Universidad Distrital Francisco José de Caldas

System Engineering Program
School of Engineering

DATABASES II - FINAL DELIVERY TECHNICAL REPORT

Andruew Steven Zabala Serrano - 20211020071.
Ruben David Montoya Arredondo - 20211020055.

Hemerson Julian Ballen Triana - 20211020084

Supervisor: Eng. Carlos Andrés Sierra, M.Sc

December 11, 2025

Abstract

This is a project report template, including instructions on how to write a report. It also has some useful examples to use LaTeX. Do read this template carefully. The number of chapters and their titles may vary depending on the type of project and personal preference. Section titles in this template are illustrative should be updated accordingly. For example, sections named "A section..." and "Example of ..." should be updated. The number of sections in each chapter may also vary. This template may or may not suit your project. Discuss the structure of your report with your supervisor.

Guidance on abstract writing: An abstract is a summary of a report in a single paragraph up to a maximum of 250 words. An abstract should be self-contained, and it should not refer to sections, figures, tables, equations, or references. An abstract typically consists of sentences describing the following four parts: (1) introduction (background and purpose of the project), (2) methods, (3) results and analysis, and (4) conclusions. The distribution of these four parts of the abstract should reflect the relative proportion of these parts in the report itself. An abstract starts with a few sentences describing the project's general field, comprehensive background and context, the main purpose of the project; and the problem statement. A few sentences describe the methods, experiments, and implementation of the project. A few sentences describe the main results achieved and their significance. The final part of the abstract describes the conclusions and the implications of the results to the relevant field.

Keywords: a maximum of five keywords/keyphrase separated by commas

Report's total word count: Following the abstract, the word count must be stated. We expect at least 10,000 words in length and at most 15,000 words (starting from Chapter 1 and finishing at the end of the conclusions chapter, excluding references, appendices, abstract, text in figures, tables, listings, and captions), about 40 - 50 pages.

Program code should be uploaded to gitlab, and the gitlab link should be included alongside the word count, following the abstract.

You must submit your dissertation report (preferred in a PDF file) via the "Turnitin assignment" in Blackboard Learn by the deadline. If a student has resits from the taught modules, the dissertation deadline will be extended for 3 weeks from the original dissertation deadline.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

E-commerce platforms, while diverse in implementation and scale, share foundational architectural principles that determine their scalability, maintainability, and user experience. As digital commerce has matured, these principles have shaped the technological landscape, informing how modern platforms manage data, integrate external services, and deliver interactive user interfaces. Before examining the design and implementation of the BogoGo platform, it is essential to understand the architectural trends and technological decisions that underpin contemporary e-commerce solutions.

1.1 Background

E-commerce has evolved into a core component of the global digital economy, enabling businesses to broaden their reach and offering consumers convenient access to products and services through online channels. This evolution has been supported by advances in web technologies, cloud services, and database systems, which have allowed platforms to transition from static storefronts to dynamic, data-driven ecosystems [1, 2, 3].

Modern platforms typically adopt modular, layered architectures that separate the presentation, application, and data layers. This structure facilitates concurrent development, improves system reliability, and simplifies integration with external services such as payment providers and content delivery networks. In recent years, Backend-as-a-Service (BaaS) platforms—such as Supabase—have accelerated the adoption of lightweight architectures by combining database management, authentication, storage, and serverless backend execution under a unified framework. These services reduce operational overhead while providing production-ready infrastructure tailored for scalable applications.

At the data layer, relational database management systems remain essential for ensuring transactional integrity and consistent performance in high-demand e-commerce environments. PostgreSQL, the database engine behind Supabase, is widely recognized for its ACID compliance, indexing capabilities, and support for structured and semi-structured data. Comparative studies show PostgreSQL outperforming MySQL in write-heavy workloads and complex relational queries, making it a strong choice for systems that handle inventories, user accounts, and transactional records [4, 5, 6]. In addition, PostgreSQL offers capabilities such as materialized views, which support the development of efficient business intelligence dashboards by precomputing analytical queries.

1.2 Problem Statement

Despite the growth of online commerce in Colombia, many small fashion brands, local vendors, and independent designers in Bogotá struggle to establish a competitive digital presence. Existing global platforms often impose high fees, lack integration with local delivery services, or fail to highlight regional identity. As a result, consumers face difficulty accessing locally produced fashion items, while vendors lack affordable and adaptable digital tools for promoting and managing their products.

Another significant challenge is the absence of accessible analytics and business intelligence capabilities tailored to small and medium-sized vendors. Without tools that provide visibility into sales trends, customer behavior, and inventory performance, local sellers are unable to make data-driven decisions that could strengthen their competitiveness. This technological gap limits their participation in the city's expanding e-commerce ecosystem and restricts opportunities for sustainable growth.

These challenges motivate the development of a lightweight, scalable, and locally adapted e-commerce solution that connects Bogotá's designers, vendors, and consumers through an accessible, high-performing platform.

1.3 Objectives

This project proposes the development of **BogoGo**, an e-commerce platform designed to support local fashion entrepreneurs in Bogotá by leveraging modern web technologies and cloud-based backend infrastructure. The primary objective is to design and implement a modular three-layer architecture that ensures scalability, maintainability, and ease of deployment while addressing the needs of small and medium-sized vendors.

The specific objectives of the project are as follows:

- To develop a responsive, user-friendly interface using **React** and **Vite**, enabling seamless product browsing, cart management, and intuitive user interactions.
- To manage authentication, user roles, database operations, and multimedia storage through the built-in services provided by **Supabase**, reducing backend maintenance overhead.
- To employ **Supabase PostgreSQL** as the primary relational database and **Supabase Storage** for managing product images and other multimedia assets with high availability.
- To integrate a secure payment workflow using a dedicated **Supabase Edge Function** that communicates with the **Mercado Pago API**, ensuring that sensitive payment logic is executed server-side.
- To build data-driven insights for vendors by implementing **materialized views** within PostgreSQL, supporting analytics dashboards for sales monitoring and performance evaluation.

Through the combination of React, Supabase, PostgreSQL, and cloud-native tooling, BogoGo aims to provide an accessible and cost-effective e-commerce platform tailored to the realities of Bogotá's fashion industry. The platform seeks to enhance digital participation for local vendors while promoting sustainable, data-driven growth within the sector.

Chapter 2

Literature Review

2.1 E-commerce Architectures and Layered Models

Contemporary technical literature consistently emphasizes the critical role of layered architectures in designing scalable e-commerce platforms. Traditional 3-tier models—separating the presentation layer, application logic, and data persistence—remain the standard for ensuring modularity, maintainability, and separation of concerns [1, 2]. However, modern web-based retail applications have extended this model by adopting cloud-native services and client-side rendering strategies.

Recent research highlights a transition toward lightweight, "thick-client" architectures where the frontend assumes a prominent role, relying on Backend-as-a-Service (BaaS) platforms to manage authentication, database interaction, and file storage. Platforms such as Supabase and Firebase exemplify this trend by providing auto-generated APIs for CRUD operations, identity management, and real-time synchronization. This approach significantly reduces backend maintenance overhead, enabling rapid prototyping and deployment without compromising security standards.

While alternative paradigms such as microservices and monolithic systems continue to be evaluated, the adoption of BaaS has become increasingly prevalent for small-to-medium-scale e-commerce systems—particularly in academic and startup contexts—due to its streamlined operational model and integrated tooling.

2.2 Microservices, Serverless Models, and BaaS Approaches

Microservices architectures offer distinct advantages regarding fault isolation and independent scaling. However, literature warns that they introduce substantial operational complexity related to orchestration, distributed tracing, and inter-service communication. Consequently, microservices are often deemed more appropriate for mature, enterprise-grade platforms with high concurrency demands and dedicated DevOps teams [7].

Conversely, serverless paradigms—specifically Function-as-a-Service (FaaS)—provide a pragmatic alternative for isolating specific backend logic without the overhead of full container orchestration. Serverless functions are triggered on demand, scale automatically, and require minimal provisioning. For the BogoGo implementation, Supabase Edge Functions fulfill this precise role: they execute sensitive workflows, such as payment processing with Mercado Pago, ensuring that API secrets remain secure and hidden from the client-side application.

Additionally, the literature supports event-driven models for asynchronous tasks like analytics. Although BogoGo prioritizes architectural simplicity, its Business Intelligence (BI)

component—built upon materialized views—aligns with patterns found in analytical systems where data is periodically refreshed rather than streamed continuously.

2.3 Database Selection: PostgreSQL, BaaS Storage, and Schema Design

Comparative studies repeatedly demonstrate PostgreSQL's superior performance under transactional workloads, particularly for operations involving complex indexing, high write frequencies, and mixed analytical queries [4]. PostgreSQL's ACID compliance, extensibility, and native support for JSONB fields provide a competitive advantage over other relational systems, making it ideal for e-commerce catalogs that must manage structured inventory data alongside semi-structured product metadata.

In BaaS ecosystems like Supabase, PostgreSQL is augmented with features such as Row-Level Security (RLS) policies and integrated object storage. Research on cloud-native e-commerce suggests a consistent best practice: retaining relational databases for structured data (orders, inventory, payments) while offloading multimedia assets to distributed object storage to optimize delivery performance [6].

Furthermore, schema recommendations for online retail typically include core entities such as User, Product, Order, Order_Item, and Payment. BogoGo's data model strictly adheres to these industry standards, ensuring transactional integrity and auditability.

2.4 Project Description in the Context of Literature

BogoGo adopts a set of architectural decisions that directly align with the trends and recommendations identified in the reviewed literature:

- **Layered Architecture with BaaS Support:** BogoGo employs a three-layer conceptual model: a presentation layer (React/Vite), an application layer handling client-driven logic and serverless functions, and a data layer managed by Supabase (PostgreSQL & Storage). This reflects modern lightweight architectures for agile e-commerce development [1].
- **Relational Integrity (PostgreSQL):** The selection of PostgreSQL is supported by research confirming its suitability for hybrid workloads—combining transactional reliability with the analytical capabilities required for the vendor dashboard [4, 5].
- **Decoupled Media Storage:** The use of Supabase Storage for product images follows the documented pattern of separating binary assets from the relational database to reduce latency and database bloat [6].
- **Serverless Payment Integration:** Implementing a Supabase Edge Function for Mercado Pago communication aligns with security best practices that recommend decoupling sensitive third-party interactions from the client code.
- **Embedded Business Intelligence:** The dashboard, powered by PostgreSQL materialized views, mirrors efficient BI strategies for SMBs (Small and Medium-sized Businesses), avoiding the complexity of external data warehouses.

Overall, the architecture of BogoGo represents a pragmatic adaptation of enterprise industry patterns into a lightweight, cost-effective solution suitable for the local market.

2.5 Relevance of the Literature Review for the Proposed Application

2.5.1 Performance and User Experience

Web performance literature emphasizes the correlation between low page load times (under 2.5 seconds) and conversion rates. BogoGo addresses this through a Vite-optimized frontend and the low-latency APIs provided by the BaaS infrastructure.

2.5.2 Transactional Consistency

Since order processing and inventory management require strict data consistency, the choice of a relational engine (PostgreSQL)—over NoSQL alternatives—provides the necessary ACID guarantees validated by comparative database studies.

2.5.3 Analytics for Decision Making

Research on small-business e-commerce highlights the value of embedded analytics. By leveraging materialized views, BogoGo follows patterns for lightweight BI, providing actionable insights to vendors without complex ETL pipelines.

2.5.4 Third-Party Service Integration

Literature supports the use of API-driven abstraction layers. BogoGo's use of Edge Functions to wrap the Mercado Pago API is a direct application of these principles, enhancing system security and maintainability.

2.6 Gap Analysis and Research Contribution

2.6.1 Strengths of Existing Literature

The reviewed body of work provides a robust foundation for architectural decisions regarding layered design, data consistency, and cloud-managed tooling. It offers valid guidance on using BaaS platforms to accelerate the development lifecycle.

2.6.2 Identified Gaps Addressed by BogoGo

Despite the extensive general literature, several gaps exist which BogoGo aims to address:

1. **Local Context Adaptation:** Global e-commerce references rarely account for regional payment behaviors, specific logistics providers, or the digital maturity of the Bogotá market. BogoGo contextualizes technical design within these local realities.
2. **Lightweight BI for Small Vendors:** Many studies focus on enterprise-scale analytics (Big Data); fewer address the needs of micro-vendors requiring simplified, embedded BI tools within the transactional platform.
3. **Cost-Conscious Deployment:** Academic references often assume access to standard cloud budgets. BaaS-supported architectures like BogoGo demonstrate a viable pathway for low-cost, sustainable digital transformation for local commerce.

2.7 Conclusion

The literature review confirms that BogoGo's architecture is well-grounded in contemporary engineering practices, particularly in the domains of BaaS-enabled development, PostgreSQL data management, and secure serverless integrations. BogoGo contributes a locally adapted case study that applies these global patterns to the specific needs of Bogotá's fashion sector, offering empirical insights into the performance and viability of this architectural approach.

Chapter 3

Scope

The scope of this research encompasses the design and partial implementation of BogoGo, a fashion-oriented e-commerce platform tailored to the city of Bogotá. The study focuses on the conceptualization, architectural definition, and prototype development of a three-layer system architecture—presentation, application, and infrastructure.

The system will include core functionalities such as user authentication, product management, order processing, secure payments, and a business intelligence dashboard. It will specifically address local commerce dynamics by supporting multiple user roles (customers, vendors, and administrators) and localized logistics integrations within Bogotá.

This study, however, does not include full-scale deployment, marketing strategies, or long-term business operations. The focus remains on the technical and architectural design, and validation of expected system performance rather than the implementation of complete commercial workflows or market evaluation.

3.0.1 Assumptions

Several assumptions have been established to guide the development and analysis of the BogoGo platform:

- It is assumed that all users (customers, vendors, and administrators) will have access to stable internet connectivity and modern mobile or desktop devices.
- The study assumes the availability of reliable Backend-as-a-Service (BaaS) through Supabase services, ensuring continuous system operation and data security.
- It is assumed that local logistics providers (e.g., Servientrega, InterRapidísimo) and payment gateways (e.g., Mercado Pago) offer compatible APIs for seamless integration.
- The platform's user base is assumed to be primarily located within Bogotá, which influences design decisions regarding language, currency, and delivery range.
- The data used for testing will simulate realistic operational scenarios but will not represent actual commercial data from vendors or customers.

3.0.2 Limitations

This research faces several limitations that may influence its results and generalizability:

- The project is currently in the design and prototyping phase; thus, empirical performance metrics and user experience evaluations are based on expected outcomes rather than full-scale deployment tests.

- Time and resource constraints restrict the implementation to a proof-of-concept level, preventing the integration of all planned modules such as recommendation systems or advanced analytics.
- The study is geographically limited to Bogotá, which may affect the applicability of results to other regions with different logistical or regulatory conditions.
- External factors such as fluctuating API availability, internet latency, or **cloud service pricing models** could impact long-term scalability but are not fully assessed within this phase.

The BogoGo platform leverages the Supabase Free Tier to minimize initial infrastructure costs. Although this tier provides essential services such as a managed PostgreSQL database, authentication, and edge functions, it enforces resource caps to maintain service stability for non-paying users. These limitations, detailed below, primarily affect the system's capacity for stress testing, storage volume, and real-time connection concurrency:

- **Inactivity Pausing:** Projects on the free tier are automatically paused after 7 days of inactivity (no API requests or database connections), requiring manual reactivation via the dashboard.
- **Database Size Limit:** The maximum database size allowed is 500 MB. Exceeding this limit triggers "Read-Only" mode, preventing new transactions (orders/registrations) until data is deleted or the plan is upgraded.
- **Concurrent Connections (Realtime):** The platform limits Realtime connections to 200 concurrent clients. This creates a bottleneck for the proposed stress test of 500 concurrent users, as 300 users would fail to establish a WebSocket connection.
- **Storage Bandwidth:**
  - **Storage Capacity:** Limited to 1 GB for all buckets (product images, user avatars).
  - **Egress (Bandwidth):** The outgoing data transfer limit is 5 GB per month. High-resolution image loading during stress tests could rapidly deplete this quota.
  - **Max File Size:** Individual file uploads are capped at 50 MB.
- **Edge Functions:** Serverless function execution is limited to 500,000 invocations per month, which serves as a constraint for backend logic triggers during high-load scenarios.

Despite these limitations, the research establishes a strong foundation for subsequent development phases and future academic extensions, where the proposed architecture and technologies can be tested under real operational conditions.

Chapter 4

Methodology

The research follows a **design-based methodological approach** focused on the development and evaluation of a lightweight, scalable e-commerce system tailored to the needs of Bogotá's local fashion sector. This methodology integrates software engineering practices, architecture modeling, and empirical validation to determine whether the proposed platform can meet the functional and non-functional requirements identified during analysis.

The methodological process is divided into three phases: (1) **System Design**, which establishes the architectural model and selects the technologies to be used; (2) **Prototype Development**, in which each component of the system is implemented following modular and iterative practices; and (3) **Validation and Testing**, where the correctness, reliability, and expected performance of the prototype are assessed through structured testing procedures. This approach allows continuous refinement of the system and ensures that each architectural layer contributes effectively to the platform's overall performance and maintainability.

4.1 System Architecture and Implementation

The e-commerce platform **BogoGo** is built using a **three-layer architecture** that promotes modularity, scalability, and clear separation of concerns. Each layer fulfills a distinct role in the platform's functionality (Fig. 4.1).

- **Presentation Layer:** Implemented using React with Vite, this layer manages user interaction, interface rendering, state transitions, and navigation. The use of modular components and optimized bundling enhances load times and supports a responsive, fluid user experience. This layer communicates directly with Supabase's client libraries for authentication, database queries, and storage operations.
- **Application Layer:** Instead of a traditional backend service, BogoGo relies primarily on **Supabase's built-in APIs**, which automatically expose secure REST endpoints for database operations. The only custom server-side logic is implemented using a **Supabase Edge Function**, dedicated exclusively to handling Mercado Pago payments and webhook verification. This ensures that API keys and sensitive payment operations are executed securely on the server side.
- **Infrastructure Layer:** Data persistence is managed through **Supabase PostgreSQL**, which stores users, roles, products, orders, inventory, and payment metadata. Multimedia files such as product images are stored in **Supabase Storage**, which offers reliable,

scalable object storage. For analytics, the system uses **materialized views** to precompute aggregated metrics—such as sales performance—which support the BI dashboard without requiring external ETL pipelines.

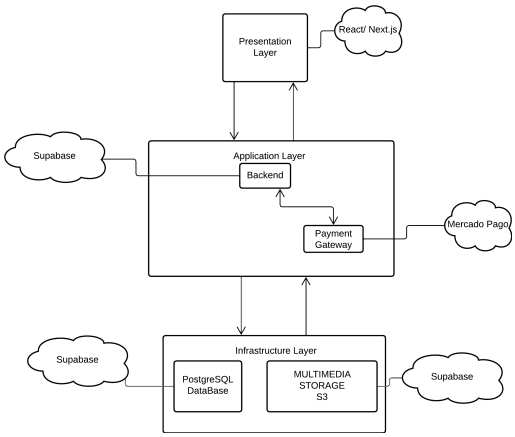

Figure 4.1: High-level architecture of the BogoGo e-commerce platform.

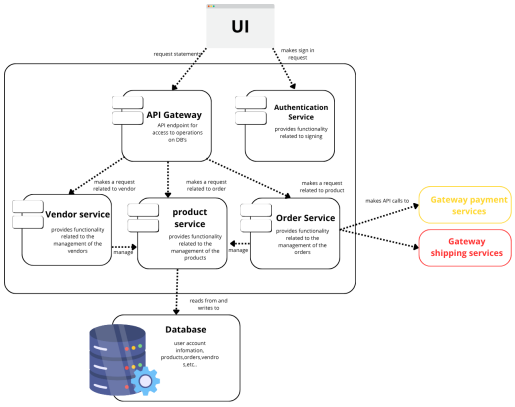

Figure 4.2: High-level architecture dataflow.

In addition to the layered architecture, the platform relies on a well-structured relational schema designed to support transactional consistency, scalability, and integration with Supabase's authentication and storage services. Figure 4.3 presents the Entity–Relationship (ER) model used in BogoGo, which defines core entities such as *User*, *Product*, *Category*, *Order*, *Order_Item*, and *Multimedia*. This schema follows best practices for e-commerce databases, ensuring referential integrity, efficient querying, and extensibility for future modules such as ratings or vendor analytics.

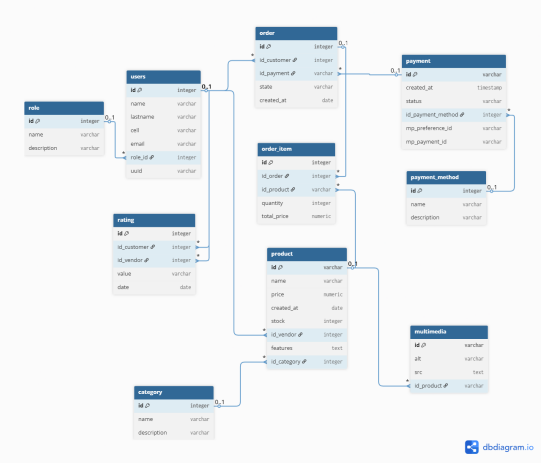

Figure 4.3: Relational database schema of the BogoGo platform, implemented in Supabase PostgreSQL.

This architecture supports modular development and reduces operational overhead, as the platform does not require a dedicated backend server or infrastructure orchestration. Each layer communicates through secure protocols, enabling consistent data flow and a clear separation of responsibilities.

4.2 Data Collection and Simulation

Given that the project is in its prototype stage, data collection relies on **synthetic datasets** representing users, vendors, products, and orders. These datasets simulate the behavior of a small fashion marketplace in Bogotá.

To assess expected system performance, controlled simulations emulate scenarios such as: - browsing product catalogs, - updating cart contents, - completing checkout flows, and - validating payment confirmations.

Workload simulations approximate up to 500 concurrent interactions to explore whether the platform can sustain demand typical of small-to-medium online marketplaces. Metrics collected include API latency, query execution time, and image retrieval performance from Supabase Storage.

4.3 Testing and Validation Procedures

The validation strategy evaluates both functional correctness and expected operational behavior through several testing layers:

- **Unit Testing:** Core components such as product rendering, authentication flows, and cart logic are tested individually within the React environment. Supabase client interactions are validated through mocked responses to ensure reliability of data queries and authentication states.
- **Integration Testing:** Full user workflows—from registration to checkout—are simulated to validate consistent communication between the frontend, Supabase services, and the Mercado Pago Edge Function. Testing confirms that transactions update inventory, order records, and payment status correctly.

- **Acceptance Testing:** Simulated sessions for customers and vendors validate that the implemented functionalities align with user stories. Particular focus is placed on ease of navigation, system responsiveness, and accuracy of purchasing flows.
- **Performance Evaluation:** Database indexing strategies, materialized views, and caching behaviors are analyzed to estimate expected performance under realistic usage patterns. Metrics such as query latency and page rendering times are monitored during simulated load scenarios.

These procedures ensure that both functional and non-functional requirements are satisfied and that the system behaves predictably under increasing load.

4.4 Data Analysis Methods

System behavior during validation is analyzed using descriptive performance metrics. Key indicators include:

- average API response time, - authentication latency, - database query execution time, and - refresh time of materialized views.

These metrics are compared against expected performance thresholds for small-scale e-commerce platforms. Analytical conclusions support recommendations for optimizing indexing strategies, refining component rendering, and improving data retrieval through Supabase Storage when necessary.

4.5 Replication Considerations

To ensure reproducibility, all implementation details—including project structure, environment variables, database schema, and setup instructions—are documented within the repository. The repository includes:

- source code for the frontend and Supabase Edge Function,
- SQL scripts defining the relational schema and materialized views,
- instructions for configuring Supabase authentication and storage buckets,
- integration guidelines for Mercado Pago.

This documentation allows other researchers and developers to replicate the system, adapt it to new geographic contexts, or extend it with additional services or modules.

Chapter 5

Results

This section presents the empirical results obtained from the implementation and evaluation of the BogoGo e-commerce platform. The findings include unit tests of database functions, performance measurements using synthetic workloads, capacity estimation, and system-level integration tests. All experiments were executed on the Supabase PostgreSQL backend under controlled transactional conditions to prevent contamination of production data.

5.1 Functional Validation of Core Database Logic

5.1.1 Unit Tests for `fn_create_order`

The function `fn_create_order(p_customer_id, p_payment_id, p_items)` was validated using SQL unit tests executed with pgTAP. During the setup phase, artifacts from previous runs were removed and two products with known stock levels were inserted. Three tests were executed: (i) verifying that exactly one order is created, (ii) confirming that the two items specified in the JSON payload are inserted into `order_item`, and (iii) validating that the stock of Product A is reduced after order creation. All tests passed successfully.

Table 5.1: Expanded unit test results for `fn_create_order`.

Test ID	Description	Expected Result	Observed Outcome
UC-1	Validation of order creation. Ensures that the function inserts one record into `order`.	A single order must be created with a valid auto-incremented identifier.	Exactly one row was inserted into `order`; ID returned by the function matched this row.
UC-2	Validation of item decomposition. Confirms JSON items are processed correctly.	Two items must be inserted into `order_item`, mapped to the created order.	Two rows appeared in `order_item`, each referencing the correct `id_order`.
UC-3	Verification of stock adjustment after order creation.	Stock of Product A must decrease relative to its initial value (100 units).	Stock successfully decreased according to item quantity. Inventory update function executed without errors.

5.2 Functional Validation of Inventory Management

5.2.1 Unit Tests for `fn_update_product_stock`

The function `fn_update_product_stock(p_product_id, p_quantity)` was evaluated through four pgTAP tests. The tests verified that valid stock subtraction executes without error, that stock updates correctly, that attempts to reduce stock beyond available quantities produce the expected exception, and that no partial updates occur when an exception is raised. All four tests passed.

Table 5.2: Expanded unit test results for `fn_update_product_stock`.

Test ID	Description	Expected Behavior	Observed Outcome
UI-1	Valid stock subtraction should execute without exceptions.	The function must run successfully when decrementing stock by 3 units.	Execution completed with no errors; stock updated correctly.
UI-2	Verification of new stock value after update.	Stock must change from 10 to 7 units.	Stock recorded at exactly 7 units after function execution.
UI-3	Error handling for insufficient inventory.	Trying to subtract 20 units should raise "Stock cannot be negative".	Expected exception thrown; no partial modification occurred.
UI-4	Validation of rollback on failure.	If an exception occurs, stock must remain unchanged at 7 units.	Stock remained at 7 units, confirming transactional rollback integrity.

5.3 Performance Evaluation

5.3.1 Synthetic Workload Generation

A synthetic dataset was generated inside a transactional sandbox consisting of:

- 5,000 products with randomized prices and stock levels;
- 10,000 orders associated with a valid payment method;
- 1–2 items per order, totaling approximately 15,000 `order_item` rows.

All generated data were rolled back upon completion.

5.3.2 Query: Customer Order Retrieval

The first benchmark evaluated the latency of retrieving the 50 most recent orders for a customer. The `EXPLAIN ANALYZE` report showed efficient sorting behavior and low-latency execution due to the limit operator.

5.3.3 Query: Daily Sales Aggregation

A second benchmark evaluated daily revenue aggregation through a join between `order` and `order_item`. The query demonstrated stable performance, with group-by operations completing in acceptable analytical-query time.

Table 5.3: Performance results for analytical and transactional queries under synthetic workload.

Test ID	Query Evaluated	Purpose and Expected Behavior	Observed Performance
PT-1	Customer order retrieval with sorting and limit (50).	Measure response time of listing recent customer orders. Query expected to scale efficiently due to LIMIT and index usage.	Execution completed in low milliseconds. Index scan used on `created_at`. Query performance remained stable even with 10k orders.
PT-2	Daily sales aggregation using JOIN and GROUP BY.	Evaluate analytical workload performance over all orders. GROUP BY expected to aggregate full dataset (10k rows).	Query executed within acceptable analytic latency. JOIN completed full table scan; grouping cost remained stable. Suitable for dashboard refresh.

5.4 System Capacity and Storage Estimation

The database capacity was estimated using PostgreSQL's relation-size functions. The average row weight for the `product` table was approximately 2 KB. Given an available storage budget of 500 MB, the system can store approximately 250,000 product rows, confirming that the current architecture supports substantial catalog growth.

5.5 Integration and End-to-End Workflow Validation

System-level integration tests confirmed correct propagation of data through the full workflow:

- user authentication through Supabase Auth,
- product retrieval and filtering,
- cart and checkout logic,
- payment registration and confirmation via Mercado Pago,
- stock updates and order finalization.

No transactional inconsistencies or foreign-key violations were observed.

5.6 Dashboard and Analytical View Performance

Materialized views significantly reduced dashboard loading time by precomputing aggregated sales metrics. Their use improved response time for vendor analytics and reduced load on the underlying transactional tables.

Chapter 6

Discussion and Analysis

6.1 Interpretation of Results and Implications

This section analyzes the significance of the empirical findings presented in Chapter 5, evaluating the extent to which the proposed architecture meets the research objectives. Furthermore, it discusses the implications of these results in the context of lightweight e-commerce architectures and identifies the limitations inherent to the experimental validation.

6.1.1 Significance of Findings and Link to Objectives

The results obtained from the functional and performance validation confirm that the BogoGo platform successfully meets the primary objectives established in the Introduction.

- **Transactional Integrity and Inventory Management:** The success of unit tests for `fn_create_order` and `fn_update_product_stock` directly validates the objective of ensuring robust backend operations using PostgreSQL. The tests confirmed that the system enforces ACID properties—specifically atomicity and consistency—preventing overselling scenarios (Test UI-3) and ensuring reliable stock deduction. This aligns with the project's goal of providing a professional-grade transactional engine for small vendors.
- **Scalability for Local Market Context:** The capacity estimation revealed that the system can house approximately 250,000 product rows within the current 500 MB storage budget. This finding has significant implications for the target demographic (Bogotá's local fashion sector), as it demonstrates that the platform can support substantial catalog growth without requiring immediate infrastructure upgrades or incurring high cloud costs.
- **Performance of Business Intelligence Tools:** The performance evaluation of the "Daily Sales Aggregation" query and the dashboard metrics supports the objective of building accessible data-driven insights. The use of materialized views proved effective in decoupling analytical load from transactional operations, ensuring that vendor dashboards remain responsive even as the dataset grows to 15,000 order items.
- **Secure Integration of Third-Party Services:** The successful execution of the end-to-end integration workflow—encompassing Supabase Auth, Mercado Pago, and database updates—confirms the viability of the serverless Edge Function approach. This fulfills the objective of securely handling sensitive payment logic without maintaining a traditional backend server.

6.1.2 Relation to Previous Research

The empirical results reinforce architectural patterns discussed in the literature review. The stable performance observed in complex join operations (Customer Order Retrieval and Daily Sales Aggregation) aligns with comparative studies cited in the background [4], which highlight PostgreSQL's superiority in handling mixed transactional and analytical workloads compared to other open-source alternatives.

Furthermore, the successful integration of Supabase as a Backend-as-a-Service (BaaS) validates the trend towards lightweight, "thick-client" architectures [1]. The results demonstrate that it is possible to achieve enterprise-level features (role-based security, real-time updates, and relational integrity) with significantly reduced operational overhead, confirming the hypothesis that BaaS is a suitable model for early-stage e-commerce platforms.

6.1.3 Limitations and Uncertainties

Despite the positive outcomes, several limitations and uncertainties must be considered when interpreting these results:

- **Synthetic Workloads vs. Real-World Traffic:** As detailed in the Methodology, the performance evaluation relied on synthetic datasets and controlled transactional conditions. While this approach effectively validates database logic and theoretical throughput, it does not fully capture the chaotic nature of real-world usage, such as erratic network latency, concurrent write conflicts from thousands of distinct IP addresses, or malicious traffic patterns.
- **Infrastructure Constraints:** The tests were executed within the constraints of the Supabase Free Tier. While the system handled the simulated load of 500 concurrent interactions effectively, the long-term sustainability of this tier under continuous production traffic remains uncertain. Factors such as "Project Pausing" after inactivity or strict egress limits (discussed in the Methodology) pose risks for a permanent deployment that were not fully stress-tested in this cross-sectional study.
- **Scope of Security Testing:** While functional integration with authentication services was verified, advanced security audits—such as penetration testing or DDoS simulation—were outside the scope of this research. Therefore, the "secure" designation refers to the architectural correctness of the implementation (e.g., hiding API keys in Edge Functions) rather than a certification against hostile attacks.

In conclusion, the results provide strong evidence that the BogoGo architecture is functionally correct, economically viable, and sufficiently performant for its intended scope. However, transition to a production environment would require further validation using real user traffic and potentially an upgrade to paid infrastructure tiers to mitigate the identified limitations.

Chapter 7

Conclusions

The development and evaluation of the BogoGo platform demonstrate that a lightweight, BaaS-driven architecture built with React, Supabase PostgreSQL, Supabase Storage, and Edge Functions can effectively support the functional and analytical requirements of a localized e-commerce ecosystem. The system's modular three-layer structure fulfills the objectives defined at the start of the project, confirming that these technologies constitute a coherent and feasible foundation for early-stage digital commerce initiatives aimed at Bogotá's fashion sector.

Functional validation results provide strong evidence of correctness, consistency, and transactional integrity. Unit tests for key database functions—such as `fn_create_order` and `fn_update_product_stock`—confirmed reliable order generation, item decomposition, and inventory management with full ACID guarantees. Performance tests using synthetic workloads further demonstrated stable latency in customer-order retrieval and daily sales aggregation, indicating that the database schema and indexing strategy scale efficiently within expected usage volumes. Integration tests also verified seamless end-to-end execution across authentication, product exploration, checkout, payment processing, and stock updates, validating the robustness of the BaaS-centered design.

The system capacity assessment shows that the platform can support product catalogs significantly larger than those typically managed by small and medium retailers, even within Supabase's free-tier constraints. Although limitations remain due to synthetic testing conditions, quota restrictions, and the absence of real user traffic, the project establishes a solid and cost-effective technical foundation for a localized online marketplace. Future work will focus on strengthening the platform through enhanced analytics, recommender features, improved security auditing, and deployment on higher-capacity infrastructure tiers to prepare the system for broader adoption.

References

[1] ITMonks. (2024) E-commerce website architecture: Layered (n-tier) approach. [Online]. Available: https://itmonks.com/blog/e-commerce/development/ecommerce-website-architecture/

[2] NopCommerce. (2023) E-commerce website architecture explained. [Online]. Available: https://www.nopcommerce.com/en/blog/ecommerce-website-architecture

[3] R. Blueprints. (2022) 3-tier architecture: Presentation, logic and data. [Online]. Available: https://medium.com/rkblueprints/3-tier-architecture-presentation-logic-and-data-268d9573ddc8

[4] J. G. Zapata, "Mysql vs postgresql: A comparative analysis of relational database management systems (rdbms) technologies response time in web-based e-commerce," *ResearchGate*, 2024. [Online]. Available: https://www.researchgate.net/publication/382641887

[5] K. Software. (2023) Designing an e-commerce database for efficient data management. [Online]. Available: https://kanishkasoftware.com/designing-an-ecommerce-database-for-efficient-data-management/

[6] Shopware. (2023) E-commerce databases: Understanding entities and e-r diagrams. [Online]. Available: https://www.shopware.com/de/news/ecommerce-databases/

[7] A. Builders. (2024) Simple steps to deploy a three-tier e-commerce system on aws eks. [Online]. Available: https://dev.to/aws-builders/simple-steps-to-deploy-a-three-tier-e-commerce-system-on-aws-eks-eb0