ChatGPT

Implementation Guide for *The Open Box / La Boîte Ouverte* Marketplace

Welcome to the comprehensive technical and business implementation guide for **The Open Box / La Boîte Ouverte**, a bilingual (English/French) online marketplace for second-hand goods. This guide covers platform selection, system architecture, development steps, AI integration, automation workflows, DevOps, compliance considerations, hosting strategy, and a sample timeline with team roles. By following this guide, a development team should be able to implement the platform with minimal ambiguity.

Platform Selection and Justification

Recommended Platform: After evaluating modern e-commerce solutions, we recommend using **Saleor**, an open-source headless e-commerce platform, as the foundation of the marketplace. Saleor provides a robust GraphQL API backend (built with Python/Django) and a customizable React/Next.js frontend, offering a modern, scalable, and feature-rich base [1]. Key reasons for choosing Saleor include:

- **Multilingual & Multi-Currency Support:** Saleor supports product data in multiple languages out-of-the-box [2], which aligns with the bilingual English/French requirement. By contrast, alternatives like Medusa.js lack native multi-language fields [3]. Saleor also supports multiple currencies and regional settings (via its Channels feature), making it easier to localize prices and taxes per region.
- **Marketplace Capability:** While Saleor is primarily a single-vendor store engine, it can be extended to a multi-vendor marketplace. Saleor integrates with third-party marketplace systems such as **Jetti** to convert a single store into a multi-vendor marketplace [4]. This allows **The Open Box** to onboard multiple sellers and manage vendor-specific orders and payouts. (Alternatively, Medusa.js would require a custom marketplace plugin using Medusa's extensibility [5], but Saleor's approach leverages a proven service.)
- **Feature Completeness:** Saleor comes with built-in features like product catalog management, categories/collections, inventory, discounts, customer accounts, and order management. It supports variant products, attributes, and has an **admin dashboard** for managing the store. The admin panel is React-based and can be localized (it supports multiple languages in the admin UI) [6] [7].
- **Scalability and Performance:** Saleor is designed with a service-oriented, horizontally scalable architecture [8]. It can handle high traffic and large catalogs by scaling its GraphQL API and worker processes. The platform uses a modern tech stack (GraphQL, Django, PostgreSQL) optimized for performance. This ensures the marketplace can grow in user base and data volume without major refactoring.
- **Developer Experience & Customization:** Saleor's use of GraphQL API means front-end developers can query exactly the data they need, enabling fast and dynamic UIs. Developers can extend Saleor via its plugin/app system or the GraphQL API without forking core code. There is an active community and extensive documentation. Saleor is also provided as Docker images for easy setup and cloud deployment [9], simplifying DevOps.
- **Hosted Option:** If needed, Saleor offers a hosted cloud solution, but since we aim for an open-source stack, self-hosting is possible. (If the team prefers a Node.js solution and is ready to

implement missing features like i18n, Medusa.js is an alternative. Medusa is a headless Node/TypeScript platform with a flexible plugin system and built-in REST APIs [10] [11]. However, given the bilingual requirement and Saleor's out-of-the-box capabilities, Saleor is the more **turnkey choice**.)

Justification: In summary, **Saleor** is chosen as the core e-commerce engine due to its multilingual support, rich features, and scalability which perfectly suit the needs of *The Open Box / La Boîte Ouverte*. It provides a stable backbone so the development team can focus on customizations like the AI chatbot and automation workflows rather than reinventing core commerce features.

Solution Architecture Overview

Figure: Example headless commerce architecture (decoupled front-end and back-end, with integrations for database, third-party services, and AI/automation components). In our case, the Saleor backend (GraphQL API & database) serves a Next.js frontend, while additional services (n8n automation, AI chatbot, etc.) integrate via APIs.

The system architecture is designed as a **modular, end-to-end solution** where each component is decoupled yet interoperable. The high-level architecture for *The Open Box* marketplace is as follows:

- **Frontend:** A consumer-facing web application built with **Next.js** (React). This will be a **headless storefront** that communicates with the backend via API. Next.js is chosen for its support of SSR/SSG for performance, built-in internationalization (i18n) routing, and seamless deployment on Vercel. The frontend renders the product catalog, search results, product pages, shopping cart, checkout, etc., by querying the Saleor GraphQL API. It also includes the **chatbot UI** (chat interface on the site) for customer support. Additionally, an admin or seller portal front-end can be provided (either using Saleor's built-in dashboard or a custom admin UI) for managing products and orders.
- **Backend (E-commerce API):** The **Saleor** core application serves as the backend. It exposes a **GraphQL API** that the frontend and other services will use. Saleor handles business logic for product management, user accounts, shopping cart, orders, payment processing (via integrations), etc. The backend is containerized (Docker) and connected to a **PostgreSQL** database for persistent storage. All product data, user data, and transaction records reside in the database. The backend also includes background workers (for tasks like sending order confirmation emails, inventory updates, etc.). We will extend Saleor via its **App** mechanism or webhooks for custom needs (e.g., notifying the chatbot or triggering n8n flows on certain events).
- **Database: PostgreSQL** is used as the main database (the default for Saleor). It stores structured data for products, categories, users, orders, inventory, etc. We will enable **read replicas** or caching if needed for scalability [12]. For certain features like full-text search, we might integrate an Elasticsearch or use Saleor's built-in search capabilities (Saleor can integrate with Elastic via an extension, or we use Postgres full-text search for simplicity initially).
- **Automation Workflow Engine (n8n):** We incorporate **n8n** as a standalone service for automating product ingestion and other workflows. n8n is an open-source workflow automation tool with 400+ integrations and a visual editor [13]. We will deploy n8n (via Docker container or as a managed cloud instance) and use it to create **automated ingestion workflows** that pull in product data (images, descriptions) from external sources or through user submissions. For example, if bulk uploading products via a spreadsheet or integrating with another platform's API, n8n can fetch that data and then call the Saleor GraphQL API to create products. One community use-case is using n8n to *"scrape data from websites … to populate/update internal tables"* [14] – similarly, our setup can scrape or receive second-hand product info and create listings automatically. n8n will also be used for other

automations, like sending notifications or performing scheduled maintenance tasks (e.g., unpublishing stale listings).

- **AI Chatbot Service:** A dedicated AI service implements the **Retrieval-Augmented Generation (RAG) chatbot**. This is separate from Saleor for flexibility. The chatbot service consists of:
- A **Vector Database** (or document index) storing embeddings of relevant knowledge (product descriptions, FAQ, policies, etc.) for retrieval.
- Integration with a **Language Model** (such as OpenAI GPT-4 via API, or a self-hosted LLM) to generate responses. The chatbot uses RAG: it will retrieve relevant data from the knowledge base and feed it into the LLM prompt to ground the responses in factual information [15].
- This service can be built with Python (using frameworks like FastAPI for an API, and libraries like **LangChain or LlamaIndex** to handle RAG logic). The chatbot exposes an API endpoint (e.g., `/api/chat`) that the frontend calls when the user sends a message. It may also use WebSockets for real-time streaming responses.
- The chatbot service will interface with Saleor's API (or database) for certain queries – for example, to fetch live product info or inventory during a conversation, or to check a user's order status for support queries.
- **Third-Party Integrations:** Various external services will be integrated:
- **Payment Gateway:** We will use a PCI-compliant payment provider (e.g., **Stripe**) for handling credit card payments. The frontend will use Stripe's SDK (Elements/Checkout) so that sensitive card data is sent directly to Stripe, keeping our servers out of PCI scope. Saleor can store payment intents and transactions but not raw card data.
- **Email/SMS Service:** For notifications (order confirmations, password resets, etc.), integrate an email service (like SendGrid or AWS SES) and possibly SMS for 2FA or alerts. Saleor can trigger webhooks or use its plugins to send emails on events.
- **Image Storage & CDN:** Product images and other media will be stored on a cloud storage (like **AWS S3** or GCS) and served via a CDN (AWS CloudFront or a service like Cloudflare) for fast global delivery. This offloads media serving from our app servers and improves performance.
- **Logging & Monitoring Tools:** For system logs and metrics, we integrate services like **AWS CloudWatch** or **Elastic Stack** for logs, and **Prometheus/Grafana** for metrics. Additionally, **Sentry** will be used for error tracking on both frontend and backend [16].
- **Auth Services:** Saleor has its own authentication for customers and admin. For enhanced security and single sign-on options, we could integrate OAuth providers (Google, Facebook login for customers) via Saleor's APIs or use AWS Cognito if needed for user pools. (However, Saleor's built-in auth may suffice with email/password and confirmation flows.)

All components communicate primarily over secure APIs. The Next.js frontend talks to Saleor via GraphQL HTTPS calls; it talks to the chatbot via REST API or WebSocket; n8n workflows call Saleor or external APIs; Saleor sends/receives webhooks for events (e.g., to trigger n8n when a new product is added, or to allow the chatbot to retrieve certain info). This decoupled design follows the headless architecture principle – the front-end presentation is separated from back-end commerce logic, communicating only through API endpoints [17] [18]. Such separation provides flexibility to modify or replace components independently (for example, add a mobile app front-end later without changing back-end logic). According to AWS's well-architected guidance, a headless e-commerce app with decoupled UI and backend layers is ideal for scalability and agility [19].

High-Level Data Flow: A typical user request (e.g., viewing a product) goes from the browser to our Next.js frontend, which fetches product data via the Saleor API. The data is rendered and served to the user. If the user asks the chatbot, the query goes to the chatbot service, which retrieves relevant info (from Saleor's

data or policy docs) and responds with an AI-generated answer. If a seller uploads a product image and info, n8n picks it up (via a trigger such as an incoming email or an API call), then calls Saleor's API to create the product in the catalog. Each service is loosely coupled, communicating through secure endpoints and messaging, ensuring the system is **flexible, scalable, and maintainable**.

Automation Workflows with n8n for Product Ingestion

One standout feature of *The Open Box* is the **automated product ingestion** pipeline powered by **n8n**. This allows new second-hand product listings to be added to the catalog with minimal manual effort, by leveraging automation and AI to process images and descriptions.

n8n Setup: We will deploy n8n as a microservice (Docker container). It provides a web-based visual editor where we can design workflows that integrate various data sources and APIs. Some key workflows to implement:

- **Image & Description Ingestion:** Sellers can provide product information by either filling a form, emailing in details, or uploading to a cloud folder. n8n will be configured to trigger when a new input is received. For example:
- *Email ingestion:* n8n's IMAP trigger watches a mailbox for new emails with attachments. When a seller emails `listings@theopenbox.com` with a product image and description text, n8n retrieves the email.
- *Cloud Storage trigger:* Alternatively, if sellers upload images to a specific S3 bucket or Google Drive folder, n8n can trigger when a new file is added.
- *Web form/API trigger:* A custom submission form (or a Google Form/Sheet) could send data to n8n via a webhook.
- Once triggered, n8n will retrieve the image and text. We then use an **AI service within n8n** (leveraging n8n's integration with AI APIs or custom code) to enhance this data:
- **Image Recognition & Auto-tagging:** Pass the image to a computer vision API (e.g., AWS Rekognition or Google Vision) to detect the item type or features. This can suggest categories or attributes (e.g., brand, color, condition).
- **Description Enhancement:** If the seller provided a very short description, use an LLM (like GPT via n8n's OpenAI node) to generate a fuller product description or translate it to the other language. For instance, if the input description is in French, the automation can translate it to English as well, so we have both language fields populated.
- **Price Recommendation (optional):** Use pricing heuristics or an AI model to suggest a price based on item category and condition. (This could be a future enhancement where the system suggests a fair price range to the seller or auto-sets a price.)
- After processing, the n8n workflow uses an **HTTP Request node** to call Saleor's GraphQL API to create a new product with the gathered details. It will include:
- The product name (in EN/FR),
- Description in both languages (as product attribute translations),
- Categories/tags (from the image recognition step),
- Price,
- Images (the image can be uploaded to our S3 and the URL provided to Saleor or uploaded via Saleor's API if supported).
- **Moderation & Approval:** We can include a step for moderation – for example, the new product is created in a "draft" or "unpublished" state. An admin is notified (n8n can send an email or Slack

message to staff) to review the listing. Once approved, a separate workflow (or manual action in admin) can mark it as active. This ensures quality control.
- **Bulk Import:** If we have existing catalogs or partner inventories to ingest, we can create an n8n workflow to read from a CSV or external API (say another marketplace's feed) and loop through items to create products in Saleor. This is especially useful during initial seeding of the marketplace.
- **Keeping Data in Sync:** n8n can also run scheduled jobs to ensure data consistency. For example, a nightly job could check for products that have been sold or inactive and flag or archive them. Another could periodically validate that all products have both language descriptions, etc., and fix or report any gaps.

Using n8n in this way greatly reduces the manual workload. It essentially acts as an **ETL (Extract-Transform-Load)** pipeline for marketplace listings. This approach is inspired by how others have used n8n for similar automation, e.g., scraping product info and populating internal databases [14].

Deployment Considerations: The n8n service will have access to our Saleor API (using an admin API key for authentication on GraphQL) and to any third-party APIs needed (credentials stored securely in n8n). We will secure the n8n instance (basic auth or behind VPN) since it has powerful access. For reliability, n8n can be set up with its own small database (SQLite or PostgreSQL) to store workflow state.

Example Workflow: *A seller submits an item via email.* The email has subject "New Listing: Vintage Lamp" and body "Old lamp, good condition, works. $30". They attach a photo of the lamp. The n8n workflow triggers on email receive -> extracts the title, description, and photo -> calls Vision API to identify it as "Lamp" and condition "Used - good" -> the description is short, so it calls OpenAI to generate a more detailed description (and translate it to French) -> uploads the photo to S3 -> calls Saleor's `productCreate` GraphQL mutation with name "Vintage Lamp", English description, French description, category "Home Decor > Lighting", price $30, and image URL. Saleor creates the product (unpublished). The admin is notified on Slack to review. Once approved in Saleor's dashboard, the product goes live. This entire flow can happen in seconds automatically.

Through such automation, *The Open Box* can scale its catalog ingestion and ensure bilingual content for each listing without burdening staff with manual data entry.

AI Chatbot with RAG (Design & Capabilities)

A flagship feature of *The Open Box / La Boîte Ouverte* is its **AI-powered chatbot** that provides interactive customer support in both English and French. This chatbot uses **Retrieval-Augmented Generation (RAG)** to ensure accurate and contextually relevant responses, combined with negotiation logic to handle price bargaining. Below we detail the design and capabilities of this AI chatbot:

Objectives of the Chatbot:

1. **Answer Customer Queries:** The bot will instantly answer questions about products (e.g., "What are the dimensions of this table?"), shipping ("How long does delivery take to Toronto?"), return policies, payment options, and other FAQs. Instead of making customers read through policy pages, the bot will fetch the relevant information and provide a clear answer.
2. **Guide and Inform:** It can act as a shopping assistant – helping users navigate the site ("Show me available smartphones under $200"), recommending products, and checking stock availability.

3. **Negotiate Prices:** Uniquely, the chatbot can engage in **price negotiation** for second-hand items, within limits defined by the business rules or sellers. This adds a "haggling" feature to the marketplace experience, simulating how one might bargain in person. The bot will follow predefined discount rules to ensure price offers remain within acceptable margins.
4. **Multilingual Conversations:** The chatbot can seamlessly interact in English or French, detecting the user's language and responding accordingly. This ensures a native experience for users in both language demographics.

RAG Architecture: Retrieval-Augmented Generation means the bot combines a knowledge search with generative AI. Instead of relying solely on an LLM's trained knowledge (which might be outdated or hallucinate), it **retrieves** relevant data from an external source and uses that to ground its answers [20] [15]. Our implementation includes:

- **Knowledge Base:** A collection of documents and data that the bot can draw from. This includes:
- Product information: titles, descriptions, specs, and possibly recent pricing or stock info. We will index all product descriptions (likely the English version, and possibly French as separate entries) in a vector database for semantic search.
- Company policies: shipping policy text, return policy, terms and conditions, privacy policy – these will be indexed so the bot can quote exact policy language when asked.
- FAQs and help articles: We can compile an FAQ document (in both languages) and include it in the knowledge base.
- Conversation history: To some extent, the bot can consider prior conversation context (limited to avoid long prompts) for continuity.
- **Vector Store:** We will use a vector database (such as **Pinecone**, **Weaviate**, or an open-source solution like **Chroma** or **FAISS**) to store embeddings of the knowledge base texts. When a user asks a question, the bot generates an embedding of the query and performs a similarity search in the vector DB to retrieve the top relevant pieces of context (e.g., the paragraph of the shipping policy that mentions "2-5 days in Ontario", or the product description of the item in question).
- **LLM (Generative Model):** We will utilize a Large Language Model to actually generate the answer. This could be an OpenAI GPT-4 API (which is state-of-the-art and multilingual), or a hosted open-source model (like LLaMA 2 fine-tuned for chat) if we want to avoid external dependencies. The LLM is prompted with a combination of the user's question (in their language) and the retrieved context documents. Because the model sees relevant context, it can produce a correct and specific answer, citing the details from our knowledge base. *This approach improves factual accuracy by grounding the model's output in real data* [21].
- **Multilingual Handling:** The retrieval step can be done either in a single language or multilingual. One approach: maintain the knowledge base primarily in English for internal retrieval, and if the user asks in French, translate the query to English for search, retrieve English context, then prompt the LLM to respond in French. Alternatively, we index both English and French content so that it can retrieve in the same language. Given modern models can handle bilingual prompts well, we might use the translate approach for simplicity. The LLM (especially GPT-4 or similar) can output in whichever language the user communicates [22] [23], ensuring a fluent, grammatically correct response in that language.
- **Negotation Module:** For price bargaining, the bot will be augmented with rules. Each product (or product category) will have negotiation parameters defined, such as:
- **Min Price** – the lowest price the bot is allowed to offer (could be set by the seller or as a percentage of listing price).
- **Max Discount** – e.g., at most 15% off, or $20 off, etc.

- **Negotiation Strategy** – e.g., whether to allow counter-offers and how many rounds of negotiation.

When a user attempts to negotiate ("Is the price negotiable?" or "Will you take $50 for this item?"), the bot will check these rules from the database. The negotiation logic can be implemented as a function that the chatbot service calls. For instance, if item is listed at $100, min price $80: - If user offers $60 (which is below $80), the bot will refuse that offer, perhaps counter with the minimum $80: *"I'm sorry, I cannot go that low. The best I can do is $80."* - If user offers $85, the bot might accept (since it's above min) or counter slightly higher (e.g., split difference: $90) to simulate a bargain and then settle at $85 if user persists.

The chatbot's LLM prompt will incorporate the price rules, e.g., "The item's price is $100, minimum allowed is $80. If user's offer >= 80, accept or counter within that range...". This way the generative model will produce a response that follows the logic. Alternatively, we use a more deterministic approach: have the application logic determine the next price and then just have the LLM phrase the response politely.

Research Basis: Negotiation chatbots have been shown to increase customer engagement and satisfaction by allowing customers to feel they got a better deal [24]. The negotiation is bounded by predefined limits – the seller's minimum price and the original price set – which define the range for the algorithm [25]. Our bot will automate this negotiation process, simulating a human-like haggling experience while ensuring the final price never goes below the seller's minimum [25] [26].

- **Conversation Flow & Memory:** The chatbot will maintain short-term memory during a session (e.g., using the conversation ID or user session ID). This means if a user asks *"Can I get a discount?"* and then follows up *"How about free shipping?"*, the bot remembers context (that they are talking about the same product). We will use either the LLM's ability to take conversation history or manage a memory state in our service (like storing last asked product). However, to prevent long conversations causing context overflow, we might summarize or limit the history we send each time.
- **Fallback and Safety:** The bot will have fallbacks for when it cannot find an answer (e.g., a very obscure question) – it might respond with a polite default or forward the query to a human agent. We will also implement safety filters: the bot should refuse to answer inappropriate requests and comply with ethical guidelines. Using OpenAI's models helps as they have built-in moderation, but if self-hosted, we'll incorporate content filtering on user inputs.

Example Interactions:

- *Q:* "**What condition is the *Canon EOS 80D camera* in?**"
  Bot retrieval: finds the product "Canon EOS 80D" description which says "Excellent condition, barely used, comes with original packaging."
  Bot answer: "This Canon EOS 80D camera is listed as being in excellent condition, with very minimal use, and it includes the original packaging." (Multilingual: If asked in French, it answers in French: *"Il est indiqué en excellent état, très peu utilisé, avec l'emballage d'origine."*)
- *Q:* "**What's your return policy?**"
  Bot retrieval: fetches the Returns & Refunds policy from knowledge base (e.g., "Returns accepted within 30 days of delivery, buyer pays return shipping if item as described...").
  Bot answer: "You can return items within 30 days of delivery as long as they are in the same condition. Note that the buyer is responsible for return shipping costs, unless the item was not as described [27]. I can send you the full policy page if you'd like."
- *Q:* "**I live in Montreal. How much is shipping?**"
  Bot: The bot can either use a shipping rates table or call an API. Suppose standard shipping to QC is

$10 for that item. It answers: “Shipping to Montreal for this item would be approximately $10 and it typically takes 5-7 business days to arrive.”

- *Q:* “**The price is $200 – will you take $150?**” (Negotiation on a product with min price $180)
  Bot logic: $150 is below $180 minimum, so bot should counter.
  Bot answer: “I’m afraid $150 is a bit low for this item. I can offer it for **$180** if that works for you.”
  User: “How about $170?”
  Bot: Now $170 is still below min $180, so bot holds firm at $180 (or might drop slightly to show compromise, but not below $180).
  Bot answer: “I’ve already applied the biggest discount I can for you – $180 is the lowest I can go. This is a great deal for the condition and value of the item.”
  (If the user agrees, we could generate a one-time discount code or adjust the price in the cart for them. The bot could respond with something like “Great! I’ve applied the discount – you can checkout for $180 now.” Implementation-wise, upon agreement, we could have the bot trigger an action via Saleor’s API to create a custom discount for that user’s cart. This is an advanced feature that can be added.)

This AI chatbot will greatly enhance user engagement on the platform. By providing quick answers and a personalized experience (including the thrill of negotiating a deal), it can increase conversion rates and customer satisfaction [28] [24]. Moreover, being multilingual ensures we cater to both English and French speakers equally well, which is crucial for our market [23].

Under the hood, the combination of RAG and controlled generation allows the bot to stay accurate and up-to-date (e.g., always using current product data rather than potentially outdated training data) [29] [30]. Because the knowledge base can be updated continuously (new products, changed policies), the bot’s responses remain current without retraining the model – only the index needs updating, which is straightforward to automate [31]. This architecture ensures that as *The Open Box* grows, the AI assistant will scale and remain a trustworthy, useful tool for all users.

Frontend Development Guide (Web App)

The frontend of *The Open Box* is a critical component, as it directly interfaces with users. We will develop a modern, responsive web application with an emphasis on performance, accessibility, and bilingual user experience. Below are step-by-step development instructions and best practices for the frontend:

1. **Scaffold the Project:** Initialize a new **Next.js** project (using `create-next-app`) with TypeScript. Next.js is ideal for our use-case because it supports hybrid rendering (static generation for catalog pages, server-side rendering for dynamic pages), and has built-in support for internationalized routing. The project structure will be organized into pages (or use the App Router with server components as appropriate) for main views like Home, Product Listing, Product Details, Cart, Checkout, User Profile, etc. We will also create custom components as needed (e.g., ProductCard, NavBar, Footer, ChatbotWidget).
2. **Integrate Saleor GraphQL API:** Set up Apollo Client or URQL to interact with the Saleor GraphQL endpoint (`/graphql/`). Generate TypeScript types from Saleor’s GraphQL schema for type-safe queries (Saleor provides an API playground and docs for queries [32]). Create a client instance that points to the backend URL (for local dev, `http://localhost:8000/graphql/` if Saleor runs on 8000). Handle authentication for API calls: for public data (products, etc.), Saleor’s API can be queried without auth (if permissions are set accordingly). For user-specific data or mutations (adding to cart, etc.), you’ll use the user’s auth token (obtained on login) in the headers. Next.js can use API routes or

server-side functions to keep secrets if needed (like an API token for server-to-server communication).

3. **Routing and Pages:** Implement pages for all major flows:
4. **Home Page:** Highlights categories or featured products, search bar, etc. It will fetch some featured listings via GraphQL (e.g., list of latest products or best deals).
5. **Category/Browse Pages:** Pages showing lists of products, possibly paginated or infinite scroll. We can use Saleor's filtering and pagination queries to fetch products by category, price range, search keyword, etc.
6. **Product Detail Page:** Displays product images, description (toggle or tabs for English/French description if needed or auto-switch based on user language), price, condition, seller info, and a button to add to cart or make an offer (if negotiating via chatbot or offer system). It will query the product by ID or slug from Saleor. If the user is viewing the French site, it will display the French fields (Saleor's API can return translated fields if requested).
7. **Cart Page:** Shows items user added, allow update quantity or remove, and proceed to checkout. Cart state can be managed via Saleor's cart API (Saleor uses the concept of checkout or order creation via API) or managed locally then submitted.
8. **Checkout Page:** Collects shipping info, selects shipping method, and payment. Here we integrate the Stripe payment. Likely, we'll redirect to a Stripe Checkout session for simplicity. Alternatively, use Stripe Elements for card input within our site (which requires setting up client and server payment intents). Using Stripe's hosted checkout is simplest and keeps us most PCI compliant.
9. **User Account Pages:** For login/register (though could be modals), profile details, order history. Saleor's API has mutations for authentication (obtain token) and queries for orders by user.
10. **Admin/Seller Pages:** If we allow sellers to login and manage their items, we might create a basic seller dashboard (or simply direct them to Saleor's own dashboard app). For phase 1, an admin can use Saleor's React Admin app (Saleor has an official admin interface separate from the storefront [33] [34]).
11. **Internationalization (i18n):** Configure Next.js i18n with locales `en` and `fr`. Determine locale either from sub-path (`/en/` vs `/fr/` in URL) or domain (if using separate domain per language). All UI strings need to be translated – use a library like `next-i18next` or Next.js built-in i18n routing with JSON message catalogs. We will create English and French translation files for all static text (navigation labels, form labels, etc.). For dynamic content (product data), Saleor already provides the product fields in both languages via API. Ensure to request the correct language fields based on current locale.
12. Provide a language switcher on the site so users can toggle language.
13. Make sure direction is LTR for both (since French/English both LTR).
14. Dates, numbers, currency formats should be localized. Use locale-specific formatting (e.g., `$1,234.56` vs `1 234,56 $`). The currency will likely be CAD or USD – ensure using appropriate symbol and comma/decimal formats via Intl API.
15. **State Management:** Use React context or a state management library (like Zustand or Redux) to handle global state such as the shopping cart (if not fully relying on backend cart) and the user's auth status. Next.js can also leverage server-side to initialize cart data on page load.
16. **Chatbot Integration (Frontend):** Implement a chat widget component – e.g., a button that expands a chat window overlay. This will interface with our AI chatbot service. Possibly use a WebSocket or polling for real-time stream of responses. When user sends a message:
17. The frontend calls our chatbot API endpoint (for example, a Next.js API route that proxies to the chatbot service).
18. Display the user's message in the chat UI, and show a loading indicator while awaiting response.

19. Stream in or display the bot's response. Support markdown or rich text (if the bot responds with links or lists).
20. If the conversation includes an "offer" (negotiation), the bot might send a special payload (e.g., "offer price accepted: $X"). The frontend could then reflect that in the UI (for instance, generate a discount code or adjust the cart price). This requires some additional handling in the protocol between bot and front-end (maybe a simple pattern like if the bot message contains "OFFER_ACCEPT" token, we know to act on it).
21. Ensure the chat interface is bilingual too – if the user is on French site, initial greeting is in French, etc.
22. **UI/UX and Theming:** Develop a responsive UI following modern design practices. Since it's a marketplace, the design should emphasize product images and simple navigation. We might use a UI library or CSS framework (TailwindCSS or Material-UI) to speed up development. Key considerations:
23. **Responsive Design:** Test on mobile, tablet, desktop. Ensure grids of products adjust, navigation becomes a burger menu on mobile, etc.
24. **Accessibility (WCAG):** Use semantic HTML for all components (e.g., `<button>` for clickable, proper form labels). Ensure color contrasts meet WCAG AA guidelines (important for users with visual impairments). All images (especially product images) should have appropriate `alt` text (which can be dynamically set to the product name). The site should be navigable via keyboard (e.g., for modals, ensure focus trapping, etc.). This not only widens the audience but also is legally important in many regions (and simply good practice).
25. **Performance:** Use Next.js features like Image Optimization for product images (serving responsive sizes via `<Image>` component). Enable caching for data fetching where possible. Use code-splitting such that heavy components (maybe admin pieces or chatbot logic) don't bloat initial bundle for users who might not use them.
26. **SEO:** Ensure server-side rendering of important pages (home, category, product) so that crawlers can index content. Use proper meta tags and JSON-LD structured data (for products) to improve search appearance.
27. **Testing the Frontend:** During development, write component tests (using Jest/React Testing Library) for critical UI logic. Also perform end-to-end tests using a tool like **Cypress** to simulate a user flow: browsing, adding to cart, logging in, and checking out. This ensures the integration with the backend is working (possibly use a staging environment of the Saleor API or mock responses).
28. **Iteration:** Collect feedback on the UI/UX, especially for bilingual usage. Perhaps run tests with both English and French users to ensure the language toggle is intuitive and all content is translated. Optimize any workflow that seems slow or clunky (e.g., refine search autocompletion, etc.). Also, style the chatbot nicely so it's a welcoming feature (e.g., a chat bubble icon, with company mascot maybe).
29. **Deployment of Frontend:** We plan to host the frontend on **Vercel** for its excellent support of Next.js. We will connect the repository to Vercel so that every push to main triggers a deployment. Configure environment variables on Vercel for any needed API endpoints (Saleor's URL, chatbot URL, Stripe public key, etc.). Vercel will handle CDN and edge caching for us, making the site globally fast. Alternatively or additionally, we could export a static version of parts of the site (Next can pre-render many pages if data is known at build) and serve via CloudFront or Netlify. But using Vercel will likely suffice and simplifies the process.

By following these steps, the front-end development will produce a high-quality, maintainable web application. The choice of Next.js + React aligns well with Saleor's headless setup (Saleor itself often pairs

with a Next.js storefront [35]). Our front-end will deliver a polished user experience, from browsing items to interacting with the AI assistant, all while being performant and accessible to a broad audience.

Backend Development Guide (E-commerce & Server Logic)

The backend is primarily powered by the **Saleor** platform, which will be customized and extended to meet the marketplace requirements. This section provides step-by-step guidance on setting up and developing the backend, including the core e-commerce functionality, integrations, and any custom server-side logic.

1. **Initial Setup of Saleor:** Begin by setting up a development instance of Saleor.
2. Clone the Saleor repository (or use the Saleor CLI). Saleor provides a Docker Compose setup for local development [9] . Using Docker, spin up the services: the Saleor API (Python/Django server) and a PostgreSQL database. Verify that you can access the GraphQL API (e.g., at `localhost:8000/graphql/`) and the Saleor dashboard (if running the separate saleor-dashboard).
3. Run database migrations to create the schema. Seed the database with initial data: create at least one example category, and one admin superuser.
4. Check the Saleor configuration (it uses environment variables for settings): set the `ALLOWED_HOSTS`, database connection, email backend (for dev you might use console email backend), etc. Also, configure media storage (in dev, local; in prod, will use S3 via settings).
5. **Core Configuration:**
6. **Multi-Language**: Enable multiple languages in Saleor's settings. Saleor by default might support this via the `LANGUAGES` setting or via its Site settings in dashboard. We'll ensure English and French are active. This will allow entering translations for product fields.
7. **Currencies**: Configure currencies (Saleor supports multi-currency by creating separate channels or listing multiple currencies per channel). For example, define a channel "Canada" with currency CAD, and maybe "USA" with USD if needed. Since bilingual likely targets Canada, we might stick to CAD but still show currency symbol properly for French (which is still $ but can use locale-specific formatting).
8. **Taxes**: Decide on how to handle taxes. Saleor can be configured to use **VAT (Value added tax)** style or manual taxes. For Canada, likely we'll not include tax in prices and just treat prices as final (since second-hand might not always charge tax, but that depends). If needed, integrate a tax service or define tax rates per province. Saleor has a plugin interface for taxes (e.g., an Avalara plugin for automated tax calculation in some editions).
9. **Shipping Methods**: Set up shipping zones (e.g., domestic Canada, perhaps US/international if needed). Define shipping methods (flat rate, weight-based, etc.) and their costs. This is configured via Saleor dashboard or API.
10. **Payment Gateway Integration**: Saleor supports payment gateways through a plugin system. We will integrate **Stripe**: configure the Stripe plugin with API keys in Saleor's settings (Saleor's documentation provides steps to add Stripe, Braintree, etc.). Ensure in dev you use test keys. This will allow the checkout to create payment intents via Stripe's API. Alternatively, we can bypass Saleor's payment plugin and handle payment entirely on front-end (Stripe Checkout) and just record the payment status via a webhook.
11. **Emails & Notifications**: Configure SMTP or an email service for sending order emails. Saleor can send order confirmation emails if properly configured. We might hold off initially and use our own system or just rely on transactional emails from the payment gateway.
12. **Marketplace (Multi-Vendor) Customization:** By default, Saleor doesn't have separate vendor accounts for a marketplace. We have two main approaches:

13. **Use Jetti (recommended)**: Jetti is a third-party service that overlays multi-vendor on platforms. If using Jetti, we'd integrate via API and likely handle vendor onboarding through Jetti. Given this is a technical guide, we will outline a lighter approach as well:
14. **Custom Vendor Field**: Extend Saleor's data model to mark which user is the "seller" of a product. For example, use Saleor's attribute or metadata on Product to store a vendor ID (which corresponds to a User who is a seller). Also, create a group or permission in Saleor for "Vendor" that allows limited dashboard access (only to their products/orders). Saleor has a notion of staff accounts with restricted permissions; we could create a staff user for each vendor, restricted to managing their products (though Saleor's permission granularity might not support per-product restrictions easily). Another simpler method is to not give vendors direct dashboard access initially, but manage their listings via the ingestion workflows or a simplified custom portal.
15. Implement logic so that when an order is placed, if items belong to multiple vendors, the system splits the order for fulfillment/tracking purposes. This is non-trivial: Saleor doesn't natively split one checkout into multiple vendor orders. Jetti or a custom solution is needed to route orders to the right vendor for fulfillment. For an MVP, we might enforce one order = one vendor (shopping cart constrained to one vendor at a time) to avoid complexity.
16. Ensure that financials for vendors are tracked (Saleor can record payments, but paying out to vendors might be off-platform – perhaps handled by Jetti or manually).
17. In summary, for a full-fledged marketplace, integrating a service like Jetti (which connects via Saleor's APIs to manage multi-vendor logic) is the robust path [4]. For this guide, we acknowledge this need and plan accordingly, even if initial implementation might treat the marketplace in a simpler fashion (e.g., the site itself is the seller for all items and later attribute them to users).
18. **Product Catalog & Data Entry:** Use Saleor's dashboard or GraphQL mutations to set up initial categories (e.g., Electronics, Furniture, etc.), and example products. This helps validate that bilingual fields and everything work. The admin user can use the Saleor admin UI to input translations for product info (or use the API if needed).
19. If using the n8n ingestion, now is the time to test creating a product via API and see that it appears with correct data.
20. Ensure that all necessary product fields are present (Saleor allows additional metadata on products if we need to store custom fields like "condition" of item or "vendor id". We can add a ProductType and assign attributes for Condition, etc.).
21. **AI Chatbot Backend Integration:** The chatbot service (described in the AI section) will need to interface with Saleor:
22. **Read Operations:** The chatbot may call Saleor's API to get product details (e.g., if the user asks about a product by name or ID). We can create a restricted saleor API token for the chatbot to use for queries only. Alternatively, we can nightly export product data to the chatbot's vector index to reduce real-time calls.
23. **Price Negotiation Hooks:** If we implement the price adjustments via the platform, we could use Saleor's discount code system. For example, when the bot and user agree on a price, the chatbot service can call a Saleor mutation to create a **Voucher (promo code)** of a specific amount for that product for that user. Or it can initiate a draft Order with a manual discount. This requires careful use of Saleor's API (ensuring not everyone can use that discount, etc.). A simpler way is the bot says "Ok, we have a deal at $X. Please proceed to checkout and use code ABC123 to get that price." and we generate a one-time code. Implementing this will involve writing a small function in the chatbot backend that uses Saleor's GraphQL to create a voucher (Saleor allows vouchers limited to email or user, or limited to a specific product or variant).
24. **Webhooks:** Configure Saleor to send webhooks for events like Order Created, so that the chatbot (or another service) could notify the seller or just log interactions. Not strictly needed for MVP, but worth

noting. Also, if we want the chatbot to greet logged-in users by name or reference their last order, it might query user info (within privacy limits).

25. **Custom Business Logic:** Some aspects might require extending Saleor beyond configuration:
26. If we want to enforce that users can only checkout items from one vendor at a time, we might need to add validation in the checkout process (could be done on frontend or via a checkoutCreate mutation wrapper).
27. If we want to automatically mark an order as paid once Stripe confirms, ensure the webhook from Stripe calls Saleor's payment API to update order status (Saleor's Stripe plugin should handle this).
28. Implement any data transforms needed for the front-end: for example, a small proxy to convert GraphQL responses to a simpler shape if needed, but likely not needed as front can consume GraphQL directly.
29. Setup logging of important actions (maybe integrate Sentry for exceptions in the Saleor backend).
30. Monitoring: Integrate APM (like NewRelic or open-source alternatives) if needed to trace performance.
31. **Testing the Backend:**
32. Write unit tests for any custom logic added (Saleor itself comes with its test suite, but our extensions and configurations should be tested). For example, if we add a custom function for negotiation or voucher creation, test it with various scenarios.
33. Test GraphQL queries from the frontend perspective. Ensure a product query returns both languages as expected. Test that an unauthorized user cannot do restricted actions.
34. Integration test: Run the whole system locally and simulate a full purchase: user signs up, adds product to cart, checkout with Stripe test card, ensure Saleor marks order paid, stock decremented, email sent. Also test a negotiation flow: user asks bot for discount, bot generates code, user applies code in checkout, sees reduced price.
35. **Security Hardening:**
36. Make sure to disable any default credentials, and ensure the admin account has a strong password.
37. If exposing the GraphQL API, consider rate limiting or depth limiting queries to prevent abuse. (Saleor might have built-in protections or can be configured with GraphQL query cost analysis).
38. Ensure that media uploads are secured (Saleor can be configured to only allow certain file types for upload to prevent malicious files).
39. Backup strategy for the database: set up regular backups (if using AWS RDS, enable automated backups and snapshots).
40. If running on AWS, use Secrets Manager or environment variables for sensitive config (API keys, database passwords) rather than hardcoding.
41. **DevOps for Backend:** We will containerize the Saleor app for production deployment. Write a Dockerfile (if not using provided one) that encapsulates the Saleor API (gunicorn server). Use multi-stage to build assets if any (Saleor's admin is a separate frontend app – if we use it, it might be deployed separately or served by the backend via static files). Ensure the image is minimal and secure (use Python slim base image).
42. Prepare a Kubernetes manifest or Docker Compose for production with Postgres, Saleor app, possibly a Redis (Saleor can use Redis for caching or background tasks).
43. Alternatively, one can use AWS Elastic Beanstalk or ECS to deploy the container. We'll detail deployment in a later section.
44. **Extending to Production Data:** When going live, we'll import actual product data. The n8n workflows will likely do a bulk import. Make sure the system can handle a large number of products

and users. We might need to enable search indexing (Saleor might integrate with Elastic; if not, at least ensure database indexing on name/sku for search queries).
    - Also plan for media storage in cloud: configure the S3 bucket and credentials in Saleor's settings so that when images are added (via dashboard or API), they are stored in S3 and served via CDN.

Throughout backend development, keep in mind maintainability and upgradability. Since Saleor is actively developed, we should avoid modifying core code and instead use its extension mechanisms (apps, webhooks, API) to implement custom features. This way, we can upgrade the Saleor version in the future to get security patches and improvements, without losing our custom logic.

By leveraging Saleor for the heavy lifting of e-commerce features and focusing our custom development on marketplace-specific needs and AI integration, we ensure a robust backend. The backend will enforce business rules (like negotiation limits, stock availability), handle transactions securely, and provide data to the frontend efficiently through the GraphQL API. It's the engine that will keep *The Open Box* running smoothly behind the scenes.

AI Services Development Guide (Chatbot & Automation)

In this section, we provide a focused development guide for the AI components – namely, the chatbot service with Retrieval-Augmented Generation and the integration of AI into automation workflows (like generating product descriptions). These services are separate from, but closely integrated with, the main application.

1. Setup the AI Chatbot Service:

We will implement the chatbot as a standalone service (e.g., a Python FastAPI app). Key steps:

- **Environment & Libraries:** Set up a Python virtual environment. Use libraries such as:
- `fastapi` (to create API endpoints for the chatbot queries),
- `uvicorn` (to run the server),
- `langchain` or `llama_index` (to facilitate building RAG pipelines),
- `transformers` or OpenAI SDK (to interface with an LLM, if using open-source model or OpenAI API),
- `faiss` or `chromadb` (for vector search if we self-manage it, or use `pinecone-client` if using Pinecone).
- **Knowledge Data Pipeline:** Write a script or use langchain's utilities to **ingest data** into the vector store. This involves:
- Export product data from Saleor: we can write a script to fetch all products (name, description in EN and FR, perhaps concatenating both language texts or storing separately). Alternatively, read from the database or use Saleor's GraphQL to get all products.
- Load policy and FAQ documents: e.g., have markdown or text files for "shipping_policy_en.txt", "shipping_policy_fr.txt", etc.
- Split these texts into chunks (maybe 100-200 words per chunk) for embedding.
- Compute embeddings: Use a pre-trained multilingual model like `sentence-transformers` (e.g., model `all-MiniLM-L6-v2` for English, or a multilingual model like `distiluse-base-`

`multilingual-cased`) to vectorize each chunk [36]. Store the embeddings with references to their source (which document/product and maybe an ID).
- Either push these into a **vector database** (if using Pinecone: create an index and upsert vectors; if using a local FAISS, build the index and serialize to disk; if using Chroma, it manages storage).
- This ingestion script should be rerun whenever data updates (or better, set up a webhook from Saleor so that when a product is updated, we update the index for that product).
- **API Endpoint:** Implement an endpoint `/chat` which expects a POST with user message (and perhaps a conversation ID or history). The handler will:
- Detect language of the user query (could use a simple approach like checking for presence of French-specific characters or use `langdetect` library).
- If French and our knowledge base is primarily English, translate the query to English using a translation model or API (or possibly just do cross-lingual embedding if using a multilingual model).
- Perform the vector search: find top 3-5 relevant chunks from the index.
- Compose the LLM prompt. A recommended structure: include a **system prompt** that sets the chatbot's role and guidelines (e.g., "You are an assistant for an online marketplace. Answer questions truthfully using the provided information. If negotiating, follow the pricing rules given. Respond in the user's language."). Then include a brief summary of relevant context: e.g., for each retrieved chunk, add it as a quote or an excerpt with reference. Then user's question goes at the end.
- Call the LLM (e.g., OpenAI's API with the composed prompt, or if using local model, run it through the `transformers` pipeline). If using OpenAI, ensure to set model (e.g., `gpt-4` or `gpt-3.5-turbo`), and maybe use function calling feature for structured outputs (though not strictly needed).
- Get the response. Parse it if needed (for negotiation, perhaps define a format like the bot should respond with a JSON if it's an offer, but more simply, we'll parse text).
- Return the response back to the frontend.
- **Negotiation Logic in Backend:** To support negotiation without relying solely on the LLM, implement a utility function `calculate_counter_offer(product_id, user_offer)`:
- Fetch product's base price and min price (store min price in product metadata or a separate mapping).
- If `user_offer >= base_price`, the user is essentially offering full price or more (which is rare); the bot can just accept: counter = user_offer (or base_price if user overpaid).
- If `user_offer < min_price`, then counter_offer = min_price (or maybe min_price + a small buffer to leave room for one more negotiation).
- If `user_offer` is between min and base, then perhaps counter at something like halfway between base and user_offer, or just a small discount from base. For simplicity, we could say if user_offer is reasonable (within, say, 10% of base), accept it; otherwise, counter at maybe (user_offer + base_price)/2 (rounded).
- This function returns a number (or flag that offer is too low to even counter, etc.).
- The chatbot prompt can incorporate the result: e.g., "User offered $50, you should counter with $60 as that's the minimum acceptable." or if the function says accept, then prompt says "User's offer is acceptable, go ahead and accept it."
- This way the final decision is rule-based, and the LLM's job is just to word the response.
- **Testing Chatbot:** Simulate various inputs by calling the `/chat` endpoint with sample questions. Ensure that:
- With known product query, it finds and returns info correctly.
- With a policy question, it cites the right policy.
- In French queries, it responds in French.

- For negotiation, try different offer values and see that the response aligns with the rules (not giving too high discount).
- If the LLM gives any hallucinated or incorrect response, adjust the system prompt to be more strict (like “If you don’t know, say you’ll refer to a human” etc.), or ensure more relevant context is included.
- **Performance:** The chatbot’s response time depends on the LLM. Using GPT-4 might be a 1-2 second API call typically. Vector search is fast (ms range). This is acceptable for a support chat. If volume grows, consider running multiple instances or using a streaming response to show partial answer.
- **Multilingual AI Note:** Our approach uses one model for both languages. GPT-4 for example can handle both languages in one prompt. The retrieval in English and then answering in French works because the model can translate/integrate context. If using an open model, consider one like `mGPT` or a fine-tuned bilingual model.

2. AI in n8n Workflows (Product Description AI):

We already touched on using AI in the ingestion process. To implement that: - For image recognition, use an API like Google Vision. In n8n, you can use an HTTP node to call Google Vision API with the image (requires Google Cloud Vision API enabled and credentials). Parse the JSON response for labels. - For description generation or translation, use n8n’s built-in OpenAI node or an HTTP request to OpenAI’s endpoint. Provide a prompt like: “Here is a brief description: 'Old lamp, works well'. Expand this into a compelling product description (50-100 words) for a marketplace listing, in English. Then provide a French translation.” The AI will return text which we parse (could even ask it to format output as JSON with `{"en": "desc in English", "fr": "desc in French"}` for easy parsing). - Alternatively, use a local model: n8n can run custom JavaScript, so one could theoretically call a local AI service (like our chatbot service if it had a route for description generation without retrieval). - Ensure these AI calls have fallbacks (if API fails or times out, maybe just use the original description and mark that it needs manual editing later). - Test the output quality with a few samples. Make sure it doesn’t produce inaccuracies (for instance, the AI shouldn’t add features the product doesn’t have). - Also, run a **content filter** on AI outputs: e.g., ensure no offensive or policy-violating text is generated (OpenAI’s API usually won’t for such prompts, but good to be cautious).

3. Deployment & Integration:

- Containerize the chatbot service (Dockerfile with Python environment). This will run alongside other services in production. Expose its port (maybe behind an internal reverse proxy or as an AWS ECS service).
- Secure the chatbot API – at least use an internal token or network isolation so that only the frontend (or our site) can use it, not the public (to avoid misuse).
- For n8n and AI, since they involve third-party API keys (OpenAI, etc.), store those keys securely (as secrets in n8n, or env variables in the AI service).
- Integrate the chatbot with the website: We already covered the frontend widget. We just need to ensure the URL and any auth for the chatbot API is configured in the front-end app (possibly via env var).
- Logging for AI: Have the chatbot service log conversations (at least on a temporary basis) for us to review and improve the bot. We must also consider privacy – maybe anonymize user info. But logs help debug when the bot says something weird, we can adjust.
- Scale considerations: If the chatbot becomes popular, we can scale it horizontally (multiple instances behind a load balancer) as it’s stateless (each request independent except for what it reads/writes

from DB). The heavy part is the LLM call which is external (OpenAI can scale on their side). If using a local model, scaling would be more complex (need multiple GPUs or servers).

By following these steps, we will have robust AI services integrated into our platform. The chatbot will be a sophisticated piece of the system, but with the above methodology, it remains manageable (leveraging existing AI infrastructure). This AI layer is what will set *The Open Box* apart, providing a modern, smart shopping experience.

Importantly, we maintain **human oversight**: the bot encourages automation but we will monitor interactions, especially price negotiations, to ensure the rules yield profitable outcomes and adjust if needed. As AI evolves, we can further train/tune our models on our domain data to improve performance (e.g., fine-tune a model on past Q&A logs for better accuracy).

DevOps, CI/CD, and Infrastructure Strategy

A strong DevOps strategy ensures that development, testing, deployment, and scaling of *The Open Box* platform are efficient and reliable. In this section, we outline how to set up continuous integration/continuous deployment (CI/CD) pipelines, infrastructure as code, and environment configurations, as well as strategies for scaling and maintaining the application in production.

1. Source Control and Branching: All code for the project (frontend, backend, AI services, etc.) will be stored in a source control system (e.g., GitHub or GitLab). We will use a branching strategy like GitFlow or simple feature branching: - The `main` (or `master`) branch represents stable, deployable code. - A `develop` branch (optional) for integration testing. - Feature branches for new features/bugfixes which are merged via Pull Requests with code reviews.

This ensures that only reviewed code is deployed, and we can track changes.

2. Continuous Integration (CI): Set up CI pipelines (using GitHub Actions or GitLab CI) that trigger on pushes and pull requests: - **Build and Test:** The pipeline will install dependencies and run tests for each part: - Backend: run Saleor's test suite (if we have any custom tests or rely on core tests), run linting (flake8/black for Python). - Frontend: run `npm run build` to ensure it compiles, run unit tests (jest) and lint (ESLint). - AI Service: run any Python tests for chatbot functions, lint (pylint/black). - We also incorporate type-checking (TypeScript check for front, mypy for Python) to catch errors early. - **Report:** If tests or linters fail, the CI marks the build as failed, preventing merge/deployment. Use CI badges to track status.

3. Continuous Deployment (CD): We will employ automated deployment for certain branches: - The **staging environment** will auto-deploy when code is merged into a `staging` branch (for example). This pushes the latest code to a staging instance of the app for QA. - The **production environment** deployment from `main` can be configured to either auto-deploy on merge or require a manual approval step (especially if we want to control release timing).

Using GitHub Actions, we can set up workflows that: - On push to main: build Docker images for backend, chatbot, etc., push to a container registry (like ECR or DockerHub), and then trigger a deployment (e.g., update ECS service or trigger Vercel deployment for front). - On push to front-end main: Vercel auto-deploys (since Vercel integrates with GitHub).

Alternatively, use infrastructure like **AWS CodePipeline** or GitLab's CD to deploy.

4. Infrastructure as Code (IaC): Define our infrastructure using code for consistency and easy reproducibility: - Use **Terraform** or **AWS CloudFormation** to script creation of AWS resources (VPCs, ECS clusters, RDS database, etc.). This way, environments (staging, prod) can be stood up with the same configuration. It also allows versioning of infra. - In Terraform, for example, define modules for networking (subnets, security groups), compute (ECS tasks or EC2 instances), services (RDS, S3, CloudFront, etc.). - Store IaC code in the repository and maybe have pipelines apply it (with manual approval for production changes).

5. Environments Setup: - **Development:** Local dev can run with Docker Compose for convenience. Each developer can spin up the stack on their machine (Saleor, DB, maybe a local AI server stub or use cloud AI). - **Staging:** An environment that mirrors production on a smaller scale. Use a separate DB instance, perhaps smaller ECS tasks, and maybe enable debug mode for more logging. Staging will use a staging stripe account and other sandbox credentials. Data can be dummy or a scrubbed copy of prod. - **Production:** The live environment with full scale and proper credentials. Ensure separation of secrets between staging and prod (don't share API keys).

6. Hosting Infrastructure: We will utilize a mix of **AWS** for backend services and **Vercel** for the frontend: - **Frontend on Vercel:** As mentioned, the Next.js app will be on Vercel. Vercel handles the CDN, SSL, and scaling of the frontend globally. Every deployment gets a unique preview URL for testing PRs, which is great for QA reviewing feature branches. - **Backend on AWS (ECS Fargate):** Containerize the Saleor backend and deploy it on AWS Fargate (serverless containers) behind a load balancer. Fargate will allow running tasks without managing EC2 servers. We can configure auto-scaling on CPU/memory usage if needed (e.g., scale out additional tasks if high load). The AWS Reference Architecture for web apps suggests decoupling UI and back-end and using auto-scaling groups [37] [38] – our approach aligns with that. - Use an **Application Load Balancer (ALB)** to route requests to the Saleor tasks (for `/graphql` and any other endpoints). The ALB also handles SSL termination (use ACM for cert). - **Database:** Use **AWS RDS (PostgreSQL)** for a managed, reliable database. In production, run a multi-AZ RDS instance for high availability. Enable automated backups. For caching, if needed, an **Elasticache (Redis)** can be used (Saleor can use Redis for caching or session storage). - **Media storage:** Use **S3** for product images and any user-uploaded files. Set up an S3 bucket and configure Saleor's settings to use S3 for storing media (Saleor docs have a guide for S3 storage [39]). Set up **CloudFront CDN** in front of the S3 bucket for faster delivery to users across regions. - **n8n and Chatbot on AWS:** These two services can also be containerized and run on ECS Fargate or as AWS Lambda if possible. n8n is a long-running workflow server, so it's better on ECS. The chatbot (FastAPI) also on ECS behind the same or another ALB (could even be a path-based route on same ALB, e.g., `/api/chat` forwarded to chatbot service). - **Networking:** All these containers will be in a private subnet (except ALB in public). Use security groups to allow only necessary traffic (ALB to backend on port, backend to DB, etc.). The chatbot and n8n might not need public endpoints except via ALB or for n8n if we want to access its UI (could restrict by IP). - **Domain and DNS:** Use Route 53 to manage the domain DNS (e.g., `theopenbox.com` and `laboiteouverte.ca` if separate). Point the apex or subdomain to Vercel for the front-end (Vercel provides aliases). For the API endpoints (Saleor, chatbot), you can either: - Have the frontend call them via a subdomain like `api.theopenbox.com` which points to the ALB. Set up DNS accordingly. - Or just call the ALB's domain directly if we prefer not to expose a friendly name (but better to use subdomain for SSL). - Ensure SSL for all domains (Vercel auto handles for frontend, AWS ACM certificate for `api` subdomain on ALB).

7. Monitoring & Logging: - **Application Monitoring:** Use **AWS CloudWatch** for ECS logs and metrics. All container stdout/stderr can go to CloudWatch Logs. Set up custom metrics if needed (like request counts, latencies). Use CloudWatch Alarms to notify if CPU is high or memory or if ECS tasks restart unexpectedly. - **Error Tracking:** Implement **Sentry** in the frontend and backend. In frontend, Sentry will catch JS errors or UX issues (with user info, if consented). In backend, integrate Sentry Python SDK to capture exceptions. - **Uptime Monitoring:** Use an external service (Pingdom, UptimeRobot) to hit the site and API periodically to ensure they are up. Also consider AWS Route 53 health checks. - **Analytics:** For business, integrate Google Analytics or similar on the front-end to track user behavior, conversion funnels etc., while respecting privacy laws (cookie consent banners etc. as needed for GDPR). - **Performance Monitoring:** Optionally, use APM tools (DataDog, NewRelic) to trace backend performance and front-end performance (Web Vitals). - **Cost Monitoring:** Set up AWS budgets/alerts to track spending, since multiple services will incur costs (ECS, RDS, etc.). This ensures we catch any anomaly (like if a bug triggers a million API calls to OpenAI, etc. – that cost should be monitored too via OpenAI usage dashboard).

8. Scaling and High Availability: - The system is built to scale horizontally: - We can increase ECS task counts for Saleor if traffic grows (behind ALB). Saleor being stateless (except DB) allows that easily. Same for chatbot service. - The DB can be scaled vertically to a larger instance or add read replicas if needed for read-heavy loads (Saleor can be configured to use read replica for select queries [40] [41]). - Use CloudFront caching for static assets and maybe pages if possible to reduce load. - For high availability, run multiple ECS tasks in different AZs and multi-AZ RDS, such that even if one AZ goes down, the app stays up. Also store media across AZs (S3 by design is multi-AZ). - If using Kubernetes (EKS) instead of ECS, the approach is similar but more devops heavy. ECS Fargate is simpler to start.

9. Release Management: - Use versioning for releases. Tag releases in Git and maybe use semantic versioning (v1.0, v1.1 etc.). Each deployment could be tied to a release tag. - Blue-Green deployment: For zero downtime deploys, consider running two sets of tasks and switching over. AWS CodeDeploy for ECS can manage that. Alternatively, ensure new tasks start before old ones stop (so ALB always has a healthy target). - Database migrations: When deploying a new Saleor version or our extensions that require DB migrations, handle carefully: run migrations as a separate step (in CI or manually) before deploying new code. In zero-downtime scenario, you'd run migrations that are backwards-compatible, then update app, etc.

10. Backup and Recovery: - Database: Automated daily snapshots on RDS and the ability to restore quickly. Also, for safety, occasional logical backups (pg_dump) stored offsite. - Media: S3 is durable, but enabling versioning is a good idea so files can be recovered if deleted. Could also back up S3 to another bucket. - Configs: Backup environment config (like in a secure repo or parameter store). - n8n workflows: Back up the workflows (n8n can export workflows JSON; we should store those in git or an S3). - Disaster Recovery Plan: Document procedures to rebuild in a new region if needed (since infra as code helps here).

By following these DevOps practices, we ensure that the engineering team can continuously deliver improvements to the marketplace reliably. Automated CI/CD fosters rapid iteration but with safety nets (tests and approvals). Monitoring and logging give us visibility into the system's health, allowing quick response to issues. The infrastructure choices (AWS & Vercel) provide a balance of control and convenience, and can scale with the business.

In summary, our deployment pipeline will enable us to **ship updates frequently and confidently**, our infrastructure will be **robust and secure**, and our operations will be proactive through monitoring and alerts. This means the platform can evolve and grow without compromising stability or security.

Testing, QA, and Monitoring

Delivering a high-quality marketplace requires rigorous testing and ongoing monitoring once live. In this section, we outline testing strategies (for functionality, performance, and security) and how we will monitor the system in production to promptly detect and address issues.

1. Testing Strategy Overview: We will adopt a multi-layered testing approach: - **Unit Testing:** Test individual functions/modules in isolation. For example, test the price negotiation function logic with various inputs, test utility functions (like translation or formatting helpers). Both front-end and back-end should have unit tests (front-end using Jest for utils and React components with shallow rendering; back-end using Python unittest/pytest for any custom functions or model methods). - **Integration Testing:** Test how components work together. E.g., simulate a GraphQL query hitting the Saleor API (possibly using a test database), or test an n8n workflow end-to-end with dummy input to see if it creates a product in a test Saleor instance. On the front-end, integration tests could involve rendering a page and mocking network calls to see that it displays data correctly. - **End-to-End (E2E) Testing:** Simulate user flows in a staging environment. Use **Cypress** or **Playwright** to automate a browser to: go to homepage, navigate categories, add item to cart, go to checkout, fill info, and (possibly simulate payment or use Stripe test mode), then verify order confirmation. Also E2E test the chatbot: open chat, ask a question, get an answer. These tests ensure the entire stack (frontend, backend, AI) work in concert. - Because E2E tests can be time-consuming, we'll run a core subset on each build (like a smoke test: can the site load, can a product be viewed). Then run full regression E2E suite nightly or before a release. - **Regression Testing:** Before each release, run through a checklist of core scenarios manually as well to catch any UX issues automation might miss. Particularly new features or any critical flows like payment should be double-checked. - **Performance Testing:** Use tools like **JMeter** or **k6** to load test the site's APIs. For instance, simulate 100 concurrent users browsing products and adding to cart to see how the system performs (especially on the backend and DB). Load test the chatbot with multiple parallel conversations to ensure the AI service and vector DB hold up. This will inform if we need to scale up resources. - **Security Testing:** Perform a security audit: - Use static analysis (linters, dependency vulnerability scanners via GitHub Dependabot or `npm audit` etc.). - Penetration testing: attempt common web vulnerabilities (SQL injection on API, XSS on inputs, CSRF, etc.). Many of these are mitigated by using frameworks (Saleor sanitizes inputs and uses GraphQL which is type-safe; Next.js has built-in XSS protections). Nonetheless, we should test. For example, ensure that an authenticated user cannot access another's order by changing an ID (should be prevented by permission checks). - If budget allows, consider a third-party security assessment or use open-source tools like OWASP ZAP to scan the application. - **Accessibility Testing:** Use tools (like Lighthouse, axe-core) to run automated accessibility tests on pages. Also do manual keyboard-only navigation and screen reader testing on key flows to ensure compliance with WCAG 2.1 AA.

2. Testing Environments and Data: - Maintain a separate **test database** with seed data for automated tests (like a few products, a test user, etc.). This can be SQLite or an ephemeral Postgres for unit/integration tests. - The staging environment can double as a testing environment for E2E; populate it with sample data. - Use test API keys (Stripe, etc.) in non-prod so that tests don't trigger real transactions or notifications.

3. Continuous Testing in CI: - Ensure that the CI pipeline runs the test suites on every push. If any test fails, the build fails and we don't deploy that code [42] (we treat tests as gatekeepers for quality). - Set up output of test results and coverage reports for transparency. Aim for a healthy coverage percentage (though 100% is not required, focus on critical logic).

4. QA Process: - Adopt an agile approach where each user story has acceptance criteria that QA verifies. The QA engineer (or team member in charge of QA) will test the feature in an isolated environment or local, then again when it's on staging. - Use a tracking tool (Jira, etc.) to document test cases and mark off passes/failures. - For any bugs found, create bug tickets with steps to reproduce, and address them before release. - Perform **user acceptance testing (UAT)** with a small group (maybe internal team or a friendly customer group) on staging to get feedback especially on the AI chatbot behavior and any confusing UI element.

5. Monitoring in Production: Once live, we need proactive monitoring: - **Real-time Application Monitoring:** As mentioned in DevOps, we use CloudWatch and application logs. Set up CloudWatch alarms for: - High error rate (for example, if 5xx errors on the API exceed a threshold). - High latency (if p95 response time goes beyond X ms). - Resource utilization (CPU > 80% for a sustained period -> maybe scale up). - **Sentry Alerts:** Configure Sentry to alert on new errors or spikes in errors. For example, if a specific error (say, null pointer or GraphQL resolver error) happens 50 times in an hour, alert the dev team via Slack/Email. This catches issues that slip by testing once real users hit edge cases. - **AI Monitoring:** Monitor the chatbot usage. Keep track of: - Number of conversations, questions asked. - The AI API usage (OpenAI has a dashboard for usage – track to manage cost and see if usage matches expectations). - Failures: if the AI service fails to respond or returns error, log that and potentially alert if it's frequent. - Content monitoring: if possible, log or review conversation transcripts occasionally to ensure the bot is behaving and not giving improper answers. (This should be done in compliance with privacy – perhaps only review if a user flags a conversation or in anonymized form). - **User Feedback:** Incorporate a feedback mechanism on the site. For example, after a chat, ask "Was this answer helpful?" and gather thumbs up/down. Or provide a way for users to report an issue. Monitor this feedback to improve the AI and the platform in general. - **Analytics Monitoring:** Keep an eye on key metrics via Google Analytics or an e-commerce analytics tool: - Conversion rate (how many product views lead to purchase). - Drop-off points (if many start checkout but don't finish, investigate issues). - Popular search terms (if users search and get no results, maybe we need to add those items or synonyms). - Use these insights to continuously refine UX (could involve adding features or adjusting AI responses to common questions). - **Scheduled Audits:** Periodically (say monthly) do an audit of security (review access logs, ensure no suspicious logins), performance (any pages consistently slow?), and cost (are we within our cloud budget, any inefficiencies?).

6. Incident Response: - Develop an incident response plan. If the site goes down or a major bug appears: - Have contact info of responsible engineers on call. - Use status pages or banners to inform users if needed. - Rollback plan: Because we use versioned deployments, if a new release is problematic, be ready to rollback to previous stable version (either via `git revert` and redeploy, or if using a container, redeploy the previous image tag). - Database issues: have a plan to restore from backup if needed, though that's last resort (and data loss would be a concern, so better to fix forward with minimal downtime). - Practice a drill of recovering from backup in a staging environment to ensure our backups are valid.

Through these testing and monitoring practices, we aim for **high reliability** and **quality** of the platform. Testing catches issues before they hit users, and monitoring catches issues that only appear under real-world conditions. Combined, they create a feedback loop: data from monitoring might lead to new test

cases (e.g., if an error happened in prod, write a test to cover that scenario in future). This culture of quality will help maintain user trust – crucial for an e-commerce marketplace.

Security, Compliance, and Accessibility

Building user trust and meeting legal/ethical obligations is as important as building features. This section details how *The Open Box / La Boîte Ouverte* will comply with security best practices, data protection laws (like GDPR), payment security standards (PCI-DSS), and accessibility guidelines (WCAG), as well as handle localization aspects like currency and taxes properly.

1. Payment Security (PCI-DSS Compliance): Handling payments requires adherence to the Payment Card Industry Data Security Standard (PCI-DSS). Our strategy is to minimize our scope: - **Use Stripe (or equivalent)**: By using Stripe's hosted checkout or elements, we ensure that credit card data never touches our servers directly. Stripe's systems are PCI-DSS Level 1 certified, shifting most compliance burden to them. Our site will use either: - **Stripe Checkout** (hosted payment page): After the user enters shipping info, we redirect them to a secure Stripe page to input card details and complete payment, then back to our site on success. This approach keeps us mostly out of scope (just need to secure redirect and verify webhook). - Or **Stripe Elements** (embedded card fields): If we use this, card info is sent from browser direct to Stripe via JS, and we get a token. Our server then uses that token to charge via Stripe API. In this approach, our server still does not see raw card data, but since it handles the token and interacts with Stripe's API, we must ensure HTTPS and proper secret management. - **PCI Questionnaire:** Even with Stripe, we will fill out the appropriate SAQ (Self-Assessment Questionnaire), likely SAQ A or A-EP depending on integration. This is a checklist of security measures (like using TLS, not storing card details, etc.). - **Secure Data Handling:** We will *not store any sensitive payment info* on our side. Payment tokens or customer IDs from Stripe can be stored, as they are useless outside Stripe context. We ensure all payment-related communication is over TLS 1.2+ and our servers are hardened. - **Firewall & Network Security:** Our infrastructure will have security groups to isolate the database from public internet, use OS-level firewalls on instances if any. Use AWS WAF on the ALB to filter common web attacks (AWS WAF can mitigate SQLi, XSS attempts at the edge) [43]. - **Regular Scans:** Use vulnerability scanning on the server images, and if possible, periodic PCI scans by an ASV (Approved Scanning Vendor) if required by bank.

2. Data Privacy and GDPR: Since we will have users possibly from the EU or certainly in Canada (PIPEDA applies similarly), we need compliance with privacy regulations: - **Privacy Policy and Consent:** Draft a clear Privacy Policy explaining what data we collect and why. On first visit, use a cookie consent banner if we use any tracking cookies (for analytics or for personalized content). Allow users to opt out of non-essential cookies (especially in EU jurisdictions). - **User Data Rights:** Implement features (or at least a support process) for: - **Data Export (Article 15 GDPR):** If a user requests their data, we should be able to provide an export (account info, order history, etc.). This can be facilitated by a script or manual admin process initially. - **Data Deletion (Article 17 Right to be Forgotten):** Allow users to delete their account from the settings. This process should remove personal data from our systems. Perhaps we implement a "delete account" button that anonymizes or deletes user record, while retaining orders in an anonymous form for bookkeeping. (We must balance legal record-keeping vs deletion; usually, we can scrub personal identifiers but keep order transactions for financial records.) - **Minimization:** Only collect necessary personal data. For example, require shipping address only when needed (at checkout), not for casual browsing or when not needed. - **Storage & Encryption:** Store personal data in the database encrypted at rest (RDS has encryption). Backup data is also encrypted. Ensure any sensitive fields (like user password hashes – Saleor uses PBKDF or similar by default – API keys, etc.) are properly hashed or encrypted. - **Data Transfer:** If we

have EU users but host in North America, consider legal mechanism (e.g., standard contractual clauses) in privacy policy. Possibly host EU user data in EU region if expanding. - **Train Staff:** If admin staff have access to user data, ensure they understand privacy and handle data appropriately (not copying it out, etc.). - **Breach Response:** Have a plan in case of data breach (as required by GDPR to notify users and authorities within 72 hours). Hoping to never need it, but be prepared.

3. Account Security: - **Passwords:** Saleor handles authentication – it stores salted password hashes. Enforce strong password policy on registration (min length, etc.). Possibly integrate "HaveIBeenPwned" API to reject known leaked passwords for extra security. - **2FA:** Consider offering two-factor authentication for user accounts, at least for admin/seller accounts. Saleor might not have built-in 2FA, but we can integrate an authenticator app or SMS 2FA via a plugin or custom code for admin login. - **Session Security:** Use HTTP-only, secure cookies for session tokens. Set proper session timeout and idle expiration. Use CSRF tokens for forms if needed (Saleor and our front-end should be using GraphQL with tokens which is not susceptible to CSRF if done right, but still take precautions for any state-changing operations via forms). - **Input Validation:** Ensure that any user input (product listings from sellers, chat messages, etc.) is sanitized or escaped to prevent XSS in our UI. For instance, if a seller includes a `<script>` in description, our front should escape it when rendering (or disallow HTML in descriptions to be safe). We will use libraries or built-in frameworks to handle these (React auto-escapes content by default). - **Authorization checks:** Only allow data access if the user is authorized. E.g., ensure one user cannot access another's order via API. Saleor's permissions and our access control for any custom endpoints must be thoroughly tested.

4. GDPR for AI (if applicable): Our chatbot might log conversations. Under GDPR, chat content from users could be personal data if they mention something. We should: - Disclose the use of AI and data usage in privacy policy. - Possibly allow opting out of chatbot data collection. Or ensure it's anonymized and primarily for service improvement. - Not use personal data in training AI without consent. For instance, we won't be feeding user-specific chat logs into a model training pipeline except for troubleshooting. If we did, we'd anonymize it. - Provide a way to delete chat data upon user request, if we store it linked to them.

5. Accessibility (WCAG Compliance): We aim to meet **WCAG 2.1 AA** guidelines: - Ensure proper use of headings, landmarks, and roles in HTML for screen readers. - All interactive elements (links, buttons) must be reachable via keyboard (tab order logical, and `:focus` styles visible). - Provide text alternatives for all non-text content: - Product images: use alt text that describes the item briefly (we can generate alt text automatically from product title or allow seller to input it). - Icons: if using icon fonts or SVGs for buttons, give them `aria-label` or screen reader text. - Color contrast: Use a color palette that has sufficient contrast (we'll test with contrast checkers). For example, text on backgrounds should generally have contrast ratio >= 4.5:1 for normal text. - No reliance on color alone: if an error is shown with a red outline, also include an icon or text. - Forms: each input has a label, and provide helpful error messages. E.g., label the search bar, label checkout fields, etc. - Language attributes: set `lang="en"` or `lang="fr"` on the HTML element appropriately so screen readers know which language. - Testing: Use screen reader (NVDA/JAWS for Windows, VoiceOver for Mac) on key flows to ensure they read everything sensibly. Use keyboard navigation to ensure no traps. - Accessibility for the chatbot: The chat widget should be accessible too. Possibly allow using it via keyboard (open chat with keyboard, focus moves to chat area, etc.). Also, consider users who cannot use chat – ensure critical info (like policies) is also available in text on site (which it will be in FAQ pages). - Continuously include accessibility in our testing plan. It's easier to build it in from start than retrofit.

6. Localization (Currency, Tax, Units): - **Currency Display:** As mentioned, support multi-currency if needed. If we stick to one (CAD), ensure that in French locale, we format as 1 234,00 $ (with space and comma). We can use Intl.NumberFormat with locale to format currency. If expanding to multi-currency, Saleor's multi-channel pricing can handle that (e.g., a USD channel vs CAD channel). - **Tax Calculation:** The platform should calculate taxes correctly based on locale: - For Canada, if we charge tax, need to apply GST/HST or provincial tax depending on province of buyer and maybe where seller is. This can be complex, but since second-hand maybe often individuals (maybe platform might not be required to charge tax if it's a consumer-to-consumer sale? If platform is merchant of record, likely need to handle). We can integrate a tax service or set a rule: e.g., use one tax rate for all to keep it simple if legally acceptable. - If expanding to EU, then need VAT by country and possibly IOSS for cross-border. - **Language translations:** Ensure that all content including emails sent to users are in the appropriate language. We might need to send bilingual order confirmation emails or send in the language the order was placed. Saleor's emails can be customized to include translations. - **Date/Time/Units:** Display dates in local format (for French, day-month-year format if needed). Times typically in e-commerce are just dates (like order date). If showing any measurement units (dimensions, weight), consider unit conversion if needed (in Canada both metric and imperial might be understood; in France strictly metric). - **Legal Compliance:** For localization, consider legal requirements: e.g., in Quebec, French must be provided not as a lesser version than English (so ensure French content is equally complete). Also currency: In Canada, we might only transact in CAD; ensure currency conversion if showing to US visitors (maybe just display approximate conversion as info). - **Content Moderation:** A side note on compliance: since it's a user-generated content scenario (sellers adding product descriptions/images), we should have moderation policies. Prohibit illegal items, hate speech in descriptions, etc. We can enforce via terms of service and implement checks (perhaps using AI to flag inappropriate listings).

7. Audit and Updates: - Conduct periodic security audits (review user roles, access logs, rotate secrets like API keys periodically, update dependencies to patch vulnerabilities). - Keep software up to date: Apply patches to the OS images, regularly update Saleor to latest minor version for security fixes (with testing). - Ensure the team is aware of social engineering risks (phishing, etc., for admin credentials).

By covering these bases, *The Open Box* will not only be a feature-rich platform but also a trustworthy one. Customers will feel safe entering their payment information, knowing their personal data is handled properly, and that the site can be used by anyone regardless of disabilities or language. Compliance is an ongoing effort – we will stay informed of legal changes (like new privacy laws) and tech changes (like new WCAG versions) and adapt accordingly.

This approach is not just about avoiding penalties; it's about building a **reputable brand** that values its users' safety and inclusivity, which in turn can be a competitive advantage.

Cloud Hosting and Deployment Best Practices

Deploying *The Open Box / La Boîte Ouverte* to the cloud requires careful planning to ensure reliability, scalability, and cost-effectiveness. Here we outline our cloud hosting strategy, primarily using AWS and Vercel, and enumerate deployment best practices to follow.

1. AWS Cloud Hosting Strategy:

As mentioned earlier, AWS will host the majority of our backend infrastructure: - **AWS Region:** Choose a primary region close to our user base. If our marketplace targets Canada, the **Canada Central (Montreal)**

region is a good choice for low latency within Canada (and it keeps data in-country which can be beneficial for data sovereignty). Alternatively, US East (N. Virginia) or East (Ohio) could be used if most traffic is North America broadly. We'll also consider using CloudFront which edges content globally. - **Compute (ECS vs. EKS vs. EC2):** We decided on **AWS ECS Fargate** for running containers (Saleor backend, chatbot, n8n). Fargate is serverless for containers, meaning no EC2 management and it auto-scales fairly easily. We'll use an ECS cluster with services for each component: - Saleor service: e.g., 2 tasks (containers) minimum, scale out to 4 on high load. - Chatbot service: maybe 1 task to start, scale to more if usage increases (the LLM calls might be the bottleneck but those are external). - n8n service: 1 task is fine (workflows run sequentially unless we specifically scale it; if ingestion loads grow, we can scale it or separate critical workflows to another instance). - **Database: AWS RDS for PostgreSQL**. E.g., start with a db.m5.large (2 vCPU, 8GB) which is enough for small to medium traffic. Enable multi-AZ for failover. Storage with auto-scaling. Turn on Performance Insights to monitor queries. In production, we might also enable read replicas if needed to offload read-heavy operations (Saleor can use them for some queries [40]). - **Networking:** Use a VPC with both public and private subnets. Public subnets for Load Balancer and maybe bastion if needed; private for ECS tasks and RDS. Use NAT Gateway for tasks to access the internet (for updates or calling external APIs, though OpenAI and such can be called through NAT). - **S3 & CloudFront:** - Create an S3 bucket `theopenbox-media` for user-uploaded media. Configure it to only allow read via CloudFront (private bucket with CloudFront origin access identity). - CloudFront distribution: Serve `https://media.theopenbox.com/<path>` to deliver images. This will cache images globally, improving load times [44]. Set long cache TTL for images (since product images rarely change). - Use another S3 for any logs or backups if needed. - **Lambda/Serverless Functions:** We might use AWS Lambda for small tasks: - If using webhooks (Stripe or others), we could catch them with API Gateway + Lambda if we don't want to expose the Saleor API publicly for them. But likely easier to allow Stripe to call Saleor webhook endpoint directly. - Perhaps an Lambda for sending emails if using AWS SES (though Saleor can send directly). - If we have cron jobs, AWS EventBridge + Lambda could do periodic tasks (or use ECS Scheduled Tasks). - **Cognito for Auth (optional):** If we want to offload user authentication, we could integrate Amazon Cognito for user sign-up/sign-in and then use Cognito tokens in our app. But since Saleor already provides user auth, we probably stick with that to avoid complication. - **Logging:** Use CloudWatch Logs for ECS tasks. Also, set up an export of logs to S3 for longer retention or to an ELK stack if needed for advanced searching. We might not need a full ELK if CloudWatch Insights suffices. - **Secrets Management:** Use AWS Secrets Manager or SSM Parameter Store for storing sensitive config (DB passwords, API keys). ECS can fetch these and inject into container env at runtime. This avoids hardcoding secrets in images or code. - **Cost Optimization:** Choose appropriate instance sizes and use AWS auto-scaling to scale down when idle. For example, maybe at night traffic is low, we could scale Saleor tasks to 1. Also, use AWS's Savings Plans or Reserved Instances for RDS or any fixed usage to reduce cost. Turn on S3 lifecycle rules to move old logs to Glacier. Monitor AWS bills initially carefully to find any inefficiencies (like a misconfigured service doing chatty calls).

2. Vercel Hosting for Frontend: - Connect the GitHub repo for the Next.js frontend to Vercel for CI/CD. - Configure build settings (should detect Next.js automatically). Set environment variables on Vercel: e.g., `NEXT_PUBLIC_API_URL` (pointing to our API endpoint), `NEXT_PUBLIC_STRIPE_KEY`, etc., and secure ones like Sentry DSN, etc. - Vercel will build and deploy on every push to main (and deploy previews for PRs). We will protect production deployment by maybe linking it to main or a specific branch only. - Vercel provides global edge network, so our site will be fast worldwide by default. We should still specify caching headers in Next.js for static assets, which Vercel will honor. - Domain setup: Add our custom domain in Vercel (e.g., `theopenbox.com`). Since root domain might be tricky (apex on Vercel uses A records with static IPs or alias records), we might use `www.theopenbox.com` for the app and redirect apex to www via Route53. Or use ANAME/ALIAS if Route53 supports it. In either case, Vercel will provision SSL certificate via

Let's Encrypt automatically for that domain. - Monitor Vercel's analytics/insights (if enabled) for front-end performance. It can show metrics like TTFB, etc.

3. Deployment Best Practices: - **Automate Everything:** No manual steps in deployment. Use the CI/CD pipeline to push out new releases. This reduces human error. For infrastructure changes, use Terraform apply via CI if confident, or run manually with peer review. - **Blue-Green Deployments:** For critical services like Saleor, consider Blue-Green deployments. That is, deploy a new version to a parallel set of containers, run health checks, then switch the load balancer to the new ones, then decommission old. AWS CodeDeploy can assist with ECS blue-green deployments. This ensures zero downtime and quick rollback if needed (just switch back). - **Rolling Deployments:** Alternatively, for small changes, a rolling update in ECS (the default) is fine: it will bring up a new task, wait for healthy, then kill an old one. - **Database migrations in deployment:** Use a strategy to apply DB migrations safely. Possibly run migrations as a separate task prior to deploying app tasks. Could automate with a small ECS job definition that runs `alembic upgrade` or Django migrate for Saleor. Only after success, continue deployment. In a blue-green, you might run migrations that are backward-compatible, then deploy new code that uses them. - **Feature Flags:** To reduce risk, consider using feature flags for big features. That way, you can deploy code turned off by default and enable for testing or gradually to users. This is more of an advanced practice but useful for controlling rollout. - **Backup before changes:** For major deployments (like a version upgrade of Saleor), create a DB snapshot beforehand. Also perhaps test the upgrade in staging with a copy of prod DB if possible. - **Observation Post-Deploy:** After deployment, monitor closely (maybe have a checklist to verify key pages load, check error dashboards). Also, possibly do a canary release: deploy to one task, route a small % of traffic (if ALB supports weighted target groups or via splitting traffic by domain or some method) to new version for monitoring, then increase to 100% if all good. - **Documentation:** Document the deployment process so any team member or new DevOps engineer can pick it up. Include how to handle emergency rollbacks. - **Resilience:** Place critical services in at least two AZs. Use health check probes so if one container goes unhealthy, ECS/ALB can replace it automatically. - **Cross-Region Strategy (if needed):** Currently likely not needed, but if we want DR, could have a standby environment in another region that can be spun up if primary fails. Or use CloudFront to distribute read-only content even if main API is down (not much use if API down though). At least backups in another region to restore from.

4. Domain and SEO: - The marketplace being bilingual could either run on one domain with language subpaths (`/en` and `/fr`), or potentially separate domains (like `.com` for English and `.ca` for French, or `.com/en` and `.com/fr`). We'll likely use one domain for simplicity and avoid duplicate site management. - Ensure correct hreflang tags in HTML to let search engines know there are English and French versions of pages to serve the right audience. - Create an XML sitemap (Next.js or Saleor can help) listing both language URLs. - Use a robots.txt to allow crawling of all product pages.

5. Third-Party Services Deployment: - We use Stripe, which is external. We ensure our webhooks (like payment success webhook) are deployed (Saleor's plugin or our endpoint) and reachable (if local dev, use a tool like ngrok to test webhooks; in staging/prod, ensure the path is publicly accessible and secure). For reliability, Stripe webhooks will retry if our endpoint was down during deploy, so that's fine. - If using any external like Jetti for marketplace, ensure API keys configured and test that integration on deploy (maybe a health check that calls Jetti API lightly). - If using Algolia for search (just as an idea), we'd deploy indexing tasks similarly.

6. Failover and Maintenance: - Plan maintenance windows for things like DB upgrades. Inform users via a banner if downtime is expected (though aim for zero-downtime deploys mostly). - Use read replicas to

offload reads during backup jobs etc., if needed. - Implement a caching layer on queries that are heavy (Saleor GraphQL can be cached at CDN if queries are GET and we include them in cache key properly, or use Varnish. Possibly out of scope for MVP, but could consider GraphQL query caching for product data since it changes rarely). - The stateless parts (front-end, application servers) are easy to redeploy; stateful part (DB) is single point of failure mitigated by multi-AZ and snapshots.

Following these practices ensures that when we deploy the marketplace to production, we have a robust, scalable setup. AWS gives us building blocks that follow the **Well-Architected Framework**, focusing on security, reliability, performance efficiency, cost optimization, and operational excellence [45]. Vercel provides a no-fuss way to deploy the frontend globally.

In summary, **The Open Box** will be hosted in a way that users experience a fast and reliable website, while the team experiences a manageable and automated deployment process. We'll continuously review the infrastructure as the project grows, tuning it for better performance and lower costs, and leveraging more AWS services as needed (for example, introducing AWS SageMaker for AI in the future if we bring AI fully in-house, etc.). The guiding principle is to use managed services where possible (databases, containers, CDN) so we can focus on application development and not on undifferentiated heavy lifting.

Project Timeline and Team Roles Distribution

Implementing *The Open Box / La Boîte Ouverte* is a significant project. Breaking it into phases with a timeline and assigning clear roles will help manage progress and ensure all aspects are covered. Below is a proposed timeline (assuming we start at Month 0) and the key team roles involved in each phase:

Phase 1: Inception and Planning (Weeks 1-2)
- *Activities:* Requirements finalization, project kickoff, and stack decision (platform selection done – Saleor). Create project plan, milestones, and set up project management tools. Define the data model (what categories, attributes for products, etc.). - *Deliverables:* Project charter, high-level architecture diagram, and this implementation guide as a reference for the team. - *Team Roles:* - **Product Manager / Business Analyst:** finalize requirements, user stories. - **Solution Architect:** confirms technology choices, outlines architecture. - **Project Manager (PM):** sets timeline, coordinates resources.

Phase 2: Environment Setup (Weeks 3-4)
- *Activities:* Set up development environments for all components. Repo initialization for frontend, backend, etc. Implement CI pipeline scaffolding (basic build/test jobs). Prepare base AWS environment via Terraform (set up VPC, perhaps RDS instance, etc.). - *Deliverables:* Running "Hello World" services: e.g., Saleor running locally, Next.js connecting to it, trivial deployment to staging environment. - *Team Roles:* - **DevOps Engineer:** sets up AWS infra, CI/CD. - **Backend Developer:** configures Saleor instance. - **Frontend Developer:** scaffolds Next.js app and connects to Saleor API. - **QA Engineer:** starts writing test plan outline.

Phase 3: Core E-commerce Development (Weeks 5-8)
- *Activities:* Implement all core marketplace features: - Backend: Configure Saleor (currencies, languages, shipping, payments). Implement any needed extensions (like vendor mapping or integrate Jetti if decided at this stage). Write API scripts for initial data loading. - Frontend: Implement product listing pages, product details, search, cart, and checkout pages integrated with Saleor's API. Set up Stripe payment in frontend (test mode initially). - Ensure bilingual content is showing correctly (toggle language, etc.). - Set up Saleor's admin and test basic product management there. - *Deliverables:* A functional skeleton of the marketplace:

one can browse items, add to cart, and simulate a checkout (payment perhaps in test mode). - *Team Roles:* - **Backend Developers (1-2):** Work on Saleor config, custom code for multi-vendor or any logic, integration of Stripe plugin, etc. - **Frontend Developers (1-2):** Build out UI components and pages, ensure i18n working. - **UX/UI Designer:** (if separate role) Should be designing the interface concurrently (wireframes, visuals) for front-end devs to implement; also ensure design accommodates French text lengths etc. - **QA Engineer:** Begin writing test cases for each user story (even if feature incomplete, plan tests).

Phase 4: AI Integration (Weeks 9-11)
- *Activities:* Develop the AI chatbot service and n8n workflows: - Set up n8n and create a couple of sample ingestion workflows (perhaps start with a simple CSV import workflow for initial products, and an email-triggered one). - Develop the chatbot backend: implement retrieval index creation and a basic endpoint that can answer a couple of hardcoded FAQs as proof of concept. - Integrate the chatbot front-end widget (initially perhaps connect to a dummy endpoint) on the site. - Test negotiation logic in isolation (e.g., a script or simple text interface). - *Deliverables:* - n8n is running in dev and can import a product automatically from a test source. - Chatbot can answer sample queries using a small knowledge base. - Multilingual response verified with a couple of inputs. - *Team Roles:* - **AI/ML Engineer:** leads the chatbot development (setting up vector DB, writing retrieval logic, etc.). - **Backend Developer:** supports integrating Saleor with chatbot (e.g., for any API calls needed) and sets up n8n workflows (n8n can be low-code, but developer may need to script certain transformations or ensure it calls Saleor API correctly). - **Frontend Developer:** implements chat UI and ensures it can send/receive messages. - **QA:** Start formulating test cases for chatbot (like specific Q&A pairs, negotiation scenarios).

Phase 5: Refinement and Integration (Weeks 12-14)
- *Activities:* Now that core features and AI are in place, refine and integrate everything: - Complete any remaining front-end flows (user registration, profile management, password reset, etc.). - Ensure the AI chatbot is fully integrated: test it on the live data (initial product catalog) and tweak prompt and retrieval as needed. - Enhance n8n workflows: perhaps include AI description generation in them now and test with various inputs. - Implement any remaining marketplace features (ratings/reviews if planned, or wishlist, etc., if those are in scope). - Cross-browser testing on front-end (Chrome, Firefox, Safari, mobile browsers). - Performance tuning: e.g., ensure images are optimized, database has indices for any slow queries (monitor with logs). - *Deliverables:* Feature-complete application in a staging environment. Begin internal testing as if it were live. - *Team Roles:* - **All Developers:** Bug fixing, code review, and fine-tuning features together. Might involve pairing (frontend-backend) to solve integration issues. - **QA Engineer:** Conduct thorough testing on staging: run through all test cases, log bugs in issue tracker. Verify fixed bugs. Also test edge cases (e.g., incomplete data, network failure scenarios). - **Product Manager:** Start preparing launch plans, marketing content, etc., and verifies the product meets requirements.

Phase 6: Security and Compliance Review (Week 15)
- *Activities:* Conduct focused audit: - Security testing (penetration test either internally or by third-party if possible in timeline). - Accessibility audit (developer and QA go through site with accessibility tools). - Ensure all compliance to-dos are done (privacy policy drafted, terms of service drafted; cookie consent mechanism implemented; emails and PDFs (if any) have correct info). - Performance test with simulated load to ensure infrastructure can handle at least expected launch load. - *Deliverables:* Security report (with any issues fixed), accessibility compliance report, readiness of legal documents on the site. - *Team Roles:* - **Security Specialist:** (could be external consultant or an internal dev with security background) performs the pen test and code review for vulnerabilities. - **Developers:** Fix any issues found (e.g., adjust code to mitigate vulnerabilities or improve accessibility). - **Legal Advisor:** (if available) Reviews terms and privacy for compliance. - **DevOps:**

Double-check infrastructure security settings (private subnets, SG rules). - **QA:** verifies that fixes for any found issues did not break functionality.

Phase 7: Deployment and Launch (Week 16)
- *Activities:* Final deployment to production environment: - Ensure all environment variables for prod are set (real API keys, etc.). - Do a dummy run of deployment on staging with production-like settings. - Migrate any production data if needed (for launch, maybe we have seeded some initial listings). - DNS changes to point domain to Vercel and AWS ALB as needed. - "Soft launch" the site (perhaps initially to a limited audience or just no advertisement yet) to see if any unexpected issues arise in live environment. - Official launch with marketing push. - *Deliverables:* Live website at official domain, accessible to public, with all systems go. - *Team Roles:* - **DevOps Engineer:** Executes deployment, monitors the process, and verifies all services (Saleor, DB, etc.) are healthy. - **Backend/Frontend Devs:** On standby to quickly fix any last-minute issues that appear live. - **QA Engineer:** Runs smoke tests immediately post-deploy on production site to ensure core flows still work with production config. - **Product Manager/Marketing:** Announce launch, monitor user feedback from day one.

Phase 8: Post-Launch Stabilization (Weeks 17-20)
- *Activities:* Monitor the platform as users start using it. Address any bug reports or performance bottlenecks that real usage uncovers. - Perhaps implement minor improvements that didn't make initial launch. - Ensure the team can respond to support requests (maybe set up a support ticket system or at least an email). - Plan next set of features (based on user feedback, maybe mobile app or more AI capabilities). - *Deliverables:* A stable system with initial user base and a backlog of enhancements for next iteration. - *Team Roles:* - **All team**: rotating on-call to handle any urgent issues. - **Support/Community Manager:** (if we have someone in that role) communicates with early users, gathers feedback.

Roles Distribution & Responsibilities:

- **Project Manager:** Oversees timeline and coordinates between teams. Ensures milestones are met and liaises with stakeholders.
- **Product Manager/Owner:** Defines features, prioritizes backlog, clarifies requirements. In a smaller team this might be same as PM or an involved founder.
- **Solution Architect:** (could be a senior developer) Makes high-level design decisions, ensures all components integrate properly, and that non-functional requirements (scalability, security) are addressed.
- **Backend Developers:** Implement and configure Saleor, develop any custom backend code (like plugins, webhook handlers, etc.), build and maintain the API, ensure data integrity and performance of the backend.
- **Frontend Developers:** Build the Next.js app UI/UX, integrate with backend API, ensure the application is responsive and accessible. Also responsible for implementing localization on the client side.
- **AI/ML Engineer:** Focused on the chatbot and any machine learning components (like image recognition integration, description generation). Ensures the RAG pipeline is working and tunes the AI responses. Also might handle integrating any ML services (like training a custom model if needed down the line).
- **DevOps Engineer:** Handles AWS infrastructure, CI/CD pipelines, and overall deployment process. Ensures environment consistency (dev/staging/prod), monitors systems, and optimizes performance from an infrastructure side.

- **QA Engineer:** Writes and executes test plans, does manual and automated testing, and works with devs to get issues resolved. Also checks compliance aspects (like going through an accessibility checklist, verifying language toggles everything).
- **UX/UI Designer:** (If part of team) Designs the user interface layout, visual elements, and possibly contributes to front-end with styles. Ensures design is user-friendly and consistent in both languages. Also likely designs any branding elements (logos, etc.).
- **Security/Compliance Expert:** Perhaps not full-time on project, but consults to ensure we meet GDPR, PCI, and other compliance. Could be an external audit role.
- **Customer Support / Community Manager:** As we launch, this role becomes important to interact with users (answer questions, moderate listings, etc.). Could be shared among team initially.

This timeline is an estimate; actual development could overlap phases (e.g., while core e-commerce is being built, the AI engineer can parallel work on chatbot). The total timeline roughly 4-5 months to launch an MVP. If the team is larger or smaller, adjustments would be made (e.g., a smaller team might take longer and combine roles, a larger one could do some phases in parallel).

Agile Methodology: We can implement this in sprints (say 2-week sprints). Each sprint should ideally deliver some increment (e.g., Sprint 1: basic product listing works; Sprint 2: cart and checkout; Sprint 3: chatbot MVP; etc.). Regular sprint reviews can keep stakeholders updated.

Communication: Have regular stand-ups, use tools like Slack for communication, and JIRA or similar for task tracking. Ensure bilingual team members or translation for any content creation (someone might need to translate all the static text to French during development).

By clearly defining phases and roles, everyone knows their responsibilities and the project can progress in a structured way. The above roles might be multiple hats on one person in a small team, but the responsibilities remain. Regular checkpoints (end of each phase or sprint) will allow us to adapt the plan if needed (maybe allocate more time to testing if new issues are found, etc.).

Finally, upon successful launch, we would conduct a retrospective to learn from the project execution and improve processes for future iterations. Then continue into maintenance mode or next phase of features as needed.

Sources:

- Saleor vs Medusa feature comparison 2 3
- Saleor marketplace integration and multi-vendor via Jetti 4
- Headless architecture and decoupling front/back for scalability 19
- n8n usage for data scraping and automation 14 13
- RAG benefits in e-commerce (accurate, up-to-date responses) 29 30
- Negotiation chatbot concept and limits (min price and original price bounds) 25 26
- Importance of multilingual AI chatbots for customer experience 22 23
- AWS Well-Architected e-commerce guidance (decoupled services, chatbots, etc.) 45

---

1 What is Saleor? A Modern, Headless Commerce Platform Built for ...
https://www.netguru.com/blog/what-is-saleor

2 3 6 7 10 11 12 Headless Commerce Platforms
https://headlesscommerceplatforms.com/

4 5 9 33 34 Review: Saleor vs Medusa Two Opensource Headless Ecommerce Platforms - DEV Community
https://dev.to/citizengovind/review-saleor-vs-medusa-two-opensource-headless-ecommerce-platforms-2h4b

8 16 39 40 41 Architecture | Saleor Commerce Documentation
https://docs.saleor.io/setup/architecture

13 n8n - Workflow Automation - GitHub
https://github.com/n8n-io

14 What's the most useful thing you're using n8n for at work?
https://community.n8n.io/t/whats-the-most-useful-thing-youre-using-n8n-for-at-work/5424

15 36 Building an Intelligent E-commerce Chatbot with RAG Technology | by Ankita Singh | Medium
https://medium.com/@aannkkiittaa/building-an-intelligent-e-commerce-chatbot-with-rag-technology-f3c1e07531bf

17 18 Headless Commerce Definition & Architecture Diagram
https://virtocommerce.com/blog/headless-commerce

19 37 38 42 43 44 45 Guidance for Web Store on AWS
https://aws.amazon.com/solutions/guidance/web-store-on-aws/

20 21 27 29 30 31 How Retrieval Augmented Generation (RAG) Is Used in E-Commerce | Tensorway
https://www.tensorway.com/post/rag-ecommerce-innovation

22 23 Building a Multilingual AI Chat-bot for eCommerce with Retrieval Augmented Generation
https://www.linkedin.com/pulse/building-multilingual-ai-chat-bot-ecommerce-retrieval-vasko-pozharski-11f6c

24 25 26 JETIR Research Journal
https://www.ijnrd.org/papers/IJNRD2403513.pdf

28 [PDF] E-COMMERCE CHATBOT FOR PRICE NEGOTIATION - IRJMETS
https://www.irjmets.com/uploadedfiles/paper/volume_3/issue_11_november_2021/17105/final/fin_irjmets1637170535.pdf

32 GraphQL API (Beta) — Saleor documentation - Read the Docs
https://saleor-fork.readthedocs.io/en/latest/architecture/graphql.html

35 Next.js Commerce, Vue Storefront, Saleor, Magento PWA Studio
https://naturaily.com/blog/next-js-commerce-vue-storefront-saleor-magento-pwa-studio-modern-headless-storefront-platforms-compared