Demo 4: Architectural Requirements

GreenCart
Client: BBD Software

Team member	Student Number
Nikhil Govender	u22504193
Shayden Naidoo	u23599236
Corné de Lange	u23788862
Tshegofatso Mahlase	u22580833
Samvit Prakash	u23525119

Introduction

GreenCart is a sustainable eCommerce platform facilitating transparent and responsible trade between eco-conscious consumers and environmentally verified retailers. The architecture supports critical system capabilities including performance, modularity, usability, scalability, and security — all informed by stakeholder needs.

Architectural Design Strategy

Design and architectural decisions for the GreenCart system were made by the development team based primarily on the quality requirements established through consultation with the client and stakeholders. These quality requirements were used as the foundation for all major design choices in the system's architecture. Other architectural strategies such as functional decomposition and test-driven architecture will be used during system implementation to support modularity, scalability, and parallel development workflows.

However, the core architectural design strategy employed during the design phase was designing based on quality requirements — ensuring that the architecture would meet the expectations around performance, scalability, usability, and modularity from the very beginning of the system lifecycle.

Reasons for this choice:

1. Alignment with the Client's Vision and Project Goals

- **Client-Driven Design Focus**: The client emphasized a need for a highly responsive,

scalable, and sustainable eCommerce platform. Designing around these quality requirements ensures that GreenCart meets these expectations from the architectural level down to the implementation.

- **Quality-Centered Requirements Gathering**: From the first iteration, quality attributes such as low-latency product browsing, secure user handling, and modular feature integration (e.g., cart and order management) were prioritized during requirement elicitation.

- **Stakeholder Satisfaction**: Meeting explicitly stated quality expectations (e.g., “must support high traffic periods,” “must be fast,” “must be easy to test and extend”) is critical for stakeholder approval, including clients, end-users, and mentors.

2. Cost and Time Efficiency in Agile Development

- **Avoiding Technical Debt**: By designing with quality in mind from the start, we reduce the likelihood of encountering issues later that would require significant architectural rewrites (e.g., performance bottlenecks due to synchronous data flows).

- **Lean and Targeted Development**: The team focuses its implementation effort on high-priority quality features, such as efficient cart handling and scalable

---

backend services. This ensures optimal use of development resources and adherence to tight project timelines (Demo 2 and beyond).

- **Parallel Team Contributions**: Architecting around clean, testable boundaries (based on quality attributes like modularity) supports distributed development — with different members working concurrently on cart, orders, auth, and product subsystems.

3. Informed and Purposeful Technical Decisions

- **Framework and Technology Selection**: Choices such as using **FastAPI** (for async performance and strong typing), **Supabase Auth** (for secure RBAC-based access), and **Redis** (for cache-aside scalability) were directly influenced by performance, security, and modularity goals defined in the QR analysis.

- **Pattern and Strategy Alignment**: The system architecture incorporates multiple complementary strategies (MVC on the frontend, layered + microservice decomposition on the backend) that are known to directly support core QRs like maintainability, performance, and reusability.

- **Predefined Testing Paths**: Quality requirements informed the test plan and coverage strategy — e.g., response time metrics for performance testing, isolated unit tests for modularity validation, and integration tests focused on feature boundaries.

In conclusion, designing based on quality requirements ensures that the GreenCart system is technically robust, meets stakeholder expectations, and is positioned for

---

iterative enhancement. This strategy integrates quality assurance directly into the architectural DNA of the system and forms the basis for all future development and deployment decisions.

Architectural Strategy

Chosen Strategy: Service-Oriented Architecture (SOA)

Overview

Service-Oriented Architecture (SOA) is an architectural style in which a system is composed of distinct, loosely coupled services that encapsulate business logic and expose well-defined interfaces. These services communicate over standard protocols—typically HTTP using RESTful APIs—enabling interoperability, scalability, and independent deployment.

Rationale for GreenCart

GreenCart is designed as a modular and scalable e-commerce platform that incorporates sustainability insights, user personalization, and real-time carbon tracking. The decision to adopt SOA was based on a combination of system quality requirements, deployment flexibility, and long-term maintainability.

1. Modularity and Loose Coupling

Each subsystem in GreenCart has a clear responsibility, for example:

- **User Management and Authentication**
- **Product Catalog and Search**
- **Sustainability Insights Engine**
- **Cart and Checkout System**
- **Order Management**
- **Carbon Footprint Reporting**

This separation allows each service to be developed, tested, and deployed independently, reducing system interdependencies and enabling faster iterations.

2. Alignment with MVC and API-First Design

The SOA strategy complements the system's MVC-style frontend, which is built with React and consumes backend data through FastAPI REST endpoints. The backend services act as the controller and model layers, while the frontend handles all user interaction and presentation logic.

By following an API-first approach, the system ensures clear contract boundaries between services, facilitating frontend-backend decoupling and future third-party integrations.

3. Scalability and Cloud-Readiness

GreenCart is hosted on AWS, which supports service-oriented deployment using tools such as:

- **AWS EC2 and RDS** for scalable compute and data persistence
- **S3 and CloudFront** for frontend hosting and content delivery
- **Lambda (future roadmap)** for serverless computation, such as sustainability badge calculation or donation triggers
- **Amazon SNS/EventBridge** (future consideration) for enabling asynchronous event-driven processing across services

This architectural style enables the team to scale specific components—such as the product search or recommendation engine—based on usage patterns without overprovisioning the entire system.

4. Flexibility and Maintainability

SOA makes it easier to replace or refactor parts of the system in the future. For

example, the sustainability engine could be migrated to a machine learning microservice without affecting the rest of the platform. This modularity improves long-term maintainability and reduces technical debt accumulation.

Summary

Service-Oriented Architecture (SOA) was selected for GreenCart because it aligns closely with the project's goals of modularity, scalability, and cloud-native development. It provides a robust foundation for building a sustainable and extensible e-commerce platform that delivers personalized, eco-conscious experiences to users while supporting continuous delivery and future innovation.

Architectural Quality Requirements

The following five quality requirements were identified by the GreenCart team in collaboration with the client (BBD Software) based on core platform goals and ethical sustainability principles. They are listed in order of priority and are quantified for testability.

1. **Security**
2. **Performance**
3. **Usability**
4. **Modularity**
5. **Scalability**

1. Security

GreenCart must ensure that sensitive user data, credentials, and platform access points are well-protected against unauthorized access and common security threats. The system should implement strong authentication, secure data handling, and protection against common web vulnerabilities to safeguard the platform and maintain user trust.

Stimulus Source	**Admin, attacker, or unauthorized user**
Stimulus	Attempt to access, manipulate, or exfiltrate restricted data or system functionality
Response	- Enforce JWT-based authentication- Use HTTPS for all communication- Hash passwords using bcrypt- Sanitize all user inputs- Apply role-based access control (RBAC)- Log suspicious or failed access attempts
Response Measure	- Unauthorized requests return 401 or 403- All sensitive data stored using salted hashing- No direct database access from

client- Zero OWASP Top 10 vulnerabilities triggered in audit

Environment Login endpoints, admin panel, database queries, API routes exposed to public

Artifact FastAPI auth routes, user model, database schema, middleware, deployment configs

---

2. Performance

Performance is essential for providing a seamless and engaging shopping experience. The system must allow shoppers to browse, search, and check out quickly without delays.

Website user or guest shopper

Stimulus Source

Stimulus Wants pages and interactions to load quickly

Response - Optimize backend queries - Use Redis cache for product and cart - Lazy-load product images

Response Measure listing page loads in ≤ 1.5s - Checkout completes in ≤ 3s

- 95% of API requests < 400 ms - Product

Environment During peak usage of the platform **Artifact** FastAPI

backend, PostgreSQL, Redis, frontend UI

---

3. Usability

Usability is a priority for GreenCart's target audience, which includes users of various technical skill levels. The UI should feel intuitive and informative, encouraging users to explore and commit to sustainable choices without confusion.

	End-users including shoppers and eco-conscious buyers
Stimulus Source	
Stimulus Wants to intuitively navigate and use the platform	
Response - Clear navigation and filtering Sustainability help sections - Accessible design and tooltips	
Environment Browsing on mobile, desktop, or tablet using common browsers	
Artifact React frontend, product filtering interface, help page	

---

4. Modularity

Given GreenCart's complex feature set (product browsing, sustainability ratings, carbon offsetting, retailer management), the system must support modular development. A modular architecture allows team members to develop, test, and scale different services independently.

	Developers and system maintainers
Stimulus Source	

Stimulus Wants to add new features or services without breaking others

Response - Backend organized into product, cart, auth, order services - Frontend components are reusable - RESTful API boundaries maintained

Response Measure - New features integrate with ≤ 2 modules - ≥ 80% unit test coverage	per service - Services deployed independently

Environment During ongoing development and feature rollouts

Artifact Backend service structure, API layer, testing suite

---

5. Scalability

GreenCart must be prepared for organic user growth and seasonal usage spikes (e.g., Earth Day campaigns). The platform should scale in terms of concurrent users, product inventory, and sustainability metrics without degrading performance or stability.

	Admin or system under user load
Stimulus Source	

Stimulus System should handle more users and product data

Response - Use FastAPI async handling - Redis caching

Response Measure - Handle ≥ 100 concurrent users -	Product queries ≤ 600 ms with large dataset - Cart/order latency ≤ 500 ms under load

Environment High traffic events, campaign launches, or data import periods **Artifact**

Backend containers, PostgreSQL, Redis, deployment infrastructure

---

Architectural Patterns

System Overview

The GreenCart system is composed of several interrelated yet modular components, each mapped to distinct architectural patterns to optimize maintainability, extensibility, scalability, and performance. The main components include:

- The frontend interface, developed in React.js with Vite, where users interact with products, carts, and dashboards.

- The backend RESTful API, built using FastAPI in Python, which handles business logic, user authentication, sustainability scoring, and recommendations.

- Several subsystems, including sustainability insights, carbon footprint tracking, authentication, and order management, which are distributed as independent

services.

These components incorporate three major architectural patterns: Model-View-Controller (MVC), Service-Oriented Architecture (SOA) and Layered Architecture.

Layered Architecture

The GreenCart backend is organized using a clear layered architectural approach that promotes separation of concerns and maintainability. This pattern structures the system into distinct horizontal layers, where each layer only communicates with adjacent layers, ensuring loose coupling and high cohesion.

The implementation follows these distinct layers:

- **Presentation Layer (Routes/Controllers)**: Implemented in `app/routes/*.py` files, this layer handles HTTP requests and responses, input validation, and API endpoint management. It serves as the entry point for all client interactions and delegates business logic to lower layers.
- **Business Logic Layer (Services)**: Located in `app/services/*.py`, this layer contains the core business rules, sustainability calculations, recommendation algorithms, and orchestration logic. It processes data from the presentation layer and coordinates with the data access layer.
- **Data Access Layer (Models/Schemas)**: Comprising `app/models/*.py` and `app/schemas/*.py`, this layer defines data structures, ORM mappings, and database interaction patterns. It abstracts database operations and provides a clean interface for business logic.
- **Database Layer**: Managed through `app/db/session.py`, this layer handles database connections, session management, and low-level data persistence operations with PostgreSQL.

This layered approach enhances modularity by allowing each layer to be developed, tested, and maintained independently. It also supports the system's scalability requirements by enabling horizontal scaling of individual layers and improves security by centralizing data access controls at the appropriate layer.

Model-View-Controller (MVC) Pattern

The MVC pattern is applied in the separation of frontend interface logic, business rules, and data models. This pattern helps enforce separation of concerns across the system's architecture, allowing parallel development across the frontend and backend teams. It also significantly improves testability, scalability, and maintainability of the platform.

- **Model**: Contains the core business logic and application data.
  This includes sustainability metadata, product listings, eco-ratings, and carbon impact analytics — all persisted in the PostgreSQL database.

- **View**: Refers to the interactive user-facing elements designed using Figma and implemented in React.js.
  These include product catalog pages, dashboards (e.g., eco-impact summaries), cart and checkout pages, and sustainability visualizations (e.g., progress graphs and certification badges).

- **Controller**: Implemented via FastAPI endpoints that respond to user actions, orchestrate business logic, and return dynamic data.
  This layer manages authentication (JWT, registration/login), product filtering and search logic, order processing, and eco-friendly product recommendations.

This architectural pattern improves usability and performance by allowing independently testable and updatable components across different system concerns.

Service-Oriented Architecture (SOA)

GreenCart follows a Service-Oriented Architecture by structuring the system as independent functional services, each responsible for a specific domain. These services are loosely coupled and communicate via RESTful APIs. Examples include:

- **Authentication Service**

- **Product Management Service**
- **Order and Checkout Service**
- **Sustainability Analytics and Carbon Tracking Service**
- **Recommendation Engine**

SOA enables horizontal scalability, independent deployment, and rapid development cycles, as different services can be updated or scaled without affecting the rest of the system. It also enhances modularity, allowing each team member to work on an isolated service while maintaining API-level consistency.

Architectural Constraints

The GreenCart platform is subject to several key architectural constraints which influence the design decisions, technology stack, and user interface considerations. These constraints were established to ensure that the system meets client expectations, complies with legal requirements, and provides a seamless user experience across diverse environments.

1. Responsive Design Constraint

> *The platform must be designed to be fully responsive on all browsers and optimized for common screen sizes.*

This constraint ensures that GreenCart can be accessed and used effectively across a wide range of devices — including mobile phones, tablets, and desktops — without loss of functionality or clarity. The architecture must therefore support:

- Use of responsive layout frameworks (e.g., CSS Flexbox, Grid)
- Design testing on common screen resolutions (e.g., 1920×1080, 1366×768, 768×1024, 390×844)
- Cross-browser compatibility, including Chrome, Firefox, Safari, and Edge ● Mobile-first component design in the React.js frontend, integrated via Vite

By enforcing responsive design at the architectural level, the system ensures consistent usability and performance, regardless of the user's device.

2. Data Privacy and Regulatory Compliance Constraint

> *The system should ensure data privacy and comply with relevant data*

protection regulations, such as POPIA & GDPR, for user information.

GreenCart handles sensitive user information (e.g., personal details, address, purchase history), making regulatory compliance a mandatory architectural concern. The system must be designed to meet the following obligations:

- **POPIA (Protection of Personal Information Act – South Africa)**
- **GDPR (General Data Protection Regulation – EU)**

This entails implementing:

- Secure user authentication mechanisms (JWT-based access control)
- Encryption of sensitive data in storage and transit
- User consent flows for data collection and communication preferences
- Mechanisms to allow data access, rectification, and deletion (user rights)

Additionally, database schemas, API design, and logging policies must ensure minimal data exposure and traceability in line with these regulations.

---

3. Usability Constraint

The user interface should be intuitive and easy to navigate, catering to users of varying technical expertise.

As an eCommerce platform aimed at promoting sustainable consumer behavior, GreenCart must be accessible to users with varying levels of digital literacy. This constraint imposes several design requirements:

- Clear, simple navigation structures using recognizable UI patterns
- Progressive disclosure: hiding advanced options unless required
- Use of icons, labels, and tooltips to guide users
- Contextual help for sustainability features (e.g., what a carbon badge means)
- Prioritizing task completion flows such as product search, filtering, checkout, and

sustainability tracking

This constraint influences the frontend architecture (e.g., component hierarchy, route layout), selection of UI libraries, and UX testing methodology.

Architectural Diagram

Requirement	Architectural Strategies	Architectural Pattern
Security	● JWT-based authentication ● HTTPS for all communication ● bcrypt password hashing ● Role-Based Access Control (RBAC) ● Sanitize inputs- Logging suspicious activity	Client-Server + SOA (Auth Service), API Security Layer, Layered Architecture (security at each layer)
Performance	● Optimize backend queries ● Redis caching for products & carts ● Lazy-load product images ● FastAPI async handling	Architecture (asynchronous tasks, background jobs), Layered Architecture (optimized at each tier)

Usability	● Clear navigation & filtering ● Accessibility features & tooltips ● Sustainability help sections ● Responsive frontend design	MVC (React frontend with FastAPI backend), Layered Architecture (presentation tier focus)
Modularity	● Independent backend services (auth, cart, orders, sustainability) ● Reusable frontend components ● RESTful API boundaries	Service-Oriented Architecture (SOA), Layered Architecture (modular layers)
Scalability	● Horizontal scaling on AWS (EC2, RDS, S3, CloudFront) ● Redis cache for scale-out reads	SOA + Layered Architecture (independent tier scaling)

	● API-first design for third-party integration	
Availability	● AWS-managed services with redundancy ● PostgreSQL with high availability	Cloud-Native Deployment
Deployment & Cloud Readiness	● AWS EC2, RDS, S3, CloudFront ● CI/CD pipelines with GitHub Actions ● Potential for serverless (Lambda)	Cloud-Native SOA
Data Privacy & Compliance	● POPIA & GDPR compliance ● Encrypted data storage & transit ● User consent management ● Data rectification/deletion workflows	SOA

Technology Choices

1. Frontend Development

Consideration 1: React with Vite

Overview

React is a popular JavaScript library for building interactive user interfaces through a component-based architecture. Vite is a modern build tool and development server that provides instant server start and lightning-fast hot module replacement (HMR). Combined, they allow developers to rapidly build and scale responsive front-end applications with excellent performance and modularity.

Advantages

- Fast development with Vite's optimized bundling and HMR.
- Highly reusable and maintainable components using React's modular structure.
- Large ecosystem and community with access to thousands of UI libraries (e.g., Material UI, Tailwind CSS).
- Seamless integration with REST APIs and state management tools (Redux, Context API).
- JSX enables flexible UI composition and logic co-location.

Disadvantages

- Steeper learning curve for advanced hooks and modern React concepts.
- Performance optimization is needed in large applications if not carefully managed.
- Requires additional libraries for routing and state management (e.g., React Router).

Suitability

React with Vite aligns well with the Model-View-Controller (MVC) architectural pattern, as it separates concerns through reusable UI components (View), integrates easily with a RESTful API (Controller), and supports state-driven rendering (Model). The integration with Figma ensures a clean design-to-development workflow, and the toolchain's modularity supports our goals for performance, scalability, and testability.

Consideration 2: Angular

Overview

Angular is a robust frontend framework maintained by Google. It offers built-in solutions for routing, forms, and dependency injection, supporting large-scale enterprise applications.

Advantages

- Full-featured framework with built-in services, routing, and RxJS.
- Strong support for TypeScript and structured architecture.
- Comprehensive documentation and tooling (CLI, Angular Material).

Disadvantages

- Steep learning curve due to verbosity and framework complexity.
- Can introduce performance overhead in smaller projects.

Suitability

While Angular supports MVC and scalable architectures, its complexity and rigidity are excessive for our modular and fast-paced development needs in GreenCart.

Consideration 3: Svelte

Overview

Svelte is a compiler-based JavaScript framework that compiles components into efficient vanilla JavaScript at build time.

Advantages

- Minimal runtime overhead and extremely fast performance.
- Easy to learn with concise syntax.
- No virtual DOM improves rendering speed.

Disadvantages

- Smaller ecosystem and less community support.
- Limited third-party libraries and tools.

Suitability

Svelte is best suited for lightweight projects. It lacks the maturity and flexibility required for GreenCart's dynamic frontend and integration needs.

Final Choice: React with Vite

React with Vite was chosen due to its excellent modularity, performance, and alignment with our MVC architecture. Its ecosystem, community support, and seamless backend integration make it ideal for building GreenCart’s responsive and scalable user interface.

2. Backend Development

Consideration 1: FastAPI (Python)

Overview

FastAPI is a modern, high-performance Python web framework for building APIs with automatic OpenAPI documentation and dependency injection. It is designed for speed and ease of development, supporting asynchronous endpoints, type hints, and automatic data validation. FastAPI is ideal for building scalable RESTful services in a microservices architecture.

Advantages

- **High Performance**: Built on Starlette and Pydantic, FastAPI offers asynchronous capabilities suitable for concurrent workloads.
- **Automatic Documentation**: Generates OpenAPI and Swagger documentation out-of-the-box.
- **Type Safety**: Enforces data validation and clear typing through Python type hints.

- **Developer Experience**: Clean, minimal syntax with async support and robust error handling.
- **Easy Testing and Integration**: Supports Pytest and TestClient for smooth API test automation.

Disadvantages

- **Steeper Learning Curve for Async Concepts**: Developers must understand asynchronous programming models.
- **Less Mature Ecosystem**: Compared to Django or Flask, fewer plugins or pre-built components exist.
- **Manual Work for Common Patterns**: FastAPI offers flexibility, but some features (e.g., admin panel, ORM integration) must be explicitly configured.

Suitability

FastAPI perfectly aligns with GreenCart's microservice-oriented and event-driven

architecture. Its support for modular API development allows us to scale independently developed subsystems like authentication, cart management, and sustainability tracking. It fits our quality requirements for performance, responsiveness, and maintainability.

Consideration 2: Django (Python)

Overview

Django is a high-level Python web framework that encourages rapid development and follows the "batteries-included" philosophy. It includes ORM, admin panel, authentication, and templating out of the box.

Advantages

- Full-featured and production-ready framework.
- Built-in ORM, admin dashboard, and user auth system.
- Strong documentation and developer community.

Disadvantages

- Monolithic structure can be limiting for microservice-based architectures. ● Heavier memory usage and slower response time than async frameworks. ● Less suitable for modern frontend integration compared to lightweight API-first frameworks.

Suitability

While Django offers many useful features, its monolithic nature and synchronous processing are less suited to the lightweight and modular GreenCart architecture. It would create tight coupling between components, hindering scalability.

Consideration 3: Express.js (Node.js)

Overview

Express.js is a minimalist Node.js framework used to build backend APIs with JavaScript. It's widely used in full-stack JavaScript applications.

Advantages

- Minimal setup and fast startup time.
- Massive community and library ecosystem.

- Easy to integrate with JavaScript-based frontends.

Disadvantages

- Lacks native type safety without additional setup (e.g., TypeScript).
- Manual configuration for routing, validation, and security.
- Less efficient for CPU-bound tasks or data-heavy applications.

Suitability

Although suitable for full-stack JS development, Express lacks the strong typing and high-performance async support offered by FastAPI. It is less ideal for our data-driven platform where clarity, validation, and response time are critical.

Final Choice: FastAPI (Python)

FastAPI was selected as the primary backend framework due to its asynchronous capabilities, built-in validation, and automatic documentation features. It fits GreenCart's service-oriented architecture and performance goals while supporting modular deployment of authentication, product, and sustainability services.

3. API & Security

Consideration 1: FastAPI + JWT (JSON Web Tokens) + OAuth2

Overview

The GreenCart system uses FastAPI to expose RESTful API endpoints, and JWT (JSON Web Tokens) for stateless authentication. OAuth2 is used as the foundational flow for token issuance, enabling secure login, registration, and permission handling. This stack offers modern, lightweight, and scalable authentication mechanisms while keeping APIs clean, standardized, and testable.

Advantages

- **Stateless Authentication**: JWT enables scalable, session-free user management.
- **Built-In OAuth2 Flow Support**: FastAPI natively supports OAuth2 with password and token flows.
- **Auto-Generated OpenAPI Documentation**: Clear API specs improve developer collaboration and testing.
- **Granular Role-Based Access**: JWT claims can encode user roles or permissions for custom authorization rules.
- **Secure Communication**: JWTs are signed (and optionally encrypted) to prevent tampering.

Disadvantages

- **Token Management Overhead**: Requires correct handling of token expiry, refresh

tokens, and revocation.

- **Security Sensitivity**: Misconfigured JWT (e.g., using weak secrets or skipping verification) can lead to vulnerabilities.
- **No Built-In Revocation**: JWT is stateless, so manual implementation is needed for logout or force-expire scenarios.

Suitability

FastAPI with JWT and OAuth2 aligns tightly with GreenCart's modular architecture and constraint of ensuring POPIA/GDPR compliance. Statelessness enhances scalability for future user growth, and token-based auth simplifies frontend integration. The stack's testability and OpenAPI support also fulfill maintainability and integration quality goals.

Consideration 2: Firebase Authentication + Firebase Admin SDK

Overview

Firebase Auth provides a backend-as-a-service authentication solution, supporting email/password, phone, and federated logins (e.g., Google, Facebook). The Firebase Admin SDK allows server-side token verification and role management.

Advantages

- **Quick Setup & Integration**: Ideal for MVPs and frontend-driven projects. ● **Federated Login Support**: Handles Google, GitHub, etc., out of the box. ● **Integrated Auth + Database**: Works seamlessly with Firestore and Firebase Hosting.

Disadvantages

- **Vendor Lock-In**: Tightly coupled to Firebase infrastructure.

- **Less Transparent Logic**: Abstracts many implementation details, reducing control.
- **Limited Custom Roles**: Requires extra work for fine-grained RBAC.

Suitability

Firebase Auth offers rapid prototyping but limits backend control and extensibility. It conflicts with our open PostgreSQL schema and prevents fully transparent user access models, making it a less suitable option for GreenCart's long-term scale and compliance needs.

Consideration 3: Auth0

Overview

Auth0 is a commercial identity-as-a-service provider offering robust authentication and authorization flows with customizable UIs and strong security features.

Advantages

- **Enterprise-Grade Security**: Includes anomaly detection, MFA, and breach alerts.
- **Flexible Identity Providers**: Integrates with social, enterprise, and custom login systems.
- **Hosted Login Pages**: Reduces frontend complexity.

Disadvantages

- **High Cost at Scale**: Pricing becomes expensive after free tier.
- **Cloud Dependency**: Similar to Firebase, it introduces third-party reliance.

- **Customization Limitations**: Less flexible than self-managed solutions.

Suitability

Auth0 offers advanced features suitable for large enterprises but is excessive for GreenCart's current scope. It also doesn't offer the same level of code-level customization and full-stack ownership provided by FastAPI and JWT.

Final Choice: FastAPI + JWT + OAuth2

This combination was selected for its balance of performance, modularity, transparency, and security. It integrates directly with our existing FastAPI backend, supports role-based control and token-based authentication, and aligns with architectural patterns like SOA and client-server communication. Furthermore, it satisfies our data protection constraint under POPIA and GDPR, and allows detailed API documentation and testing via OpenAPI schemas.

4. Database Architecture

Consideration 1: PostgreSQL

Overview

PostgreSQL is an advanced open-source relational database system known for its robustness, extensibility, and adherence to SQL standards. It supports structured data with strong integrity constraints, transactional consistency (ACID), and rich query features—ideal for managing product metadata, user accounts, cart systems, and sustainability ratings in the GreenCart platform.

Advantages

- **ACID-Compliant Transactions**: Ensures data consistency during order processing and inventory updates.
- **Advanced Querying & Joins**: Supports complex relational queries, ideal for cross-referencing product ratings, sustainability data, and categories. ● **Extensibility**: Custom data types, indexing strategies, and full-text search features.
- **Strong Ecosystem & Tooling**: Compatible with ORMs like SQLAlchemy, admin

tools like pgAdmin, and scalable services like Supabase or Azure Database for PostgreSQL.

- **Relational Integrity**: Enforces schema consistency between users, products, orders, and carts.

Disadvantages

- **Vertical Scaling Limitations**: Without caching, high read traffic can degrade performance.
- **Schema Rigidity**: Less flexibility for rapid schema evolution compared to NoSQL.
- **Manual Optimization Required**: Indexing, query planning, and connection pooling need tuning at scale.

Suitability

PostgreSQL fits GreenCart's need for structured, interrelated data and strong guarantees on data integrity—essential for handling financial transactions, sustainability tracking, and user behavior analytics. Its compatibility with Supabase and Azure supports our deployment flexibility. The schema-first design also reinforces the data privacy and compliance constraints of the system.

Consideration 2: MongoDB

Overview

MongoDB is a popular NoSQL database that stores data in flexible, JSON-like documents. It is schema-less, allowing for rapid iteration and flexible data modeling—especially in early-stage development or for systems with unstructured data.

Advantages

- **Schema Flexibility**: Ideal for rapidly changing product or sustainability data formats.
- **Horizontal Scalability**: Easy to shard and scale reads/writes across clusters. ● **Developer-Friendly**: JSON-style document structure aligns with JavaScript-based

frontends.

Disadvantages

- **Weak Transactional Guarantees**: Multi-document transactions were added recently and are not as mature.
- **Data Duplication**: Denormalization can lead to inconsistent data. ● **Less Ideal for Relational Data**: Complex joins are discouraged or inefficient.

Suitability

MongoDB offers flexibility but is not well-suited for GreenCart's transactional and relational requirements. Features like sustainability ratings, product variants, and cart-item mappings demand strong referential integrity, which MongoDB handles poorly at scale.

Consideration 3: Firebase Firestore

Overview

Firebase Firestore is a NoSQL document database designed for real-time applications. It integrates tightly with Firebase Authentication and Cloud Functions, providing automatic scaling and reactive updates across clients.

Advantages

- **Real-Time Sync**: Client-side apps reflect database changes instantly.
- **Zero Server Management**: Fully managed by Firebase.
- **Integrated Security Rules**: Enforces access control directly within the database.

Disadvantages

- **Limited Querying**: Joins and aggregations are minimal or complex.
- **Vendor Lock-In**: Firebase's proprietary format limits portability.
- **Structured Relational Data Difficult**: Poor fit for systems like carts and orders.

Suitability

Firestore provides convenience but lacks the structure and transactional strength needed by GreenCart. It would also compromise our open-stack deployment goals and compliance with data laws like POPIA/GDPR.

Final Choice: PostgreSQL

PostgreSQL is the optimal choice due to its support for relational integrity, ACID transactions, extensibility, and mature tooling. It enables GreenCart to manage complex relationships between users, products, carts, orders, and sustainability ratings. Furthermore, PostgreSQL's compatibility with Supabase, Azure, and FastAPI enables smooth deployment and integration while ensuring security and compliance under POPIA and GDPR.

Deployment & Hosting

Consideration 1: Microsoft Azure

Overview

Microsoft Azure is a leading enterprise cloud provider offering a wide range of hosting, compute, and networking services. It's known for its deep compliance integrations, hybrid infrastructure support, and extensive tooling for CI/CD pipelines.

Advantages

- **Enterprise-Grade Infrastructure**: Ideal for large-scale, high-availability applications.
- **Strong Compliance Support**: Supports standards like POPIA, GDPR, ISO 27001. ● **Rich Ecosystem**: Deep integration with GitHub Actions, Azure DevOps, and Active Directory.
- **Flexible Deployment Options**: Supports containers (AKS), serverless (Azure

Functions), and VM-based hosting.

Disadvantages

- **High Complexity**: Steep learning curve, especially for small teams.

- **Resource Overhead**: Overkill for lightweight or MVP-style applications. ● **Student Limitations**: Free student resources can be restrictive and hard to configure.

Suitability

While Azure is enterprise-ready and secure, its complexity and overhead reduce its suitability for GreenCart's rapid, iterative development needs.

Consideration 2: Firebase

Overview

Firebase is Google's Backend-as-a-Service (BaaS) platform focused on frontend hosting, real-time databases, and mobile/web integrations.

Advantages

- **Instant Deployment**: Great for quickly pushing React apps live.
- **Integrated Auth & DB**: Works well with Firebase Auth and Firestore. ● **Developer Experience**: Easy CLI, generous free tier, and documentation.

Disadvantages

- **Limited Control**: Backend logic is restricted to Firebase Functions. ● **Vendor Lock-In**: Migrating away can be difficult once integrated deeply. ● **Not Ideal for Compute-Heavy Workloads**: Poor fit for custom backend or sustainability logic.

Suitability

Firebase may assist with frontend prototyping, but lacks the flexibility and backend control required for GreenCart's modular and scalable architecture.

Consideration 3: Supabase

Overview

Supabase is an open-source Firebase alternative built on PostgreSQL. It provides authentication, real-time updates, and a familiar SQL-based environment.

Advantages

- **PostgreSQL-Based**: Matches our existing backend schema.
- **Realtime Sync**: Enables live updates for dashboards and recommendations.
- **Open Source**: Avoids long-term vendor lock-in.

Disadvantages

- **Young Ecosystem**: Limited support for advanced deployments and fewer production-grade services.
- **Limited Compute**: Not suitable for complex ML or sustainability logic pipelines.

Suitability

Supabase fits the database structure and can support real-time features, but lacks the full suite of deployment and compute tools needed for future sustainability features.

Consideration 4: Amazon Web Services (AWS)

Overview

AWS is the industry-leading cloud platform known for unmatched scalability, rich service

offerings, and reliability. It supports full-stack application hosting, from storage and databases to container orchestration and serverless architecture.

Advantages

- **Unmatched Scalability**: Services like EC2, S3, RDS, Lambda, and Elastic Beanstalk support GreenCart's growth.

- **Granular Control**: Custom IAM roles, VPCs, and monitoring tools for secure, compliant deployments.

- **Wide Integration**: GitHub Actions, CodePipeline
- **Event-Driven Support**: Ideal for sustainability services that rely on user-triggered events and analytics.

Disadvantages

- **Learning Curve**: AWS services are complex and require configuration expertise.
- **Cost Escalation**: Without optimization (e.g., unused EC2), expenses can rise quickly.
- **Initial Setup Time**: Configuring IAM, VPCs, and security policies can slow early deployment.

Suitability

AWS aligns perfectly with GreenCart’s architectural vision:

- Supports microservices and event-driven services (e.g., carbon tracking and recommendation updates).
- Complies with POPIA and GDPR through secure infrastructure and data protection tooling.
- Enables serverless sustainability reports using Lambda and global content delivery via CloudFront.

Final Choice: Amazon Web Services (AWS)

AWS is the deployment platform of choice for GreenCart due to its ability to support:

- A microservices architecture (authentication, sustainability, orders).
- Stateless APIs with JWT authentication through FastAPI.
- Modular scaling and event-driven behavior (e.g., donation badge generation, sustainability impact reports).
- Integration with CI/CD pipelines for automated deployment via GitHub Actions.

While Firebase and Supabase may be considered for testing or prototyping frontend assets, AWS will host the production environment with its extensive scalability, flexibility, and future-readiness.

Service Contracts