Software Design Specification

for

WhatsApp Chatbot

Version 1.0

Prepared by

Patrick Marsden	620169874	pmarsden2k5@gmail.com
Jason Matthews	620156868	jasonmatthews057@gmail.com
Tiana Samuels	620155658	tiana.samuels@gmail.com
Jada Cole	620171844	jadanscole10@gmail.com
Tahjric Allen	620133830	tahjric.allen04@gmail.com

Course Instructor:	Dr. Ricardo Anderson
Course:	COMP2140 – Introduction to Software Engineering
Studio Facilitator:	Roshae Sinclair
Date:	November 22, 2025

TABLE of CONTENTS

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
Draft 1	Patrick Marsden Tiana Samuels Tahjric Allen Jason Matthews Jada Cole	This is the initial version of the Software Requirements Specification document.	11/8/2025
Draft 2	Patrick Marsden	Update Requirements based on Clients Feedback and Group Resources	12/1/2025

1.0 Project Overview

White Rose Interiors Ltd. relies heavily on WhatsApp for customer quotations, scheduling, and service coordination. The existing manual process leads to delays, errors, and poor data tracking. This Software Design Specification outlines the technical architecture for a WhatsApp-based chatbot system that automates quotations, appointment scheduling, invoice generation, reminders, and administrative oversight.

This SDS reflects the **final, updated system architecture**, where **all persistent data is stored in JSON files** (`orders.json`, `prices.json`, etc.) instead of using any SQL database.

2.0 Architectural Design

2.1 General Constraints

1. **WhatsApp Automation Library (whatsapp-web.js):** Requires persistent runtime and active WhatsApp Web session.

2. **Windows Server Deployment:** Must run continuously as a background service (PM2/NSSM).

3. **JSON Storage:** All persistent data (orders, reminders, pricing, user states) must be safely handled using local JSON files.

4. **Internet Connection:** Required for WhatsApp messaging, admin dashboard access, and email delivery.

2.2 Alternatives Considered

Alternative 1: Serverless Microservices (AWS Lambda / Azure Functions) Description: Breaking the bot into small functions (e.g., calculateQuote , saveBooking) that run only when triggered.

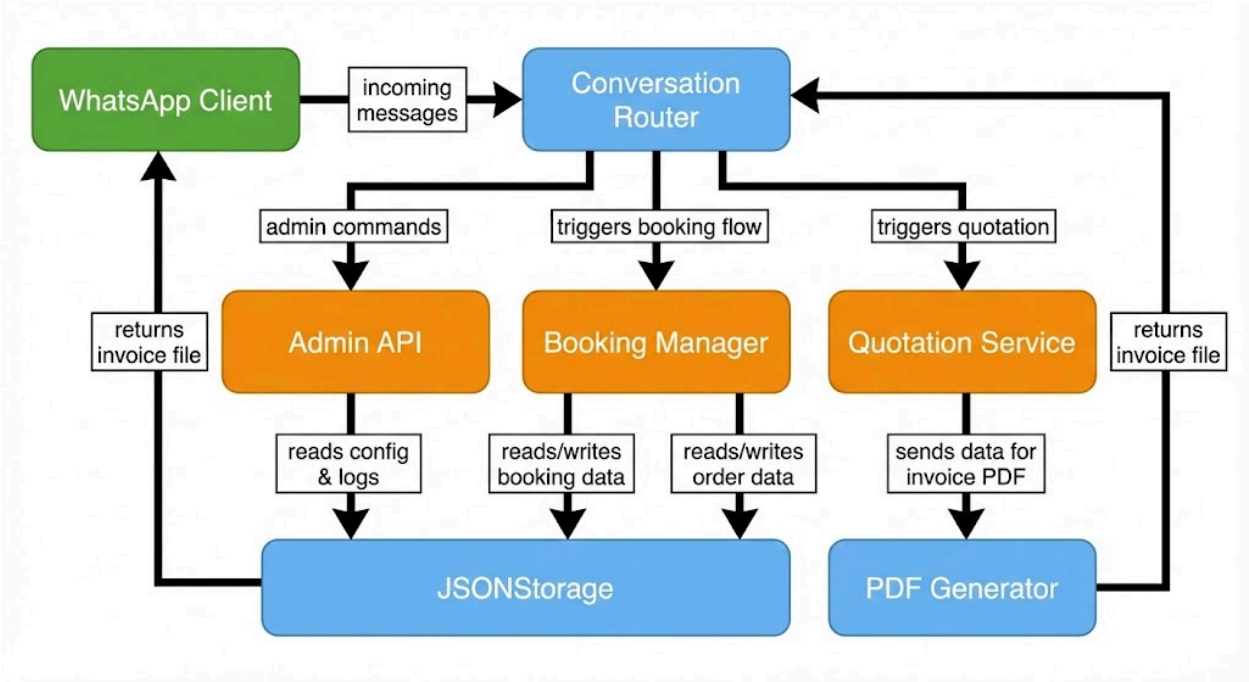
Why Rejected: The whatsapp-web.js library requires a persistent, running process to maintain the WebSocket connection to WhatsApp. It cannot "wake up" from cold storage efficiently for every message without re-authenticating (scanning QR code), which is impossible for an automated server.

Disadvantage: Incompatible with the core library dependency; high latency for session re establishment.

Alternative 2: Event-Driven Architecture (Message Bus) Description: Using a message broker (RabbitMQ/Kafka) where the WhatsApp client publishes events, and separate services (Quoter, Scheduler) subscribe and process them.

Why Rejected: While excellent for scalability, this adds massive complexity (setup, maintenance of broker) for a student project with a single deployment node. The overhead outweighs the benefits for the expected traffic volume. Disadvantage: Over-engineering; increases deployment complexity significantly

2.3 System Architecture Diagram



2.3.1 Architectural Description

- **1. WhatsApp Client (Presentation Layer)**

  The WhatsApp Client is the system's primary communication interface. Using `whatsapp-web.js`, it maintains a live WebSocket session with WhatsApp servers and forwards all incoming user messages to the Conversation Router. It also receives outgoing responses, generated PDFs, and invoice files from backend services and delivers them to customers.

- **2. Conversation Router (Logic Layer – State Machine & Intent Handler)**

  The Conversation Router is the central orchestration component of the chatbot. It interprets every user message, determines intent, updates the user's state, and routes requests to the appropriate service module.

- **3. Admin API (Presentation + Logic Layer for Staff Operations)**

  The Admin API serves administrative users through a secure dashboard. It exposes RESTful endpoints that allow managers and staff to access operational data, generate reports, and issue admin commands that affect bookings, reminders, or orders.

- **4. Booking Manager (Business Logic Layer – Scheduling & Orders)**

  The Booking Manager manages all appointment-related operations. When triggered by the Conversation Router, it handles the booking workflow: capturing date/time preferences, validating availability, storing orders, sending reminders, and updating statuses.

- **5. Quotation Service (Business Logic Layer – Pricing & Estimation)**

  This module processes user measurement inputs and calculates product prices (e.g., blinds, carpets). It applies business rules, retrieves pricing tables, and formats detailed quotation data to be passed to the PDF Generator.

- **6. JSONStorage (Storage Layer – Safe JSON I/O Manager)**

  JSONStorage acts as the system's persistent data layer. It provides safe, atomic operations around read/write access to files such as `orders.json`, `prices.json`, `reminders.json`, `logs.json`, and configuration files.

- **7. PDF Generator (Utility Layer – Document Builder)**

  The PDF Generator transforms quotation or invoice data into PDF documents. It receives structured content from the Quotation Service or Booking Manager and produces ready-to-send files.

2.3.2 Architecture Justification

The Layered Architecture is the best choice for the WhatsApp ChatBot primarily because the project's requirements demand separation of concerns, particularly concerning security and maintainability. This architectural pattern organizes functionality into layers, providing services only to the layer immediately above it, which is essential for implementing multi-level security and redundancy to enhance system dependability.

The separation also allows core business logic (like Quotation Generation) to reside in dedicated layers, insulating them from interface changes (e.g., WhatsApp vs. Admin Dashboard), thereby aiding the overall Maintainability of the system

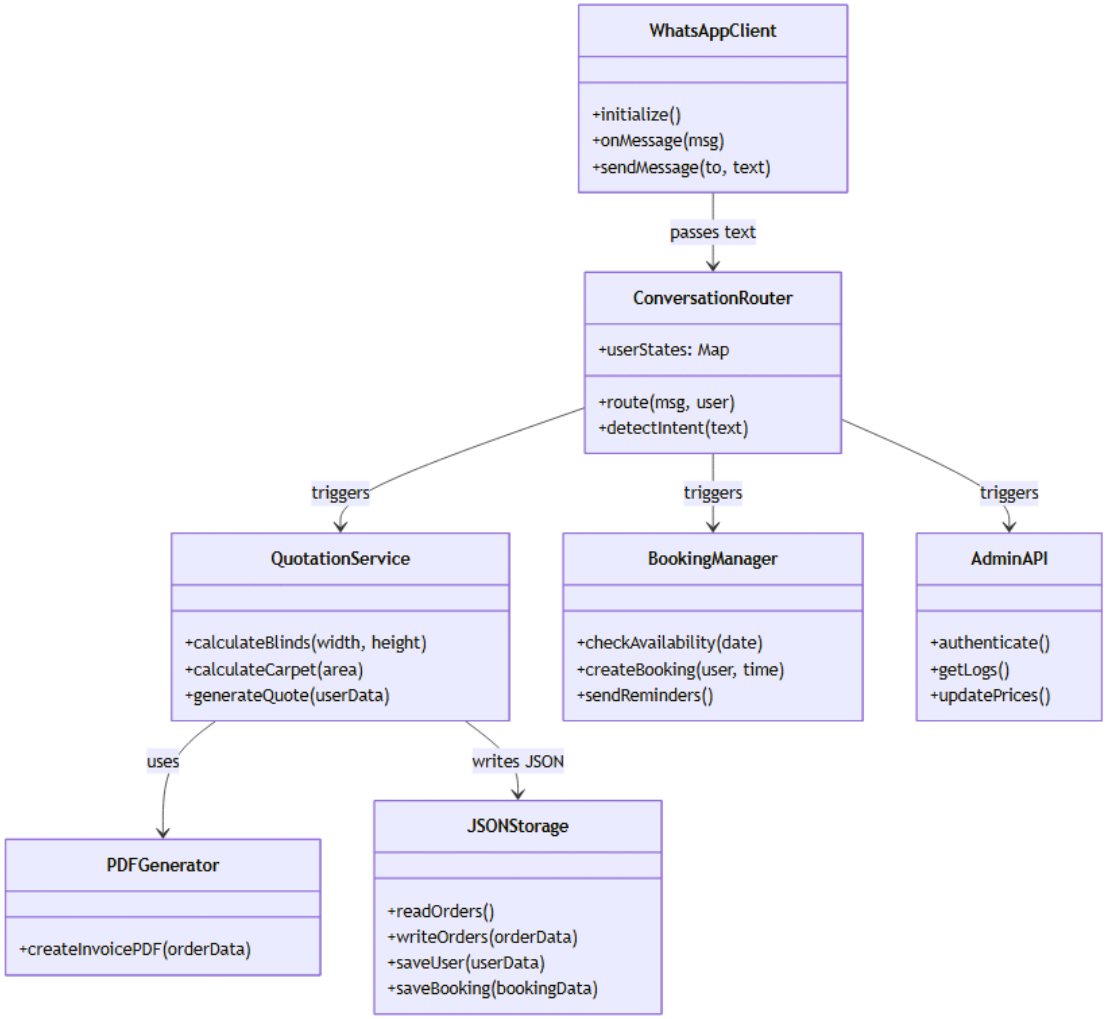
3.0 Architecture Decomposition

3.1 Component Decomposition

Requirement ID	**Architecture Component**	**Class / <Module> Name**	**Description**
Req 1	Presentation Layer	`WhatsAppClient`	Initializes the `web.js` client, manages authentication (QR stream), and standardizes incoming message events before passing them to the router.
Req 1, 7	Business Logic	`ConversationRouter`	Analyzes incoming text using a State Machine pattern. Determines user intent and routes control to the appropriate service (Quote, Schedule, or FAQ).
Req 2, 3	Business Logic	`QuotationService`	Implements the pricing algorithms for Blinds and Carpets. Validates numeric inputs (width/height) and prepares the data object for invoice generation.

Req 2, 3	Business Logic	`PDFGenerator`	A utility module that accepts an Order Object and renders it into a PDF file saved on the local file system. Returns the file path for sending.
Req 5, 8, 10	Business Logic	`BookingManager`	Manages calendar logic. Methods include `checkAvailability()`, `createBooking()`, and a cron job for `sendDailyReminders()` (Req 8).
Req 7	Business Logic	`KnowledgeBaseSvc`	Provides keyword matching logic to find answers to common questions (FAQ) from the database.
Req 4, 9	Presentation Layer	`AdminAPI`	Provides RESTful endpoints (`GET /logs`, `POST /prices`) for the web dashboard. Handles admin authentication (JWT).
All	Storage	`JSONManager`	Safe read/write operations to JSON data files

3.2 Structural Design



3.2.1 Design Notes

- State Management: The ConversationRouter class includes a userStates Map. This is a critical design decision. Because HTTP and WhatsApp are stateless protocols, the application memory must track "where" a user is in a conversation (e.g., STATE: AWAITING_WIDTH). Without this, the bot cannot handle multi-step flows like Quotations.
- Singleton Pattern: The WhatsAppClient and DatabaseManager classes are designed as Singletons. We must ensure there is only one active browser instance connected to WhatsApp and one active connection pool to the SQLite file to prevent locking errors and memory leaks.
- Dependency Direction: The diagram shows that Presentation classes depend on Logic classes, which depend on Data classes. This strictly follows the Layered Architecture pattern, preventing circular dependencies and "spaghetti code."

3.2.2 JSONStorageManager Responsibilities

1. Safely load JSON files (orders, reminders, pricing, logs).
2. Write updates atomically (sync write or write-queue).
3. Validate structure before saving.
4. Provide helper methods:
    - `getOrders()`
    - `updateOrder()`
    - `saveOrder()`

- `getPricing()`
- `logReminder()`

4.0 Additional Design Considerations

Reliability

- JSON files backed up regularly.
- Auto-repair for corrupted JSON structures.

Performance

- Cached reads reduce repetitive file access.
- Write operations optimized to avoid blocking.

Security

- Admin routes protected using token-based authentication.
- Sensitive data inside JSON encrypted when appropriate.