Documento Arquitectónico - Plataforma ePayco

Autor: Cristian Villegas

Tabla de Contenidos

1. Resumen Ejecutivo
2. Contexto y Objetivos
3. Decisiones Arquitectónicas Clave
4. Infraestructura AWS
5. Arquitectura de Backend
6. Arquitectura de Frontend
7. SAGA Pattern y Consistencia Distribuida
8. Seguridad y Resiliencia
9. Observabilidad
10. Estimación de Costos
11. Roadmap de Implementación
12. Conclusiones

---

1. Resumen Ejecutivo

Objetivos

Plataforma cloud-native para pagos electrónicos con:

- Disponibilidad 99.9% con Multi-AZ y multi-región
- Seguridad PCI-DSS Level 1
- Auto-scaling para picos de demanda
- DR con RPO 5-10 min, RTO 30-45 min
- Soporte multi-país LATAM

Decisiones Principales

Orquestación: ECS Fargate sobre EKS por menor complejidad y time-to-market más rápido.

SAGA Pattern: Step Functions sobre orquestador custom. Auditoría PCI-DSS incluida y visual debugging.

Base de Datos: Aurora PostgreSQL para ACID transaccional. DynamoDB para event sourcing.

Backend: NestJS con Clean Architecture. Facilita testing y escalabilidad enterprise.

Frontend: Next.js con SSR/SSG para mejor performance y SEO.

Multi-región: Activo-pasivo desde el inicio para DR robusto.

Stack Tecnológico

Compute: ECS Fargate, Lambda, Step Functions

Data: Aurora PostgreSQL Global, ElastiCache Redis, DynamoDB

Integration: API Gateway, ALB, SQS/SNS/EventBridge

Observability: CloudWatch, X-Ray, structured logging

---

2. Contexto y Objetivos

Contexto

ePayco es una fintech colombiana que procesa pagos online (PSE, tarjetas, wallets), dispersa pagos a comercios y ofrece integración con e-commerce vía APIs.

Desafíos principales: - Expansión multi-país en LATAM - Picos de tráfico en campañas y fechas comerciales - Cumplimiento PCI-DSS Level 1 - Consistencia en transacciones distribuidas

Requerimientos

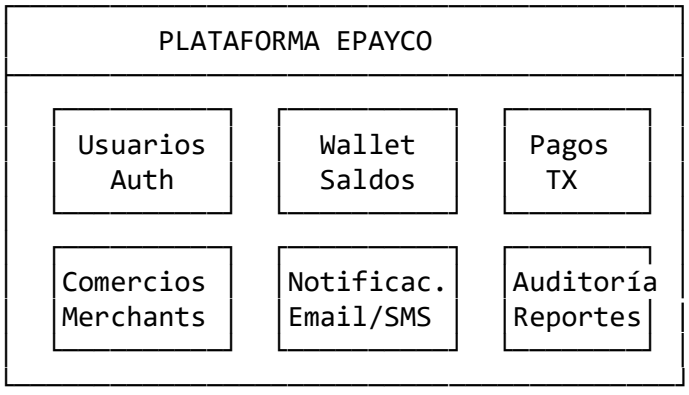
Disponibilidad: SLA 99.9% con Multi-AZ y DR multi-región.

Performance: Latencia P95 < 500ms, throughput para alto volumen, read replicas para escalabilidad.

Seguridad: PCI-DSS Level 1, TLS 1.3 en tránsito, AES-256 en reposo.

Escalabilidad: Auto-scaling horizontal automático - De carga base a picos de alta demanda - Servicios stateless

2.3 Dominios Funcionales



Bounded Contexts: ### Servicios

- **Auth Service:** Identidad y autenticación
- **Wallet Service:** Saldos, movimientos, reservas
- **Payment Service:** Transacciones y anti-fraude

- **Merchant Service:** Onboarding y configuración
- **Notification Service:** Email, SMS, webhooks
- **Audit Service:** Compliance y reportes

---

3. Decisiones Arquitectónicas Clave

Ver justifications/ para análisis completo de cada decisión.

Microservicios

Arquitectura de microservicios con bounded contexts (DDD) para escalar servicios independientemente y permitir equipos autónomos.

Beneficios: Escalado independiente por servicio, deploys sin downtime, resiliencia ante fallos aislados.

Trade-offs: Mayor complejidad operacional (mitigado con Fargate managed), consistencia distribuida (SAGA pattern), observabilidad (X-Ray tracing).

ECS Fargate sobre Kubernetes

Fargate para ~6 microservicios iniciales. Setup en días vs semanas con K8s, sin gestión de infraestructura, auto-scaling simple. Equipo pequeño (3-4 devs) puede mantenerlo sin experiencia K8s profunda.

Path de evolución: Migrar a EKS si crece complejidad o necesitamos multi-cloud. Clean Architecture facilita la migración.

Step Functions para SAGA

Orquestador SAGA con Step Functions para flujos de pago (6+ pasos distribuidos). Auditoría PCI-DSS incluida, visual debugging, retry/compensaciones built-in. Time-to-market en días vs semanas con orquestador custom.

Costo: ~$200/mes (1M tx × 8 steps)

Aurora PostgreSQL + Híbrido NoSQL

- **Aurora PostgreSQL:** ACID para transacciones financieras, read replicas, Global Database para DR
- **DynamoDB:** Event sourcing append-only
- **Redis:** Caché, sesiones, rate limiting

Multi-Región Activo-Pasivo

Primary en us-east-1, DR en us-west-2. RPO 5-10 min, RTO 30-45 min. Costo +$450/mes vs +$2000/mes activo-activo

Failover: - Route 53 health checks (30 segundos) - Aurora promotion automática - Fargate tasks escalan en DR

---

4. Infraestructura AWS

Servicios Principales

Compute: ECS Fargate, Lambda, Step Functions

Networking: VPC 10.0.0.0/16, ALB, API Gateway, Route 53

Data: Aurora PostgreSQL Global, ElastiCache Redis, DynamoDB, S3

Integration: SQS, SNS, EventBridge

Security: WAF, Shield, Secrets Manager, KMS

Observability: CloudWatch, X-Ray, CloudTrail

Topología de Red

VPC Multi-AZ con subnets públicas (ALB, NAT), privadas app (Fargate), y privadas data (Aurora, Redis sin acceso internet).

Security Groups: ALB permite 443 externo → Fargate solo desde ALB → Aurora/Redis solo desde Fargate.

Flujo: Usuario → CloudFront → Route 53 → ALB → Fargate → Aurora/Redis

Multi-Región

Primary us-east-1 procesa 100% tráfico. DR us-west-2 con Aurora read replica promocionable e infra pre-provisionada.

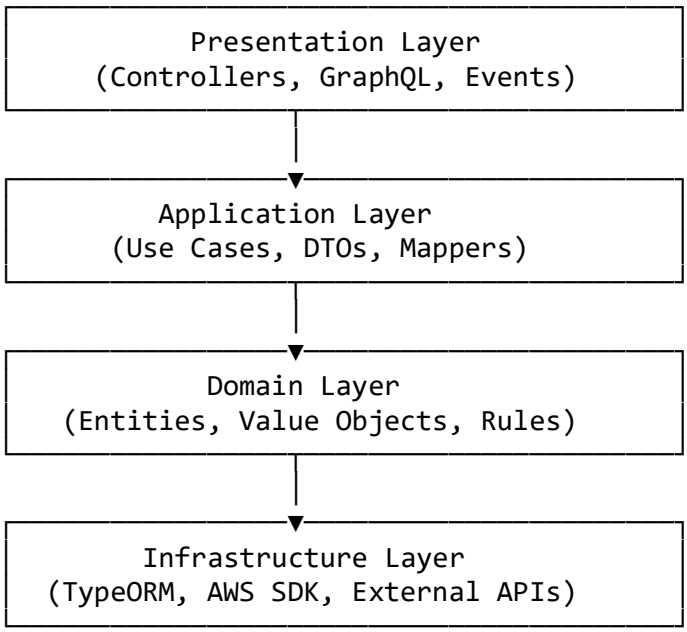
Failover: Route 53 detecta falla (30s) → DNS redirige → Aurora promociona → Fargate escala. RTO: 30-45 min.

---

5. Arquitectura de Backend

5.1 Principios de Diseño

Clean Architecture (Hexagonal):



SOLID + DDD: - Dependency Inversion: Domain no depende de infraestructura - Single Responsibility: Cada módulo una responsabilidad - Domain-Driven Design: Bounded contexts por dominio

5.2 Servicios Core

Payment Service (crítico):

```
src/
├── domain/
│   ├── entities/payment.entity.ts
│   ├── value-objects/money.ts
│   └── repositories/payment.repository.ts (interface)
├── application/
│   ├── use-cases/
│   │   ├── create-payment.use-case.ts
│   │   └── process-payment.use-case.ts
│   └── dtos/payment.dto.ts
├── infrastructure/
│   ├── database/typeorm-payment.repository.ts
│   ├── external/pse-gateway.adapter.ts
│   └── messaging/sqs-producer.ts
└── presentation/
    └── http/payment.controller.ts
```

5. Arquitectura de Backend

Clean Architecture

Cuatro capas: Presentation (Controllers) → Application (Use Cases) → Domain (Entities) → Infrastructure (TypeORM, AWS SDK).

Domain no depende de frameworks. Aplicamos SOLID + DDD con bounded contexts por dominio.

Servicios

Payment Service: Transacciones y anti-fraude con estructura domain/application/infrastructure/api.

Otros servicios: Wallet (saldos, reservas), Auth (OAuth 2.0 + JWT), Merchant (onboarding), Notification (email/SMS/webhooks), Audit (compliance).

Patrones

Repository: Interfaces en domain, implementaciones en infrastructure.

Factory: Creación de entidades complejas.

Event Sourcing: PaymentInitiated → PaymentValidated → PaymentProcessed → PaymentCompleted.

---

6. Arquitectura de Frontend

Next.js con SSR/SSG

Estrategia híbrida: SSR para dashboards con datos dinámicos, SSG para landing pages estáticas con ISR (revalidación cada hora), CSR para interacciones en tiempo real.

Performance

Optimizaciones: Image optimization, code splitting automático, font optimization, prefetching.

Métricas: LCP < 2.5s, FID < 100ms, CLS < 0.1.

---

7. SAGA Pattern y Consistencia Distribuida

Workflow de Pago

Step Functions orquesta 6 pasos: ValidateUser → CheckBalance → ReserveFunds → ProcessPayment → ConfirmTransaction → SendNotification.

Cada paso tiene compensación automática si falla: UnreserveFunds, NotifyFailure.

Idempotencia

Todas las operaciones usan idempotency-key con caché de 24 horas para prevenir duplicados.

---

8. Seguridad y Resiliencia

Seguridad

Defensa en profundidad: CloudFront + WAF (OWASP Top 10, rate limiting) → API Gateway (throttling, validation) → ALB (TLS termination) → Fargate (private subnet, IAM roles, secrets) → Aurora/Redis (cifrado en reposo KMS, en tránsito TLS, sin endpoint público).

Tokenización: Nunca almacenar datos sensibles en texto plano. Usar tokens para tarjetas.

Resiliencia

Circuit Breaker: timeout 5s, error threshold 50%, reset 30s.

Retry: Backoff exponencial (1s, 2s, 4s) con máximo 3 intentos.

Rate Limiting: 100 requests/minuto por usuario.

---

9. Observabilidad

Logging y Métricas

Logs estructurados en JSON con traceId. CloudWatch Insights para queries.

Métricas de negocio (transacciones exitosas/fallidas, revenue) y técnicas (latencia p95/p99, error rate, CPU/memoria).

Alarmas críticas: error rate > 1%, latencia > 500ms, Aurora CPU > 80%.

Tracing

X-Ray para distributed tracing con service map automático: CloudFront → ALB → payment-service → wallet-service/fraud-service/Aurora/Gateway.

---

10. Estimación de Costos

Primary (us-east-1): ~$1,720/mes (Fargate $500, Aurora $400, Redis $200, Step Functions $200, otros servicios $420)

DR (us-west-2): ~$450/mes (Aurora replica, ALB idle, S3 CRR)

Total: ~$2,170/mes (~$26,040/año)

Optimizaciones: Fargate Spot dev/staging (-$350), Aurora Reserved (-$160), S3 Intelligent-Tiering (-$10). Ahorro potencial: ~$500/mes.

---

11. Roadmap

Fase 1 (Semanas 1-2): VPC, ECS, Aurora, Redis. Auth/Wallet/Payment services MVP. Frontend Next.js básico.

Fase 2 (Semanas 3-4): Step Functions SAGA completo. Integración PSE/tarjetas. Tests unitarios e integración. Load testing.

Fase 3 (Semanas 5-6): Aurora Global DB. DR testing. CloudWatch dashboards, X-Ray. WAF rules. PCI-DSS assessment.

Fase 4 (Semanas 7-8): Auto-scaling refinement. Cost optimization. Performance tuning. Documentación.

---

12. Conclusiones

Cumplimiento de Requisitos

La arquitectura cumple todos los objetivos: disponibilidad 99.9% con Multi-AZ y multi-región, latencia P95 < 500ms con Fargate + Redis + Aurora replicas, escalabilidad horizontal automática, seguridad PCI-DSS L1 con cifrado E2E y WAF, DR con RPO 5min/RTO 30min usando Aurora Global DB, y costos optimizados ~$2,200/mes con servicios managed.

Fortalezas

Cloud-native aprovechando servicios managed de AWS. Escalable con auto-scaling en todas las capas. Resiliente con circuit breaker, retry y SAGA compensations. Observable con logs estructurados, métricas y tracing. Segura con defense in depth y cifrado E2E. Mantenible con Clean Architecture que un equipo pequeño puede operar.

Riesgos

Vendor lock-in AWS: Mitigado con Clean Architecture portable e IaC con Terraform.

Costos Step Functions: Monitoreo de costos, alternativa SAGA custom documentada si escala mucho.

Complejidad microservicios: X-Ray tracing, logs centralizados y Step Functions visual facilitan debugging distribuido.

Evolución

Corto plazo (3-6m): Completar módulos Comercios/Auditoría, portal admin, dashboards avanzados.

Mediano plazo (6-12m): Expansión multi-región según demanda, mejores integraciones, webhooks con retry, APIs versionadas.

Largo plazo (12+m): Evaluar EKS si aumenta complejidad, multi-cloud si compliance lo requiere.

Referencias

- AWS Well-Architected Framework
- NestJS Documentation
- SAGA Pattern - Microservices.io
- Clean Architecture - Robert C. Martin
- **saga-y-consistencia.md** - Workflows, compensaciones
- **seguridad-resiliencia.md** - Implementación detallada
- **observability.md** - Dashboards, queries, alertas
- **iac-terraform.md** - Módulos Terraform, CI/CD