ChatGPT

NovaPOS System Evolution Plan

Including Product Images in the Catalog

It has been confirmed that the product catalog should **include product images** wherever possible. Currently, the product data model includes fields like name, price, description, etc., but it does not yet store an image URL or blob [1]. On the frontend, however, the POS app expects an `image` field for each product (used to display the product's picture) [2]. To resolve this discrepancy, we will add an `image` **attribute** to the product schema and service API.

Schema Change: We will alter the `products` table to include an `image` URL field (e.g. `ALTER TABLE products ADD COLUMN image TEXT;`). This field will store a link or path to the product's image. Storing just a URL/path is preferred over raw binary to keep the database lean; actual image files can reside in object storage or a CDN, with the URL referencing them. Additionally, we should consider a **default image** for products that have none. For instance, if a product is created without an image URL, the service can assign a placeholder image link (so the frontend still shows a generic image instead of a broken link).

Service/API Update: The **Product Service** will be extended to handle the new image field in its create and update endpoints. This means adjusting the `NewProduct` and `Product` data structures and including the image URL in responses. For example, the `create_product` handler will accept an `image` URL in the JSON payload and insert it into the DB, and `list_products` will serialize the image URL in the output JSON. This ensures that when the POS or Admin frontend fetches products, each product comes with an image URL to display.

By including images for all products, we enhance the user experience (cashiers and customers can visually identify items). It also lays groundwork for future features like image management (e.g. uploading new images for products via the admin portal). **In summary**, the product schema and service will be updated so that every product record can include a link to an image, with sensible defaults if not provided, thereby aligning the backend with the frontend's expectations [1] [2].

Deferring Product Service Extension Until Stabilization

The next consideration is how and when to **extend the Product Service** with additional capabilities. After initial implementation of core product features (CRUD, basic fields, listing), it is prudent *not to immediately add complex extensions* until the product functionality is stable and validated. In other words, **no major extensions are done right now**; **once the product service is "situated" (stable in production)**, we can then plan and implement extensions. This staged approach ensures we don't introduce unnecessary risk or complexity during early rollout.

Planned Extensions: Once the product service is stable, one extension will be to integrate product data more deeply with other domains. A likely enhancement is linking products to supplier information. For example, after we introduce a Supplier Management component (discussed below), we can extend the

product service to record each product's primary supplier or a list of suppliers. This might involve adding a `supplier_id` field to the product or establishing a separate mapping between products and suppliers. However, doing this **prematurely (before supplier management exists)** would be futile. Thus, we schedule it for later, *after* the supplier service is in place. Another future extension could be adding product categorization or variants. We noticed the POS frontend code filters products by category if available [3], yet currently category isn't a first-class field. In a post-MVP phase, we can extend the product schema to include a category and perhaps a richer taxonomy (enabling features like category filtering, which the UI already anticipates).

Rationale: By **deferring these enhancements**, we allow the core product features (including the new image field) to solidify. The team can focus on fixing any immediate issues with product CRUD and image handling without juggling additional complexity. When usage has stabilized, we then **extend the product service** in a controlled way — adding new fields (like supplier links or category), new endpoints (e.g. product search or bulk import), or performance improvements (caching, indexing) as needed. This phased approach aligns with best practices for incremental system evolution, ensuring that each component is robust before adding more responsibilities.

Audit Logging and Supply/Demand Planning Integration

With the product basics covered, we now turn to implementing **audit-compliance logging** and scaffolding a new **Supplier/Demand Planning module**. These are significant features that affect the overall architecture, data schema, and service ecosystem. After deliberation, the decision is **"Yes"** – we will proceed with these initiatives, entailing:

- **(a) Architectural Evolution**,
- **(b) Schema Changes**,
- **(c) Service/API Additions**, and
- **(d) Scaffolded Code Sketches** (providing initial code outlines for the new components).

Each of these aspects is detailed in the subsections below.

a. Architectural Evolution

Figure: Updated NovaPOS microservices architecture with event-driven communication. The new Supplier Service (green) subscribes to inventory events to facilitate restocking. Each service persists its data to the shared database, and audit logging (not shown) captures key events across services.

The NovaPOS architecture will evolve to incorporate the new features while preserving its cloud-native, microservice-based design. Key architectural changes include introducing a **Supplier Service** and an **audit logging mechanism**, and leveraging the existing event bus (Kafka) for tighter integration:

- **New Supplier Service:** We will create a dedicated microservice responsible for **Supplier Management** (handling supplier data and procurement processes). This service will be analogous to existing domain services (e.g. Product, Order, Inventory services) – independently deployable and focused on a specific business area. In a supply chain context, it makes sense to keep supplier-related logic separate from product or inventory logic [4]. The Supplier Service will manage suppliers (name, contacts, etc.), and process **restock orders** when inventory is low.

- **Event-Driven Integration:** NovaPOS already employs an event-driven approach using Kafka as a central event bus connecting services. We will expand on this pattern for the new functionality. For example, when inventory levels drop below threshold, the Inventory Service currently raises an `"inventory.low_stock"` event on Kafka. Today, that event is only logged as a warning by the Analytics Service [5], but no further action is taken. Going forward, the **Supplier Service will subscribe** to `"inventory.low_stock"` events and react by initiating a re-stock process (e.g. creating a purchase order to a supplier). This **choreography pattern** allows the new service to react asynchronously without tight coupling – a proven approach in microservice architectures [6]. Similarly, the Product Service's `"product.created"` event (emitted when a new product is added [7]) is already consumed by Inventory Service to seed stock records. We will ensure all relevant events (product updates, order completions, etc.) are tapped for audit logging and planning logic as needed.

- **Audit Logging Mechanism:** For audit and compliance purposes, we'll introduce a cross-cutting logging capability. Architecturally, there are a couple of options:

- *Decentralized:* Each service directly writes an audit log entry to a common datastore whenever a significant action occurs.
- *Centralized via Event Bus:* Services publish audit events (e.g. "order.created", "product.updated" with user context) to a dedicated audit topic, and an Audit Logging component (could be a lightweight service or even an ETL pipeline) consumes these and writes to the audit log store.

We lean towards using the event bus for auditing as well, to minimize direct coupling. In practice, this means on each important user action, the service would produce an audit event (much like other domain events). A new **AuditLog Service** (or even the existing Analytics Service extended) could subscribe to all audit events and persist them. This way, auditing doesn't affect the latency of the main transactional flow – the log writes happen asynchronously. The audit log store could be a separate database/table optimized for queries by security/compliance teams.

- **Multi-Tenancy and Security:** All new interactions will respect the multi-tenant design (every event and log entry includes the tenant ID, and each service isolates tenant data). The audit logging especially will record *who* did *what* and *when*, which implies we need the user identity in context. We may extend our authentication token or request headers to pass a user ID to services (so the Product/Order/etc. services know which user is performing an action, and can include that in the audit record). Ensuring this propagation of user context is an architectural consideration that might involve the Integration Gateway or an API gateway injecting user info into requests.

In summary, the architecture will gain **two new components** (Supplier Service and Audit Logging facility) and new **event flows**. The figure above illustrates these changes: the Supplier Service consumes inventory events to automate supply actions, and all services produce audit events for a centralized log. This evolution maintains loose coupling and scalability, as each concern is handled by an appropriate service and asynchronous messaging.

b. Schema Changes

Implementing audit logs and supplier planning will require updates to the database schema. We need to introduce new tables and modify some existing ones. All schema changes will be managed via migrations for maintainability. The main changes include:

- **Product Image & Category:** As noted earlier, the `products` table will gain an `image` column (TEXT) to store product image URLs. We might also add a `category` column (since category filtering is expected) or a separate `categories` table with a relationship, but category support can be a future enhancement. For now, the image field is priority to support product images [1].

- **Audit Log Table:** We will create a new table (e.g. `audit_log`) to store audit trail entries. Each row in `audit_log` will capture an event such as a record creation, update, deletion, or a security action. Typical fields would be:

- `id`: primary key (UUID).
- `tenant_id`: which tenant the action pertains to.
- `timestamp`: when the event happened.
- `user_id`: the ID of the user (or service account) who performed the action.
- `entity`: what type of object or module (e.g. "Product", "Order", "Supplier").
- `action`: what was done (e.g. "CREATED", "UPDATED", "DELETED", "LOGIN", etc.).

- `description` or `details`: a JSON or text field capturing relevant details (e.g. changed fields or a summary).
  This design aligns with guidelines for audit logs, which recommend recording the **who, what, when, where** of each important event [8]. For example, an entry might say: *User 123 (Alice) updated Product 456 at 2025-09-19T10:15Z (changed price from $9.99 to $8.99).* We will ensure the audit table is indexed (e.g. by tenant and timestamp) to allow efficient querying by admins/compliance officers.

- **Suppliers and Purchase Orders:** We will design schema for supplier management. Likely two main tables:

- `suppliers`: stores supplier master data. Fields: `id` (UUID PK), `tenant_id`, `name`, `contact_info` (could be JSON with email/phone/address), and perhaps `notes` or `rating` for performance. Each tenant will manage their own supplier entries.

- `purchase_orders` (or `restock_orders`): stores orders placed to suppliers for restocking inventory. Fields: `id` (UUID PK), `tenant_id`, `supplier_id` (foreign key to suppliers), maybe `product_id` (if each PO is for a single product) or a separate line-items table if one order can cover multiple products, `quantity`, `status` (e.g. "PENDING", "ORDERED", "RECEIVED"), and timestamps (`created_at`, etc.). Initially, we might simplify and say one purchase order corresponds to one product low-stock event (one product per order). In the future, we could allow grouping multiple low-stock items for the same supplier into one order.
  We will also add any necessary **foreign keys** and indices. For example, `purchase_orders.product_id` would reference `products(id)`, and we'd index on `status` to query pending orders. The schema will ensure that when inventory is replenished, we can trace it via these tables.

- **Inventory Adjustments:** The existing `inventory` table (which tracks product quantities per tenant) might not need changes for basic supplier integration, but we may later add a column like `reorder_level` or `preferred_supplier_id`. For now, the `threshold` field in inventory is used for low-stock alerts [9] [10]. We will rely on that, but if we want dynamic thresholds or lead times, those could become fields either in `inventory` or in a new table mapping products to supplier lead times. For instance, if supplier A takes 2 weeks to deliver an item, the system might trigger a restock event earlier. These are advanced planning considerations that can be reflected in schema later (e.g. a `lead_time_days` field in `suppliers` or a join table between products and suppliers to override lead times per product).

All these schema changes will be applied via migration scripts so that they can be version-controlled and run consistently on all environments. Below is a pseudo-SQL snippet illustrating some of the new schema definitions:

```
-- Add image column to products (if not already added in a prior migration)
ALTER TABLE products
  ADD COLUMN image TEXT DEFAULT NULL;

-- Create suppliers table for supplier management
CREATE TABLE suppliers (
    id UUID PRIMARY KEY,
    tenant_id UUID NOT NULL,
    name TEXT NOT NULL,
    contact_info TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
    -- ... any other fields ...
    CONSTRAINT fk_tenant FOREIGN KEY (tenant_id) REFERENCES tenants(id)
);

-- Create purchase_orders table for restocking inventory
CREATE TABLE purchase_orders (
    id UUID PRIMARY KEY,
    tenant_id UUID NOT NULL,
    supplier_id UUID NOT NULL REFERENCES suppliers(id),
    product_id UUID NOT NULL REFERENCES products(id),
    quantity INTEGER NOT NULL,
    status TEXT NOT NULL,            -- e.g. 'PENDING','ORDERED','RECEIVED'
    created_at TIMESTAMP WITH TIME ZONE DEFAULT now(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT now()
    -- ... possibly a expected_delivery_date, etc. ...
);

-- Create audit_log table for compliance/auditing
CREATE TABLE audit_log (
    id UUID PRIMARY KEY,
    tenant_id UUID NOT NULL,
```

```
    user_id UUID,                  -- who performed the action (null if system)
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT now(),
    entity TEXT NOT NULL,          -- e.g. 'Product', 'Order', 'Supplier'
    action TEXT NOT NULL,          -- e.g. 'CREATE', 'UPDATE', 'DELETE',
'LOGIN'
    description TEXT               -- details of the event (what changed, etc.)
);
```

These changes provide the foundation for our new features: the audit_log table will capture a **chronological narrative of user actions** for security reviews [11], and the suppliers/purchase_orders tables will support end-to-end supply management from low-stock detection to reordering and receiving stock.

c. Service and API Additions

With new schema and responsibilities, we will add corresponding **services and API endpoints** to utilize them. The major additions are:

- **Supplier Service APIs:** The Supplier Service will expose REST endpoints for managing suppliers and purchase orders. For example:
- `GET /suppliers` – list all suppliers for the tenant (with pagination perhaps).
- `POST /suppliers` – create a new supplier (with name, contact info in body).
- `PUT /suppliers/{id}` – update a supplier's info.
- `DELETE /suppliers/{id}` – (possibly) remove a supplier (if no orders linked, or mark inactive).
- `GET /purchase_orders` – list purchase orders (optionally filter by status or supplier).
- `POST /purchase_orders` – create a new purchase order (this might be triggered internally by an event, but we also expose it in API in case an admin wants to manually order).
- `PUT /purchase_orders/{id}` – update order status (e.g. mark as "ORDERED" when the order is placed, or "RECEIVED" when stock arrives).

These endpoints will follow the pattern used in other services (using `X-Tenant-ID` header for multi-tenancy [12], etc.). For security, only authorized users (like a manager role) should be able to create or update suppliers and POs, but those checks might be enforced by the gateway or an auth middleware.

- **Integration with Inventory Events:** The Supplier Service will contain logic to handle inventory events through Kafka. Concretely, we will subscribe to the `"inventory.low_stock"` topic. When a low-stock message is received (containing tenant_id, product_id, current quantity, threshold [5]), the service will:
- Look up which supplier supplies that product (this implies we might need a mapping from product -> supplier; initially, perhaps we assume one primary supplier per product, which could be stored in the product record or a join table).
- Create a new purchase order in the database for that product and supplier, with an appropriate quantity. The quantity could be a fixed reorder amount or calculated (for now, perhaps reorder up to the threshold or a set batch size).
- Optionally, emit an event or notification. For instance, the service could emit a `"purchase_order.created"` event or send an email to the procurement team. In our architecture, simply creating the DB record and perhaps logging is enough for phase 1; further notifications can be added later.

Additionally, when a purchase order's status is updated to "RECEIVED", the Supplier Service can emit an `"inventory.restocked"` event with the product and quantity received. The Inventory Service (if subscribed to that) can then increment the on-hand quantity accordingly. This closes the loop from low stock -> order -> restock. We may not implement the full loop initially, but the design leaves room for it.

- **Analytics Service Enhancements:** On the demand planning side, the existing Analytics Service already provides a basic sales forecast (`/analytics/forecast`) based on average daily sales [13] [14]. We plan to enhance this capability. Potential improvements include:
- Using more sophisticated forecasting algorithms (moving averages, seasonal trends, etc.) instead of a simple average.
- Providing forecasts per product, not just overall sales. For example, a `/analytics/forecast?product_id=X` could return the predicted sales for product X for the next week or month. This would be very useful for planning inventory.
- Generating **demand reports** that combine sales trends with current stock to recommend reorder quantities. This could even be a new endpoint like `/analytics/reorder_suggestions` which lists products that are projected to stock-out soon and how much to reorder. Internally, this would use the sales forecast plus current inventory to compute days of stock remaining.

However, given the timeframe, a simpler initial step is to integrate the existing forecast logic into supplier planning decisions. For instance, when creating a purchase order on low stock, instead of a fixed quantity, we could query the Analytics forecast for that product to decide how much to order (e.g. order enough for the next 7 days of sales). This is an area where analytics and supplier services will collaborate.

- **Audit Logging in Services:** Each service (Product, Order, Inventory, etc.) will be instrumented to record audit events. In practice, this might mean:
- After a successful *create/update/delete* operation, the service calls an internal helper to log the action. For example, after `update_product` modifies the DB [15] [16], we add code to insert an audit_log record noting that product X was updated by user Y.
- For read access or authentication events, the Auth Service or Gateway could log when a user logs in or when an admin views certain data (depending on compliance requirements).
- We ensure that each audit entry captures the necessary context (user, timestamp, details) as discussed. This might involve passing the authenticated user's ID via a header or JWT claims to each service (since currently services don't have user context, only tenant). We might extend the `X-Tenant-ID` header approach to an `X-User-ID` or include user info in the auth token that downstream services can decode (if we share a JWT signing key, for example).

Optionally, instead of each service writing directly to the `audit_log` table, they could send an event to Kafka (say topic `"audit.events"`). A small AuditLog Service (or even a Kafka consumer task) would subscribe and write to the DB. The trade-off is complexity vs. decoupling. As a first iteration, it might be acceptable for services to write to the audit table directly (since they all share the database). This direct write approach is simpler to implement initially: just an extra SQL insert on the same DB connection. We will encapsulate this in a utility function to keep code clean. For example, a pseudo-Rust snippet for logging could be:

```
fn log_audit(db: &PgPool, tenant: Uuid, user: Uuid, entity: &str, action: &str,
desc: &str) {
    sqlx::query!(
```

```
        "INSERT INTO audit_log (tenant_id, user_id, entity, action,
description)
         VALUES ($1, $2, $3, $4, $5)",
        tenant, user, entity, action, desc
    ).execute(db).await.expect("Failed to insert audit log");
}
```

Then call `log_audit` from each handler as needed (non-blocking in production code, ideally).

In conclusion, these new services and APIs will enable the platform to manage its **supply side** and maintain a **comprehensive audit trail**. We will have a fully functional Supplier Service with CRUD APIs and event-driven restock automation, richer analytics for demand forecasting, and audit logging baked into all critical operations.

d. Scaffolded Code Sketches

To get things started, we will scaffold the basic code for the new components. This involves setting up project structure and some placeholder logic so that developers can later fill in the details. Here are a few code sketches illustrating the plan:

- **Supplier Service Skeleton:** We create a new Rust project under `services/supplier-service`. The `main.rs` might look like this (initially providing just a health check and Kafka subscription setup):

```
use axum::{routing::{get, post}, Router};
use sqlx::PgPool;
use rdkafka::consumer::{Consumer, StreamConsumer};
use rdkafka::Message;
use uuid::Uuid;

async fn health() -> &'static str { "ok" }

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    tracing_subscriber::fmt().with_env_filter("info").init();
    let database_url = std::env::var("DATABASE_URL")?;
    let db = PgPool::connect(&database_url).await?;

    // Kafka consumer for low_stock events
    let consumer: StreamConsumer = rdkafka::ClientConfig::new()
        .set("bootstrap.servers",
&std::env::var("KAFKA_BOOTSTRAP").unwrap_or("localhost:9092".into()))
        .set("group.id", "supplier-service")
        .create()?;
    consumer.subscribe(&["inventory.low_stock"])?;
```

```
    // Spawn a task to handle incoming low_stock events
    let db_clone = db.clone();
    tokio::spawn(async move {
        let mut stream = consumer.stream();
        while let Some(message) = stream.next().await {
            if let Ok(m) = message {
                if let Some(Ok(payload)) = m.payload_view::<str>() {
                    if m.topic() == "inventory.low_stock" {
                        if let Ok(event) =
serde_json::from_str::<LowStockEvent>(payload) {
                            // TODO: create a purchase order in DB for
event.product_id
                            // and maybe send notification or event.
                            println!("Received low stock alert for product {}
(qty {} <= threshold {})",
                                     event.product_id, event.quantity,
event.threshold);
                            // Example logic: insert into purchase_orders table
                            let _ = sqlx::query!(
                                "INSERT INTO purchase_orders (id, tenant_id,
product_id, supplier_id, quantity, status)
                                 VALUES ($1, $2, $3, $4, $5, $6)",
                                 Uuid::new_v4(), event.tenant_id,
event.product_id,
                                 /* supplier_id */ Uuid::new_v4(), /* order qty
*/ event.threshold * 2, "PENDING"
                            )
                            .execute(&db_clone).await;
                        }
                    }
                }
            }
        }
    });

    // Set up HTTP routes (e.g., list/create suppliers, list/create purchase
orders)
    let app = Router::new()
        .route("/healthz", get(health))
        .route("/suppliers", get(list_suppliers).post(create_supplier))
        .route("/purchase_orders",
get(list_purchase_orders).post(create_purchase_order))
        .with_state(db); // using PgPool as state for simplicity

    axum::Server::bind(&"0.0.0.0:8090".parse()?)
        .serve(app.into_make_service())
        .await?;
```

```
    Ok(())
}
```

Sketch: In this code, we subscribe to `inventory.low_stock` and on each event, we insert a new record into `purchase_orders` (for illustration, we use a placeholder supplier_id and quantity logic – in a real scenario, we'd determine the actual supplier and needed quantity). We also define routes for basic CRUD on suppliers and purchase orders (the handler functions `list_suppliers`, `create_supplier`, etc., would be implemented to perform SQL queries accordingly). This scaffolding ensures that the service builds and can run, even if the business logic is still rudimentary.

- **Audit Logging Example:** In the Product Service, after adding the new audit_log table, we can augment handlers. For instance, in `create_product` (Rust Axum handler), after inserting the product into DB and sending the Kafka event [17] [18], we add:

```
// Pseudo-code: Log the creation in audit_log
let user_id = /* extract from auth context, to be implemented */;
sqlx::query!(
    "INSERT INTO audit_log (tenant_id, user_id, entity, action, description)
     VALUES ($1, $2, $3, $4, $5)",
    tenant_id,
    user_id,
    "Product",
    "CREATE",
    format!("Product {} created: name='{}', price={}", product.id,
product.name, product.price)
).execute(&state.db).await.unwrap();
```

Similar snippets would be added to `update_product`, `delete_product`, and likewise in Order Service (for order placed, order cancelled), Inventory (perhaps for manual inventory adjustments), etc. Over time, we might refactor this into a shared library or macro to avoid repetition. But scaffolding-wise, placing these calls in key spots ensures audit data starts capturing events from day one.

- **Demand Forecast Extension:** To illustrate improving demand planning, suppose we want to enhance the forecast calculation. Currently, `get_forecast` in Analytics averages the last 7 days of sales [13] [19]. We could sketch out using a more advanced method or external library. For example, using a simple exponential moving average:

```
pub async fn get_forecast(State(state): State<AppState>, headers: HeaderMap)
    -> Result<Json<ForecastResult>, (StatusCode, String)>
{
    let tenant_id = extract_tenant_id(&headers)?;
    // Fetch last N days of sales
    let sales: Vec<f64> = sqlx::query_scalar!(
        "SELECT total_sales FROM daily_sales
         WHERE tenant_id = $1 AND date < CURRENT_DATE
         ORDER BY date DESC LIMIT 30",
```

```
            tenant_id
    )
    .fetch_all(&state.db).await
    .map_err(|e| (StatusCode::INTERNAL_SERVER_ERROR, format!("DB error: {}",
e)))?
    .into_iter().filter_map(|x| x).collect();

    let forecast = if sales.is_empty() {
        0.0
    } else {
        // Apply a weighted moving average: more weight to recent days
        let mut weight = 1.0;
        let mut total_weight = 0.0;
        let mut weighted_sum = 0.0;
        for &daily in sales.iter().rev() { // oldest first
            weighted_sum += daily * weight;
            total_weight += weight;
            weight += 0.1; // increase weight over time
        }
        weighted_sum / total_weight
    };
    Ok(Json(ForecastResult { next_day_sales: forecast }))
}
```

This is just a conceptual sketch of using a weighted average instead of a plain average. In the future, we might integrate a proper time-series forecasting crate or even a machine-learning model for more accuracy. But even this improved logic could yield better estimates of next-day sales. These forecasts, when provided per product, will directly support the demand planning decisions (how much stock to reorder).

- **Integration Gateway Adjustments:** Though not explicitly asked, it's worth noting that if the frontends access these new services, we may route through the **Integration Gateway** (if it's functioning as a unified API endpoint). For instance, the admin portal might call the gateway for supplier management endpoints, and the gateway would proxy or aggregate calls to the Supplier Service. As part of scaffolding, we might add routes in the gateway like `/api/suppliers -> Supplier Service` and ensure proper authentication/authorization flows.

With these sketches in place, the team has a starting point. Developers can run the new Supplier Service (it will initially register itself just like others with a `/healthz`) and observe that it consumes Kafka events (printing a log on low stock). The audit log inserts can be tested by performing a product or order operation and checking the `audit_log` table for a new entry. These scaffolds will be iteratively refined into full implementations: fleshing out the handlers with validation, hooking up real user IDs for audit, linking products to real suppliers, and so on.

Conclusion

In this plan, we've confirmed key decisions and mapped out the work ahead. **Product images** will be supported in the system so that each product can display a picture, requiring a simple schema and API

update. We will hold off on major **product service extensions** until the current functionality is stable, avoiding premature complexity. Meanwhile, we are green-lighting the implementation of **audit logging** (to bolster compliance and security) and a **supplier/demand planning module** (to automate inventory replenishment and use sales forecasts more intelligently). This involves an evolution of the NovaPOS architecture — adding a Supplier Service and leveraging our event-driven infrastructure — along with the necessary database changes, new APIs, and initial code scaffolding.

By following this plan, NovaPOS will evolve from a point-of-sale system into a more comprehensive platform that not only sells products but also smartly manages stock and suppliers, all while keeping an audit trail of operations. These enhancements set the stage for a robust, enterprise-ready solution with improved visibility and control over the retail supply chain. As we implement these steps, we will continuously test and refine, ensuring each piece (images, logs, planning) integrates seamlessly with the whole. The result will be a NovaPOS that is richer in features (visual product catalogs, smarter inventory management) and ready for stringent compliance requirements, providing value both to end-users and administrators of the system.

Sources:

1. NovaPOS Product service code (Rust) – current product schema and event publishing [1] [7].
2. NovaPOS POS frontend (React) – usage of product image and category filter in UI [2] [3].
3. NovaPOS Inventory/Analytics services – inventory threshold events and basic forecasting logic [5] [19].
4. Microservices architecture reference – role of Supplier Management Service in supply chain microservices [4].
5. Audit logging best practices – definition and importance of audit logs for compliance [8].

---

[1] [7] [15] [16] [17] [18] product_handlers.rs
https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/services/product-service/src/product_handlers.rs

[2] ProductCard.tsx
https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/frontends/pos-app/src/components/ProductCard.tsx

[3] NovaPOS_Cashier_POS.txt
https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/building_documents/NovaPOS_Cashier_POS.txt

[4] [6] Deep Dive : Supply Chain Management using Microservice Architecture | by Avinash Dhumal | Medium
https://medium.com/@avinash.dhumal/deep-dive-supply-chain-management-architecture-2dc1e347cf7b

[5] main.rs
https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/services/analytics-service/src/main.rs

[8] [11] Guide to Building Audit Logs for Application Software | by Tony | Medium
https://medium.com/@tony.infisical/guide-to-building-audit-logs-for-application-software-b0083bb58604

9 10 main.rs

https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/services/inventory-service/src/main.rs

12 inventory_handlers.rs

https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/services/inventory-service/src/inventory_handlers.rs

13 14 19 analytics_handlers.rs

https://github.com/datawrangler05/novapos/blob/73e1639823e8adf66e82499fd59fa79bfbd9a6eb/services/analytics-service/src/analytics_handlers.rs