Payment Link System – Architecture & Infrastructure Decisions

This document summarizes the key architecture and infrastructure decisions for the Payment Link system and the main tradeoffs behind them. It complements the product and technical strategy from Part 1 and focuses on how the system is implemented on AWS using Terraform.

1. High-level architecture

I organized the system into three main layers:

- Frontend
  - Public checkout page that customers open via a payment link.
  - Merchant portal where businesses create and manage links and see payments.
- Backend (Payment Link Service)
  - Payment Link API (CRUD for links and public checkout endpoints).
  - Payment Orchestrator + PSP Router (selects PSP, handles failover, orchestrates the flow).
  - Fee Engine (fixed fees, variable fees, FX markup).
  - FX Service + cache (fetches FX rates, applies markup, stores snapshots).
  - PSP Adapters (Stripe/Adyen-like, simulated).
  - Webhook Handler (processes asynchronous PSP updates).
- Data & Supporting Services
  - Relational database (PostgreSQL) for core business data.
  - Redis for FX caching and potential rate limiting.
  - Parameter store for secrets and configuration.
  - Cloud logging and metrics for observability.

This structure directly addresses the core requirements: multi-PSP orchestration, fee calculation, FX management, payment link lifecycle, and security.

2. Application architecture

Frontend

On the checkout side, I avoid sending raw card data to the backend. The browser uses a simulated PSP tokenization flow (analogous to Stripe or Adyen JS SDKs). It collects card details, calls a tokenization endpoint, and the backend only sees a **card token** or payment method ID.

This design aligns with PCI DSS: the backend never handles PAN or CVV, only opaque tokens. The tradeoff is some coupling between the frontend and PSP capabilities, but it significantly reduces PCI scope and security risk, which is more important at this stage.

The merchant portal is a separate frontend that uses authenticated APIs to manage links and view payments. This separation keeps the public checkout surface minimal and secure, and concentrates admin complexity in the internal UI.

Backend

The backend is a single Spring Boot service, internally split into clear modules:

- The **Payment Link API** exposes admin and checkout endpoints and delegates to domain services.
- The **Payment Orchestrator + PSP Router** is the core: it validates the payment link, pulls fee and FX info, chooses a PSP based on routing rules, calls the PSP adapter, and, if configured, retries with a secondary PSP on retriable errors. Routing is data-driven using tables like `PSP` and `PSP_ROUTING_RULE` (by merchant, currency, country, card brand). The client sends an idempotency key (as a header or request field), and the Payment Orchestrator first checks if a `PAYMENT` record already exists for the pair (`payment_link_id`, `idempotency_key`). If it does, the service returns the existing result instead of creating a new payment. If it does not, it creates a new `PAYMENT`, runs the orchestration flow, and stores the outcome. This prevents double charging when clients or intermediaries retry the request
- The **Fee Engine** reads `MERCHANT_FEE_CONFIG` and calculates fixed fees, percentage fees, and FX markup. The result is stored in `PAYMENT_FEE`, so the exact fee breakdown is auditable later, regardless of config changes.
- The **FX Service** gets FX rates from a simulated provider and persists a per-payment `FX_RATE_SNAPSHOT`. FX rates are cached in Redis to improve latency and reduce external calls. The snapshot in the relational DB ensures consistency and auditability.
- **PSP Adapters** implement a common interface (authorize/capture/refund) and wrap each PSP's specifics. This keeps the orchestrator independent from individual PSP APIs.
- The **Webhook Handler** receives asynchronous events (authorization, capture, refund) and stores raw payloads in `WEBHOOK_EVENT`, then updates the internal `PAYMENT` state. The Webhook Handler is also implemented in an idempotent way. Each PSP event includes a unique `psp_event_id` that is stored in `WEBHOOK_EVENT` with a uniqueness constraint. When a webhook is received, the handler checks whether that event has already been processed; if so, it skips side effects and returns success. Only the first occurrence updates the corresponding `PAYMENT` state. This makes webhook processing safe under PSP retries and duplicate deliveries.

I intentionally chose a **modular monolith** instead of microservices. For a first version and a single AWS environment, this is cheaper and easier to deploy, monitor, and change. The boundaries (orchestrator, routing, FX, fees) are explicit in code and data, so they can be extracted into services later if scale or team structure requires it.

Data model

The relational schema is normalized around:

- `MERCHANT` – the B2B customer using the platform.
- `PAYMENT_LINK` – each payment link created by a merchant.
- `PAYMENT` – each payment attempt/transaction from a link.
- `MERCHANT_FEE_CONFIG` and `PAYMENT_FEE` – fee configuration vs. applied fees.
- `FX_RATE_SNAPSHOT` – the FX rate actually used for each payment.
- `RECIPIENT` and payout fields in `PAYMENT` – the end recipient and net payout amount/currency.
- `INCENTIVE_RULE` and `PAYMENT_INCENTIVE` – incentives like “first N transactions cheaper/free”.
- `PSP`, `PSP_ROUTING_RULE`, `WEBHOOK_EVENT` – PSP catalog, routing configuration, and PSP event log.

This model supports the core story from Part 1: a merchant issues a link, a customer pays, the system routes to PSPs, applies fees and FX, maybe applies incentives, and delivers a net payout to a recipient.

3. Infrastructure and IaC (AWS + Terraform)

For infrastructure I use **AWS**, defined with **Terraform**, focusing on a single `staging` environment.

Compute

Instead of ECS/Fargate or Lambda, I run the Spring Boot application on a single **EC2 t3.micro** instance in a dedicated VPC. Reasons:

- Simpler to understand and review in a time-boxed case study.
- Avoids extra cost and complexity of Fargate + ALB for a small staging setup.
- Still realistic enough to run end-to-end scenarios.

The main tradeoff is availability and scalability: a single EC2 instance is a single point of failure and does not auto-scale. For production, I would move to containers on ECS Fargate behind an Application Load Balancer, with multiple tasks across AZs. For this staging environment, I explicitly accept reduced availability in exchange for lower cost and simpler IaC.

Database and cache

I use **RDS PostgreSQL** with a small `db.t3.micro` instance as the relational database. PostgreSQL offers strong ACID guarantees, good numeric/monetary support, JSONB for PSP payloads, and first-class support in Spring Boot, which fits the financial nature of the domain.

For caching I provision a small **ElastiCache Redis** cluster (single node). Redis is used for FX rate caching and can later support rate limiting. In staging, Redis is an optimization: if it is temporarily down, the system can still fetch FX rates from the upstream source. That keeps Redis out of the critical path for correctness.

Networking and security

Terraform creates a dedicated **VPC** with:

- One public subnet for the EC2 instance (with public IP and Internet Gateway).
- One private subnet for RDS and Redis (no public IPs).
- Security groups so that:
  - SSH and HTTP access to EC2 are restricted (SSH only from my IP).
  - RDS and Redis only accept traffic from the application security group.

This setup keeps data stores off the public internet while staying simple enough for a staging/free-tier environment. In production, I would move the app instances to private subnets behind an ALB and use multiple AZs, but that is intentionally out of scope here.

Secrets and configuration

For secrets I chose **AWS Systems Manager Parameter Store (SSM)** instead of Secrets Manager. Terraform creates parameters for:

- Database URL, username, and password.
- Redis host and port.
- Optionally, PSP API keys and webhook secrets.

The EC2 instance reads these parameters at startup (through the AWS SDK or environment variables), so sensitive values are not hard-coded in Terraform, not committed to Git, and not baked into the AMI. The tradeoff is fewer advanced features (like automatic rotation) compared to Secrets Manager, but for staging and this scope SSM offers a good cost–security balance.

Observability

For logging and basic monitoring I rely on **CloudWatch Logs** and built-in EC2/RDS metrics:

- Application logs are sent to a dedicated CloudWatch Logs group.
- CloudWatch alarms can monitor CPU usage, RDS storage, or generic health indicators (e.g. high error rate).

I intentionally avoid a full ELK or tracing stack to limit complexity and cost. The tradeoff is less advanced observability, but sufficient for debugging and validating this staging environment.

Terraform structure

Terraform is organized into reusable modules (network, EC2 app, RDS, Redis, SSM parameters, CloudWatch) and a single `staging` configuration that wires them together. This makes it easy to replicate the same structure for future environments (e.g. production) by reusing modules with different sizes and settings.