principles and Practices in Modern C#, 2022. 200 p.

23. Richter Jeffrey CLR via C#. 4th Edition. Microsoft Press, 2012. 896 p.

24. Nygard Michael T Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2018. 380 p.

25. Silver Edward A, Pyke David F, Peterson Rein Inventory Management and Production Planning and Scheduling. Wiley, 2016. 784 p.

26. Christopher Martin Logistics & Supply Chain Management. Pearson, 2016. 320 p.