ChatGPT

NovaPOS Cloud POS Platform – Current Architecture Overview

System Architecture Overview

NovaPOS is implemented as a distributed edge/cloud system optimized for multi-tenant SaaS delivery, following the original vision of combining in-store edge clients with a cloud microservice backend. Each retail location runs a **POS Edge Client** (React-based) that can operate independently in offline mode, while centralized cloud services (written in Rust) run in containers to handle core business logic and data [1] [2]. A loosely coupled, event-driven architecture connects these components via asynchronous messaging (Kafka) – for example, when an order is voided, the Order service emits an `order.voided` event for other services to consume, rather than calling them directly [3]. This design improves scalability and fault isolation: services don't block each other, and real-time events propagate through Kafka topics (e.g. `order.completed`, `payment.completed`) that decouple workflows [4]. Multi-tenancy is enforced throughout the stack: every record is tagged with a Tenant ID and requests must include an `X-Tenant-ID` header matching the user's JWT claims, ensuring retailers' data remains isolated [5] [6]. Overall, the architecture emphasizes modular microservices, resilient offline-capable clients, and secure multi-tenant operation – aligning with the original proposal, though some implementation details have evolved as described below.

Edge POS & Offline Mode

The **Point-of-Sale (POS) Edge Client** is a React web application (packaged for tablets or kiosks) that runs in each store location. It provides an intuitive touchscreen interface for cashiers – supporting product scanning, cart management, payments, and receipt printing. Critically, the POS is designed for **offline-first** operation: it caches key data (product catalog, prices, tax rules) in the browser and logs transactions locally if the network is down [7]. In the current implementation, the POS maintains an **offline transaction queue** (persisted in `localStorage` or IndexedDB) to record sales when offline, and automatically retries syncing them to the cloud once connectivity is restored [2]. For example, if internet drops, a cashier can still ring up sales – the orders are saved locally with a generated idempotency key and later sent to the Order service when back online [2]. Similarly, any price or product updates from the cloud are deferred and applied to the POS cache when the device reconnects, keeping data consistent. This approach ensures uninterrupted store operations – even without internet, the POS can function fully, then reconcile with the cloud later (a best practice for retail systems [8]). Currently, the offline sync mechanism is functional but basic: orders are eventually posted when online; however, the UI still uses cached receipt data, which can lead to slight discrepancies once the cloud data "catches up" (a known gap to be refined) [9]. All edge-to-cloud communications use TLS encryption and device authentication, so even when operating offline-first, security is not compromised [10].

Cloud Backend & Microservices

In the cloud, NovaPOS is composed of a suite of **containerized microservices** (written in Rust for performance and safety). These are deployable on Kubernetes or similar orchestration for auto-scaling and high availability, though in development they run via Docker Compose for simplicity [11] [12]. Each microservice owns a specific domain of the system, exposing HTTP APIs (primarily RESTful JSON; GraphQL was considered but is not implemented at this stage). The core services include: **Auth Service**, **Product Service**, **Inventory Service**, **Order Service**, **Payment Service**, **Integration Gateway**, **Customer Service**, **Loyalty Service**, and **Analytics Service** (detailed in the breakdown below). Services communicate with each other asynchronously by emitting and consuming events via Kafka, as well as through direct REST calls for certain synchronous operations. For instance, the Order service does a real-time call to the Inventory service when placing an order (to reserve stock) [13], but for other workflows it relies on events – when an order is completed, it publishes an `order.completed` event to Kafka, which the Inventory and Loyalty services consume to adjust stock levels or award points in the background. This **event-driven pattern** allows the platform to remain responsive: a checkout doesn't wait for downstream updates to finish, improving throughput and decoupling components [14] [15]. All services share a single multi-tenant PostgreSQL database for now (with tables namespaced by service domain and a tenant_id on every record) [16]. This provides a straightforward data store while ensuring tenant data isolation at the row level. (In future, services could be given separate databases or schemas, but the current MVP uses one logical DB for simplicity, running migrations for each service into the shared schema [17].) Each microservice runs on its own port (e.g. Order on 8084, Product on 8081, etc.) and exposes health endpoints for monitoring [18]. Common infrastructure services are also deployed: **Postgres** for the main database [19], **Redis** for caching and transient data [20], **Kafka** (with Zookeeper) for the message broker [21], and **HashiCorp Vault** for managing secrets (API keys, encryption keys) in secure storage [22] [23]. The use of containers and infrastructure-as-code (Terraform scripts are provided for AWS provisioning) sets the stage for cloud deployment in a scalable cluster environment. In summary, the cloud backend consists of multiple independent services that collectively provide all POS functionality, communicating over secure HTTP and Kafka, and orchestrated as a cohesive platform.

Multi-Tenancy & Data Isolation

Multi-tenancy is a fundamental design aspect: NovaPOS's single deployment serves multiple independent retail clients (tenants) while keeping their data siloed. **Tenant Isolation** is enforced at both the data layer and application layer. In the database, nearly every table includes a `tenant_id` column, and indexes like `idx_products_tenant` ensure efficient tenant-scoped queries [16]. For example, the Products table and Orders table each record which tenant (retailer) the data belongs to. On the application side, all API requests require an `X-Tenant-ID` header, and the platform's JWT-based authentication includes the tenant ID as a claim. A common security middleware (`SecurityCtxExtractor`) checks that the `X-Tenant-ID` header matches the authenticated user's tenant claim, rejecting the request if not [5] [6]. This prevents any cross-tenant data access, even if a request is malformed or a client tries to spoof an ID. Additionally, **Role-Based Access Control (RBAC)** is implemented to restrict functionalities within a tenant. Roles such as "cashier", "manager", "admin", etc., are assigned to user accounts, and the frontends and services enforce role requirements on sensitive operations. For instance, the Admin Portal wraps its routes with `RequireAuth` and `RequireRoles` components – meaning pages like user management or product management are only shown to users with Manager/Admin roles [24]. On the backend, microservice endpoints use guard functions that ensure the caller has the necessary role or capability for that action (e.g. only an admin can create products or issue refunds). These role checks are centralized via the Auth service's

JWT claims and a common authorization library, so that **principle of least privilege** is maintained consistently across services. Each tenant's data and keys are also segregated in memory and cache; for example, in the Customer service, personal data fields are encrypted per-tenant using a tenant-specific encryption key (itself protected by a master key) [25] [26]. This envelope encryption scheme ensures that even at rest, one tenant's sensitive data cannot be accessed or decrypted by another tenant's context. In summary, NovaPOS's current implementation maintains strong tenant boundaries: every request and data access is scoped to a tenant ID, and code-level checks prevent any accidental leakage or unauthorized access between tenants, upholding the security and privacy requirements of a multi-tenant SaaS.

Secure Integration & Omnichannel

To support omnichannel retail scenarios and third-party integrations, NovaPOS includes an **Integration Gateway** service acting as a secure facade for inbound and outbound integrations [27]. This microservice (running on port 8083) centralizes all communication with external systems – for example, it exposes certain APIs for e-commerce platforms or mobile apps to interact with NovaPOS, and it handles incoming webhooks from partners like payment processors. All integration endpoints are protected by API keys or OAuth tokens and are rate-limited using Redis-backed token buckets [28] [29]. Currently, the Integration Gateway's primary responsibility is payments (described in the next section), but it is designed to be extensible. In an omnichannel scenario, an online store could call the Integration Gateway's API to create an order or query inventory, or a service like Shopify could send a webhook to notify NovaPOS of an online purchase (so that NovaPOS can decrement in-store inventory for "buy online, pick up in store") [30]. The Integration Gateway would translate and forward these to the appropriate internal services (publishing events or calling internal APIs) without exposing internal endpoints directly. Outbound, it can send data to external systems: for instance, pushing daily sales totals to an accounting system, or updating a loyalty partner with points earned. All such external calls are done asynchronously where possible – e.g. a sale triggers an event which the Integration service picks up and then calls the external API – so that the core POS flow for the cashier is never stalled by an external integration [15].

Security is a key focus for integrations. The Integration Gateway verifies signatures on incoming webhooks (e.g. an HMAC signature from Coinbase Commerce is checked against a known secret) to prevent tampering [31]. It also audits API usage: each API key or integration partner call is logged, and abnormal usage can trigger security alerts (there is a Kafka topic for security alerts and optional webhook notifications for suspicious activity) [32]. Additionally, rate limiting is in place via Redis – by default, a given API key or IP is allowed a certain number of requests per minute, protecting the system from abuse or accidental overload [28]. All integration traffic uses HTTPS, and any sensitive credentials (like third-party API keys) are stored in Vault and not hard-coded in configs [22]. In sum, the Integration Gateway provides a controlled, secure layer for NovaPOS's omnichannel features: it allows NovaPOS to connect with external commerce, payment, and marketing platforms without exposing internal services, ensuring that data flows in and out are authenticated, authorized, and decoupled from the core runtime of POS operations.

AI/Analytics Stack

NovaPOS incorporates an **Analytics & AI Service** to provide business insights and machine learning-driven features across the retail data. This service (port 8082) consumes transactional and inventory events from the platform to aggregate data and perform analysis. In the current MVP implementation, the analytics service's functionality is still limited – primarily logging sales data and producing simple summary metrics – but the architecture lays the groundwork for more advanced AI modules. Transactional data from all

tenants are consolidated (in a tenant-separated way) to allow generating reports like sales by day, top-selling products, etc., which the Admin Portal can display. The vision is that this service will handle tasks such as **demand forecasting**, **inventory optimization**, and **anomaly detection**. For example, as sales events (`order.completed` events) stream in, the analytics service can update time-series sales records and use basic forecasting algorithms to predict future sales for each product [33] [34]. It can also monitor for anomalies – e.g. a sudden spike in refunds or voids might be flagged as suspicious (potential fraud or error) so that managers are alerted [35]. In the current state, these AI-driven features are minimal; the service might trigger simple alerts (using threshold rules) and compute forecasts using moving averages on the data in PostgreSQL. Results can be exposed via the Admin Portal's dashboards, such as an "Insights" section showing projected stockouts or recommending reorders.

The architecture is built to expand this component over time. The analytics service has access to a separate analytical database (or schema) where it can maintain denormalized datasets suitable for heavy read/query operations, ensuring the live transactional tables aren't slowed down [36]. At present, analytics queries run on the same Postgres instance as other data, but large-scale data could be offloaded to a warehouse or BigQuery in the future. The microservice approach means more advanced analytics (like training machine learning models or using external AI APIs) can be integrated without affecting the POS transactions. For example, a Python-based ML job could be scheduled to run on the data nightly to refine forecasts, and the results would just be read by this service and shown in the UI. The Analytics service also respects multi-tenancy and privacy: any cross-tenant machine learning would use anonymized data, and it enforces that each tenant's analytics view only includes their own data (with optional global trends presented in aggregate). While the **"AI"** in NovaPOS is nascent, the presence of this dedicated service and event pipeline means the platform is structured to become smarter over time, offering retailers increasingly sophisticated insights as more data is collected [37] [38]. As of now, the MVP demonstrates basic analytics functionality (simple reports and alerts), and more complex AI models (e.g. neural network-based forecasts or computer vision integrations) are planned for future iterations once sufficient data and feedback are available.

Component Breakdown

Below is a breakdown of the major components and services in the NovaPOS platform, summarizing their responsibilities and current implementation status:

POS Edge Client (Frontend)

This is the in-store **cashier application**, a React-based web front-end that runs on tablets or register PCs in retail locations. It provides a touch-optimized UI for product lookup (with barcode scanner support), building a cart, applying discounts, and completing a sale with various payment methods. The POS app communicates with the cloud services over HTTPS, typically to the API Gateway or directly to specific service endpoints (e.g. hitting the Order service for computing totals or creating an order). It attaches the cashier's JWT token and `X-Tenant-ID` on each request for authentication. Crucially, as described, the POS supports full offline operation: it caches the product catalog and other needed data in the browser (using IndexedDB/localStorage) so that scanning and selling can continue during internet outages [7]. It queues any transactions made offline and later syncs them to the cloud. In practice, after connectivity is restored, the POS will send the saved orders to the Order service's API in the background – using idempotency keys to avoid duplicates if some were already processed [2]. The UI then reconciles by fetching updated order status/receipts from the cloud. The POS also interfaces with **local hardware**: presently, it can trigger a print of receipts (through a connected printer or network print service) and is set up to integrate with payment

terminals for card dips/taps (for now, this integration is stubbed – the POS just calls the Payment API and waits for response, but the code anticipates a future connection to physical terminal SDKs). Overall, the POS Edge Client is the user-facing tip of NovaPOS, designed to be **fast** (optimized for <5 touches to complete a sale) and **resilient** (works offline). It logs minimal data locally (to avoid PII storage on the device) and pushes everything to the secure cloud once possible. Future enhancements will further streamline offline/online reconciliation and add more offline capabilities (e.g. customer lookup offline).

Back-Office Admin Portal

The Admin Portal is a React web application for store managers and corporate users to configure and monitor the system. It runs as a separate frontend (served on port 3001) and provides an array of administrative interfaces: product catalog management (CRUD for products and prices), inventory oversight (view stock levels, update counts, initiate transfers between stores), order management (view/search orders, process returns or voids), user management (invite staff, assign roles), and settings for integrations (like entering API keys for payment providers or e-commerce plugins). The Admin Portal is secured behind login – users must authenticate (via the Auth service's JWT) and have appropriate roles. The app uses React routers with an auth guard that checks for a valid token and wraps protected pages with role-based access requirements [24]. For example, if a cashier (role "cashier") somehow accesses the portal URL, they won't pass the `RequireRoles(['admin','manager'])` check for most pages. The portal communicates with various microservices' admin endpoints (mostly via a gateway or directly to services) using the user's JWT. It always sends the `X-Tenant-ID` header so that the backend knows which tenant's data to return. Current features in the portal include an **Orders workspace** (with filters and the ability to view detailed receipts and initiate returns on completed sales) [39], a **Products page** (list all products, create or edit a product, toggle its active status; this also retrieves product audit logs to show who changed what [24]), a **Users page** (list users, invite new user by email with role assignment), and **Settings** (for tenant-wide settings like tax configuration, integration API keys, etc.). The Admin Portal also surfaces analytics: e.g. sales charts or AI-generated insights appear on a dashboard/home page for managers, showing key metrics and any alerts (like "Stock running low for Product X" or "Unusual increase in refunds today"). Many of the advanced analytics and some management features (like detailed loyalty program management) are still in progress, but the portal is structured to easily add new sections. All interactions from the portal ultimately go to the microservice APIs – for instance, when an admin updates a price, the portal calls the Product Service's admin API endpoint, which in turn writes to the DB and emits a product update event. As with the POS, all admin API calls are logged and audited. This portal ensures that non-technical users (store staff) can use NovaPOS's powerful features through a friendly UI, rather than needing direct database access or technical tools.

Authentication & User Service (Auth Service)

The Auth Service (running on port 8085) handles all aspects of user authentication, authorization, and tenant onboarding. It issues JSON Web Tokens (JWTs) for authenticated users and exposes the `.well-known/jwks.json` endpoint for public key info [40], allowing services and frontends to verify tokens. NovaPOS uses **JWTs with RS256** signatures; the Auth service maintains an RSA key pair (with rotation support) and includes the public key via JWKS so that other services can validate tokens without sharing secrets. Users authenticate either via a login endpoint (e.g. username/password) or via SSO if integrated (future possibility). For the MVP, the Auth service verifies credentials (the current implementation uses Argon2 hashing for passwords and can integrate with OIDC providers as needed) and returns an access token and refresh token pair. It encodes claims such as `sub` (user ID), `tid` (tenant ID), roles, and

expiration in the JWT. All other services trust these JWTs; on each request, a common middleware checks the token and extracts an `AuthContext` that includes the user's roles and tenant [5].

In addition to login, the Auth Service manages **user accounts and roles**. It provides APIs for creating tenants (onboarding a new client organization), inviting users to a tenant, and listing or updating user roles. For example, an admin in the Admin Portal can invite a new manager – the portal calls Auth service to generate an invite (which might email a signup link) and assign the "manager" role on that user once they register. Role definitions (Admin, Manager, Cashier, etc.) and their permissions are centralized here, and enforced platform-wide (the *capabilities* system is evolving to map roles to fine-grained permissions in code) [41] [42]. The Auth service also implements **multi-factor authentication (MFA)** for privileged users: managers and admins can be required to use TOTP (authenticator app) codes on login. The security logic monitors login attempts and issues (the system has a concept of "suspicious activity" – e.g. many failed logins or logins from a new location – which can raise an alert for review) [43] [44]. All authentication events and key user actions are written to an audit log (and sent to the common audit event topic). The Auth service also issues and manages **API keys** for integrations: an admin can create an API key for, say, a partner system, which the Auth service will hash and store (only showing a prefix to the user) [45]. These keys are required for external services calling the Integration Gateway. In summary, the Auth Service is the gatekeeper of NovaPOS – it ensures only valid, authenticated users and partners access the system, asserts their identities and roles to other services via JWTs, and provides user management functions. This service significantly upholds the security posture by centralizing auth logic, enabling easy updates to policies (like password rules or token lifetimes) across the platform.

Product & Inventory Services

NovaPOS separates product catalog management and inventory tracking into two closely related microservices: the **Product Service** (port 8081) and the **Inventory Service** (port 8087). The **Product Service** is responsible for the master product catalog of each tenant. It manages items with fields like name, SKU, price, tax code, descriptions, etc. The service offers APIs to create, update, retrieve, and list products (with proper role checks – typically only managers or admins can create/update products). It also handles product audit logs: for every change (price update, deletion, etc.), an entry is added to an audit trail recording who made the change and when [24]. The Product Service emits events to notify other parts of the system of changes – e.g. when a new product is created or a price changes, it can emit a `product.created` or `product.updated` event (Kafka topics are set up for these) [4]. Downstream services like Inventory or Analytics can consume these to update their caches or data models. The **Inventory Service** manages stock levels for products, including multi-location inventory. It maintains records of how many units of each product are available at each store location (and a legacy "global" inventory for tenants not using location-specific tracking). The service provides APIs to adjust inventory (e.g. receiving shipments, manual adjustments), to transfer stock between locations, and to **reserve** stock. A key function is the **reservation system**: when an Order is being placed, the Order service calls Inventory's reservation API to put a hold on the items (so that concurrent sales don't oversell the stock) [13]. The reservation is a short-lived lock on a quantity which will either be finalized (on order completion) or released (on order failure/cancellation) – the Inventory service runs a background sweeper to expire any stale reservations after a TTL.

Inventory also emits events: for instance, it tracks low-stock thresholds. If an update causes a product's inventory to drop to or below a predefined threshold, the Inventory service will emit an `inventory.low_stock` event with the product info [46] [47]. This prevents spam – it only emits when crossing the threshold once, not on every single sale. That event could trigger, say, an email alert or an

update in the Admin Portal's dashboard to reorder that item. The Inventory service keeps an **audit log of inventory changes** as well (who adjusted stock, etc.) for accountability. Both services enforce tenant isolation on all queries, of course, using tenant_id filters. They also both utilize the **common money library** for price calculations and rounding, to ensure consistent currency handling across the platform [48]. In the current implementation, the core product and inventory flows are functional: products can be created/edited, inventory decrements when sales occur (or increments on returns), and basic reservation logic prevents overselling. A combined feature worth noting is **tax calculation**: the Order service, when computing totals, uses data from products (tax codes) and tenant tax settings to calculate taxes. The product has a tax classification (taxable or exempt), and tenants can set custom tax rates (even per location or POS instance) in a table; the Order compute API uses a resolver that checks overrides by POS or location, then tenant default [49] [50]. This involves the Product/Inventory domain as well. Going forward, these services will support more advanced catalog features (variants, bundles) and inventory workflows (purchase orders, automated replenishment suggestions via the AI). But as of now, they provide the essential catalogue and stock management underpinning the POS.

Order/Transaction Service

The **Order Service** (port 8084) is the transaction ledger of NovaPOS – it records every sale (and return or exchange) and orchestrates the checkout process. When a POS submits a checkout, it's the Order service that receives the order creation request. The Order service's responsibilities include: validating the order (ensuring products exist, quantities are available), calculating the final totals (by applying tax rules and any discounts via a `/orders/compute` endpoint), reserving inventory, interacting with the payment workflow, and ultimately persisting the order with all line items and payment status. Each order record includes details like which store location, which user/cashier processed it, timestamp, items sold (line items with quantity and price), subtotals, tax, total, and payment breakdown (cash, card, crypto, etc.). The service writes the order to the database and returns an order ID and summary to the POS. If the order is completed (paid), it marks it accordingly; if payment is asynchronous (e.g. waiting on a card authorization or crypto confirmation), it marks the order as `PENDING` and will update it later when the payment is confirmed [51]. Importantly, the Order service enforces **idempotency** for order creation: clients (like the POS) can provide an `idempotency_key`, and if a duplicate request comes in (e.g. due to a retry after a network glitch), the service will detect it and not create a duplicate order [2]. The POS offline mode leverages this by using deterministic idempotency keys for queued orders, so the same sale isn't double-booked when connectivity returns [2].

Once an order is created, the Order service emits events for other services: an `order.completed` event if the sale is finalized, or `order.voided` if it was voided, etc. Inventory service listens for `order.completed` to decrement stock (as a backup to the direct reservation call) and Loyalty service listens to award points. The Order service also has logic for **refunds and returns**: it can mark an order (or specific line items) as refunded, and it ensures that the same item isn't refunded twice. Currently, refund processing updates the order status (partial or full refund) and emits events, but it does **not** yet integrate back to payment gateways – meaning if a credit card sale is refunded in NovaPOS, the system records it, but the actual card refund has to be done externally (this is a gap to address) [52]. Similarly, **voids** (cancelling a sale in progress) are allowed – an API exists to void an order that was authorized but not captured, for example, which sets it to a Voided state and emits an event so inventory or loyalty can roll back any changes [3]. The Order service is also where **receipts** are generated: there's an endpoint to retrieve an order receipt (in plaintext or PDF format) which collates all info – this is used by the POS to print or email receipts after syncing.

In the current architecture, the Order service relies on synchronous calls to Inventory for reservations and to the Integration/Payment service for initiating payments, but otherwise uses events for async updates. Notably, for **pending payments** (card or crypto), NovaPOS uses a **publish/subscribe mechanism**: when an order is pending payment, the Order service will wait for a `payment.completed` or `payment.failed` event on Kafka (with the order ID) rather than blocking the HTTP request. The POS gets an immediate acknowledgment that the order is pending, and a background process (a Kafka consumer within Order service) will update the order to Completed or Failed when the payment service signals the result [53] [51]. This prevents a stalled UI if a payment takes a while (network latency, customer fiddling with card). It is implemented via the Kafka integration in the Order service (enabled with a feature flag for Kafka in non-Windows environments) which listens to those topics. If the Order service or Payment service restarts, this design ensures an in-flight payment can still complete later (the event will eventually be processed). The Order service also maintains some **administrative endpoints** for the admin portal: listing orders with filters, viewing an order's details, listing returns, etc., all scoped by tenant and with proper auth. It supports pagination and searching (by date range, or by transaction ID).

In summary, the Order/Transaction Service is the central hub for sales transactions in NovaPOS. It coordinates between the front-end, inventory, and payment components to turn a cart into a finalized sale record. It ensures consistency (via idempotency and reservation checks) and durability (writing to the DB and using events to propagate results). The current implementation covers basic POS transactions well, with improvements underway for handling more complex cases (like true exchanges, multi-order quotes, and deeper integration with payment refunds). Still, it provides the backbone ledger that any retail system needs, storing the truth of "who bought what, when, for how much, and how they paid."

Payment Services (Card & Crypto Payments)

Handling customer payments is split between two microservices in NovaPOS: the **Payment Service** (port 8086) and the **Integration Gateway** (port 8083), working in tandem. Together, they implement the platform's support for both traditional payment terminals and cryptocurrency payments, albeit with some interim stubs in the current MVP.

Card Payments (Traditional Payments): NovaPOS is designed to integrate with physical payment terminals (such as **Valor PayTech** devices) for in-person credit/debit transactions. In the current architecture, the **Payment Service** acts as a dedicated microservice that will interface with these payment gateways or terminal SDKs. At present, a full integration is not complete – instead, NovaPOS has a placeholder (stub) for this connection. When a POS transaction includes a card payment, the following occurs: the Order service marks the order as pending and calls the **Integration Gateway's** payment API (e.g. `/pay/card`) to initiate the card authorization [51]. The Integration Gateway, in turn, communicates with the Payment Service (over HTTP) which contains the logic to interact with the terminal or gateway. Since hardware integration is not yet implemented, the Payment Service currently simulates an immediate approval for card transactions (or could be configured to simulate declines). Essentially, it's a Valor integration stub – no actual card data is processed yet, which keeps NovaPOS out of PCI scope for now [51]. The Payment Service ensures that no sensitive PAN (card number) or CVV is ever stored or even transmitted through NovaPOS unencrypted; in a real integration, the card terminal would handle the card data and just return a token or approval code to Payment Service. Once the Payment Service (stub) "approves" the transaction, it publishes a `payment.completed` event into Kafka. The Order service consumes this event and updates the order from Pending to Completed (and triggers any receipt finalization) [53]. If the Payment Service had reported a failure, a `payment.failed` event would be published instead. This event-driven

completion allows the POS to be responsive – the cashier sees that the payment is processing and then gets an update that it succeeded or failed without the entire system blocking.

It's worth noting the **limitations** of the current stub: features like chip/PIN entry, contactless tap, signature capture, or EMV fallback logic are not yet integrated [52]. The Payment Service's stub does not handle edge cases (e.g. partial approvals, network timeouts, reversals of an authorization) – all such scenarios are acknowledged as gaps [52]. These will be addressed when the actual Valor (or another provider) SDK is connected. The architecture is ready for it: the Payment Service is the encapsulated component for that integration, isolating all PCI-sensitive operations away from other services, and once implemented, NovaPOS will remain mostly out of PCI scope since card data entry/encryption happens on the Valor terminal itself [54]. The Payment Service also logs necessary info such as transaction reference IDs and can handle **voids/refunds** by communicating with the gateway (currently, voids/refunds don't reach the external processor – voiding just stops our local order, and refunds are recorded internally but would need manual external refund – this is also on the roadmap to automate [52]).

Crypto Payments: NovaPOS natively supports cryptocurrency transactions, specifically stablecoins like **USD Coin (USDC)** via **Coinbase Commerce**. This is handled through the **Integration Gateway** service primarily. When a customer chooses to pay with crypto, the POS calls the Integration Gateway's crypto checkout endpoint (e.g. `/pay/crypto`), including the amount. The Integration service uses Coinbase's API (with the merchant's API key stored in Vault) to create a charge – essentially generating a payment request for a certain USDC amount [55]. Coinbase returns a payment **checkout URL or QR code**, which the Integration Gateway relays to the POS. The POS then displays the QR code for the customer to scan with their crypto wallet app, or it can open the hosted Coinbase checkout page. At this point, the sale is in a pending state (just like a card payment) – the Order service has it as Pending waiting for confirmation. Coinbase, upon the customer actually sending the USDC, will send a **webhook** to NovaPOS's Integration Gateway (to a preconfigured callback URL) indicating that the payment was successful (or expired/failed if not paid in time). The Integration Gateway verifies this webhook (using the shared secret to ensure it's genuine) and then publishes a `payment.completed` event (or `payment.failed`) similar to card, which the Order service consumes to complete the order [51]. Alternatively, there is also an internal polling or timer in case webhooks are delayed – but generally Coinbase's webhook drives the flow. Once the payment is confirmed, funds are in the merchant's Coinbase account in USDC, and NovaPOS could automatically initiate conversion to USD if configured [56]. In the MVP implementation, this crypto flow is partially in place: the Integration Gateway has the logic to handle Coinbase webhooks and mark orders paid. It was tested in a sandbox mode. It does highlight security – currently, the webhook verification and idempotency for crypto could be improved (e.g. ensure we don't accept duplicate notifications, handle key rotation) [57], but these are known items to harden. From the cashier's perspective, the crypto payment is just another option – they don't need to know the blockchain details. The receipt will note that it was paid in USDC and include a transaction hash or Coinbase reference.

Both Payment Service and Integration Gateway ensure that **no sensitive secrets or data are exposed**. Card data isn't stored on NovaPOS servers at all, and crypto private keys are not handled by NovaPOS (Coinbase manages the wallets). The Integration Gateway stores only tokens or API keys (encrypted via Vault) and references to transactions. Communications between services (Order, Payment, Integration) use secure channels internally as well. The system logs events for audit – e.g. a `order.completed` will include method "crypto" or "card". As of now, NovaPOS can handle cash (immediate complete), card (via stub -> event complete), and crypto (via Coinbase -> webhook complete) in a unified way from the order perspective [51]. **Cash** is simplest: the POS flags it as cash, the Order service immediately marks the order

paid and complete in one step (no external call, it just records it and emits `order.completed` inline) [51]. Thus the platform was built to accommodate **multiple tenders** seamlessly. The code generalizes payment status: pending vs completed, with events driving the finalization, which sets the stage for adding other methods (gift cards, PayPal, etc.) down the road.

In conclusion, the Payment portion of NovaPOS's architecture, though currently relying on stubs and third-party sandbox integration, provides a clear pathway for full production readiness. It isolates payment logic in specialized services, uses asynchronous events to avoid coupling and delays, and supports an emerging range of payment methods (traditional and crypto). Once the few remaining integration gaps – like hardware terminal integration, robust failure handling, and upstream refund automation – are closed [52], NovaPOS will have a comprehensive, flexible payment processing capability fully integrated into the POS experience.

Customer & Loyalty Services

NovaPOS includes dedicated services for managing **Customer profiles** and **Loyalty programs**, reflecting the importance of customer-centric features (beyond just transactions). The **Customer Service** (port 8089) maintains a database of end-customer information for each tenant: think of customers as the shoppers who buy from the retailer, as opposed to the users (staff) in the Auth service. This service stores details like customer name, contact info, loyalty membership ID, purchase history references, and any CRM-type data. Given privacy requirements, the Customer service is built with **PII encryption** in mind – sensitive personal fields (email, phone, addresses) are stored encrypted using per-tenant data encryption keys [26]. The environment variable `CUSTOMER_MASTER_KEY` is provided (as a base64 string) to derive and protect those tenant keys [58]. This means if someone were to access the raw database, they wouldn't easily read customer personal data without the keys. The service provides APIs to create/update customers (e.g. adding a new customer profile during checkout or updating contact info) and to fetch customer info (for instance, the POS might search by phone or email to find a customer for loyalty point lookup). It also supports **GDPR compliance** features – endpoints to export a customer's data or delete a customer entirely (which would be soft-delete or anonymize in practice) are planned, ensuring the platform can honor data privacy regulations. Role-based access is enforced here too: usually cashiers can look up customers and enroll them, but only managers or certain roles can delete or export data (mapped to a `GdprManage` capability) [41] [59]. The Customer service integrates with sales: each Order record can optionally link to a customer (the Order service stores a `customer_id` if provided), allowing building of purchase history. Currently, linking a customer to a sale in the POS will let the admin portal later show that sale under the customer's profile. The Customer service does not duplicate order data, but it can query orders by customer via the Order service if needed.

The **Loyalty Service** (port 8088) is focused on loyalty programs – typically tracking points or rewards that customers accrue and redeem. In the MVP, this service is relatively simple: it can log points earned per purchase and track a points balance for each customer (by tenant). For example, a retailer can configure a rule like "1 point per $1 spent". When an order.completed event comes in, the Loyalty service will consume it and calculate points to add to the customer's balance (if that order had an associated customer) [60]. It stores these in a `loyalty_points` table (with columns for customer, order, points, etc.). The service provides APIs to get a customer's loyalty balance and history, and to adjust points (administrators can do manual adjustments or corrections via an admin API). It might also handle **reward redemption**: e.g. an endpoint to redeem points for some benefit (this part is likely in development – the MVP Gaps analysis notes that redemption workflows and tiering are not fully implemented yet) [61]. The POS, when ringing up

a sale, can display the customer's current points and perhaps allow applying a reward (for instance, 100 points for a $5 discount). In such a case, the POS would call an API (maybe in Order or Loyalty service) to apply a reward, which would deduct points and apply a discount line item. The specifics of this flow are still being ironed out; at minimum the infrastructure to track and query points exists.

Both Customer and Loyalty services work hand-in-hand: typically a customer profile is required to track loyalty points, and thus the Loyalty service references customers by ID. These two services were separated to allow the loyalty logic to evolve (or even be modular per tenant – some tenants might not use loyalty features). They also separate potentially heavy read/write patterns: loyalty transactions could be frequent (every sale) and we may want to scale that service or optimize it independently of the core customer directory. As it stands, after each sale, an event is sent that eventually results in a loyalty update, all asynchronously. If the loyalty service is down for a bit, sales still go through; the points can catch up when it's back (eventually consistent). The architecture envisions possibly integrating with external loyalty or CRM systems too – the Integration Gateway could forward events to an outside loyalty provider if a tenant uses one, but still NovaPOS would update its local record for the unified view.

Currently identified gaps in this domain include loyalty **tiering** (e.g. Silver/Gold customer levels) and offline visibility (ensuring the POS can show loyalty info when offline by caching some data) [61] . Those are slated for future development. Nevertheless, NovaPOS now has a solid foundation for customer relationship management: the Customer service ensures all shopper data is in one place and protected, and the Loyalty service builds on that to drive repeat business through rewards. These services emphasize NovaPOS's approach to not just process transactions, but also foster ongoing customer engagement by remembering customers and incentivizing them to come back.

Integration Gateway Service

(Covered in Secure Integration & Omnichannel section above – the Integration Gateway is a key piece for external API access and is already described. Its role spans multiple domains, notably payments, and future e-commerce or third-party system integrations, so it has been detailed earlier rather than in isolation.)

Reporting & Audit Logging

(Note: In the original architecture proposal, a separate Reporting/Audit service was envisioned. In the current implementation, dedicated services for reporting or auditing have not been split out – instead, these concerns are handled within existing components or common libraries.) NovaPOS does implement extensive **audit logging** and has foundational support for reporting. Every microservice uses a common **Audit** library to produce audit events for significant actions (e.g. user logins, permission failures, data changes) which are sent to a Kafka topic for audit logs [62] [63] . Although a standalone "audit-service" consumer (to read those events and write to an immutable audit store) is not yet in production, the groundwork is there – an audit log table schema exists in the design, and the plan is to have a background worker (or future microservice) consume the `audit.events` topic and persist events in an append-only, tamper-evident log store [62] [63] . For now, some services directly record audit trails in their own domain (e.g. product changes in a product_audit_log table [24]). These audit trails allow administrators to trace actions – e.g. "who changed the price of item X" or "which user deleted customer Y".

On the **reporting** side, the Admin Portal currently retrieves operational reports by querying the Order service (for sales totals, etc.) and the Analytics service (for aggregated insights). There isn't a separate

reporting service generating PDFs or complex reports yet; however, the data is all in Postgres and can be accessed via APIs. The system can generate on-demand exports like CSV lists of products or sales, and those are done by the respective services (Product or Order service providing an export endpoint). The need for complex reports (financial statements, multi-store comparisons, etc.) might lead to a reporting service or leveraging the Analytics service as it matures. Already, some scheduled jobs (not user-facing) in Analytics service can compute daily summaries.

Observability: All services expose Prometheus `/metrics` endpoints and are configured for log aggregation. A Prometheus and Grafana stack is included in the deployment for monitoring [64] [65]. Key metrics like HTTP request counts, error rates, and domain-specific metrics (e.g. number of failed payments, low-stock alerts triggered) are recorded. For example, a standardized counter `http_errors_total{service,...}` is incremented on each service whenever an API returns an error, labeled by error code [66] [67]. This enables dashboards to quickly flag if any service is experiencing an unusual volume of errors. Tracing is not fully implemented yet, but correlation IDs (trace_ids) are passed in the logs and responses to assist with tracing a transaction across services [68]. The logs themselves are structured JSON, and critical events (like security alerts) can be forwarded to external systems (the Integration Gateway can call a webhook for serious security events if configured [32]).

In summary, while NovaPOS doesn't yet have a dedicated reporting microservice or a fully standalone audit service, it has integrated these needs into the current architecture: robust audit logging via events and tables, and basic reporting via existing services and the budding analytics component. This ensures that even at the MVP stage, the system can answer the essential questions ("what happened, when, and by whom") and support monitoring and compliance, with room to grow these capabilities into separate services as the system scales.

Conclusion: The NovaPOS platform's architecture, as implemented today, aligns closely with the initial architectural proposal's goals: a cloud-native, multi-tenant POS with offline resilience, modular microservices, and a forward-looking integration of AI and multi-channel capabilities. There have been some adjustments in implementation – for example, splitting certain services (Product vs Inventory, Customer vs Loyalty) for clarity, introducing an Integration Gateway layer specifically for external interactions, and deferring some planned features like a dedicated audit service or full hardware payment integration. These changes have been made to better accommodate the practical development workflow and to incrementally deliver a working MVP. The result is an architecture that is **modular** (each service can be developed and scaled independently), **resilient** (offline mode and asynchronous messaging prevent single points of failure in store operations), and **secure** (multi-tenant isolation, encryption of sensitive data, and thorough auditing). The system is on a solid foundation for the next phases of development – as new requirements emerge (from scaling up during the World Cup launch, or adding new features like advanced loyalty tiers or AI models), the existing architecture can support those with minimal disruption due to its decoupled, event-driven nature [69]. In essence, NovaPOS has evolved from the proposal into a living system that reflects both the original vision and the practical lessons learned during implementation, providing a clear picture of "where we are now" in our architectural journey.

Sources: The above architectural description is based on the current NovaPOS codebase and design documents, including the original Architecture Proposal [1] [54], development runbooks and analysis of MVP gaps [70] [2], and the repository's README/configuration which illustrate how services and infrastructure are set up [12] [71]. These references and in-line citations point to the exact sections of the

code or docs that substantiate each aspect of the architecture, ensuring this documentation of the existing system is accurate and traceable to the source implementation.

---

1 7 8 10 14 15 27 30 33 34 35 36 37 38 54 55 56 60 69 NovaPOS Cloud POS Platform – Architecture & Implementation Proposal.docx
file://file-G4ZUkQVgpuLxYi51GaCLeF

2 3 9 13 24 39 51 52 53 57 61 70 MVP_Gaps.md
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/doc/analysis/MVP_Gaps.md

4 11 12 19 20 21 22 23 31 32 40 58 64 65 71 docker-compose.yml
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/docker-compose.yml

5 6 guards.rs
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/services/common/auth/src/guards.rs

16 1001_create_products.sql
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/services/product-service/migrations/1001_create_products.sql

17 migrate-all.ps1
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/migrate-all.ps1

18 app.rs
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/services/order-service/src/app.rs

25 26 28 29 43 44 45 62 63 security_plan.txt
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/docs/security/security_plan.txt

41 42 59 capabilities.md
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/docs/development/capabilities.md

46 47 README.md
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/services/inventory-service/README.md

48 README.md
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/README.md

49 50 runbook.md
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/docs/runbook.md

66 67 68 services_notes.txt
https://github.com/datawrangler05/novapos/blob/d797e8874712bc716599dc65e627a6fe2a0a1c73/services_notes.txt