Implementing RFM Configuration in Herethere Loyalty App

To implement the RFM (Recency, Frequency, Monetary) configuration feature for our Shopify loyalty app, targeting small and medium-sized merchants and leveraging our tech stack (NestJS with TypeScript, Vite + React, Rust with Shopify Functions, PostgreSQL, Redis), we'll provide a detailed task list based on the plan outlined earlier. This task list breaks down the implementation into actionable steps, covering backend, frontend, Shopify Functions, and testing, with considerations for usability, performance, and merchant needs. The tasks are prioritized to deliver a minimum viable feature quickly while allowing for iterative improvements.

# Phase 1: Planning and Setup

## Goal:

Establish the foundation for RFM configuration, ensuring alignment with merchant needs and tech stack.

## Enhancements & Best Practices

Interview 2–3 merchants before finalizing requirements to ensure default thresholds and UI match real-world needs.

### 1. Define Feature Requirements

**Description:** Finalize the scope of RFM configuration, focusing on small and medium-sized merchants (100–10,000 customers, AOV $20–$200).

**Tasks:**

1. Document RFM threshold inputs (e.g., Recency: days, Frequency: orders, Monetary: spend)
2. Specify tier configuration (e.g., 2–5 tiers, name, RFM criteria, rewards)
3. Define adjustment frequency options (daily, weekly, monthly, quarterly, event-based)
4. Include notification settings (e.g., Klaviyo integration, email templates)
5. Confirm analytics preview (e.g., segment size, repeat purchase rate)

**Deliverable:** Requirements document (e.g., in Notion or Google Docs).

### 2. Analyze Merchant Data Patterns

**Description:** Study typical purchase cycles and AOV for small/medium merchants to suggest default RFM thresholds.

**Tasks:**

* Use Shopify Analytics or test store data to calculate median purchase interval and AOV.
* Example: For AOV $50, suggest Monetary thresholds (5: $250+, 4: $100–$249, etc.).
* Validate with 2–3 merchant personas (e.g., pet store, fashion boutique).

**Deliverable:** Default RFM thresholds and tier suggestions.

### 3. Set Up Development Environment

**Description:** Configure your NestJS (TypeScript) backend and Vite + React frontend for development.

**Tasks:**

1. Initialize a new feature branch (`feature/rfm-config`).
2. Set up Vite with React, TypeScript, and Shopify Polaris for the admin UI.
3. Configure NestJS server with TypeScript and PostgreSQL (using herethere\_full\_schema.sql).
4. Install Shopify CLI for Rust-based Shopify Functions.
5. Set up Redis for caching RFM scores and segment sizes

**Deliverable:** Working dev environment with basic app scaffold.

# Phase 2: Backend Development (NestJS/TypeScript)

## Goal:

Build backend logic to calculate RFM scores, manage tiers, and handle adjustments.

## Enhancements & Best Practices

* Use API versioning for RFM endpoints (e.g., `/api/v1/rfm-config`) for future compatibility.
* Add input validation and robust error handling for all config endpoints.

### 4. Integrate Shopify APIs

**Description:** Fetch customer and order data for RFM calculations.

**Tasks:**

1. Set up Shopify GraphQL Admin API client in NestJS (e.g., using `@shopify/shopify-api`)
2. Create endpoints to query `Customer` and `Order` data (e.g., `GET /api/customers`)
3. Cache results in PostgreSQL (customers table) and Redis to reduce API calls
4. Handle pagination for stores with >100 customers

**Deliverable:**API service to fetch purchase data.

### 5. Implement RFM Calculation Logic

**Description:** Calculate RFM scores based on merchant-defined thresholds.

**Tasks:**

* Define TypeScript interfaces for RFM config and scores in NestJS
* Write a NestJS service to compute RFM scores from order data and store in customers.rfm\_score (JSONB)
* Cache scores in Redis for fast retrieval.

**Deliverable:** RFM calculation service.

### 6. Develop Tier Assignment Logic

**Description:** Assign customers to tiers based on RFM scores and merchant criteria.

**Tasks:**

1. Create a NestJS service to map RFM scores to tiers
2. Update customer records in PostgreSQL (customers.rfm\_score) with tier assignments

**Deliverable:** Tier assignment service.

### 7. Set Up Adjustment Scheduling

**Description:** Implement scheduled and event-based tier adjustments.

**Tasks:**

1. Use `@nestjs/schedule` for scheduled updates (e.g., monthly on the 1st)
2. Subscribe to Shopify `orders/create` webhook in NestJS for event-based updates
3. Implement grace period logic (e.g., delay demotions by 30 days) in program\_settings (JSONB)

**Deliverable:** Scheduling and webhook handlers.

### 8. Integrate Notifications

**Description:** Enable tier change notifications via Klaviyo or Shopify email.

**Tasks:**

1. Create NestJS API endpoints to send emails (e.g., `POST /api/notifications`)
2. Integrate with Klaviyo API for email templates (e.g., "Welcome to Gold!")
3. Allow merchants to customize notification templates in program\_settings.email\_templates (JSONB)

**Deliverable:** Notification service.

# Phase 3: Shopify Functions (Rust)

## Goal:

Use Rust for performance-critical RFM updates.

## Enhancements & Best Practices

* Add logging and error reporting to Rust functions for easier debugging in production.

### 9. Develop RFM Score Update Function

**Description:** Create a Shopify Function to update RFM scores in real-time.

**Tasks:**

1. Set up a Rust project with Shopify Function CLI
2. Implement logic to calculate RFM scores on `orders/create` events (see herethere\_full\_schema.sql for rfm\_score structure)
3. Update customer records in PostgreSQL via webhook callbacks
4. Test with Shopify's Function testing tools

**Deliverable:** Deployed Shopify Function.

### 10. Optimize for Large Stores

**Description:** Ensure scalability for merchants with 5,000–10,000 customers.

**Tasks:**

1. Use batch processing for RFM calculations in Rust
2. Cache results in Redis to reduce API calls
3. Test performance with simulated data (e.g., 10,000 customers)

**Deliverable:** Optimized test report.

# Phase 4: Frontend Development (Vite/React)

## Goal:

Build an intuitive configuration UI for merchants.

## Enhancements & Best Practices

* Ensure React/Polaris UI is accessible (labels, keyboard navigation, color contrast)
* Confirm the RFM config UI is mobile responsive for merchants using the Shopify app.

### 11. Design RFM Configuration UI

**Description:** Create a React form using Polaris for RFM thresholds, tiers, and frequency.

**Tasks:**

* Extend the provided React component with Vite, TypeScript, and Tailwind CSS
* Add inputs for:
  + RFM thresholds (e.g., sliders or text fields for days/orders/spend)
  + Tier definitions (name, RFM criteria, rewards)
  + Adjustment frequency (dropdown: daily, weekly, monthly, quarterly, event-based)
  + Notification settings (email templates, integration toggle)
* Use Polaris components (e.g., `TextField`, `Select`, `FormLayout`)
* Implement validation (e.g., ensure Monetary 5 > Monetary 4).

**Deliverable:** RFM configuration form component.

### 12. Add Analytics Preview

**Description:** Display segment sizes and metrics to guide merchants.

**Tasks:**

1. Create a NestJS API endpoint to calculate segment sizes based on draft config
2. Use Chart.js in the React frontend to visualize segment distribution (e.g., bar chart of customers per tier)
3. Example: "Gold: 100 customers (10%), Silver: 300 customers (30%)"

**Deliverable:** Analytics preview component.

**Enhancements & Best Practices**

* Add analytics to track how merchants interact with the RFM config UI (e.g., which fields are changed, where they drop off)
* Track the performance of RFM segments over time (e.g., repeat purchase rate by tier) and surface this in analytics.

# Phase 5: Testing and Validation

## Goal:

Ensure the feature works reliably for small and medium-sized merchants.

## Enhancements & Best Practices

* Test with edge-case data (e.g., merchants with no orders, very high/low AOV, or customers with no email)
* For batch processing, simulate even larger stores (e.g., 20,000+ customers) to future-proof.

### 13. Unit Test Backend Logic

**Description:** Test RFM calculations and tier assignments.

**Tasks:**

1. Write Jest tests for NestJS RFM calculation and tier assignment services
2. Mock Shopify API responses to simulate customer data

**Deliverable:** Test suite with 80%+ coverage.

### 14. Test Shopify Function

**Description:** Validate Rust function for real-time updates.

**Tasks:**

1. Use Shopify CLI to test Function with sample order data
2. Verify PostgreSQL updates for tier assignments

**Deliverable:** Tested Shopify Function.

### 15. Test UI and UX

**Description:** Ensure the configuration UI is intuitive.

**Tasks:**

1. Conduct usability testing with 2–3 mock merchants (e.g., pet store, fashion)
2. Verify validation (e.g., prevent invalid thresholds)
3. Test form submission and API integration

**Deliverable:** Bug-free UI.

### 16. End-to-End Testing

**Description:** Test the full workflow from configuration to tier updates.

**Tasks:**

1. Simulate a merchant configuring RFM thresholds and tiers
2. Trigger order events to test real-time updates
3. Verify notifications (e.g., Klaviyo emails)
4. Test with 1,000 customers to ensure scalability

**Deliverable:** End-to-end test report.

# Phase 6: Deployment and Documentation

## Goal:

Launch the feature and support merchants.

## Enhancements & Best Practices

1. For deployment, have a rollback plan in case the new RFM config causes issues
2. Prepare support scripts/FAQs for common merchant questions about RFM
3. Generate and publish OpenAPI/Swagger docs for your RFM endpoints
4. Include a "Best Practices" section in your help article (e.g., how to choose thresholds, common mistakes)

### 17. Deploy Feature

**Description:** Release the RFM configuration to production.

**Tasks:**

1. Deploy NestJS backend, Vite + React frontend, and Shopify Function updates to VPS using Docker Compose
2. Configure Nginx to route API requests and serve frontend assets
3. Monitor logs for errors using tools like Sentry

**Deliverable:** Live feature deployment.

### 18. Create Documentation

**Description:** Provide guides for merchants to use the feature.

**Tasks:**

1. Write a help article explaining RFM setup (e.g., "How to Configure RFM Tiers")
2. Include examples for common merchant types (e.g., pet store: monthly, AOV $40)
3. Add screenshots of the UI

**Deliverable:** Help article in your app's support portal.

### 19. Pilot with Merchants

**Description:** Test with real merchants to gather feedback.

**Tasks:**

1. Recruit 3–5 small/medium merchants for a beta test
2. Monitor usage (e.g., segment sizes, engagement rates)
3. Collect feedback via surveys or calls

**Deliverable:** Beta feedback report.

# Phase 7: Pricing and Rollout

## Goal:

Make the feature accessible and profitable.

## Enhancements & Best Practices

* Consider using feature flags to enable RFM config for select merchants before a full rollout

### 20. Define Pricing Strategy

**Description:** Price the feature to appeal to small/medium merchants.

**Tasks:**

1. Offer basic RFM config (2–3 tiers, monthly adjustments) in free plan
2. Include advanced features (e.g., real-time updates, custom notifications) in a paid plan ($15–$29/month)
3. Compare with competitors (e.g., Loyal: free, LoyaltyLion: $399/month)

**Deliverable:** Pricing model.

### 21. Roll Out to All Merchants

**Description:** Launch the feature to all users.

**Tasks:**

* Announce via email and in-app banner
* Provide a setup wizard for new merchants
* Monitor adoption and support queries.

**Deliverable:** Launch campaign.

# Timeline and Resource Estimate

**Total Duration:** ~30–35 days (assuming 1–2 developers).

**Breakdown:**

* Phase 1: 5–6 days
* Phase 2: 13–14 days
* Phase 3: 5–6 days
* Phase 4: 6–7 days
* Phase 5: 9 days
* Phase 6: 3 days
* Phase 7: 3 days

**Resources:** 1 full-stack developer (NestJS/React, Rust), Grok AI (handling both).

# Considerations for Small/Medium Merchants

1. **Simplicity:** Provide a "Quick Setup" wizard with pre-filled thresholds (e.g., based on AOV $50, purchase cycle 30 days)
2. **Affordability:** Keep advanced features affordable ($15–$29/month) to compete with Loyal (free) and Rivo (free plan)
3. **Usability:** Use Polaris for a familiar Shopify admin experience; include tooltips and examples
4. **Scalability:** Optimize for 100–10,000 customers, as medium-sized merchants may grow quickly
5. **Support:** Offer live chat or email support to guide setup, critical for small merchants.

# Example Merchant Workflow

For a pet store (AOV $40, 1,000 customers):

**Configuration:**

1. Recency: 5 = <30 days, 4 = 31–60 days, etc
2. Frequency: 5 = 5+ orders, 4 = 3–4 orders
3. Monetary: 5 = $200+, 4 = $100–$199
4. Tiers: Gold (5-4-4), Silver (3-2-2), Bronze (1-1-1)
5. Frequency: Monthly, 30-day grace period
6. Outcome: 10% in Gold (100 customers), 25% increase in repeat purchases

# Next Steps

1. Start with Phase 1: Finalize requirements and analyze merchant data (5–6 days)
2. Prioritize Backend: Build Shopify API integration and RFM calculation first to validate data flow
3. Test Early: Use a test store to simulate RFM configs before building the full UI
4. Seek Feedback: Share the UI prototype with 1–2 friendly merchants for early input.

# Summary Table of Key Suggestions

* **Planning:** Interview merchants before finalizing requirements
* **Backend:** API versioning, input validation, error handling
* **Rust:** Add logging/error reporting to Shopify Functions
* **Frontend:** Ensure accessibility and mobile responsiveness
* **Analytics**: Track UI usage and segment performance
* **Testing:** Test edge cases and larger stores
* **Deployment:** Prepare rollback plan, support scripts/FAQs
* **Docs**: Generate OpenAPI docs, add "Best Practices" to merchant docs
* **Rollout**: Use feature flags for gradual enablement |

If you need a specific task expanded (e.g., detailed Shopify Function code, full UI mockup, or Klaviyo integration steps), let me know. Would you like me to provide a sample Shopify Flow automation for tier updates or a mockup for the setup wizard?