LoyalNest Microservice Design

This document outlines the design of LoyalNest's 15 microservices, built with Nx monorepo, NestJS, gRPC/Kafka, and Docker Compose for Shopify App Store TVP submission (February 2026). It ensures scalability (10,000 orders/hour), GDPR/CCPA compliance (AES-256 PII encryption, audit logging), and email delivery via third-party providers (Klaviyo, Postscript, AWS SES). Each service includes its purpose, endpoints, technology stack, database schema, and inter-service communication.

Microservices

1. Core Service

- **Purpose**: Centralizes business logic, configuration management, and inter-service coordination, including global settings, plan enforcement (freemium-to-Plus funnel), and usage thresholds.
- **Endpoints**:
  - REST: `/v1/api/core/settings`, `/v1/api/core/usage`, `/v1/api/core/plan`
  - gRPC: `/core.v1/CreateCustomer`, `/core.v1/UpdateSettings`
- **Tech**: NestJS, PostgreSQL (`program_settings`, `customer_import_logs`), Redis Cluster (`config:{merchant_id}`, `usage:{merchant_id}`), Kafka (event publishing), `socket.io` (WebSocket).
- **Database Schema**:
  - `program_settings` (PostgreSQL):
    - `merchant_id`: UUID, primary key
    - `settings`: JSONB (loyalty rules, e.g., points per dollar)
    - `created_at`: TIMESTAMP, indexed
    - `updated_at`: TIMESTAMP
  - `customer_import_logs` (PostgreSQL):
    - `import_id`: UUID, primary key
    - `merchant_id`: UUID, indexed
    - `status`: ENUM(pending, completed, failed)
    - `log_details`: JSONB
    - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
  - **Consumes**: `rfm.updated` (Kafka, from RFM-Service), `gdpr_request.created` (Kafka, from AdminCore), `user.created`, `user.updated` (Kafka, from Users-Service).
  - **Produces**: `customer.created`, `customer.updated`, `plan_limit_warning` (Kafka, to Points, Referrals, RFM-Service, AdminFeatures).
  - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service for customer data), `/auth.v1/ValidateMerchant` (gRPC, Auth for validation).
  - **Called By**: Points, Referrals, RFM-Service, Frontend, AdminFeatures (REST/gRPC for settings).
- **Interactions**: Provides configuration to all services, coordinates upgrade nudges via WebSocket, logs imports to AdminCore.

2. Auth Service

- **Purpose**: Manages Shopify OAuth, JWT authentication (15-minute expiry, revocation list in Redis), MFA via Auth0, and RBAC integration.

- **Endpoints**:
  - REST: `/v1/api/auth/login`, `/v1/api/auth/refresh`, `/v1/api/auth/mfa`, `/admin/v1/auth/revoke`
  - gRPC: `/auth.v1/ValidateToken`, `/auth.v1/ValidateMerchant`
- **Tech**: NestJS, `@shopify/shopify-app-express`, Redis Cluster (`jwt:{merchant_id}`, `revoked_jti:{token_id}`), PostgreSQL (`merchants`, `admin_users`, `admin_sessions`, `impersonation_sessions`).
- **Database Schema**:
  - `merchants` (PostgreSQL):
    - `merchant_id`: UUID, primary key
    - `shop_domain`: VARCHAR, unique, indexed
    - `access_token`: TEXT (AES-256 encrypted)
    - `created_at`: TIMESTAMP, indexed
  - `admin_users` (PostgreSQL):
    - `user_id`: UUID, primary key
    - `email`: TEXT (AES-256 encrypted, pgcrypto), indexed
    - `merchant_id`: UUID, indexed
    - `created_at`: TIMESTAMP
  - `admin_sessions` (PostgreSQL):
    - `session_id`: UUID, primary key
    - `user_id`: UUID, indexed
    - `jwt_token`: TEXT
    - `expires_at`: TIMESTAMP, indexed
  - `impersonation_sessions` (PostgreSQL):
    - `session_id`: UUID, primary key
    - `admin_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `expires_at`: TIMESTAMP
- **Inter-Service Communication**:
  - **Consumes**: `user.created`, `role.assigned` (Kafka, from Users-Service, Roles-Service).
  - **Produces**: `merchant.created`, `auth.revoked` (Kafka, to Core, AdminCore).
  - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service for user validation), `/roles.v1/GetRole` (gRPC, Roles-Service for RBAC).
  - **Called By**: API-Gateway, Core, Points, Referrals, RFM-Service, Frontend, AdminCore, AdminFeatures (gRPC for token validation).
- **Interactions**: Validates tokens for all services, integrates with Roles-Service for RBAC, supports MFA for Shopify Plus.

3. API-Gateway Service

- **Purpose**: Routes Shopify webhooks and external REST/gRPC requests, enforces rate limiting, and validates tokens.
- **Endpoints**:
  - REST: `/v1/api/webhooks/orders/create`, `/frontend/*`, `/v1/api/*`
  - gRPC: Forwards to service-specific endpoints (e.g., `/points.v1/*`, `/users.v1/*`).
- **Tech**: NestJS, Redis Cluster (`rate_limit:{merchant_id}:{endpoint}`), Nginx (load balancing), Kafka (event publishing).
- **Database Schema**: None (uses Redis for rate limiting, no persistent storage).

- **Inter-Service Communication**:
  - **Consumes**: None.
  - **Produces**: `webhook.received` (Kafka, optional, to Points, Referrals, RFM-Service).
  - **Calls**: `/auth.v1/ValidateToken`, `/auth.v1/ValidateMerchant` (gRPC, Auth for validation).
  - **Called By**: External clients (Shopify, Frontend), routes to Core, Points, Referrals, Users, Roles, RFM-Service, AdminCore, AdminFeatures.
- **Interactions**: Routes `/webhooks/orders/create` to Points, Referrals, RFM-Service; proxies Frontend requests; enforces rate limits (Redis).

4. Points Service

- **Purpose**: Manages points earning/redemption, Shopify POS with offline mode, checkout extensions, and campaign discounts.
- **Endpoints**:
  - REST: `/v1/api/points/earn`, `/v1/api/points/redeem`, `/v1/api/points/adjust`, `/v1/api/rewards`, `/v1/api/points/sync`
  - gRPC: `/points.v1/GetPointsBalance`
  - WebSocket: `/api/points/stream`
- **Tech**: NestJS, Rust/Wasm (Shopify Functions for discounts), MongoDB (`points_transactions`, `reward_redemptions`, `pos_offline_queue`), Redis Cluster (`points:customer:{id}`), `socket.io`.
- **Database Schema**:
  - `points_transactions` (MongoDB):
    - `_id`: ObjectId, primary key
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `points`: NUMBER
    - `type`: ENUM(earn, redeem, adjust)
    - `created_at`: TIMESTAMP, indexed
  - `reward_redemptions` (MongoDB):
    - `_id`: ObjectId, primary key
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `reward_id`: UUID
    - `status`: ENUM(pending, completed)
    - `created_at`: TIMESTAMP
  - `pos_offline_queue` (MongoDB):
    - `_id`: ObjectId, primary key
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `points`: NUMBER
    - `created_at`: TIMESTAMP
- **Inter-Service Communication**:
  - **Consumes**: `webhook.received` (Kafka, from API-Gateway), `user.created` (Kafka, from Users-Service).
  - **Produces**: `points.earned`, `points.redeemed` (Kafka, to RFM-Service, AdminCore).
  - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service for customer data), `/auth.v1/ValidateMerchant` (gRPC, Auth), `/core.v1/GetSettings` (gRPC, Core for rules).

    - **Called By**: API-Gateway, Frontend (REST/gRPC), AdminFeatures (adjustments).
- **Interactions**: Processes `orders/create` webhooks, syncs POS data, applies discounts, streams updates via WebSocket.

5. Referrals Service

- **Purpose**: Manages SMS/email referrals, referral status with progress bar, and error handling with fallback to AWS SES.
- **Endpoints**:
    - REST: `/v1/api/referrals/create`, `/v1/api/referrals/complete`, `/v1/api/referrals/status`, `/v1/api/referrals/progress`
    - gRPC: `/referrals.v1/GetReferralStatus`
- **Tech**: NestJS, Klaviyo/Postscript (SMS/email, 5s timeout, 3 retries), AWS SES (fallback), Bull queues, PostgreSQL (`referrals`), Redis Streams (`referral:{code}`, `referral:status:{id}`).
- **Database Schema**:
    - `referrals` (PostgreSQL):
        - `referral_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `referrer_id`: UUID, indexed
        - `referral_link_id`: UUID, indexed
        - `status`: ENUM(pending, completed, expired)
        - `created_at`: TIMESTAMP, indexed
        - `updated_at`: TIMESTAMP
- **Inter-Service Communication**:
    - **Consumes**: `webhook.received` (Kafka, from API-Gateway), `user.created` (Kafka, from Users-Service).
    - **Produces**: `referral.created`, `referral.completed` (Kafka, to RFM-Service, AdminCore).
    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service for referrer data), `/auth.v1/ValidateMerchant` (gRPC, Auth), `/core.v1/GetSettings` (gRPC, Core for rules).
    - **Called By**: API-Gateway, Frontend (REST/gRPC), AdminFeatures (management).
- **Interactions**: Generates referral links, sends notifications via Klaviyo/Postscript/AWS SES, tracks 7%+ conversion, logs errors to PostHog.

6. Users-Service

- **Purpose**: Manages merchant and customer accounts, including PII encryption and audit logging.
- **Endpoints**:
    - REST: `/v1/api/users/create`, `/v1/api/users/update`, `/v1/api/users/get`
    - gRPC: `/users.v1/GetUser`, `/users.v1/UpdateUser`
- **Tech**: NestJS, PostgreSQL (`users`, `audit_logs`), Redis Cluster (`user:{user_id}`), Kafka (event publishing).
- **Database Schema**:
    - `users` (PostgreSQL):
        - `user_id`: UUID, primary key
        - `email`: TEXT (AES-256 encrypted, pgcrypto), indexed
        - `merchant_id`: UUID, indexed
        - `role_id`: UUID, indexed

        - `created_at`: TIMESTAMP, indexed
        - `updated_at`: TIMESTAMP
        - Trigger: Logs changes to `audit_logs`
    - `audit_logs` (PostgreSQL, shared with AdminCore):
        - `log_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `user_id`: UUID, indexed
        - `action`: ENUM(create, update, delete)
        - `details`: JSONB
        - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
    - **Consumes**: `merchant.created` (Kafka, from Auth).
    - **Produces**: `user.created`, `user.updated` (Kafka, to Core, Points, Referrals, RFM-Service, AdminCore).
    - **Calls**: `/roles.v1/GetRole` (gRPC, Roles-Service for role validation), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Core, Points, Referrals, RFM-Service, Frontend, AdminCore, AdminFeatures (gRPC/REST for user data).
- **Interactions**: Manages user accounts, logs changes for GDPR/CCPA, caches user data in Redis.

7. Roles-Service

- **Purpose**: Manages RBAC with role-based permissions for merchants and admins.
- **Endpoints**:
    - REST: `/v1/api/roles/create`, `/v1/api/roles/update`, `/v1/api/roles/get`
    - gRPC: `/roles.v1/GetRole`, `/roles.v1/UpdateRole`
- **Tech**: NestJS, PostgreSQL (`roles`, `audit_logs`), Redis Cluster (`role:{role_id}`), Kafka (event publishing).
- **Database Schema**:
    - `roles` (PostgreSQL):
        - `role_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `permissions`: JSONB (e.g., `["admin:full", "admin:analytics"]`)
        - `created_at`: TIMESTAMP, indexed
        - `updated_at`: TIMESTAMP
        - Trigger: Logs changes to `audit_logs`
    - `audit_logs` (PostgreSQL, shared with AdminCore, Users-Service):
        - Same as Users-Service schema.
- **Inter-Service Communication**:
    - **Consumes**: `merchant.created` (Kafka, from Auth).
    - **Produces**: `role.assigned`, `role.updated` (Kafka, to Auth, AdminCore).
    - **Calls**: `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Auth, AdminCore, AdminFeatures, Frontend (gRPC/REST for RBAC).
- **Interactions**: Defines roles, supports RBAC for Auth, logs changes for compliance.

8. RFM-Service

- **Purpose**: Provides RFM analytics with time-weighted recency, lifecycle stages, and visualizations (e.g., Recency vs. Monetary scatter plot).
- **Endpoints**:
  - REST: `/v1/api/rfm/segments`, `/v1/api/rfm/segments/preview`, `/v1/api/rfm/nudges`
  - gRPC: `/rfm.v1/GetSegments`, `/rfm.v1/GetCustomerRFM`
  - WebSocket: `/api/rfm/visualizations`
- **Tech**: NestJS, Rust/Wasm (real-time updates), TimescaleDB (`rfm_segment_deltas`, `rfm_segment_counts`, `rfm_score_history`, `customer_segments`), Redis Streams (`rfm:customer:{id}`, `rfm:preview:{merchant_id}`).
- **Database Schema**:
  - `rfm_segment_deltas` (TimescaleDB, hypertable):
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `recency`: NUMBER
    - `frequency`: NUMBER
    - `monetary`: NUMBER
    - `created_at`: TIMESTAMP, indexed
  - `rfm_segment_counts` (TimescaleDB, materialized view):
    - `merchant_id`: UUID, indexed
    - `segment`: ENUM(high_value, at_risk, new)
    - `count`: NUMBER
    - `created_at`: TIMESTAMP, indexed
  - `rfm_score_history` (TimescaleDB):
    - `history_id`: UUID, primary key
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `rfm_score`: JSONB
    - `created_at`: TIMESTAMP, indexed
  - `customer_segments` (TimescaleDB):
    - `customer_id`: UUID, indexed
    - `merchant_id`: UUID, indexed
    - `segment`: ENUM(high_value, at_risk, new)
    - `created_at`: TIMESTAMP, indexed
    - Trigger: Updates on `orders/create`, `points.earned`
- **Inter-Service Communication**:
  - **Consumes**: `points.earned`, `referral.completed`, `user.created`, `user.updated` (Kafka, from Points, Referrals, Users-Service).
  - **Produces**: `rfm.updated` (Kafka, to Core, AdminFeatures, Frontend).
  - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service for customer data), `/auth.v1/ValidateMerchant` (gRPC, Auth).
  - **Called By**: Core, Frontend, AdminFeatures (REST/gRPC for analytics).
- **Interactions**: Calculates RFM scores, refreshes daily (`0 1 * * *`) and incrementally on `orders/create`, caches in Redis Streams, streams visualizations.

9. Event Tracking Service

- **Purpose**: Tracks feature usage and events for analytics and merchant engagement.

- **Endpoints**:
  - REST: `/v1/api/events`
  - gRPC: `/event_tracking.v1/CreateTask`
- **Tech**: NestJS, PostHog, PostgreSQL (`queue_tasks`), Kafka.
- **Database Schema**:
  - `queue_tasks` (PostgreSQL):
    - `task_id`: UUID, primary key
    - `merchant_id`: UUID, indexed
    - `event_type`: VARCHAR (e.g., `points_earned`, `user.created`)
    - `status`: ENUM(pending, completed, failed)
    - `payload`: JSONB
    - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
  - **Consumes**: `points.earned`, `referral.created`, `user.created`, `role.assigned`, `rfm.updated` (Kafka, from Points, Referrals, Users-Service, Roles-Service, RFM-Service).
  - **Produces**: `task.created`, `task.completed` (Kafka, to AdminCore).
  - **Calls**: None.
  - **Called By**: API-Gateway, AdminCore (REST/gRPC for event logging).
- **Interactions**: Captures events, sends to PostHog via Kafka, queues tasks for async processing.

10. AdminCore Service

- **Purpose**: Manages merchant accounts, GDPR requests, and audit logs.
- **Endpoints**:
  - REST: `/admin/merchants`, `/admin/logs`
  - gRPC: `/admin_core.v1/GetMerchants`, `/admin_core.v1/GetAuditLogs`, `/admin_core.v1/HandleGDPRRequest`
- **Tech**: NestJS, PostgreSQL (`gdpr_requests`, `audit_logs`, `webhook_idempotency_keys`), Redis Streams (`audit_logs:{merchant_id}`), Kafka, `socket.io`, Nginx (IP allowlisting, HMAC).
- **Database Schema**:
  - `gdpr_requests` (PostgreSQL):
    - `request_id`: UUID, primary key
    - `merchant_id`: UUID, indexed
    - `user_id`: UUID, indexed
    - `status`: ENUM(pending, completed)
    - `created_at`: TIMESTAMP, indexed
  - `audit_logs` (PostgreSQL, shared with Users-Service, Roles-Service):
    - Same as Users-Service schema.
  - `webhook_idempotency_keys` (PostgreSQL):
    - `key_id`: UUID, primary key
    - `merchant_id`: UUID, indexed
    - `webhook_id`: UUID, indexed
    - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
  - **Consumes**: `user.created`, `role.assigned`, `points.earned`, `referral.created` (Kafka, from Users-Service, Roles-Service, Points, Referrals).
  - **Produces**: `gdpr_request.created` (Kafka, to Core).

    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service), `/roles.v1/GetRole` (gRPC, Roles-Service), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: AdminFeatures, Frontend (REST/gRPC for logs, GDPR).
- **Interactions**: Handles GDPR webhooks, logs audits from Users-Service and Roles-Service, streams logs via WebSocket.

11. AdminFeatures Service

- **Purpose**: Manages points adjustments, referrals, RFM segments, customer imports, notification templates, rate limits, integration health, onboarding, multi-currency settings, and feedback.
- **Endpoints**:
    - REST: `/admin/points/adjust`, `/admin/referrals`, `/admin/rfm-segments`, `/admin/rfm/export`, `/admin/notifications/template`, `/admin/rate-limits`, `/admin/customers/import`, `/admin/settings/currency`, `/admin/integrations/square/sync`, `/admin/v1/feedback`
    - gRPC: `/admin_features.v1/UpdateNotificationTemplate`, `/admin_features.v1/GetRateLimits`, `/admin_features.v1/ImportCustomers`, `/admin_features.v1/SyncSquarePOS`
    - WebSocket: `/admin/v1/setup/stream`
- **Tech**: NestJS, PostgreSQL (`email_templates`, `integrations`, `setup_tasks`, `merchant_settings`), Redis Streams (`setup_tasks:{merchant_id}`), Bull queues (imports, exports), `socket.io`, Nginx (IP allowlisting, HMAC), Klaviyo/Postscript/AWS SES.
- **Database Schema**:
    - `email_templates` (PostgreSQL):
        - `template_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `template_data`: JSONB
        - `created_at`: TIMESTAMP, indexed
    - `integrations` (PostgreSQL):
        - `integration_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `credentials`: TEXT (AES-256 encrypted)
        - `type`: ENUM(shopify, square, klaviyo)
        - `created_at`: TIMESTAMP
    - `setup_tasks` (PostgreSQL):
        - `task_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `status`: ENUM(pending, completed)
        - `created_at`: TIMESTAMP, indexed
    - `merchant_settings` (PostgreSQL):
        - `merchant_id`: UUID, primary key
        - `currency`: VARCHAR
        - `settings`: JSONB
        - `created_at`: TIMESTAMP
- **Inter-Service Communication**:
    - **Consumes**: `user.created`, `rfm.updated` (Kafka, from Users-Service, RFM-Service).
    - **Produces**: `email_event.created` (Kafka, to Event Tracking).

    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service), `/roles.v1/GetRole` (gRPC, Roles-Service), `/rfm.v1/GetSegments` (gRPC, RFM-Service), `/points.v1/GetPointsBalance` (gRPC, Points), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Frontend, AdminCore (REST/gRPC for management).
- **Interactions**: Manages rate limits, async imports/exports, notification templates via Klaviyo/Postscript/AWS SES, onboarding progress, and integration health.

12. Campaign Service

- **Purpose**: Manages Shopify Discounts API campaigns.
- **Endpoints**:
    - REST: `/api/campaigns`, `/api/campaigns/{id}`
    - gRPC: `/campaign.v1/CreateCampaign`, `/campaign.v1/GetCampaign`
- **Tech**: NestJS, Rust/Wasm (Shopify Functions), PostgreSQL (`campaigns`), Redis Cluster (`campaign:{campaign_id}`).
- **Database Schema**:
    - `campaigns` (PostgreSQL):
        - `campaign_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `details`: JSONB (discount rules)
        - `created_at`: TIMESTAMP, indexed
        - `updated_at`: TIMESTAMP
- **Inter-Service Communication**:
    - **Consumes**: `user.created` (Kafka, from Users-Service).
    - **Produces**: `campaign.created` (Kafka, to AdminCore, Frontend).
    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Frontend, AdminFeatures (REST/gRPC for campaign management).
- **Interactions**: Creates/applies campaign discounts, caches in Redis, integrates with Shopify Discounts API.

13. Gamification Service

- **Purpose**: Manages badge awards and leaderboards (Phase 6).
- **Endpoints**:
    - REST: `/api/gamification/badges`, `/api/gamification/leaderboard`
    - gRPC: `/gamification.v1/AwardBadge`, `/gamification.v1/GetLeaderboard`
- **Tech**: NestJS, PostgreSQL (`customer_badges`, `leaderboard_rankings`), Redis Cluster (`badge:{customer_id}:{badge_id}`, `leaderboard:{merchant_id}:{page}`).
- **Database Schema**:
    - `customer_badges` (PostgreSQL):
        - `badge_id`: UUID, primary key
        - `customer_id`: UUID, indexed
        - `merchant_id`: UUID, indexed
        - `badge_type`: VARCHAR
        - `created_at`: TIMESTAMP, indexed
    - `leaderboard_rankings` (PostgreSQL):
        - `ranking_id`: UUID, primary key

        - `customer_id`: UUID, indexed
        - `merchant_id`: UUID, indexed
        - `score`: NUMBER
        - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
    - **Consumes**: `user.created` (Kafka, from Users-Service).
    - **Produces**: `badge.awarded` (Kafka, to AdminCore, Frontend).
    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Frontend, AdminFeatures (REST/gRPC for badges/leaderboards).
- **Interactions**: Awards badges, ranks customers, caches data in Redis.

14. Frontend Service

- **Purpose**: Hosts merchant dashboard, customer widget, and admin module as a single-page app, ensuring Shopify compliance and accessibility.
- **Endpoints**:
    - REST: `/`, `/customer`, `/admin`
    - WebSocket: `/admin/v1/setup/stream`, `/api/points/stream`, `/api/rfm/visualizations`
- **Tech**: Vite + React, Polaris, Tailwind CSS, App Bridge, `i18next` (multilingual with RTL for Arabic/Hebrew, fallback: English).
- **Database Schema**: None (client-side, data via API-Gateway).
- **Inter-Service Communication**:
    - **Consumes**: `points.earned`, `referral.created`, `user.created`, `role.assigned`, `rfm.updated` (Kafka, via WebSocket).
    - **Produces**: None.
    - **Calls**: `/points.v1/GetPointsBalance`, `/referrals.v1/GetReferralStatus`, `/rfm.v1/GetSegments`, `/users.v1/GetUser`, `/roles.v1/GetRole`, `/admin_core.v1/GetAuditLogs`, `/admin_features.v1/GetRateLimits` (gRPC, via API-Gateway), `/core.v1/GetSettings` (gRPC, Core).
    - **Called By**: None (client-facing).
- **Interactions**: Queries all services via API-Gateway, displays dashboards (`UsersPage.tsx`, `RolesPage.tsx`, `AnalyticsPage.tsx`), supports i18n.

15. Products Service

- **Purpose**: Manages product-related data, including product-level RFM analytics (Phase 6), campaign eligibility, and Shopify Product API integration.
- **Endpoints**:
    - REST: `/v1/api/products`, `/v1/api/products/rfm`, `/v1/api/products/campaigns`
    - gRPC: `/products.v1/GetProductRFM`, `/products.v1/ApplyCampaign`
- **Tech**: NestJS, PostgreSQL (`products`, `product_rfm_scores`), Redis Cluster (`product:{product_id}:rfm`), Shopify Product API.
- **Database Schema**:
    - `products` (PostgreSQL):
        - `product_id`: UUID, primary key
        - `merchant_id`: UUID, indexed
        - `name`: VARCHAR

        - `created_at`: TIMESTAMP, indexed
    - `product_rfm_scores` (PostgreSQL):
        - `score_id`: UUID, primary key
        - `product_id`: UUID, indexed
        - `merchant_id`: UUID, indexed
        - `rfm_score`: JSONB
        - `created_at`: TIMESTAMP, indexed
- **Inter-Service Communication**:
    - **Consumes**: `user.created`, `rfm.updated` (Kafka, from Users-Service, RFM-Service).
    - **Produces**: `product_rfm.updated` (Kafka, to AdminCore, Frontend).
    - **Calls**: `/users.v1/GetUser` (gRPC, Users-Service), `/rfm.v1/GetSegments` (gRPC, RFM-Service), `/auth.v1/ValidateMerchant` (gRPC, Auth).
    - **Called By**: Frontend, AdminFeatures, Campaign (REST/gRPC for product data).
- **Interactions**: Fetches product data, calculates product-level RFM, integrates with Campaign Service for discounts.

Service Discovery

To enhance scalability and manage dynamic service instances, Loyalnest adopts **HashiCorp Consul** as the service discovery tool. Consul enables services to register themselves dynamically, allowing the API Gateway to route requests to available instances without static configurations. This is critical for handling Shopify's variable traffic patterns (e.g., during sales events).

Integration with API Gateway

- **Service Registration**: Each microservice (e.g., `core`, `auth`, `campaign`) registers with Consul on startup, providing its IP, port, and a health check endpoint (e.g., `/health`). For example, the `core` service registers as:

```
{
  "name": "core-service",
  "address": "core-service-host",
  "port": 8080,
  "check": {
    "http": "http://core-service-host:8080/health",
    "interval": "10s"
  }
}
```

- **API Gateway Routing**: The API Gateway queries Consul's DNS interface (e.g., `core-service.service.consul`) or API to resolve service endpoints dynamically. The gateway caches endpoints for 10 seconds to reduce latency but refreshes periodically to handle service scaling.
- **Health Checks**: Each service exposes a `/health` endpoint returning `{ "status": "healthy" }` if the service and its database connection are operational. Consul marks unhealthy instances as unavailable, ensuring reliable routing.
- **Scalability**: Consul supports horizontal scaling by allowing multiple instances of a service (e.g., `campaign`) to register under the same name, with load balancing handled by the API Gateway or

Consul's built-in load balancer.

Benefits

- Eliminates manual configuration of service endpoints in the API Gateway.
- Supports auto-scaling of services during high-traffic Shopify events.
- Provides fault tolerance by rerouting requests away from failed instances.

Implementation Notes

- Use Consul's client library (e.g., `hashicorp/consul` for Node.js) in each service.
- Monitor Consul's performance using Prometheus to ensure low-latency service resolution.
- Document health check specifications in each service plan (e.g., `core_service_plan.md`).

Event-Driven Architecture

To decouple microservices and handle Shopify's high-volume events (e.g., order completions, user actions), Loyalnest adopts an event-driven architecture using **Apache Kafka** as the message broker. Kafka enables asynchronous communication between services, reducing tight coupling and improving scalability.

Message Broker Setup

- **Kafka Deployment**: Kafka is deployed on the cloud infrastructure (aligned with `system_architecture_and_specifications.md`), with topics like `points-events`, `referrals-events`, and `order-events`.
- **Event Schemas**: Events follow a standardized JSON schema, including `event_type`, `user_id`, `timestamp`, and `context`. Example:

```
{
  "event_type": "points_earned",
  "user_id": "123",
  "points": 50,
  "timestamp": "2025-07-30T22:34:00Z",
  "context": { "order_id": "shopify_order_456" }
}
```

Event Flows

- **Shopify Webhooks**: The `core` service receives Shopify webhooks (e.g., `order/created`) and publishes events to Kafka. For example, an `order_completed` event triggers updates in `points`, `referrals`, and `rfm` services.
- **Points Service**: Publishes `points_earned` events when users earn points, consumed by `gamification` (to update badges) and `rfm` (to update frequency/monetary scores).
- **Referrals Service**: Publishes `referral_created` events, consumed by `campaign` for tracking campaign performance.
- **Event Tracking Service**: Subscribes to all events for logging and analytics, storing them in the `events` table (`event_tracking.sql`).

Integration Details

- **Producers**: Services like `core` and `points` use Kafka's producer API to publish events. Example:

```
const { Kafka } = require('kafkajs');
const kafka = new Kafka({ brokers: ['kafka:9092'] });
const producer = kafka.producer();
await producer.send({
  topic: 'points-events',
  messages: [{ value: JSON.stringify({ event_type: 'points_earned', user_id:
'123', points: 50 }) }]
});
```

- **Consumers**: Services like `gamification` and `rfm` subscribe to relevant topics using Kafka's consumer API, processing events asynchronously.
- **Idempotency**: Each event includes a unique `event_id` to prevent duplicate processing. Consumers store processed `event_id`s in their databases (e.g., `gamification.sql`).
- **Reliability**: Kafka's exactly-once semantics ensure reliable event delivery. Failed events are retried via a dead-letter queue.

Benefits

- Decouples services, allowing independent scaling and development.
- Handles Shopify's high-volume webhooks asynchronously, avoiding rate limit issues.
- Enables real-time analytics by feeding events to the `event_tracking` service.

Implementation Notes

- Use a time-series database (e.g., InfluxDB) for high-volume event storage in `event_tracking.sql` to optimize performance.
- Monitor Kafka's throughput and latency using Prometheus.
- Document event schemas and flows in `event_tracking_service_plan.md`.

API Interface Design

To provide a flexible and efficient API for Shopify merchants, Loyalnest adopts **GraphQL** as the primary API interface, with **REST** as a fallback for legacy integrations. GraphQL reduces over-fetching/under-fetching issues, improving frontend performance and aligning with Shopify's GraphQL Admin API.

GraphQL Implementation

- **GraphQL Server**: The API Gateway hosts a GraphQL server (e.g., Apollo Server) exposing a single endpoint (e.g., `/graphql`). Example schema:

```
type User {
  id: ID!
  points: Int!
```

```
  referrals: [Referral]
  rfm: RFM
}
type Query {
  user(id: ID!): User
}
```

- **Resolvers**: The API Gateway's resolvers fetch data from microservices (e.g., `users`, `points`, `referrals`) via internal REST or gRPC calls. Example resolver:

```
const resolvers = {
  Query: {
    user: async (_, { id }) => {
      const user = await fetch('http://users-service:8080/users/' +
id).then(res => res.json());
      const points = await fetch('http://points-service:8080/points/' +
id).then(res => res.json());
      return { ...user, points: points.value };
    }
  }
};
```

- **Data Loaders**: Use `dataloader` to batch and cache requests, preventing N+1 query issues.
- **Shopify Integration**: The `core` service uses Shopify's GraphQL Admin API for queries (e.g., fetching orders), reducing API call overhead.

REST Fallback

- **Legacy Support**: Maintain REST endpoints (e.g., `/users/:id`, `/points/:userId`) for existing integrations or services not yet migrated to GraphQL.
- **Deprecation Plan**: Document a timeline in `api_gateway_service_plan.md` to phase out REST endpoints, encouraging clients to adopt GraphQL.

Benefits

- GraphQL's flexible queries reduce API calls, improving performance for Shopify merchants.
- Aligns with Shopify's GraphQL-based Admin API, simplifying integration.
- REST fallback ensures backward compatibility during the transition.

Implementation Notes

- Document the GraphQL schema and resolvers in `api_gateway_service_plan.md`.
- Update the frontend (`frontend_service_plan.md`) to use a GraphQL client (e.g., Apollo Client).
- Ensure rate limiting in the API Gateway supports GraphQL's single-endpoint model.

Next Steps

- Implement service discovery using Consul, starting with the `core` service.

- Deploy Kafka and prototype event flows for `points_earned` events.
- Set up a GraphQL server in the API Gateway and test with `features_1_must_have.md` queries.
- Update related service plans (e.g., `api_gateway_service_plan.md`, `event_tracking_service_plan.md`) to reflect these changes.