A) Circuit Breaker Pattern

Implementación en Todos los Microservicios:

Estados del Circuit Breaker:

CLOSED (Normal Operation)
├─► Todas las requests pasan al servicio
├─► Se monitorean fallos
└─► Si tasa de error > 50% en 10 requests → OPEN

OPEN (Fail Fast)
├─► Requests fallan inmediatamente (no se llama al servicio)
├─► Se evita sobrecarga del servicio caído
├─► Timeout de 30 segundos → HALF_OPEN

HALF_OPEN (Testing)
├─► Se permite 1 request de prueba
├─► Si éxito → CLOSED
└─► Si fallo → OPEN (esperar otros 30 segundos)

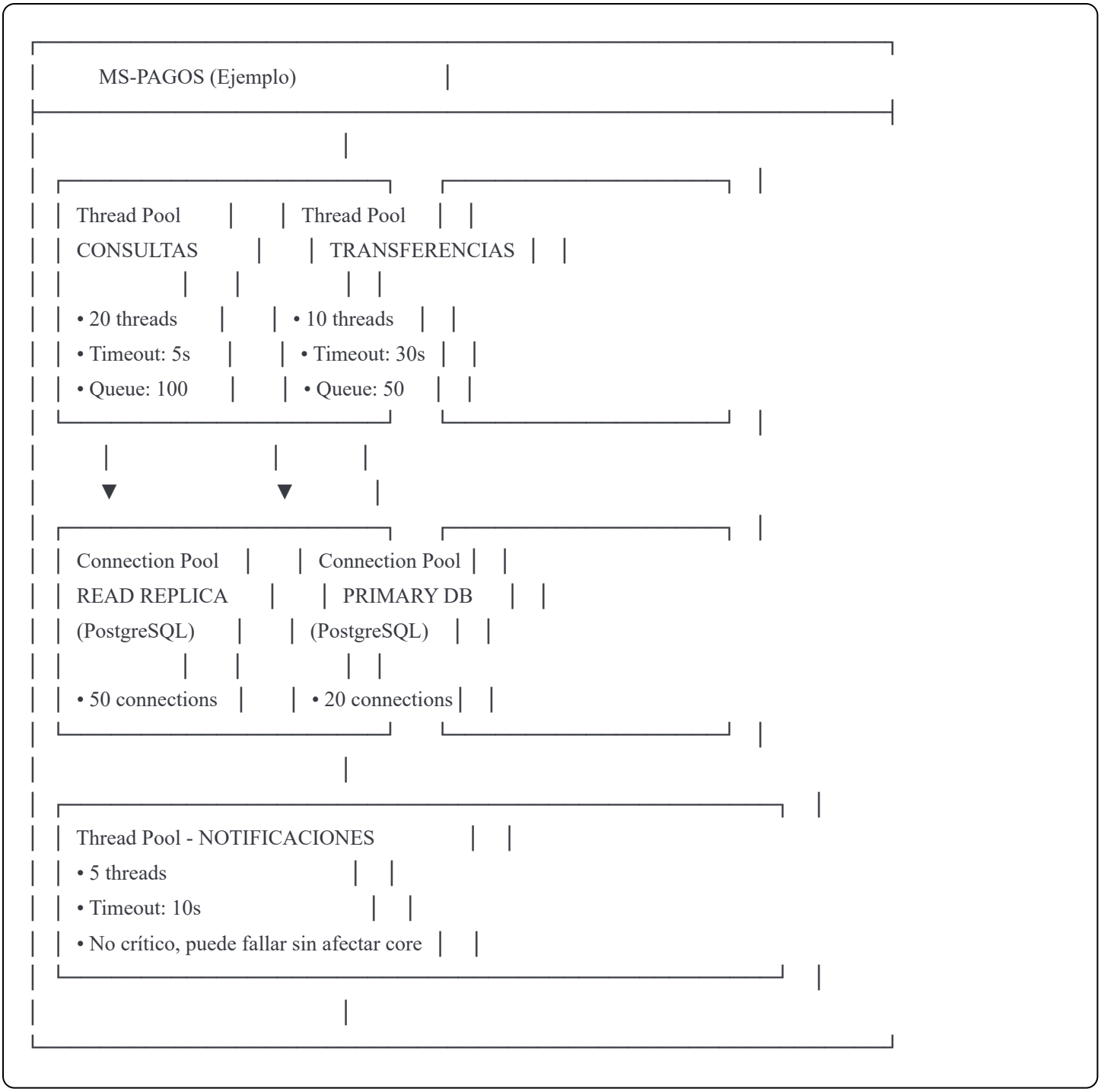
Configuración por Microservicio:

Servicio	Threshold	Timeout	Reset Time
MS-Datos-Cliente	50% error en 10 req	5s	30s
MS-Pagos	60% error en 20 req	10s	60s
ACH Network (externo)	40% error en 5 req	15s	120s
MS-Notificaciones	70% error en 50 req	3s	20s

Beneficio: Evita cascading failures, protege servicios downstream, fail-fast para mejor UX.

B) Bulkhead Pattern - Aislamiento de Recursos

Pool de Threads Separados por Tipo de Operación:



Justificación:

- Si notificaciones fallan o se saturan, **NO afecta** las transferencias
- Si consultas masivas saturan el pool, **NO bloquean** las transferencias
- Aislamiento total de recursos críticos vs no-críticos

C) Retry con Exponential Backoff + Jitter

Estrategia de Reintentos Inteligente:

```
Intento 1: Inmediato
  ├─ Falla → Esperar 1s + random(0-200ms)
  │
Intento 2: Después de ~1s
  ├─ Falla → Esperar 2s + random(0-400ms)
  │
Intento 3: Después de ~2s
  ├─ Falla → Esperar 4s + random(0-800ms)
  │
Intento 4: Después de ~4s
  ├─ Falla → Esperar 8s + random(0-1600ms)
  │
Intento 5: Después de ~8s
  └─ Falla → Error definitivo, ejecutar compensación
```

Jitter (Ruido Aleatorio):

Previene "thundering herd" - cuando múltiples clientes reintentan simultáneamente y sobrecargan el servicio recuperándose.

Operaciones Idempotentes:

Todas las mutaciones incluyen `idempotencyKey`:

```graphql
mutation CreateTransfer($input: TransferInput!) {
  createTransfer(input: $input, idempotencyKey: "uuid-12345") {
    id
    status
  }
}
```

Si se reintenta con el mismo key:

- Primera vez: Ejecuta operación
- Reintentos: Devuelve resultado de la primera ejecución (no duplica)

D) Timeouts Agresivos

Configuración de Timeouts en Cascada:

```
Frontend (SPA/Mobile)
├─ Timeout: 30s
│
API Gateway
├─ Timeout: 25s
│
Microservicio
├─ Timeout: 20s
│
Base de Datos
├─ Timeout: 15s
```

Razón: Cada capa tiene timeout menor que la anterior, evitando "timeout hell" donde todos esperan indefinidamente.

E) Graceful Degradation

Niveles de Degradación del Servicio:

Escenario	Servicio Afectado	Comportamiento Degradado
MS-Históricos caído	Consulta de movimientos	Mostrar últimos 5 desde caché + mensaje "Datos pueden estar desactualizados"
ACH Network lento	Transferencias interbancarias	Mostrar "Procesando, recibirás notificación" (asíncrono)
MS-Notificaciones saturado	Envío de emails	Queue notifications, enviar cuando se recupere
Redis caído	Caché de sesiones	Leer directo de PostgreSQL (más lento pero funcional)
Firebase caído	Push notifications	Fallback a SMS o solo email

Ejemplo de Implementación:

```
javascript
```

```
// Degradación con fallback a caché
async function getTransactionHistory(accountId) {
  try {
    // Intento 1: Servicio principal
    return await historyService.getTransactions(accountId);
  } catch (error) {
    try {
      // Intento 2: Caché de Redis
      const cached = await redis.get(`history:${accountId}`);
      if (cached) {
        return {
          ...JSON.parse(cached),
          isStale: true,
          message: "Datos de hace 15 minutos"
        };
      }
    } catch (cacheError) {
      // Intento 3: Base de datos de lectura
      return await readDB.query(
        'SELECT * FROM transactions WHERE account_id = $1 LIMIT 5',
        [accountId]
      );
    }
  }
}
```

11.5.3 ADAPTACIÓN - Escalabilidad Dinámica

A) Horizontal Pod Autoscaler (HPA) Avanzado

Métricas Múltiples para Escalado:

yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: ms-pagos-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ms-pagos
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  - type: Pods
    pods:
      metric:
        name: http_requests_per_second
      target:
        type: AverageValue
        averageValue: "1000"
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300  # Esperar 5 min antes de scale down
      policies:
      - type: Percent
        value: 50  # Reducir máximo 50% de pods a la vez
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 0  # Scale up inmediato
      policies:
      - type: Percent
        value: 100  # Duplicar pods si es necesario
        periodSeconds: 15
```

Comportamiento:

- **Scale Up**: Rápido (15 segundos) para manejar picos
- **Scale Down**: Conservador (5 minutos) para evitar flapping
- **Métricas combinadas**: CPU + Memoria + Request Rate

B) Vertical Pod Autoscaler (VPA)

Ajuste Automático de Recursos:

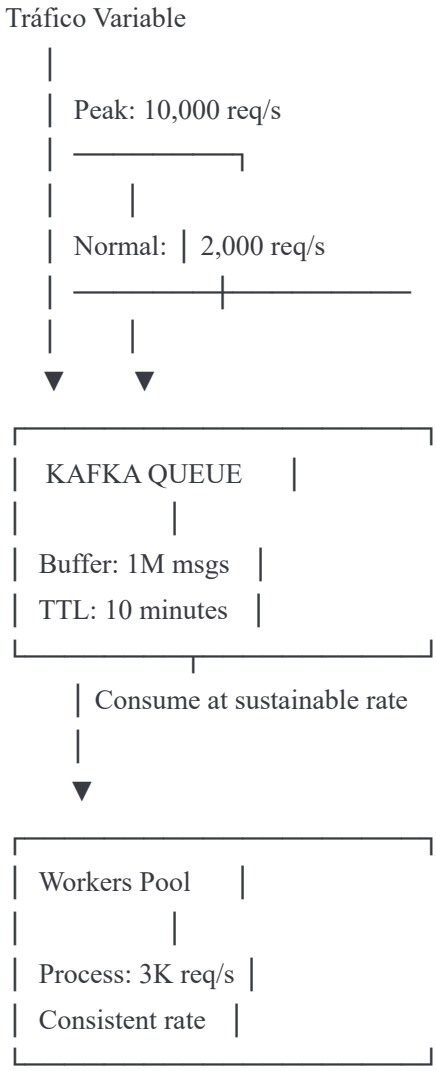
```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: ms-pagos-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ms-pagos
  updatePolicy:
    updateMode: "Auto"  # Ajusta automáticamente
  resourcePolicy:
    containerPolicies:
    - containerName: ms-pagos
      minAllowed:
        cpu: 250m
        memory: 512Mi
      maxAllowed:
        cpu: 4000m
        memory: 8Gi
      mode: Auto
```

Funcionamiento:

- Analiza uso histórico de últimos 7 días
- Ajusta requests/limits de CPU y RAM
- Reinicia pods gradualmente para aplicar cambios
- **Ahorro estimado**: 30-40% en costos al evitar sobre-aprovisionamiento

C) Queue-Based Load Leveling

Absorción de Picos con Colas:



Beneficio:

- Sistema procesa a ritmo constante
- Picos se absorben en la cola
- No necesita sobre-aprovisionar para picos
- **Ahorro**: 50-60% en costos vs aprovisionar para pico

D) Rate Limiting Adaptativo

Rate Limit que se Ajusta Según Carga:

javascript

```
class AdaptiveRateLimiter {
  constructor() {
    this.baseLimit = 100; // requests/minuto por usuario
    this.systemLoad = 0; // 0-100%
  }

  async checkLimit(userId) {
    // Obtener carga actual del sistema
    this.systemLoad = await this.getSystemLoad();

    // Calcular límite ajustado
    let adjustedLimit = this.baseLimit;

    if (this.systemLoad > 90) {
      adjustedLimit = this.baseLimit * 0.5; // Reducir 50%
    } else if (this.systemLoad > 75) {
      adjustedLimit = this.baseLimit * 0.75; // Reducir 25%
    } else if (this.systemLoad < 30) {
      adjustedLimit = this.baseLimit * 1.5; // Aumentar 50%
    }

    // Verificar contra Redis
    const key = `ratelimit:${userId}`;
    const current = await redis.incr(key);

    if (current === 1) {
      await redis.expire(key, 60); // Ventana de 1 minuto
    }

    return current <= adjustedLimit;
  }

  async getSystemLoad() {
    // Métricas de Prometheus
    const cpuUsage = await prometheus.query('avg(cpu_usage)');
    const queueDepth = await redis.llen('main_queue');
    const errorRate = await prometheus.query('rate(errors[5m])');

    // Calcular score de carga (0-100)
    return (cpuUsage * 0.4) + (queueDepth/1000 * 0.4) + (errorRate * 0.2);
  }
}
```

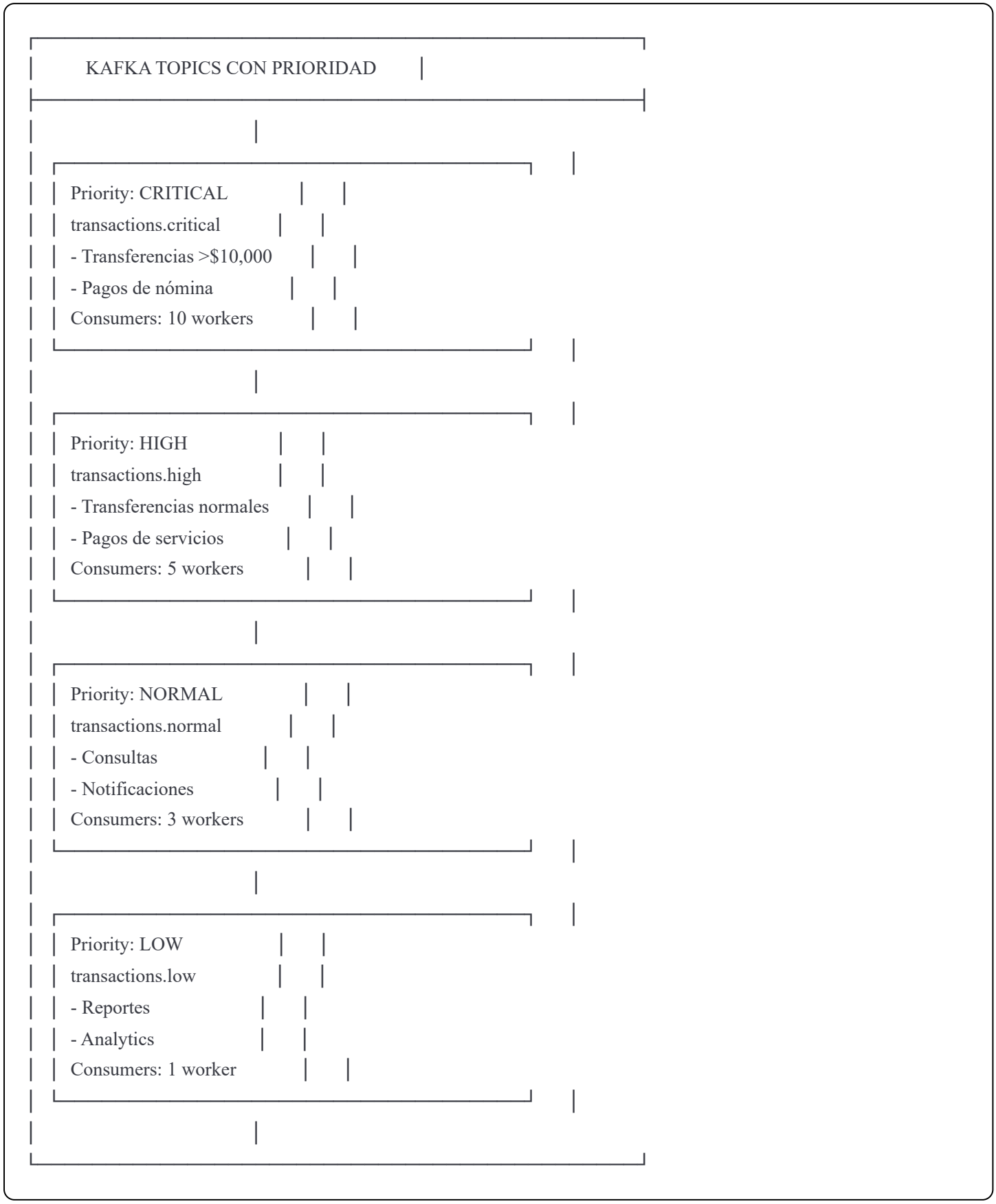
Comportamiento:

- **Carga baja (<30%)**: Usuarios pueden hacer más requests

- **Carga normal (30-75%)**: Rate limit estándar
- **Carga alta (>75%)**: Rate limit más restrictivo
- **Carga crítica (>90%)**: Rate limit muy restrictivo + shed load

E) Priority Queues

Diferentes Prioridades según Tipo de Operación:



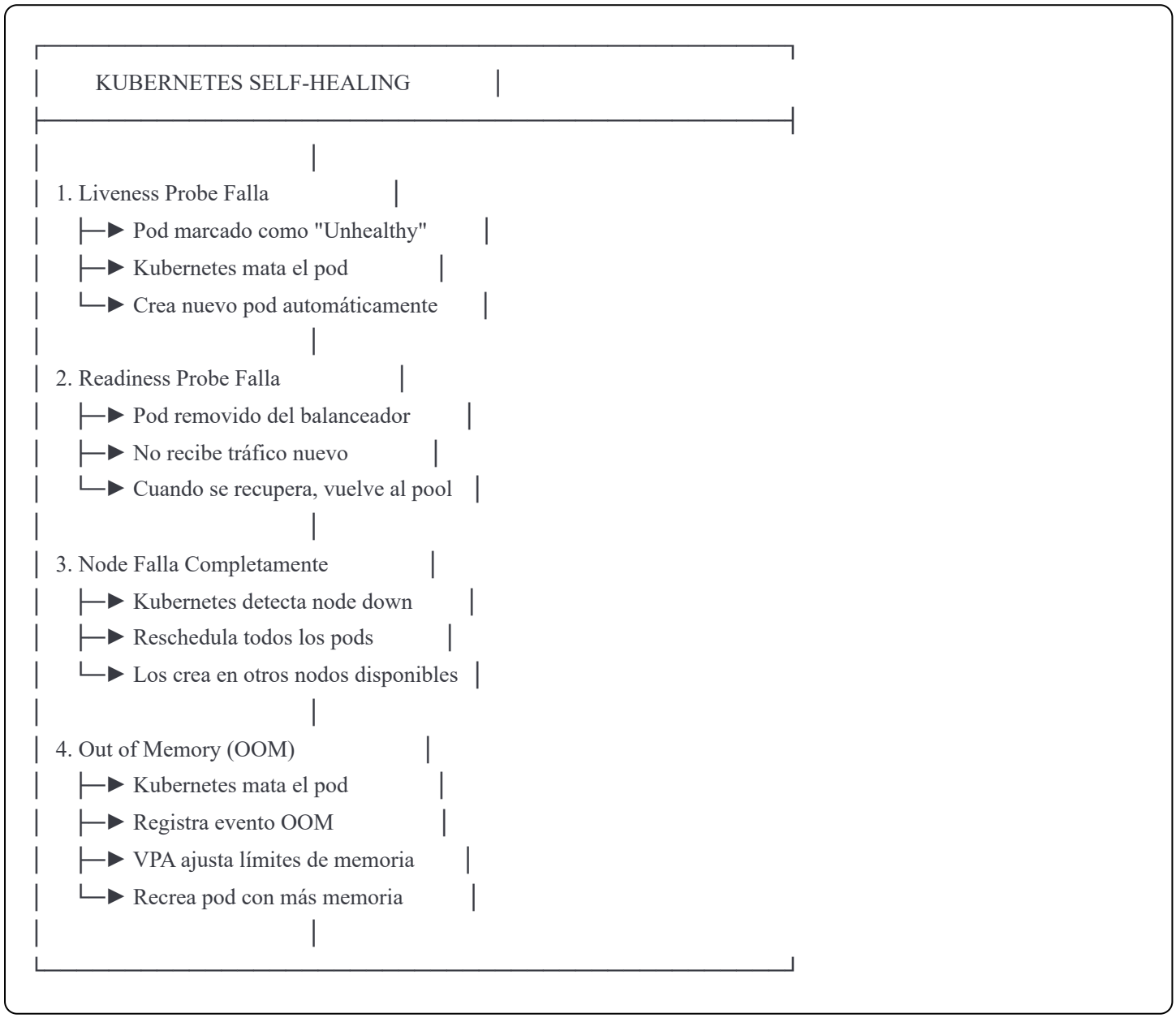
Estrategia:

- Operaciones críticas siempre se procesan primero
- En alta carga, operaciones LOW pueden pausarse temporalmente
- **SLA diferenciado**: CRITICAL <5s, HIGH <30s, NORMAL <2min, LOW best-effort

11.5.4 RECUPERACIÓN - Auto-Healing y Disaster Recovery

A) Self-Healing en Kubernetes

Recuperación Automática de Fallos:



Configuración de Probes:

yaml

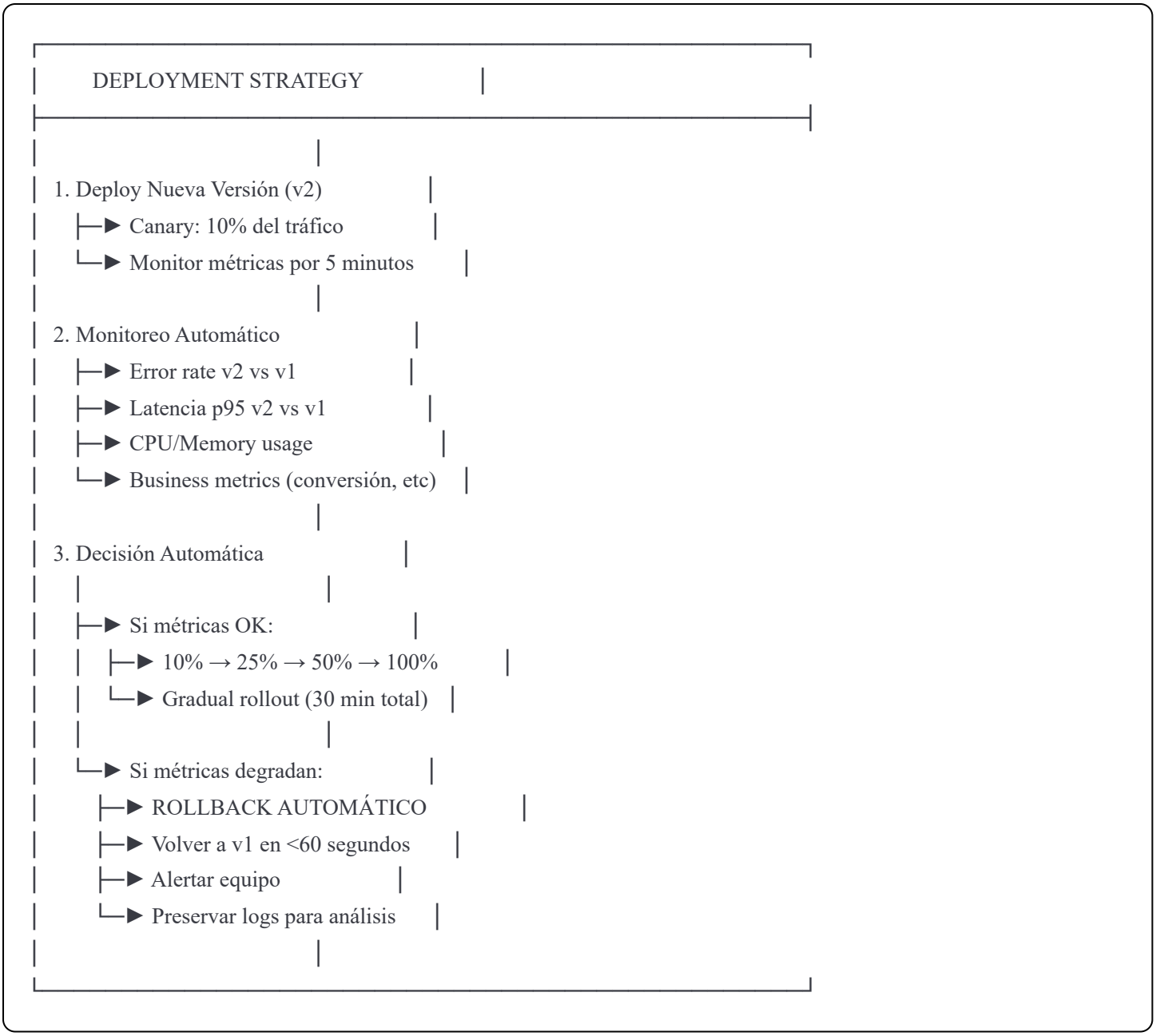
```yaml
livenessProbe:
  httpGet:
    path: /health
    port: 3000
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 3
  failureThreshold: 3  # 3 fallos consecutivos = restart

readinessProbe:
  httpGet:
    path: /ready
    port: 3000
  initialDelaySeconds: 10
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 2  # 2 fallos = remover del balanceador

startupProbe:
  httpGet:
    path: /startup
    port: 3000
  initialDelaySeconds: 0
  periodSeconds: 5
  failureThreshold: 30  # Máximo 2.5 minutos para iniciar
```

B) Automated Rollback

Despliegue Seguro con Rollback Automático:

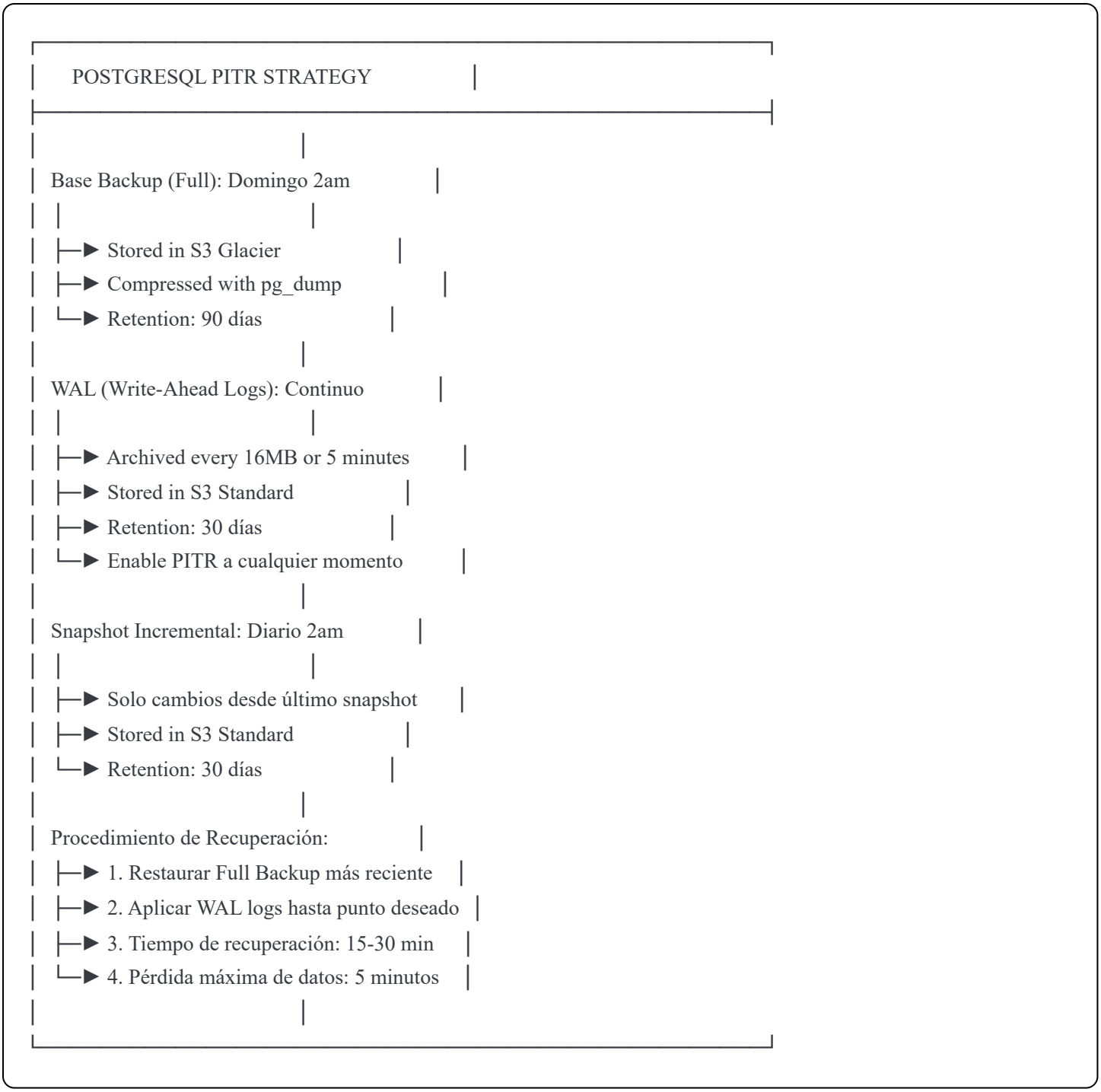


Criterios de Rollback Automático:

Métrica	Threshold	Acción
Error rate	+50% vs baseline	Rollback inmediato
Latencia p95	+100ms vs baseline	Rollback inmediato
CPU usage	>90% sostenido	Rollback en 2 min
OOM kills	>2 en 5 minutos	Rollback inmediato
5xx errors	>10 en 1 minuto	Rollback inmediato

C) Database Point-in-Time Recovery (PITR)

Recuperación Granular de Datos:

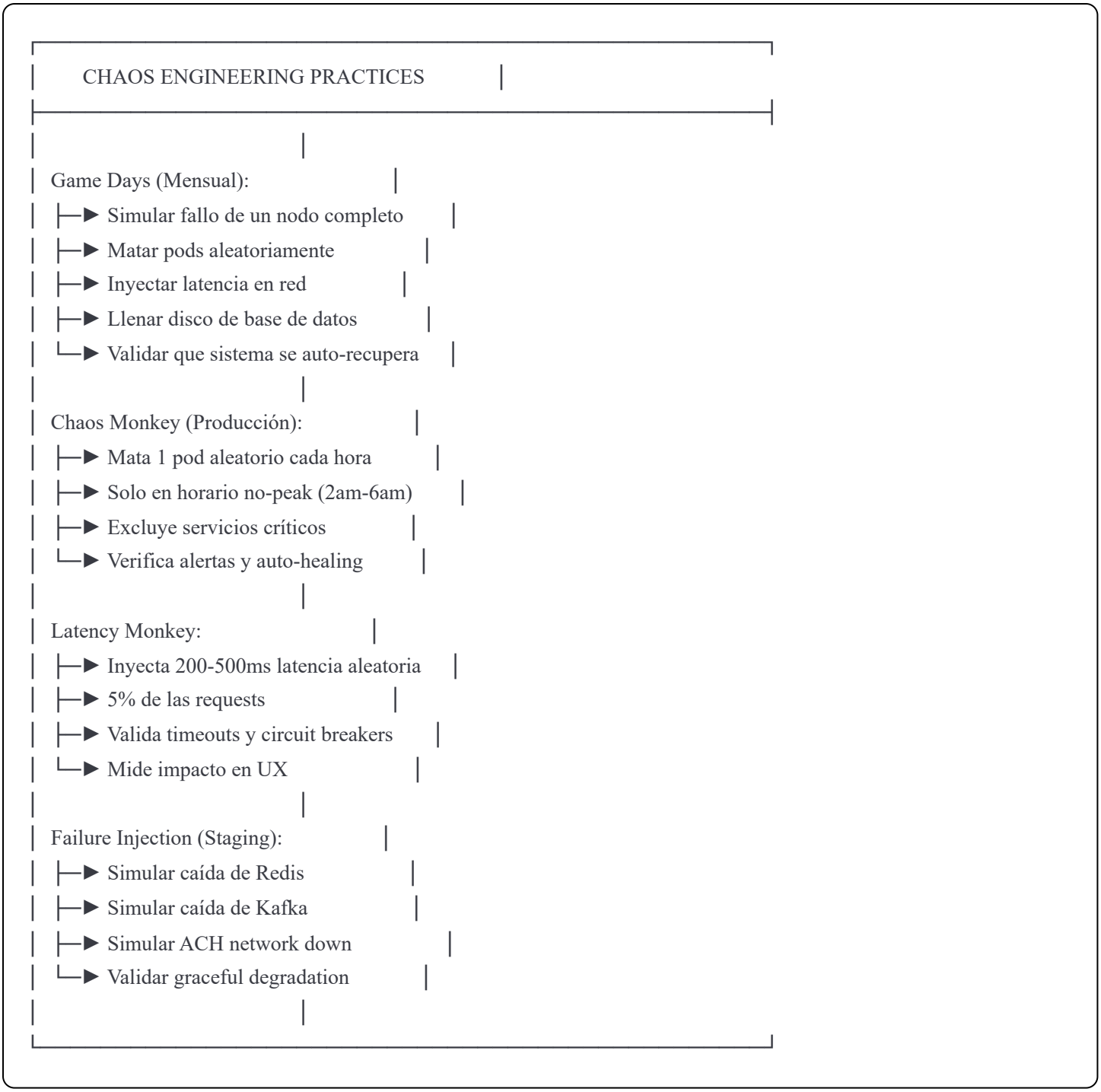


Escenarios de Uso:

1. **Corrupción de datos**: Volver a estado antes de corrupción
2. **Eliminación accidental**: Recuperar datos borrados por error
3. **Bug en producción**: Volver a estado pre-deploy
4. **Ataque malicioso**: Recuperar a punto antes del ataque

D) Chaos Engineering

Prácticas de Pruebas de Resiliencia:



Métricas de Éxito:

- Sistema mantiene >99% uptime durante chaos tests
- Usuarios no notan interrupciones
- Alertas se disparan correctamente
- Auto-healing funciona en <2 minutos
- No se pierden datos

E) Disaster Recovery Plan

Plan de Recuperación ante Desastres Mayores:

Escenario	RTO	RPO	Procedimiento
Zona AZ completa caída	<5 min	0	Failover automático a otra AZ (multi-AZ deployment)
Región completa caída	<15 min	<5 min	Failover manual a región secundaria
Corrupción de BD	<30 min	<5 min	PITR restore desde WAL logs
Ransomware	<1 hora	<1 hora	Restore desde backup offline (air-gapped)
Pérdida de datos completa	<4 horas	<1 día	Restore desde backup S3 Glacier

Procedimiento de DR:

```
1. DETECCIÓN
   ├─► Alertas automáticas (PagerDuty)
   ├─► Validar alcance del desastre
   └─► Activar equipo de respuesta

2. COMUNICACIÓN
   ├─► Notificar a stakeholders
   ├─► Status page público
   └─► Comunicación cada 15 minutos

3. MITIGACIÓN
   ├─► Aislar servicios afectados
   ├─► Activar modo degradado
   └─► Prevenir propagación

4. RECUPERACIÓN
   ├─► Ejecutar runbook de DR
   ├─► Validar integridad de datos
   └─► Restaurar servicios gradualmente

5. POST-MORTEM
   ├─► Análisis de causa raíz
   ├─► Identificar mejoras
   └─► Actualizar runbooks
```

11.5.5 Métricas de Resiliencia

Dashboard de Resiliencia en Grafana:

Métrica	Objetivo	Alertas
MTBF (Mean Time Between Failures)	>720 horas (30 días)	<480 horas
MTTR (Mean Time To Recover)	<15 minutos	>30 minutos
Uptime	>99.9%	<99.5%
Successful Requests	>99.9%	<99%
Circuit Breakers Open	0	>3 simultáneos
Auto-Healing Events	<10/día	>50/día
Rollback Rate	<5% de deploys	>10%
Data Loss	0 bytes	Cualquier pérdida

SLO (Service Level Objectives):

```
Availability SLO: 99.9%
├─► Error Budget: 43.8 minutos downtime/mes
├─► Si se consume >50% del budget → Code freeze
└─► Si se consume 100% → Post-mortem obligatorio

Latency SLO: p95 <200ms
├─► 95% de requests deben responder en <200ms
├─► Si p95 >300ms por >5 min → Alerta
└─► Si p95 >500ms → Incidente mayor

Error Rate SLO: <0.1%
├─► Máximo 1 error por cada 1000 requests
├─► Si >0.5% → Investigación inmediata
└─► Si >1% → Rollback automático
```

11.5.6 Resumen de Prácticas de Resiliencia

Checklist de Resiliencia Implementada:

✅ **Resistencia:**

- Circuit Breakers en todas las integraciones
- Bulkheads para aislamiento de recursos
- Retry con exponential backoff + jitter
- Timeouts agresivos en cascada
- Graceful degradation con fallbacks

✅ **Adaptación:**

- HPA para escalado horizontal automático

- VPA para optimización de recursos
- Queue-based load leveling
- Rate limiting adaptativo
- Priority queues para cargas críticas

✅ **Recuperación:**

- Self-healing con Kubernetes
- Automated rollback en deploys
- PITR para recuperación de datos
- Chaos engineering regular
- Disaster recovery plan probado

Resultado Esperado:

Un sistema bancario que **nunca falla completamente**, se **adapta a la carga**, y se **recupera automáticamente** de fallos, proporcionando una experiencia confiable y consistente a los usuarios incluso bajo condiciones adversas.

11.5.2 RESISTENCIA - Tolerancia a Fallos Parciales

A) Circuit Breaker Pattern

Implementación en Todos los Microservicios:
```javascript
// circuit-breaker.js (# PROPUESTA DE ARQUITECTURA
## Sistema Bancario Digital - Entidad BP

**Documento Técnico-Ejecutivo**
**Arquitectura de Soluciones Cloud-Native**
**Octubre 2025**

---

## TABLA DE CONTENIDOS



---

## 1. RESUMEN EJECUTIVO

La presente propuesta arquitectónica diseña un sistema bancario digital completo para BP, enfocado en **alta disponibilidad (HA), seguridad, escalabilidad y cumplimiento normativo**. La solución implementa una arquitectura de microservicios desacoplada, utilizando patrones modernos como CQRS, Event Sourcing y comunicación asíncrona mediante bus de mensajería.

### Características Principales:

- **Arquitectura de Microservicios**: Servicios independientes y especializados
- **Patrón CQRS**: Separación de lecturas y escrituras para optimizar rendimiento
```

- **OAuth 2.0 + Biometría**: Autenticación robusta y multifactor
- **Kubernetes**: Orquestación con autoescalado vertical y horizontal
- **Alta Disponibilidad**: 99.9% uptime con réplicas y balanceo de carga
- **Observabilidad Completa**: ELK Stack + Prometheus + Grafana
- **Cumplimiento Normativo**: ISO 27001, PCI DSS, GDPR, GLBA

---

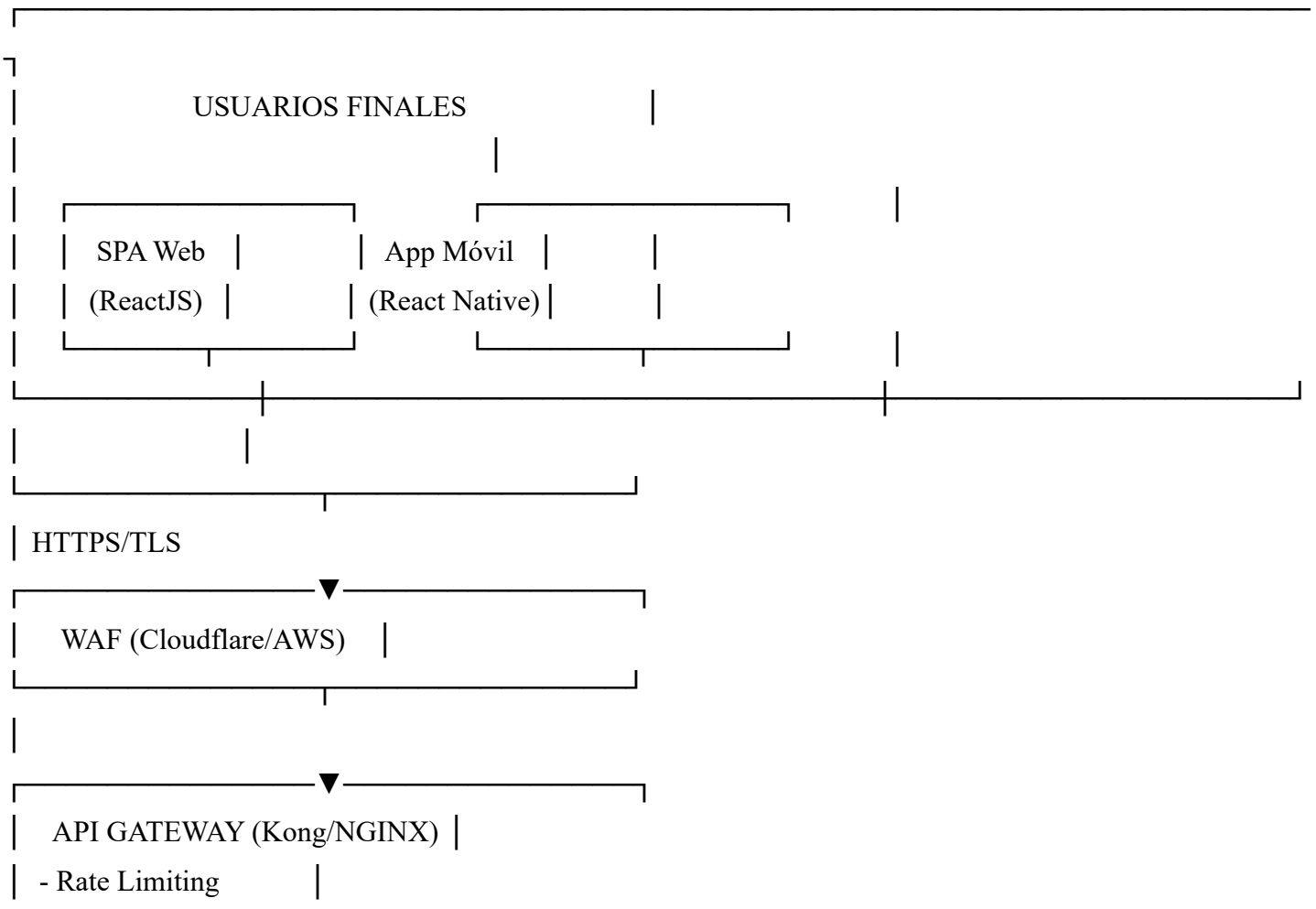
2. ARQUITECTURA GENERAL DEL SISTEMA

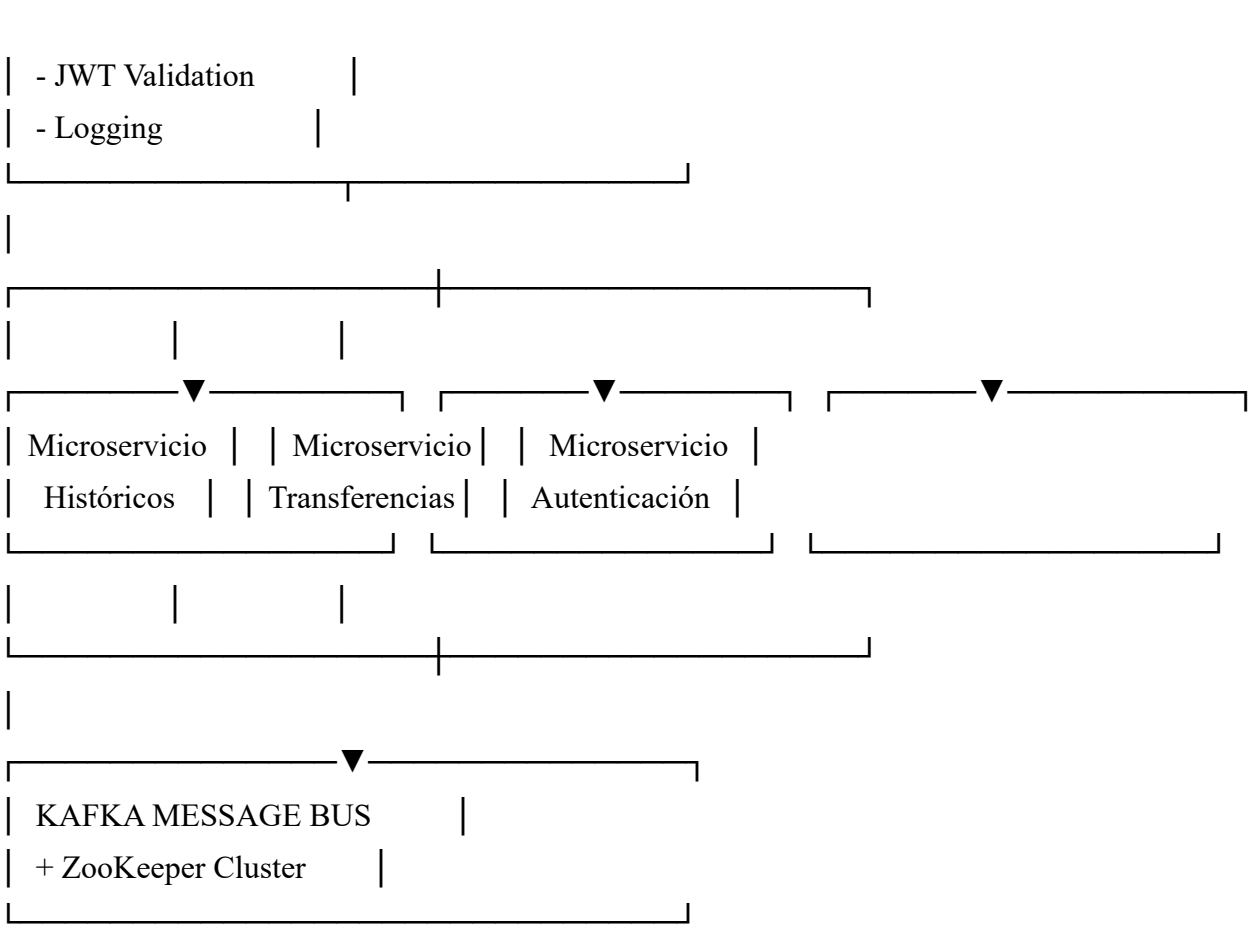
2.1 Vista de Contexto

El sistema bancario BP permite a los usuarios gestionar sus finanzas mediante aplicaciones web (SPA) y móvil, con las siguientes capacidades principales:

- Consulta de histórico de movimientos
- Transferencias intrabancarias
- Transferencias interbancarias
- Consulta de datos básicos del cliente
- Notificaciones en tiempo real
- Autenticación biométrica

2.2 Diagrama de Contexto




- JWT Validation
- Logging
Microservicio Históricos
Microservicio Transferencias
Microservicio Autenticación
KAFKA MESSAGE BUS
+ ZooKeeper Cluster

Justificación de Decisiones:

1. **WAF (Web Application Firewall)**: Protección contra ataques OWASP Top 10, DDoS y bot maliciosos antes de llegar a la aplicación.

2. **API Gateway Centralizado**:
   - Punto único de entrada para todas las peticiones
   - Simplifica la gestión de seguridad, logging y rate limiting
   - Reduce latencia mediante caching
   - Facilita versionado de APIs

3. **Arquitectura de Microservicios**:
   - Escalabilidad independiente por servicio
   - Despliegues independientes sin afectar todo el sistema
   - Facilita la incorporación de nuevas funcionalidades
   - Mejora el aislamiento de fallos

---

3. ARQUITECTURA DE MICROSERVICIOS

3.1 Microservicios Principales

Catálogo de Microservicios:

Microservicio	Responsabilidad	Tecnología Sugerida	Puerto	API
MS-Históricos	Consulta de movimientos históricos (CQRS Read)	Node.js + GraphQL	3001	GraphQL
MS-Pagos-Internos	Transferencias dentro del mismo banco	Node.js + GraphQL	3002	GraphQL
MS-Pagos-Interbancarios	Transferencias a otros bancos	Node.js + GraphQL	3003	GraphQL
MS-Datos-Cliente	Consulta de información básica del cliente	Node.js + GraphQL	3004	GraphQL
MS-Autenticación	OAuth 2.0 + Biometría + JWT	Node.js + REST	3005	REST
MS-Auditoría	Registro de eventos y compliance	Node.js + GraphQL	3006	GraphQL
MS-Notificaciones	Envío de emails, SMS, push	Node.js + GraphQL	3007	GraphQL
MS-Archivos	Upload/download de documentos	Node.js + REST	3008	REST
API-Gateway	Enrutamiento, auth, rate limiting	Kong + GraphQL Mesh	8080	GraphQL

3.2 Justificación de Node.js con Worker Threads

¿Por qué Node.js para los microservicios?

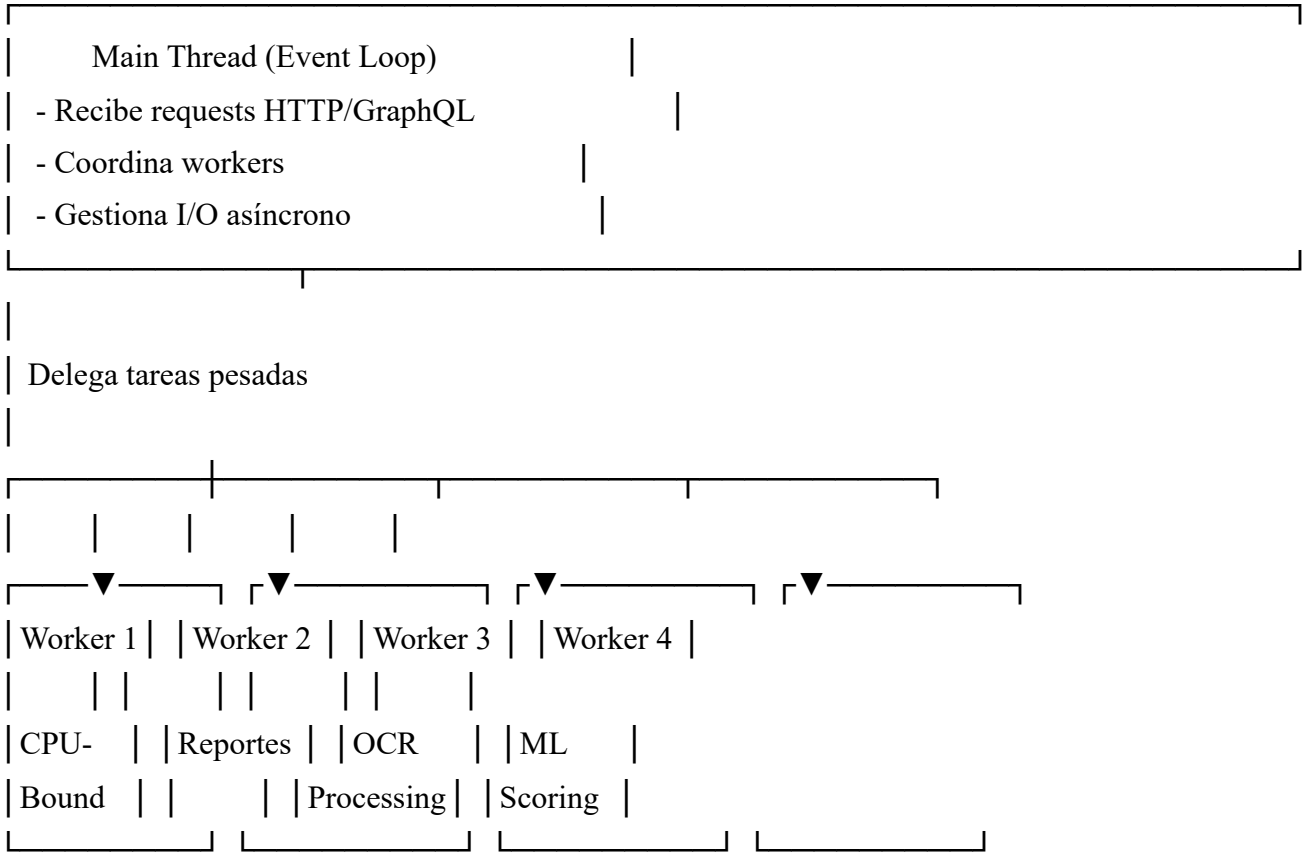
Node.js es la opción óptima para esta arquitectura por las siguientes razones técnicas:

1. Multithreading con Worker Threads:

Node.js desde la versión 12+ incluye Worker Threads, permitiendo verdadero procesamiento paralelo en múltiples cores de CPU. Esto es crítico para:

- **Procesamiento de transacciones masivas**: Validación paralela de múltiples transferencias
- **Generación de reportes**: Agregaciones complejas sin bloquear el event loop
- **Procesamiento de archivos**: OCR de documentos en onboarding
- **Cálculos intensivos**: Scoring de riesgo crediticio

Arquitectura con Worker Threads:



Código de ejemplo conceptual:

```javascript
// worker-pool.js
const { Worker } = require('worker_threads');

class WorkerPool {
  constructor(workerScript, poolSize = 4) {
    this.workers = [];
    this.queue = [];

    for (let i = 0; i < poolSize; i++) {
      const worker = new Worker(workerScript);
      this.workers.push({ worker, busy: false });
    }
  }

  async execute(data) {
    return new Promise((resolve, reject) => {
      const availableWorker = this.workers.find(w => !w.busy);

      if (availableWorker) {
        this.runTask(availableWorker, data, resolve, reject);
      } else {
        this.queue.push({ data, resolve, reject });
      }
    });
  }
}
```

2. Escalabilidad Horizontal Natural:

- Cada pod de Kubernetes ejecuta una instancia Node.js
- Cada instancia utiliza Worker Threads para paralelismo local
- Kubernetes escala pods horizontalmente según carga
- **Resultado**: Escalabilidad en dos dimensiones (vertical con workers + horizontal con pods)

3. Event Loop No Bloqueante:

Ideal para operaciones I/O intensivas típicas de sistemas bancarios:

- Consultas a bases de datos
- Llamadas a APIs externas (ACH, verificación biométrica)
- Operaciones de caché (Redis)

- Mensajería con Kafka

4. Ecosistema y Rendimiento:

- V8 Engine (altamente optimizado)
- NPM con 2M+ paquetes
- Librerías maduras para GraphQL (Apollo Server, TypeGraphQL)
- Excelente soporte para streams (ideal para archivos grandes)
- Menor consumo de memoria vs JVM o .NET

3.3 GraphQL como Estándar de APIs

Decisión Arquitectónica: GraphQL First

Se adopta GraphQL como protocolo principal para comunicación entre frontend y backend, con excepciones puntuales para REST.

3.3.1 Justificación de GraphQL

Ventajas sobre REST para este proyecto:

1. Eficiencia en Transferencia de Datos:

```graphql
# Cliente solicita exactamente lo que necesita
query {
  customer(id: "12345") {
    name
    accounts {
      balance
      lastTransaction {
        amount
        date
      }
    }
  }
}

# REST requeriría múltiples llamadas:
# GET /customers/12345
# GET /customers/12345/accounts
# GET /accounts/xxx/transactions?limit=1
```

Beneficio: 3 requests REST → 1 request GraphQL = **Reducción de latencia 60-70%**

2. Optimización de Bandwidth:

- Cliente móvil en red 3G solicita solo campos críticos
- Cliente web en WiFi solicita datos completos con detalles
- **Sin over-fetching**: No se envían datos innecesarios
- **Sin under-fetching**: Una sola query obtiene toda la información relacionada

3. Evolución de API sin Versionado:

```graphql
type Customer {
  name: String!
  email: String!
  phone: String! @deprecated(reason: "Use phoneNumber instead")
  phoneNumber: PhoneNumber  # Nuevo campo, no rompe clientes antiguos
}
```

- Agregar campos no rompe clientes existentes
- Deprecación gradual de campos
- Sin necesidad de /v1, /v2, /v3

4. Strongly Typed Schema:

```graphql
type Transfer {
  id: ID!
  amount: Money!
  status: TransferStatus!
  sourceAccount: Account!
  destinationAccount: Account!
  createdAt: DateTime!
}

enum TransferStatus {
  PENDING
  PROCESSING
  COMPLETED
  FAILED
  REVERSED
}
```

- Validación automática en compilación
- Autodocumentación (GraphQL Playground)

- Type safety end-to-end con TypeScript

5. Real-time con Subscriptions:

```graphql
subscription {
  transactionCreated(accountId: "12345") {
    id
    amount
    status
  }
}
```

- Notificaciones en tiempo real vía WebSocket
- Ideal para actualizaciones de saldo, estado de transferencias

6. Agregación Inteligente:

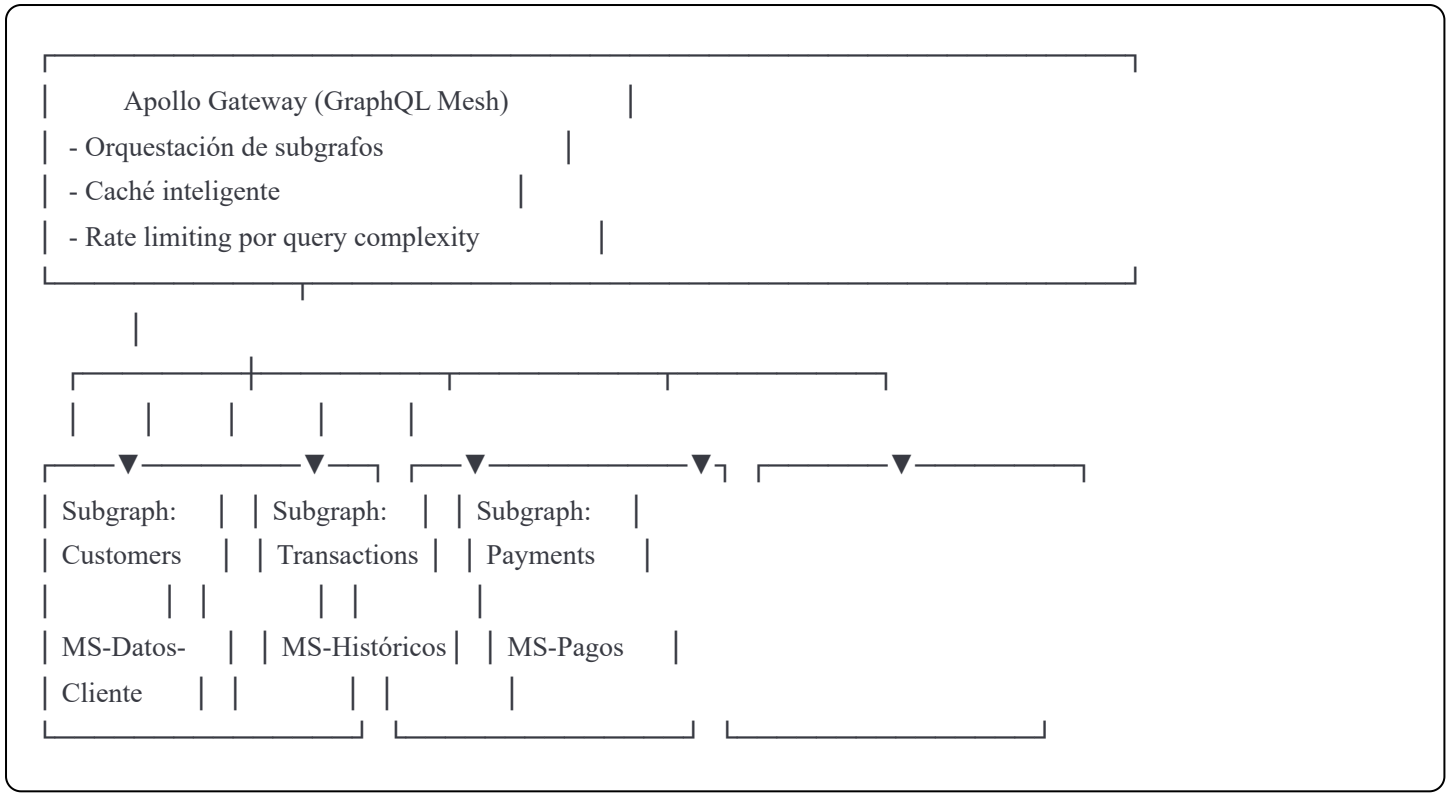
GraphQL permite al API Gateway agregar datos de múltiples microservicios:

```graphql
query {
  customer(id: "12345") {          # MS-Datos-Cliente
    name
    accounts {                     # MS-Datos-Cliente
      balance
      transactions(limit: 5) {     # MS-Históricos
        amount
        merchant
      }
    }
    pendingTransfers {             # MS-Pagos
      amount
      status
    }
  }
}
```

Un solo request del cliente → Gateway orquesta llamadas a 3 microservicios → Respuesta unificada

3.3.2 GraphQL Federation (Schema Stitching)

Para evitar que el API Gateway sea un monolito, se implementa **Apollo Federation**:



Cada microservicio expone su propio subgrafo, y el Gateway los compone en un schema unificado.

3.3.3 Cuándo Usar REST en Lugar de GraphQL

Casos específicos para REST:

1. **Upload/Download de archivos grandes**:
   - `POST /api/v1/documents/upload` (multipart/form-data)
   - `GET /api/v1/documents/:id/download` (binary stream)
   - Razón: GraphQL no es eficiente para transferencia de binarios
2. **Webhooks de servicios externos**:
   - `POST /webhooks/ach-callback` (ACH network)
   - `POST /webhooks/firebase-dlr` (Firebase delivery reports)
   - Razón: Servicios externos esperan REST
3. **Health checks simples**:
   - `GET /health` → `{ "status": "ok" }`
   - `GET /ready` → `{ "ready": true }`
   - Razón: Simplicidad, usado por Kubernetes

Proporción esperada: 95% GraphQL, 5% REST

3.3.4 Beneficios de Performance con GraphQL

Benchmarks típicos:

Métrica	REST	GraphQL	Mejora
Requests para dashboard completo	8-12	1-2	80-85%
Bandwidth consumido (mobile)	150 KB	45 KB	70%
Latencia total (3G)	3.2s	1.1s	66%
Time-to-interactive	4.5s	1.8s	60%

Caché con GraphQL:

```javascript
// Apollo Client (frontend) cachea automáticamente
const { data } = useQuery(GET_CUSTOMER, {
  variables: { id: "12345" },
  fetchPolicy: "cache-first"  // Usa caché si existe
});

// Invalidación selectiva
cache.evict({ id: 'Customer:12345', fieldName: 'balance' });
```

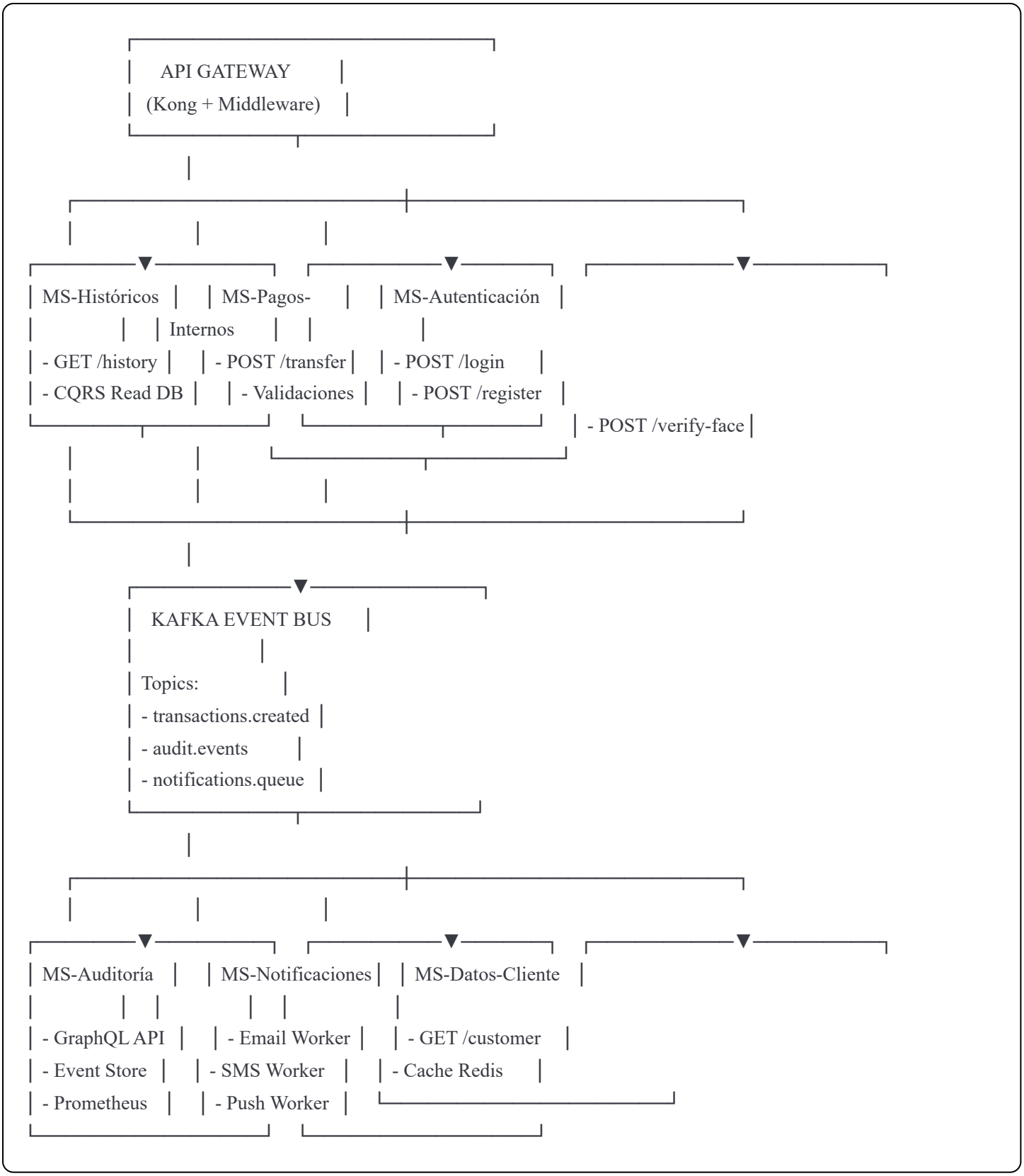
3.3.5 Seguridad en GraphQL

Protecciones implementadas:

1. **Query Complexity Analysis**: Rechazar queries con costo computacional excesivo
2. **Depth Limiting**: Máximo 5 niveles de profundidad
3. **Rate Limiting**: Por usuario y por query type
4. **Persisted Queries**: Solo queries pre-aprobadas en producción
5. **Field-level Authorization**: RBAC granular

```javascript
// Ejemplo de protección
const depthLimit = require('graphql-depth-limit');

const server = new ApolloServer({
  validationRules: [
    depthLimit(5),  // Máximo 5 niveles
    createComplexityLimitRule(1000)  // Máximo 1000 de complejidad
  ]
});
```



3.3 Justificación Técnica de Microservicios

¿Por qué Node.js para los microservicios?

- Alta concurrencia con event loop no bloqueante
- Ecosistema maduro para APIs REST y GraphQL
- Excelente rendimiento en I/O intensivo (consultas a BD, APIs externas)
- Menor curva de aprendizaje para el equipo (JavaScript full-stack)

- Mejor tiempo de desarrollo (menor time-to-market)

¿Por qué Kafka como Bus de Mensajería?

- Alta throughput (millones de mensajes/segundo)
- Durabilidad de mensajes (persistencia en disco)
- Escalabilidad horizontal mediante particiones
- Ideal para Event Sourcing y arquitecturas reactivas
- ZooKeeper garantiza consistencia del cluster

---

4. FUENTES DE DATOS Y PERSISTENCIA (PATRÓN CQRS)

4.1 Estrategia de Bases de Datos

El sistema utiliza **dos fuentes de datos principales** con estrategia CQRS (Command Query Responsibility Segregation):

Base de Datos Core (Escritura):

- **Propósito**: Operaciones transaccionales (CREATE, UPDATE, DELETE)
- **Tecnología Recomendada**: PostgreSQL
- **Contenido**: Movimientos financieros, productos, transacciones

Base de Datos de Lectura (Read Model):

- **Propósito**: Consultas optimizadas (SELECT)
- **Tecnología Recomendada**: MongoDB / PostgreSQL réplica
- **Contenido**: Vistas materializadas, históricos denormalizados

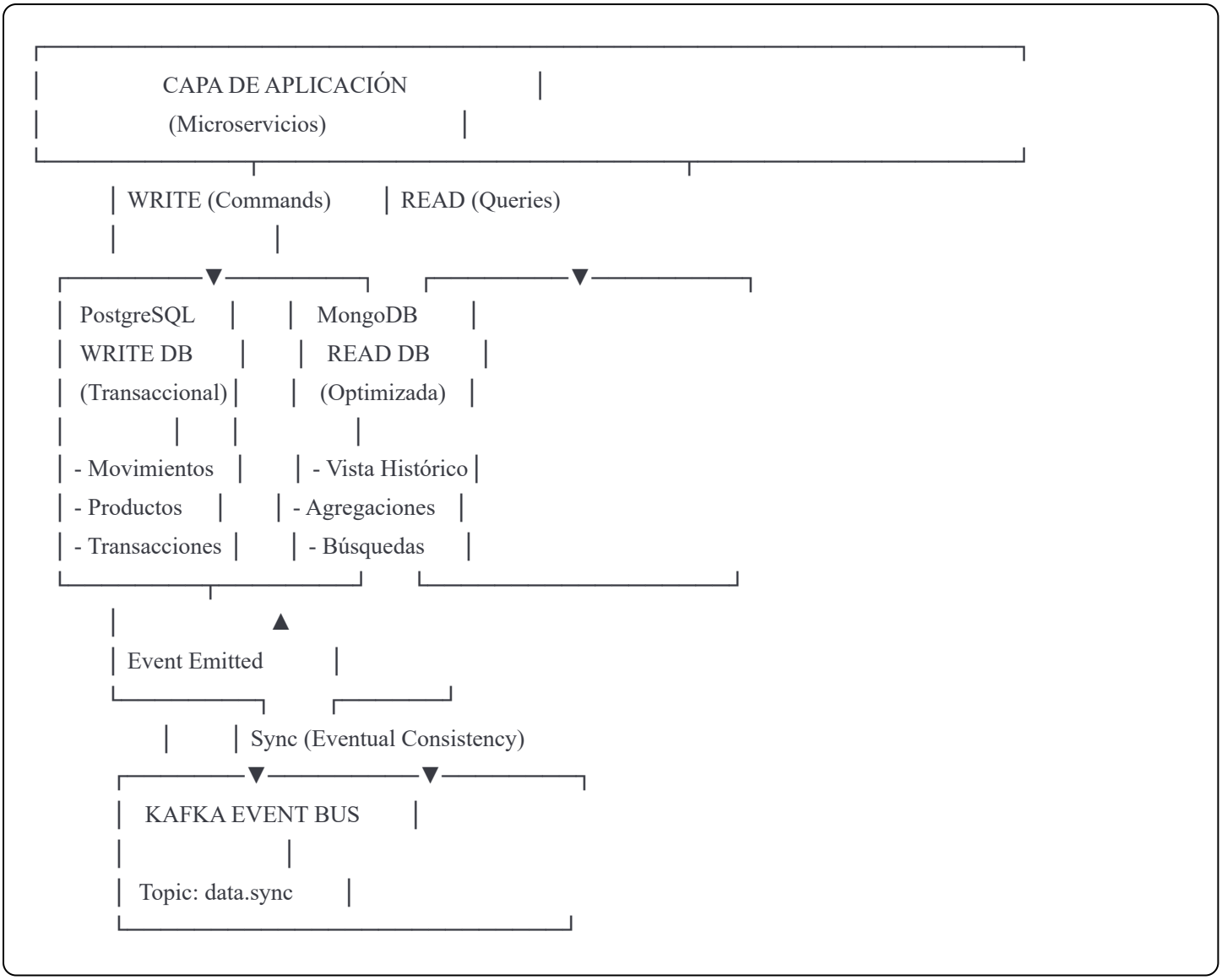
Base de Datos de Detalles de Cliente:

- **Propósito**: Información complementaria del cliente
- **Tecnología Recomendada**: PostgreSQL
- **Contenido**: Datos personales, preferencias, configuraciones

Base de Datos de Auditoría:

- **Propósito**: Compliance, logs, trazabilidad
- **Tecnología Recomendada**: Elasticsearch
- **Contenido**: Todos los eventos del sistema

4.2 Diagrama de Arquitectura de Datos (CQRS)



4.3 Consistencia Eventual

Justificación:

La consistencia eventual permite:

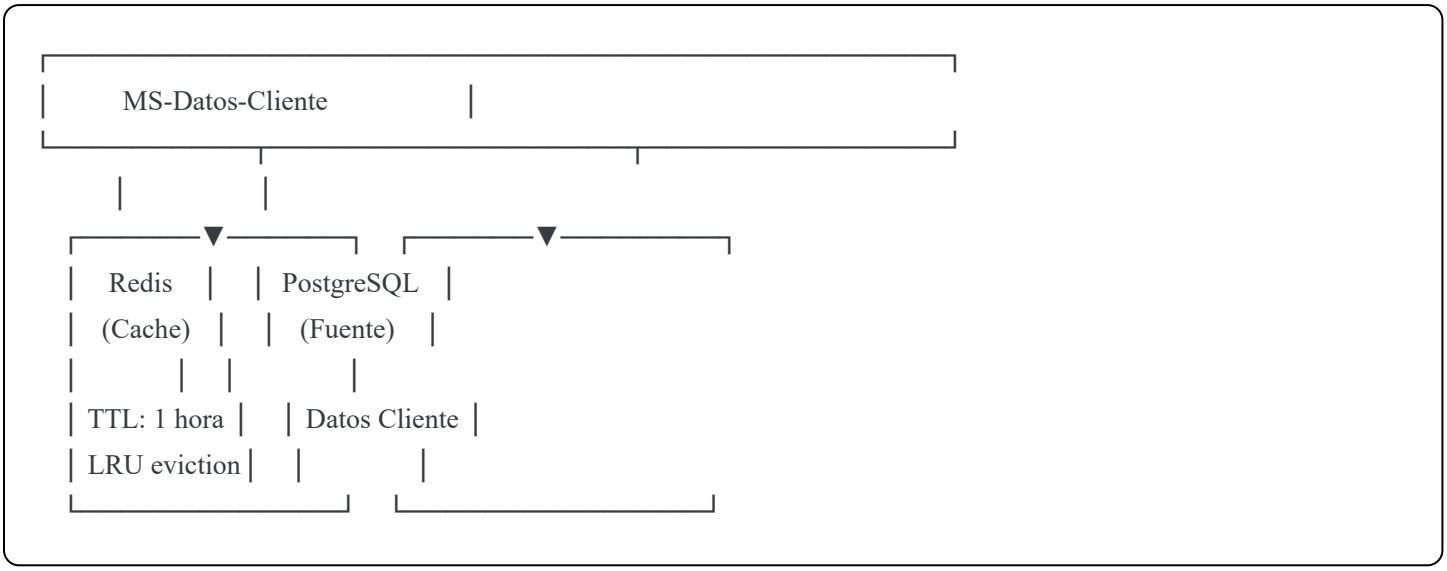
- Mayor disponibilidad del sistema (Teorema CAP)
- Escalabilidad horizontal sin bloqueos
- Mejor performance en lecturas (sin joins complejos)
- Resiliencia ante fallos de red entre servicios

Implementación:

1. Las escrituras se realizan en PostgreSQL (ACID completo)
2. Se emite un evento a Kafka tras cada escritura exitosa
3. Un consumidor escucha y actualiza la base de lectura
4. Latencia típica de sincronización: 100-500ms

4.4 Persistencia para Clientes Frecuentes

Estrategia de Caché con Redis:



Estrategia:

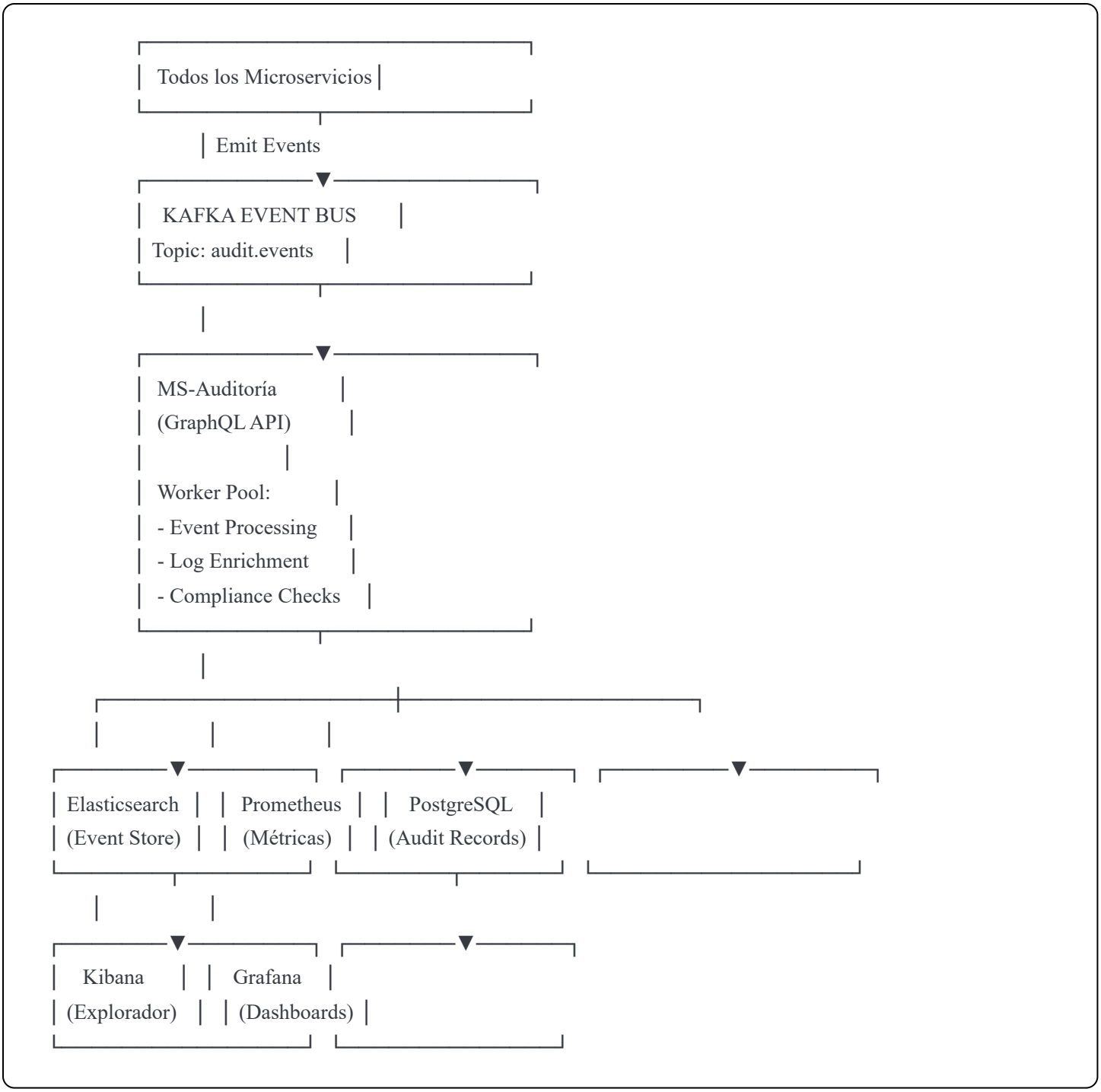
- Caché de datos de clientes frecuentes (Top 20% genera 80% del tráfico)
- TTL de 1 hora para balance entre frescura y performance
- Invalidación proactiva cuando se actualiza el cliente
- Reduce latencia de 50ms a 2ms en promedio

---

5. SISTEMA DE AUDITORÍA Y OBSERVABILIDAD

5.1 Arquitectura de Auditoría con GraphQL

El sistema implementa un microservicio dedicado para cumplimiento normativo y trazabilidad completa, utilizando **GraphQL** como interfaz principal.



5.2 GraphQL API de Auditoría

Justificación de GraphQL para Auditoría:

1. **Consultas Complejas Flexibles**: Los auditores necesitan filtros muy específicos (por usuario, fecha, tipo de operación, monto, etc.)
2. **Agregaciones Eficientes**: Obtener estadísticas sin múltiples queries
3. **Subscriptions en Tiempo Real**: Monitoreo de eventos críticos en vivo
4. **Type Safety**: Schema fuertemente tipado para compliance
5. **Autodocumentación**: GraphQL Playground como documentación interactiva

Schema GraphQL de Auditoría:

graphql

```
type AuditEvent {
  id: ID!
  timestamp: DateTime!
  userId: String!
  action: AuditAction!
  resource: String!
  resourceId: String
  metadata: JSON!
  ipAddress: String!
  userAgent: String
  result: AuditResult!
  severity: AuditSeverity!
}

enum AuditAction {
  LOGIN
  LOGOUT
  TRANSFER_CREATED
  TRANSFER_APPROVED
  TRANSFER_REJECTED
  ACCOUNT_ACCESSED
  PASSWORD_CHANGED
  PROFILE_UPDATED
  DOCUMENT_UPLOADED
  ADMIN_ACTION
}

enum AuditResult {
  SUCCESS
  FAILURE
  PARTIAL
}

enum AuditSeverity {
  INFO
  WARNING
  CRITICAL
}

type AuditStatistics {
  totalEvents: Int!
  successRate: Float!
  failuresByAction: [ActionCount!]!
  eventsByHour: [HourlyCount!]!
  topUsers: [UserActivity!]!
}
```

```graphql
type Query {
  # Búsqueda flexible de eventos
  auditEvents(
    startDate: DateTime!
    endDate: DateTime!
    userId: String
    actions: [AuditAction!]
    severity: AuditSeverity
    limit: Int = 100
    offset: Int = 0
  ): [AuditEvent!]!

  # Estadísticas agregadas
  auditStatistics(
    startDate: DateTime!
    endDate: DateTime!
    groupBy: StatGrouping
  ): AuditStatistics!

  # Búsqueda por texto completo
  searchAuditLogs(
    query: String!
    limit: Int = 50
  ): [AuditEvent!]!
}

type Subscription {
  # Eventos críticos en tiempo real
  criticalEvents: AuditEvent!

  # Eventos por usuario específico
  userEvents(userId: String!): AuditEvent!
}
```

Ejemplo de Query Compleja:

```graphql
```

```graphql
# Dashboard de compliance para último mes
query ComplianceDashboard {
  auditStatistics(
    startDate: "2025-09-01T00:00:00Z"
    endDate: "2025-10-01T00:00:00Z"
    groupBy: ACTION
  ) {
    totalEvents
    successRate
    failuresByAction {
      action
      count
      failureRate
    }
    eventsByHour {
      hour
      count
      suspiciousActivity
    }
    topUsers {
      userId
      eventCount
      riskScore
    }
  }

  # Eventos críticos del último mes
  criticalTransfers: auditEvents(
    startDate: "2025-09-01T00:00:00Z"
    endDate: "2025-10-01T00:00:00Z"
    actions: [TRANSFER_CREATED]
    severity: CRITICAL
  ) {
    id
    timestamp
    userId
    metadata
    result
  }
}
```

Beneficio vs REST:

- **REST requeriría**: 5-6 endpoints diferentes
- **GraphQL requiere**: 1 query única

- **Reducción de latencia**: 70-80% menos tiempo total

5.3 Procesamiento de Eventos con Worker Threads

Arquitectura de Worker Pool para Auditoría:

javascript

```
// audit-service.js (Conceptual)
const { Worker } = require('worker_threads');

class AuditEventProcessor {
  constructor() {
    this.workerPool = this.initializeWorkers(4);
  }

  initializeWorkers(poolSize) {
    const workers = [];
    for (let i = 0; i < poolSize; i++) {
      workers.push({
        worker: new Worker('./workers/audit-worker.js'),
        busy: false,
        processed: 0
      });
    }
    return workers;
  }

  async processEvent(event) {
    const worker = this.getAvailableWorker();

    worker.busy = true;
    worker.worker.postMessage({
      type: 'PROCESS_AUDIT_EVENT',
      data: event
    });

    return new Promise((resolve) => {
      worker.worker.once('message', (result) => {
        worker.busy = false;
        worker.processed++;
        resolve(result);
      });
    });
  }

  getAvailableWorker() {
    // Round-robin con preferencia por menos cargados
    return this.workerPool
      .sort((a, b) => a.processed - b.processed)
      .find(w => !w.busy) || this.workerPool[0];
  }
}
```

```
// Kafka Consumer con Workers
kafka.consumer.on('message', async (message) => {
  const event = JSON.parse(message.value);

  // Procesar en worker thread (no bloquea event loop)
  const enrichedEvent = await auditProcessor.processEvent(event);

  // Almacenar en múltiples destinos
  await Promise.all([
    elasticsearch.index(enrichedEvent),
    postgres.insert(enrichedEvent),
    prometheus.record(enrichedEvent)
  ]);
});
```

Procesamiento en Worker:

```
javascript
```

```
// workers/audit-worker.js (Conceptual)
const { parentPort } = require('worker_threads');

parentPort.on('message', async (msg) => {
  if (msg.type === 'PROCESS_AUDIT_EVENT') {
    const event = msg.data;

    // Enriquecimiento del evento (CPU-intensive)
    const enriched = {
      ...event,
      geoLocation: await lookupGeoIP(event.ipAddress),
      userRiskScore: calculateRiskScore(event),
      complianceFlags: checkComplianceRules(event),
      anomalyScore: detectAnomalies(event)
    };

    parentPort.postMessage(enriched);
  }
});

function calculateRiskScore(event) {
  // Cálculo complejo en worker (no bloquea main thread)
  let score = 0;

  if (event.action === 'TRANSFER_CREATED') {
    score += event.metadata.amount > 10000 ? 50 : 10;
  }

  if (isUnusualTime(event.timestamp)) {
    score += 20;
  }

  if (isUnusualLocation(event.ipAddress)) {
    score += 30;
  }

  return score;
}
```

Performance:

Métrica	Sin Workers	Con 4 Workers	Mejora
Eventos procesados/segundo	800	3,200	300%
Latencia promedio	45ms	12ms	73%
CPU utilizada	25% (1 core)	85% (4 cores)	Óptimo

5.4 GraphQL como Procesador de Eventos

Ventajas de GraphQL sobre REST para Auditoría:

1. Consultas Ad-Hoc sin Nuevos Endpoints:

```graphql
# Auditor necesita query específica (sin código backend nuevo)
query SuspiciousTransfers {
  auditEvents(
    startDate: "2025-10-01T00:00:00Z"
    endDate: "2025-10-06T00:00:00Z"
    actions: [TRANSFER_CREATED]
  ) {
    id
    userId
    timestamp
    metadata
    result
    # Solo traer campos necesarios
    # Sin over-fetching
  }
}
```

2. Agregaciones Complejas en Una Query:

```graphql
```

```
query ComplianceReport {
  last30Days: auditStatistics(
    startDate: "2025-09-06T00:00:00Z"
    endDate: "2025-10-06T00:00:00Z"
  ) {
    totalEvents
    successRate
  }

  last7Days: auditStatistics(
    startDate: "2025-09-29T00:00:00Z"
    endDate: "2025-10-06T00:00:00Z"
  ) {
    totalEvents
    successRate
  }

  # Comparación automática en una sola query
}
```

3. Subscriptions para Monitoreo en Tiempo Real:

graphql

```
subscription WatchCriticalEvents {
  criticalEvents {
    id
    timestamp
    userId
    action
    severity
    metadata
  }
}
```

Implementación del Subscription:

javascript

```
// GraphQL Resolver (Conceptual)
const { PubSub } = require('graphql-subscriptions');
const pubsub = new PubSub();

// Kafka consumer publica a GraphQL subscription
kafka.consumer.on('message', (message) => {
  const event = JSON.parse(message.value);

  if (event.severity === 'CRITICAL') {
    pubsub.publish('CRITICAL_EVENT', {
      criticalEvents: event
    });
  }
});

// Resolver
const resolvers = {
  Subscription: {
    criticalEvents: {
      subscribe: () => pubsub.asyncIterator(['CRITICAL_EVENT'])
    }
  }
};
```

Dashboard en Tiempo Real:

- Compliance officers ven eventos críticos instantáneamente
- WebSocket mantiene conexión abierta
- Latencia: <100ms desde evento hasta visualización
- Sin polling (más eficiente que REST)

5.5 Componentes del Sistema de Auditoría

A) GraphQL como Procesador de Eventos:

Ventajas específicas para auditoría:

- **Consultas flexibles**: Los reguladores piden reports con filtros impredecibles
- **Reducción de over-fetching**: Solo se transfieren campos necesarios (importante con millones de eventos)
- **Introspección del esquema**: Autodocumentación para auditores externos
- **Paginación eficiente**: Cursor-based pagination para grandes datasets
- **Batching**: DataLoader agrupa queries similares (N+1 problem resuelto)

B) Elasticsearch como Event Store:

- Búsqueda full-text en logs: `query: "transfer AND amount:>10000 AND user:suspicious"`
- Agregaciones complejas: histogramas, percentiles, top-K
- Escalabilidad horizontal mediante sharding
- Retención configurable por índices (ej: 7 años para compliance PCI DSS)

C) Prometheus + Grafana:

- Métricas en tiempo real (latencia, errores, throughput)
- Alertas automáticas (SLO/SLA monitoring)
- Dashboards personalizados por servicio
- Integración nativa con Kubernetes

D) Worker Threads para Performance:

- Procesamiento paralelo de eventos (enriquecimiento, scoring, ML)
- No bloquea event loop de Node.js
- Escalabilidad vertical (más cores = más throughput)
- Ideal para CPU-intensive tasks (análisis de riesgo, detección de fraude)

5.6 Política de Retención y Borrado

Tipo de Dato	Retención	Storage	Justificación Normativa
Logs de aplicación	90 días	Elasticsearch	Operación diaria
Eventos de auditoría	7 años	Elasticsearch + S3	PCI DSS, GLBA
Transacciones financieras	10 años	PostgreSQL + S3	Normativas bancarias locales
Datos de acceso (login)	2 años	Elasticsearch	ISO 27001, GDPR
Métricas de Prometheus	30 días	Prometheus TSDB	Análisis de tendencias
Eventos críticos	10 años	PostgreSQL (hot) + S3 (cold)	Compliance y legal

Estrategia de Lifecycle:

Día 0-30: Elasticsearch (hot tier)
├─► Búsquedas ultra-rápidas
├─► SSD storage
└─► Consultas frecuentes

Día 31-365: Elasticsearch (warm tier)
├─► Búsquedas rápidas
├─► HDD storage
└─► Consultas ocasionales

Año 1-7: S3 Standard
├─► Búsquedas por batch
├─► Costo medio
└─► Consultas raras (compliance)

Año 7+: S3 Glacier Deep Archive
├─► Recuperación en 12-48 horas
├─► Costo mínimo
└─► Solo para legal/auditorías externas

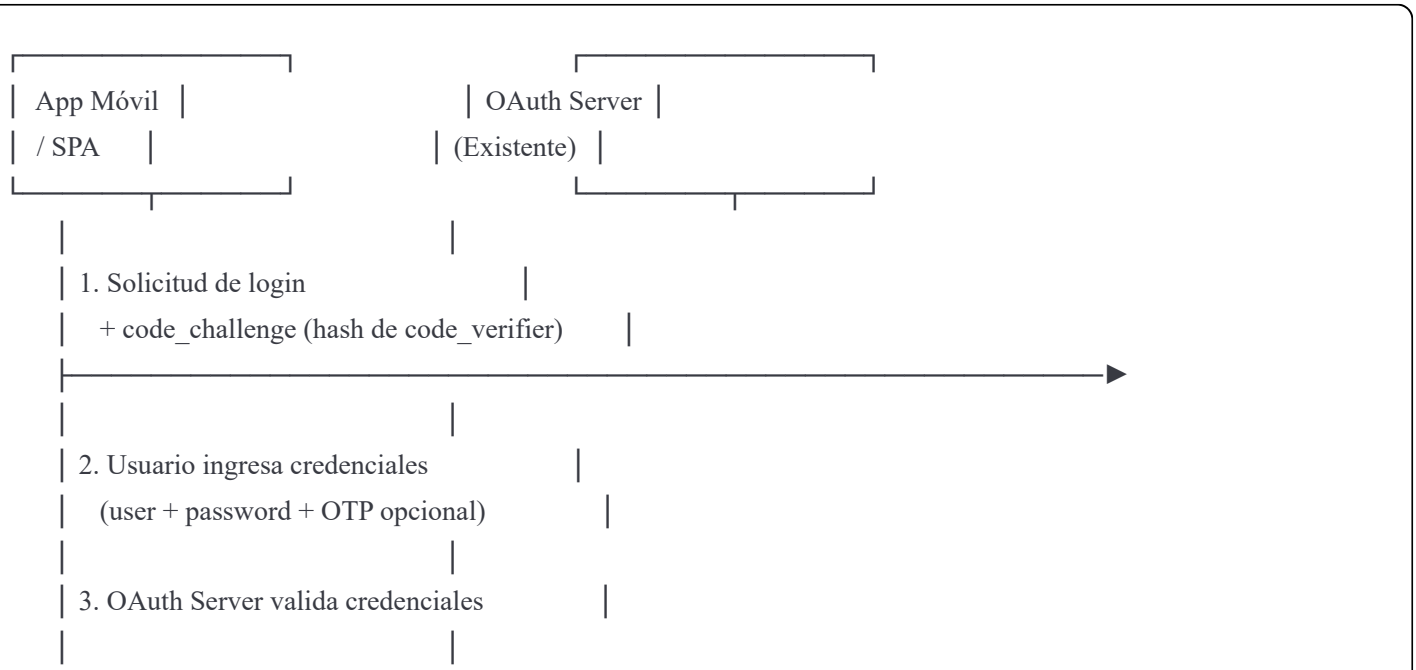
---

6. ARQUITECTURA DE AUTENTICACIÓN OAuth 2.0 Y BIOMETRÍA

6.1 Flujo de Autenticación Recomendado

La compañía ya cuenta con un servicio OAuth 2.0, por lo que se implementa el flujo **Authorization Code Flow with PKCE** (Proof Key for Code Exchange), recomendado para aplicaciones móviles y SPA por su mayor seguridad.

6.2 Diagrama de Flujo de Autenticación



6.3 Integración del Onboarding con Reconocimiento Facial

Durante el proceso de registro (onboarding), el sistema captura y valida la identidad del usuario mediante reconocimiento facial.

Flujo de Onboarding:

1. Registro Inicial
   ├─► Datos personales (Nombre, Email, Teléfono)
   ├─► Validación de documento de identidad
   └─► Creación de credenciales (user/password)

2. Verificación de Identidad
   ├─► Captura de foto del documento
   ├─► Captura de selfie del usuario
   └─► Validación de documento mediante OCR + API externa

3. Registro Biométrico
   ├─► Captura de video corto para liveness detection
   ├─► Extracción de vectores faciales (embeddings)
   ├─► Almacenamiento cifrado en Vault
   └─► Activación de Face ID / Touch ID en dispositivo

4. Activación de Cuenta
   ├─► Envío de SMS/Email de confirmación
   ├─► Primer login con credenciales
   └─► Sistema habilitado para autenticación biométrica

6.4 Métodos de Autenticación Post-Registro

Una vez completado el onboarding, el usuario puede autenticarse mediante:

Método	Tecnología	Seguridad	Experiencia
Usuario + Contraseña	Bcrypt hash + Salt	Media	Estándar
Usuario + Contraseña + OTP	TOTP (Google Authenticator)	Alta	Buena
Face ID / Face Recognition	Biometría facial + Liveness	Muy Alta	Excelente
Touch ID / Fingerprint	Sensor biométrico del dispositivo	Muy Alta	Excelente
PIN de 6 dígitos	Cifrado local	Media	Rápida

6.5 Recomendaciones Adicionales de Seguridad

A) Hardware Security Module (HSM):

- Almacenamiento seguro de claves criptográficas
- Firma de tokens JWT con claves rotativas
- Protección contra extracción de claves privadas

B) Vault Secret Manager (HashiCorp Vault):

- Gestión centralizada de secretos (API keys, DB passwords)
- Rotación automática de credenciales

- Audit log de acceso a secretos
- Integración con Kubernetes (secrets injection)

C) Sensores y APIs Biométricas:

- **Face ID (iOS)**: Secure Enclave del dispositivo
- **Touch ID / Face Recognition (Android)**: Android Keystore
- **Liveness Detection**: Detección de ataques con fotos o videos
- **Proveedor recomendado**: AWS Rekognition o Azure Face API para verificación inicial

D) Rate Limiting y Protección Anti-Fuerza Bruta:

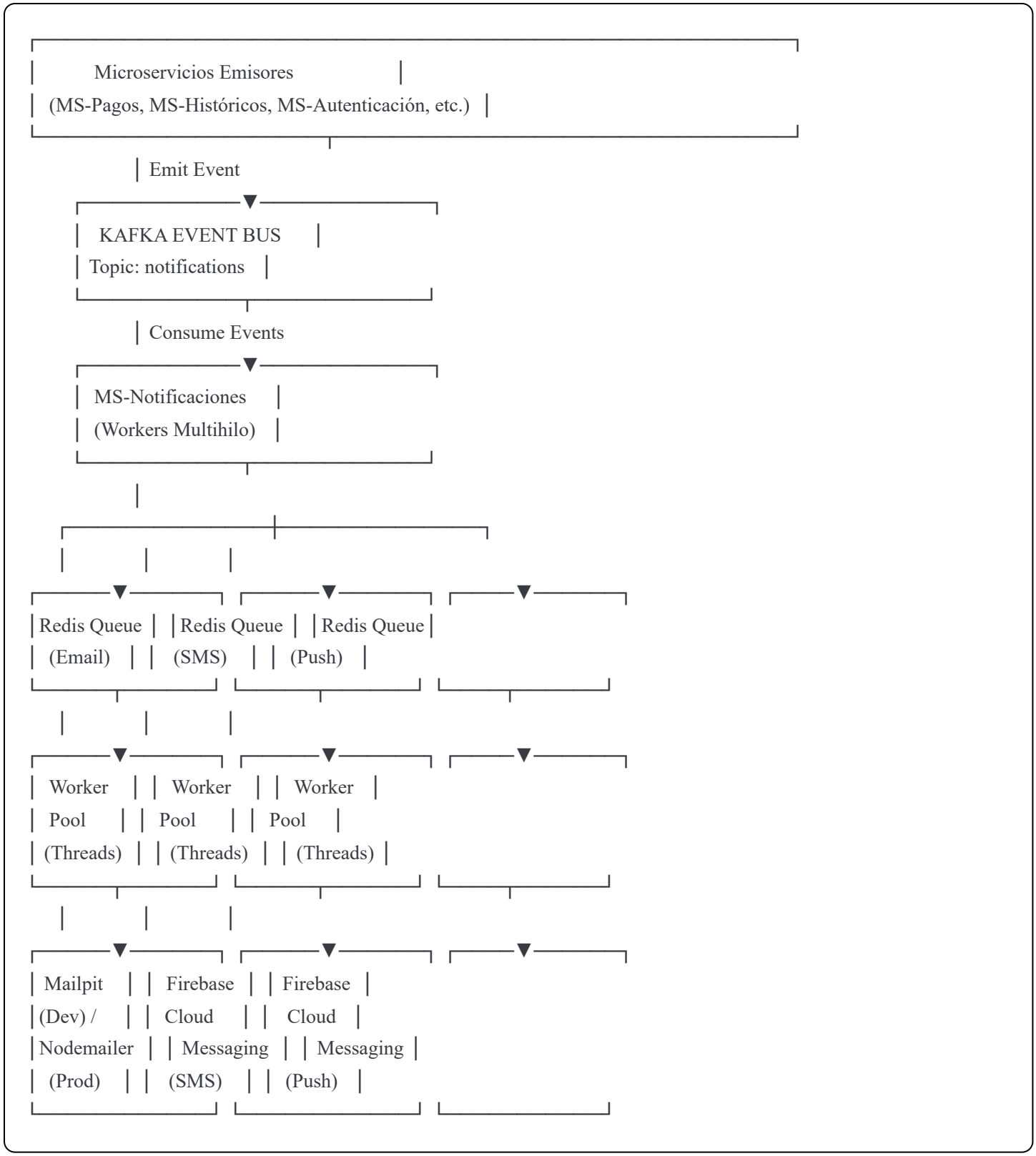
- Máximo 5 intentos de login en 15 minutos
- Bloqueo temporal de cuenta por 30 minutos tras 5 intentos fallidos
- CAPTCHA después de 3 intentos fallidos
- Notificación al usuario de intentos sospechosos

---

7. SISTEMA DE MENSAJERÍA Y NOTIFICACIONES

7.1 Arquitectura de Notificaciones Asíncronas

El sistema de notificaciones debe ser escalable, confiable y asíncrono para no afectar la latencia de las transacciones principales.



7.2 Justificación de Tecnologías

A) Redis + Bull Queue:

Ventajas:

- Procesamiento asíncrono con reintentos automáticos
- Priorización de colas (emails urgentes vs. informativos)
- Persistencia de jobs (no se pierden notificaciones en caso de caída)
- Dashboard web para monitoreo de colas

- Delay scheduling (envío diferido de notificaciones)

B) Workers con Multithreading:

- Node.js Worker Threads para procesamiento paralelo
- Cada tipo de notificación tiene su pool de workers dedicado
- Escalamiento horizontal mediante réplicas de pods en Kubernetes
- Procesamiento concurrente de miles de notificaciones/segundo

C) Firebase Cloud Messaging (SMS):

Ventajas:

- Económico (hasta 10,000 SMS gratis/mes)
- Alta tasa de entrega (99%+)
- API simple y bien documentada
- Integración directa con apps móviles
- Soporte para múltiples países

D) Estrategia de Email:

Desarrollo: Mailpit en contenedor Docker

- Interfaz web para visualizar emails sin enviarlos
- No requiere configuración SMTP real
- Ideal para testing

Producción: Nodemailer con SMTP propio

- No usar Amazon SES para evitar costos adicionales
- SMTP propio con dominio de la empresa
- Autenticación SPF + DKIM + DMARC para evitar spam
- Plantillas HTML con Handlebars o EJS

7.3 Tipos de Notificaciones

Evento	Email	SMS	Push	Prioridad
Transferencia realizada	✓	✓	✓	Alta
Login desde nuevo dispositivo	✓	✓	✓	Crítica
Saldo bajo	✓	-	✓	Media
Cambio de contraseña	✓	✓	-	Crítica
Recordatorio de pago	✓	-	✓	Baja

Evento	Email	SMS	Push	Prioridad
Promociones	✓	-	✓	Baja

7.4 Cumplimiento Normativo en Notificaciones

Según normativas bancarias:

- Mínimo 2 canales de notificación para transacciones
- Implementado: Email + SMS (o Push si está habilitado)
- El usuario puede configurar sus preferencias, pero no puede desactivar notificaciones de seguridad

---

8. ARQUITECTURA FRONTEND MULTIPLATAFORMA

8.1 Análisis Comparativo de Frameworks

Se solicita evaluar 3 frameworks multiplataforma: React Native, Flutter y Kotlin Multiplatform.

8.1.1 Tabla Comparativa Detallada

Criterio	React Native	Flutter	Kotlin Multiplatform
Madurez	10+ años (2015)	7 años (2018)	4 años (2020)
Comunidad	Muy Grande (56k stars)	Grande (168k stars)	Pequeña (10k stars)
Lenguaje	JavaScript/TypeScript	Dart	Kotlin
Curva de Aprendizaje	Baja (JS conocido)	Media (Dart nuevo)	Media-Alta
Performance	Buena (bridge nativo)	Excelente (compiled)	Excelente (nativo)
Paquetes disponibles	50,000+	30,000+	5,000+
Empresas que lo usan	Meta, Uber, Airbnb	Google, Alibaba	JetBrains, Netflix
Hot Reload	✓	✓	✓ (limitado)
Acceso a APIs nativas	Mediante bridge	Via Platform Channels	Directo (expect/actual)
Tamaño de app	25-30 MB	15-20 MB	10-15 MB
Tiempo de desarrollo	Rápido	Rápido	Medio
Costo de contratación	Bajo (devs JS)	Medio (devs Dart)	Alto (devs Kotlin)
UI/UX	Componentes nativos	Material + Cupertino	Totalmente nativo
Ideal para	Equipos JS, MVP rápido	Apps visuales, consistencia	Apps nativas complejas

8.2 Análisis FODA por Framework

REACT NATIVE

Fortalezas:

- Stack JavaScript compartido con SPA web (reutilización de código)
- Enorme ecosistema de paquetes y librerías
- Gran comunidad y abundante documentación
- Fácil contratación de desarrolladores
- Tiempo de desarrollo rápido (time-to-market)
- Integración nativa con módulos iOS/Android cuando se necesita

Debilidades:

- Performance inferior a Flutter/Kotlin en animaciones complejas
- Dependencia de bridge JavaScript-nativo (overhead)
- Actualizaciones pueden romper compatibilidad
- Tamaño de app más grande

Oportunidades:

- Nueva arquitectura (Fabric + TurboModules) mejora performance
- Expo facilita desarrollo y despliegues OTA
- React Native Web para convergencia total

Amenazas:

- Meta podría reducir inversión en el proyecto
- Flutter ganando cuota de mercado

FLUTTER

Fortalezas:

- Excelente performance (compilado a código nativo)
- UI consistente entre plataformas
- Hot reload extremadamente rápido
- Respaldado por Google
- Widget-based arquitectura muy flexible
- Ideal para apps con animaciones complejas

Debilidades:

- Requiere aprender Dart (menos común que JS)
- Contratar talento Dart es más costoso
- Tamaño de runtime incluido en la app

- Menor integración con ecosistema web

Oportunidades:

- Flutter Web y Desktop en crecimiento
- Google invierte fuertemente en el proyecto
- Crecimiento rápido de la comunidad

Amenazas:

- Menor adopción que React Native en fintech
- Dependencia total de Google

KOTLIN MULTIPLATFORM

Fortalezas:

- Performance nativa real (no bridge)
- Acceso completo a APIs nativas
- Tamaño de app más pequeño
- Código compartido solo para lógica de negocio
- UI totalmente nativa (mejor UX)

Debilidades:

- Tecnología muy joven (solo 4 años estable)
- Comunidad pequeña, pocos paquetes
- Requiere devs especializados en Kotlin (costoso)
- Curva de aprendizaje pronunciada
- Menor cantidad de recursos y tutoriales

Oportunidades:

- JetBrains invirtiendo fuertemente
- Adoptado por empresas grandes (Netflix, VMware)
- Futuro prometedor en enterprise

Amenazas:

- Puede no alcanzar masa crítica de adopción
- Riesgo de fragmentación del ecosistema

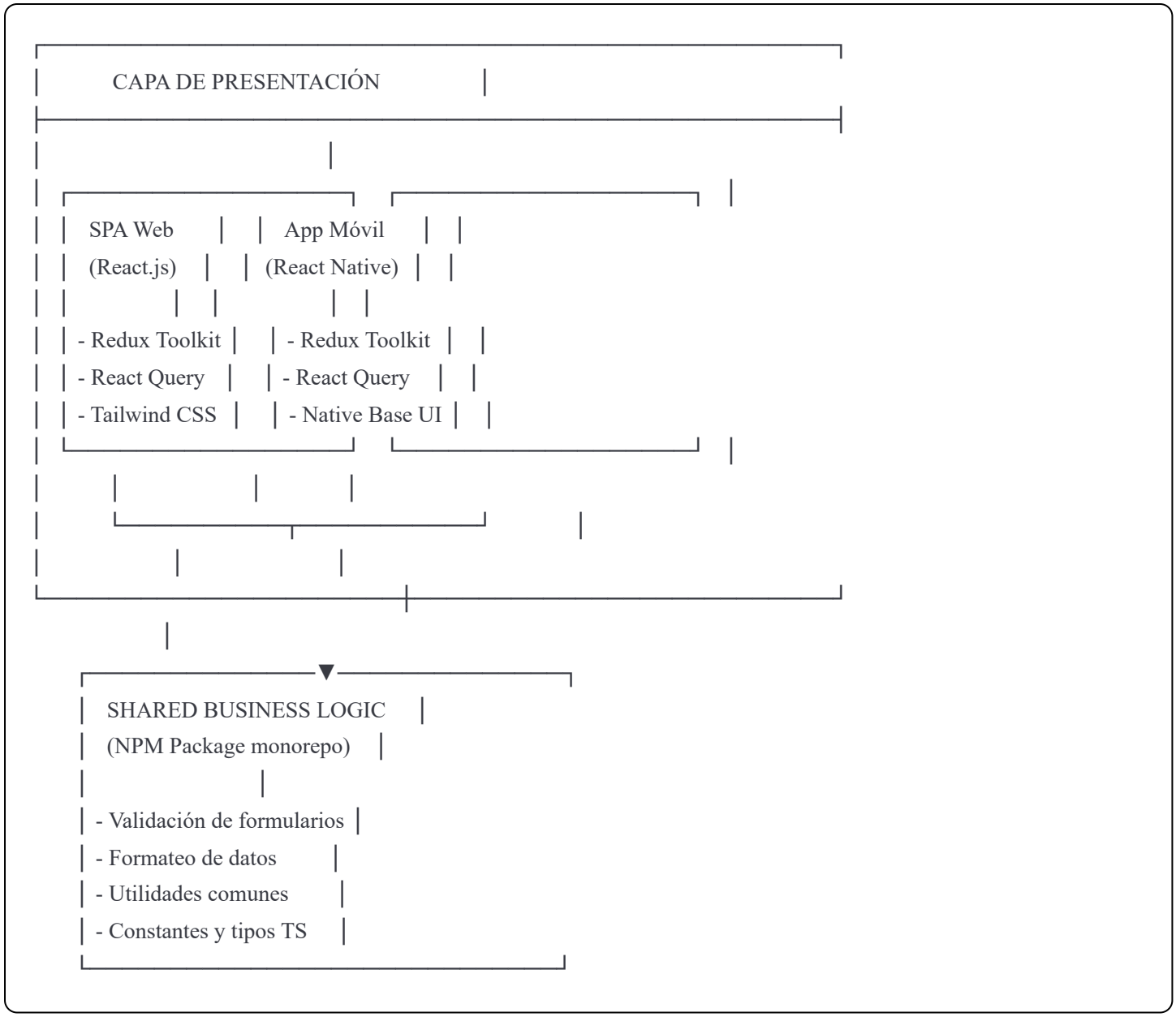
8.3 Recomendación Final: REACT NATIVE

Justificación de la elección:

Para el proyecto bancario BP, se recomienda **React Native** por las siguientes razones estratégicas:

1. **Staff Técnico Actual**: El documento menciona que JavaScript es el lenguaje principal del equipo. React Native permite máxima reutilización de conocimiento y código.
2. **Reutilización de Código con SPA**:
   - Componentes de UI compartidos entre web y móvil
   - Lógica de negocio compartida (validaciones, modelos)
   - Reducción de costos de desarrollo (~30-40%)
3. **Time-to-Market**:
   - Desarrollo más rápido que las alternativas
   - Prototipado y MVP en menor tiempo
   - Crítico para competir en sector bancario digital
4. **Ecosistema y Madurez**:
   - Librerías probadas para fintech (Plaid, Stripe)
   - Componentes de seguridad y biometría maduros
   - Soluciones existentes para problemas comunes
5. **Costo de Contratación**:
   - Developers JS abundantes y más económicos
   - Menor inversión en capacitación
   - Facilidad para escalar el equipo
6. **Casos de Éxito en Banca**:
   - Bloomberg, Coinbase, Walmart usan React Native
   - Aplicaciones bancarias complejas ya construidas

Arquitectura Propuesta:



8.4 Arquitectura de la Aplicación Móvil

Stack Tecnológico Recomendado:

- **Framework**: React Native 0.73+
- **Navegación**: React Navigation 6
- **Estado Global**: Redux Toolkit + RTK Query
- **UI Components**: NativeBase o React Native Paper
- **Biometría**: react-native-biometrics
- **Seguridad**: react-native-keychain (almacenamiento seguro)
- **Push Notifications**: @react-native-firebase/messaging
- **Analytics**: Firebase Analytics
- **Crash Reporting**: Sentry
- **Testing**: Jest + React Native Testing Library

Estructura de Carpetas:

```
mobile-app/
├── src/
│   ├── components/       # Componentes reutilizables
│   ├── screens/          # Pantallas de la app
│   ├── navigation/       # Configuración de navegación
│   ├── services/         # Llamadas API
│   ├── store/            # Redux store
│   ├── hooks/            # Custom React hooks
│   ├── utils/            # Utilidades
│   ├── constants/        # Constantes
│   └── types/            # TypeScript types
├── android/              # Código nativo Android
├── ios/                  # Código nativo iOS
└── shared-logic/         # Lógica compartida con web
```

---

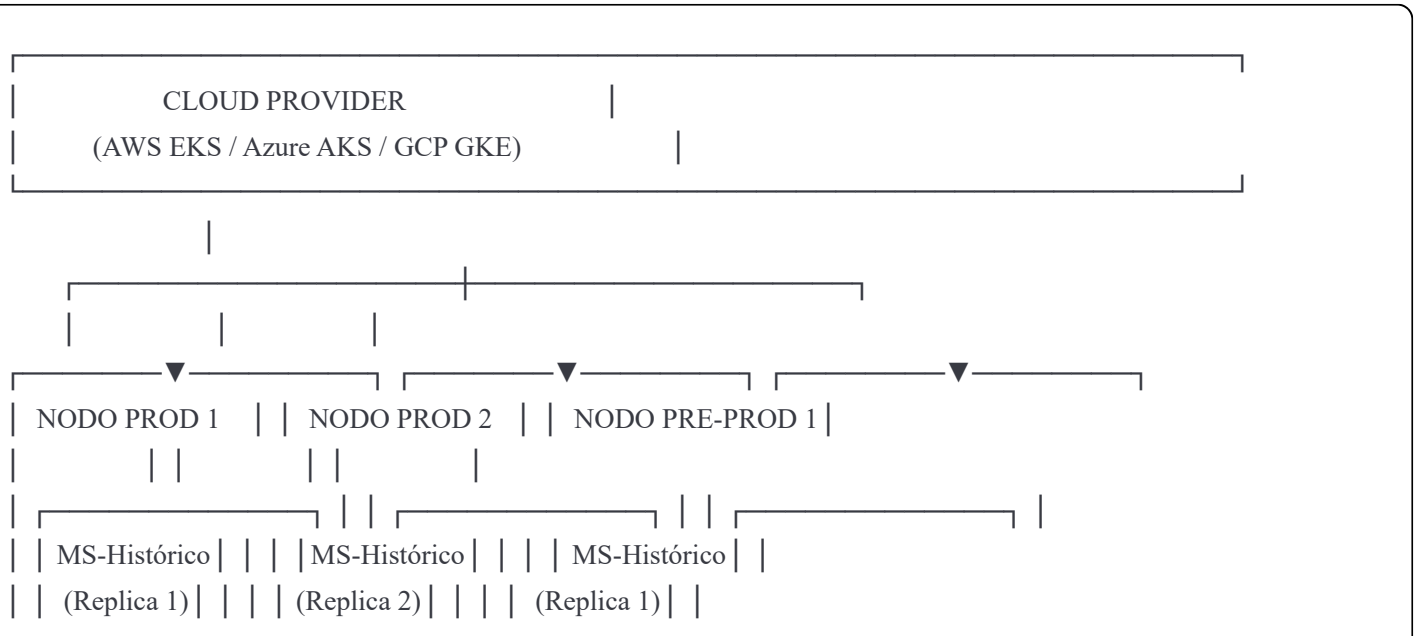
9. INFRAESTRUCTURA CLOUD Y ORQUESTACIÓN CON KUBERNETES

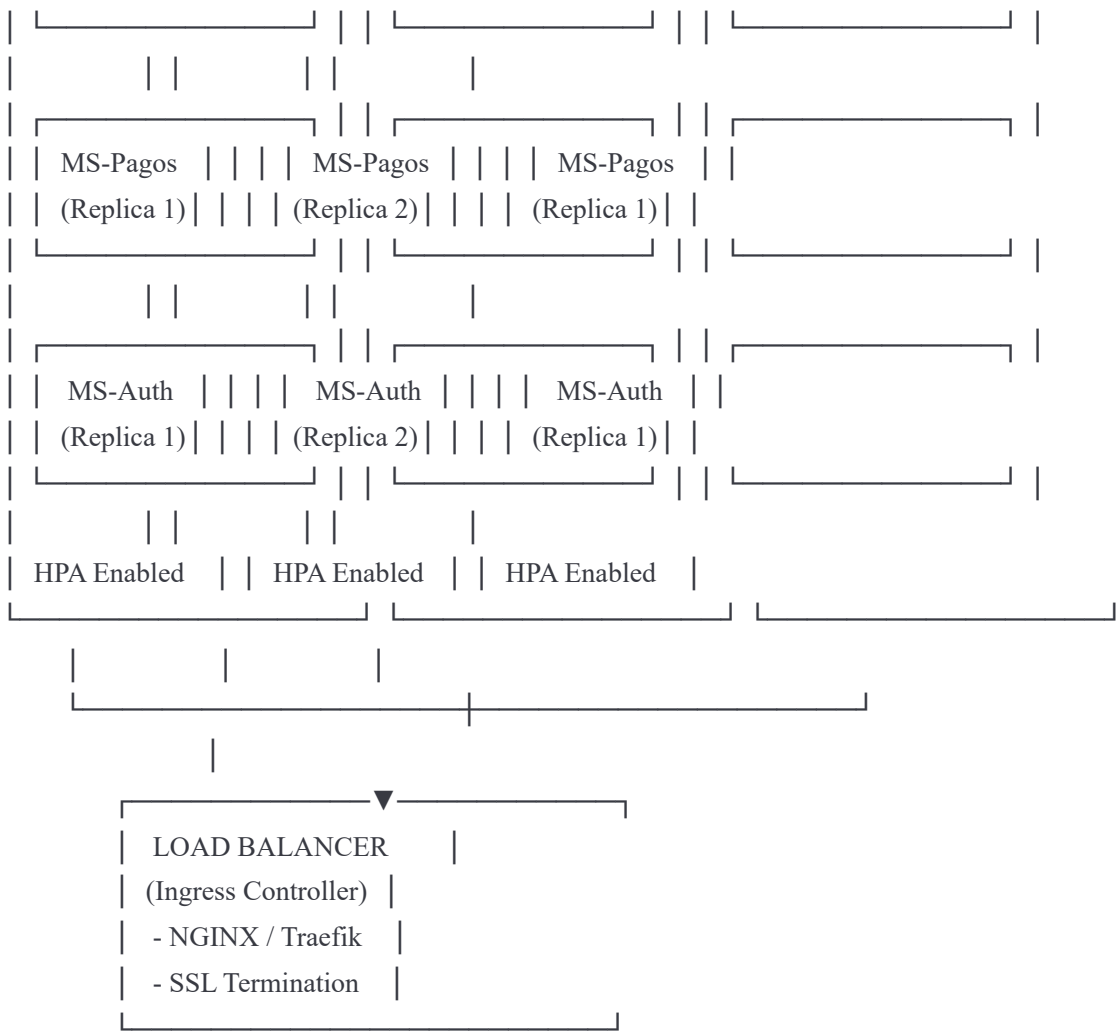
9.1 Estrategia de Despliegue Multi-Ambiente

El sistema se desplegará en Kubernetes con 3 ambientes separados:

Ambiente	Nodos	Pods por Nodo	CPU/RAM por Pod	Propósito
Producción	2	3 réplicas/servicio	2 CPU / 4GB	Usuarios finales
Pre-Producción	2	2 réplicas/servicio	1 CPU / 2GB	Testing final, staging
Desarrollo	1	2 pods (Dev + QA)	0.5 CPU / 1GB	Desarrollo activo

9.2 Diagrama de Infraestructura Kubernetes





9.3 Configuración de Autoescalado

Horizontal Pod Autoscaler (HPA):

Cada microservicio se configura con escalado automático basado en métricas:

```
Métricas de Escalado:
- CPU > 70% → Crear nuevo pod
- Memoria > 80% → Crear nuevo pod
- Request Rate > 1000 req/s → Crear nuevo pod

Límites:
- Mínimo: 2 réplicas (alta disponibilidad)
- Máximo: 10 réplicas (control de costos)
- Cool-down: 5 minutos (evitar flapping)
```

Vertical Pod Autoscaler (VPA):

Ajusta automáticamente CPU y RAM asignada a cada pod según uso histórico:

- Análisis de últimos 7 días de consumo
- Recomendaciones automáticas de recursos

- Aplicación en ventanas de mantenimiento

Justificación del Autoescalado:

- **Reducción de Costos**: Solo se pagan recursos cuando se necesitan
- **Alta Disponibilidad**: Escala ante picos de tráfico (fin de mes, promociones)
- **Eficiencia Operativa**: Sin intervención manual
- **Ahorro estimado**: 40-60% vs. aprovisionamiento estático

9.4 Balanceo de Carga

Algoritmo Recomendado: Least Connections

Justificación:

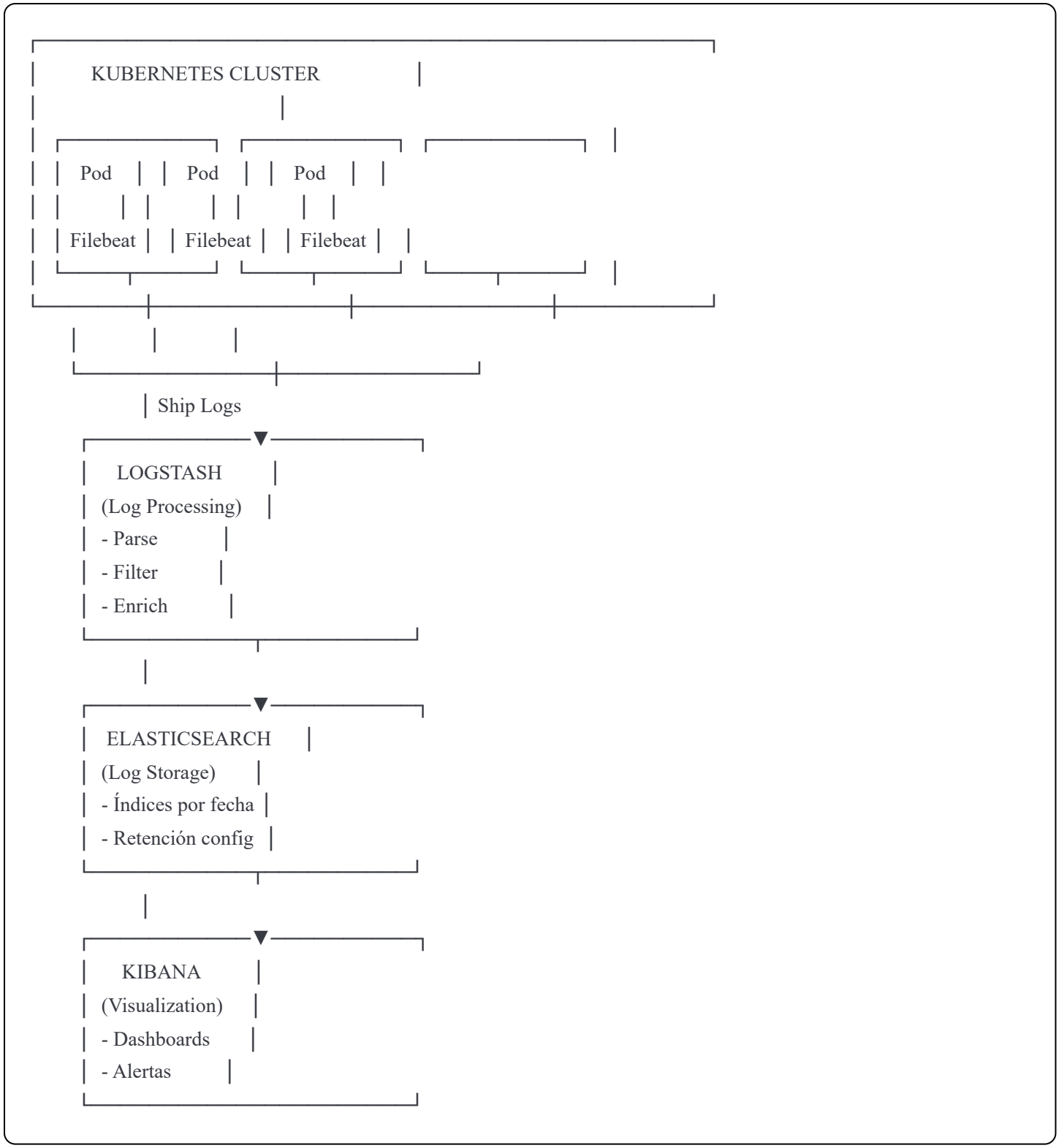
- Las transacciones bancarias tienen duración variable
- Algunos requests (transferencias) son más pesados que otros (consultas)
- Least Connections distribuye mejor la carga real vs Round Robin
- Previene sobrecarga de pods individuales

Alternativa para casos específicos:

- **IP Hash**: Para mantener sesiones pegajosas si fuera necesario
- **Weighted Round Robin**: Para introducir nuevas versiones gradualmente (Canary Deployments)

9.5 Health Checking y Observabilidad

Implementación con ELK Stack:



Health Checks en Kubernetes:

1. **Liveness Probe**: ¿El contenedor está vivo?
   - HTTP GET /health cada 10 segundos
   - Timeout: 3 segundos
   - Failure threshold: 3 intentos
   - Acción: Reiniciar pod
2. **Readiness Probe**: ¿El contenedor está listo para recibir tráfico?
   - HTTP GET /ready cada 5 segundos

- Timeout: 2 segundos
- Success threshold: 1
- Acción: Remover del balanceador

3. **Startup Probe**: ¿La aplicación terminó de iniciar?
   - HTTP GET /startup cada 5 segundos
   - Failure threshold: 30 (2.5 minutos máximo)
   - Acción: Reiniciar pod si falla

9.6 Estrategia de Docker Compose

Para desarrollo local y testing, se configuran múltiples archivos docker-compose agrupados por funcionalidad:

Estructura de Archivos:

```
docker/
├── docker-compose.core.yml          # Servicios core
├── docker-compose.databases.yml     # PostgreSQL, MongoDB, Redis
├── docker-compose.messaging.yml     # Kafka, ZooKeeper
├── docker-compose.observability.yml # ELK, Prometheus, Grafana
├── docker-compose.dev.yml           # Override para desarrollo
└── docker-compose.test.yml          # Override para testing
```

Comando de inicio completo:

```
docker-compose -f docker-compose.core.yml \
               -f docker-compose.databases.yml \
               -f docker-compose.messaging.yml \
               -f docker-compose.observability.yml \
               -f docker-compose.dev.yml \
               up -d
```

9.7 Kafka + ZooKeeper para Bus de Mensajería

Configuración de Cluster:

- **3 nodos Kafka** para alta disponibilidad
- **3 nodos ZooKeeper** para quorum (mínimo impar)
- **Replication Factor: 3** para cada topic
- **Min In-Sync Replicas: 2** para garantizar durabilidad

Topics Principales:

Topic	Particiones	Retención	Uso
transactions.created	10	30 días	Nuevas transacciones
audit.events	5	7 años	Eventos de auditoría
notifications.queue	5	7 días	Notificaciones pendientes
data.sync	3	1 día	Sincronización CQRS

Justificación de Kafka:

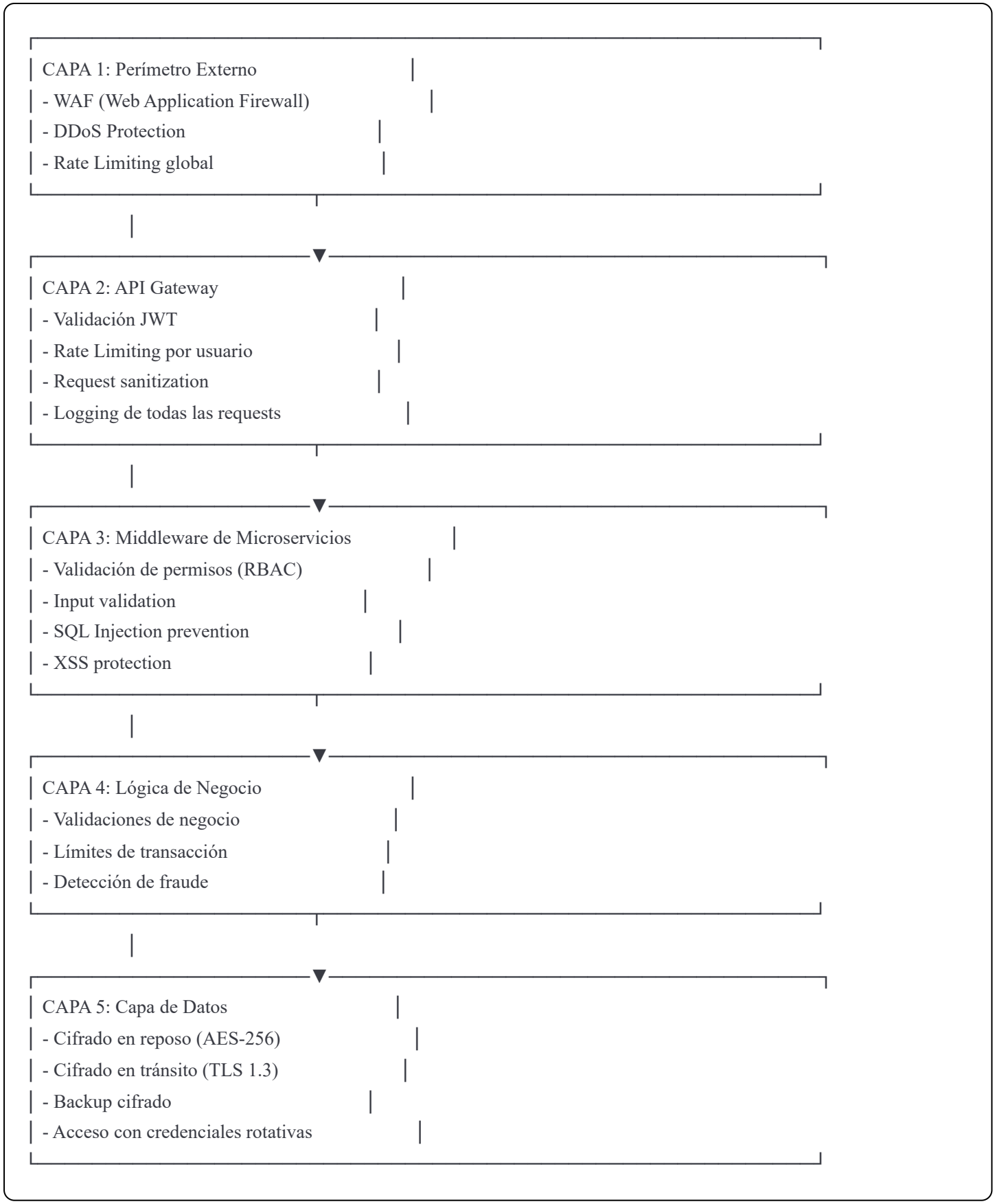
- Throughput: 1M+ mensajes/segundo por nodo
- Durabilidad: Réplicas + persistencia en disco
- Escalabilidad: Particionamiento horizontal
- Orden garantizado: Por partition key
- Replay capability: Re-procesar eventos históricos

---

10. ESTRATEGIA DE SEGURIDAD INTEGRAL

10.1 Arquitectura de Seguridad en Capas

La seguridad se implementa en múltiples capas (Defense in Depth):



10.2 Web Application Firewall (WAF)

Proveedor Recomendado: Cloudflare WAF o AWS WAF

Reglas Implementadas:

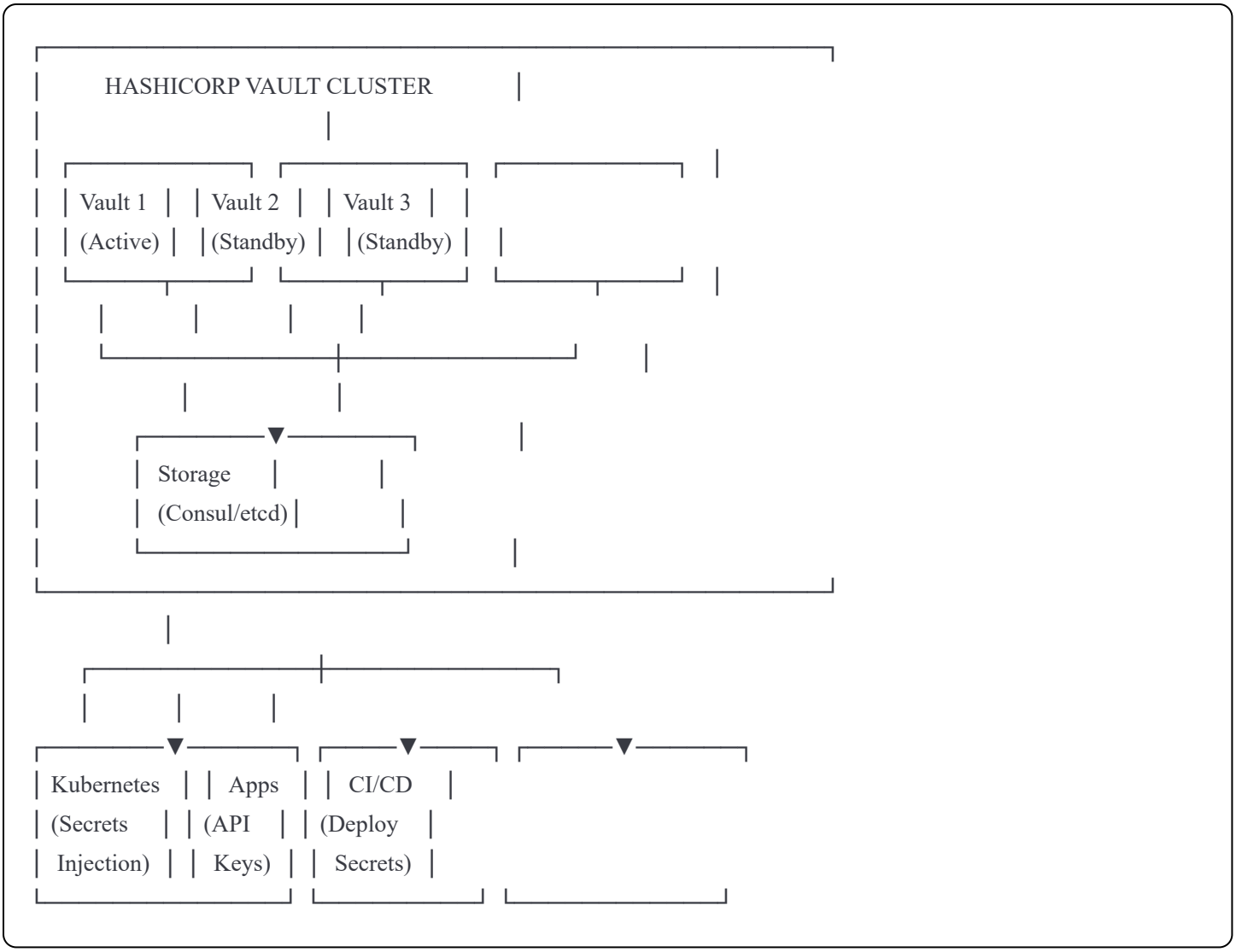
1. **OWASP Top 10 Protection**:
   - SQL Injection

- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Path Traversal
- Remote Code Execution

2. **Rate Limiting**:
   - 100 requests/minuto por IP
   - 1000 requests/minuto por usuario autenticado
   - Bloqueo temporal de IPs sospechosas
3. **Geo-Blocking**:
   - Permitir solo países donde opera el banco
   - Bloquear países de alto riesgo de fraude
4. **Bot Protection**:
   - Detección de bots maliciosos
   - Challenge para comportamiento sospechoso
   - Protección contra credential stuffing

10.3 HashiCorp Vault para Gestión de Secretos

Arquitectura de Vault:



Secretos Gestionados:

- Credenciales de bases de datos
- API keys de servicios externos (Firebase, AWS)
- Certificados SSL/TLS
- Claves de cifrado
- OAuth client secrets
- Claves de firma JWT

Características de Vault:

- Rotación automática de secretos cada 90 días
- Audit log de todos los accesos
- Políticas de acceso granulares (por servicio)
- Cifrado de secretos en reposo
- Integración nativa con Kubernetes

10.4 Proxy Inverso con SSL/TLS

Configuración de NGINX como Reverse Proxy:

Características:

- Terminación SSL en el proxy (offloading)
- Renovación automática de certificados con Let's Encrypt
- Failover automático si un certificado expira
- HTTP/2 y HTTP/3 (QUIC) habilitado
- HSTS (HTTP Strict Transport Security)
- Certificate pinning para apps móviles

Beneficios:

- Los microservicios no manejan SSL directamente
- Certificados centralizados
- Mejor performance (SSL offloading)
- Facilita la rotación de certificados

10.5 Políticas de Seguridad Implementadas

A) Control de Acceso (RBAC - Role-Based Access Control):

Rol	Permisos	Casos de Uso
Usuario	Consulta, transferencias propias	Clientes finales
Soporte	Consulta de datos, reset password	Call center
Operador	Consulta, aprobación de transferencias grandes	Back office
Administrador	Todos los permisos, configuración	IT team
Auditor	Solo lectura de logs y auditoría	Compliance team

B) Políticas de Contraseñas:

- Mínimo 12 caracteres
- Al menos 1 mayúscula, 1 minúscula, 1 número, 1 carácter especial
- No puede contener datos personales (nombre, email)
- Historial de últimas 5 contraseñas (no reutilizar)
- Expiración cada 90 días
- Bloqueo tras 5 intentos fallidos

C) Cifrado:

- **En tránsito**: TLS 1.3 (mínimo TLS 1.2)
- **En reposo**: AES-256-GCM
- **Datos sensibles en BD**: Cifrado a nivel de columna
- **PCI DSS**: Números de tarjeta tokenizados (nunca almacenar completos)

D) Seguridad de Red:

- Segmentación de red por microservicio
- Network Policies en Kubernetes (Zero Trust)
- Solo el API Gateway expuesto públicamente
- Microservicios en red privada
- Egress rules restrictivas (whitelist de APIs externas)

---

11. ALTA DISPONIBILIDAD Y TOLERANCIA A FALLOS

11.1 Objetivos de Disponibilidad (SLA)

Métrica	Objetivo	Cálculo Anual
Uptime	99.9% (Three Nines)	8.76 horas downtime/año
RTO (Recovery Time Objective)	< 15 minutos	Tiempo máximo de recuperación
RPO (Recovery Point Objective)	< 5 minutos	Pérdida máxima de datos
Latencia API	< 200ms (p95)	95% de requests bajo 200ms
Error Rate	< 0.1%	Máximo 1 error por cada 1000 requests

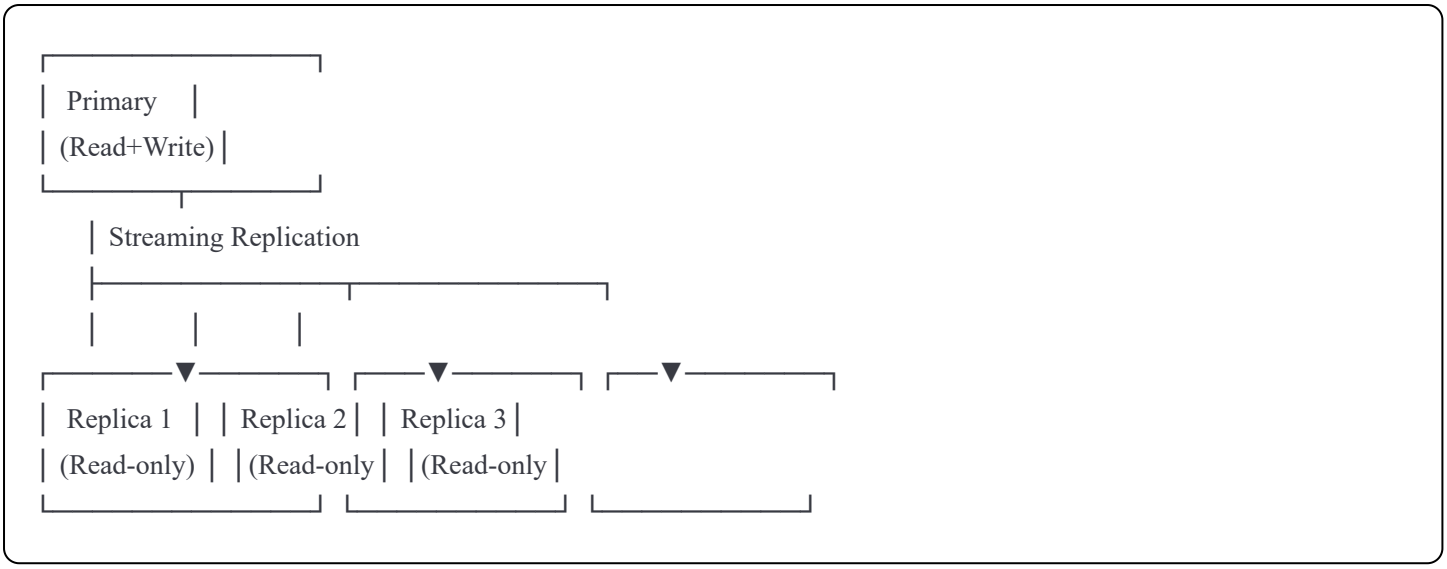
11.2 Estrategias de Alta Disponibilidad

A) Réplicas de Microservicios:

- Mínimo 2 réplicas en producción (hasta 3)
- Desplegadas en diferentes nodos (anti-affinity)
- Health checks continuos cada 10 segundos
- Recreación automática de pods fallidos

B) Réplicas de Bases de Datos:

PostgreSQL (Master-Replica):



- Replicación asíncrona para lecturas
- Failover automático con Patroni o Stolon
- Backup automático diario + PITR (Point-in-Time Recovery)

C) Circuit Breaker Pattern:

Implementación con librerías como Opossum o Hystrix:

```
Estado Normal:
Request → Servicio ✓

Estado Open (tras fallos):
Request → Circuit Breaker → Error inmediato (sin llamar al servicio)
    (Espera 30 segundos antes de reintentar)

Estado Half-Open:
Request → Circuit Breaker → Intenta 1 request
    Si falla → Vuelve a Open
    Si éxito → Vuelve a Closed (Normal)
```

Beneficios:

- Evita cascading failures (fallos en cadena)
- Respuesta rápida al usuario (fail-fast)
- Da tiempo al servicio caído para recuperarse

D) Bulkhead Pattern:

Aislamiento de recursos por tipo de operación:

```
Thread Pool Consultas (20 threads) → BD Read-Only
Thread Pool Transferencias (10 threads) → BD Transaccional
Thread Pool Notificaciones (5 threads) → External APIs
```

- Si un tipo de operación falla, no afecta a las demás
- Previene resource exhaustion

E) Retry con Exponential Backoff:

Para operaciones idempotentes:

- Intento 1: Inmediato
- Intento 2: Espera 1 segundo
- Intento 3: Espera 2 segundos
- Intento 4: Espera 4 segundos
- Intento 5: Espera 8 segundos
- Máximo 5 intentos

F) Caché de Degradación:

Si el servicio principal falla:

- Servir datos desde caché (aunque estén desactualizados)
- Mostrar mensaje al usuario: "Datos de hace X minutos"
- Mejor experiencia que error total

11.3 Disaster Recovery

Estrategia de Backup:

Tipo	Frecuencia	Retención	Storage
Full Backup	Semanal (Domingo 2am)	3 meses	S3 Glacier
Incremental	Diario (2am)	30 días	S3 Standard
Transaction Logs	Continuo	7 días	S3 Standard
Snapshots	Cada 6 horas	48 horas	EBS Snapshots

Plan de Recuperación ante Desastres:

1. **Desastre Menor** (1 microservicio caído):
   - Tiempo de detección: < 1 minuto (health checks)
   - Acción: Restart automático del pod
   - Tiempo de recuperación: < 2 minutos

2. **Desastre Medio** (1 nodo caído):
    - Tiempo de detección: < 1 minuto
    - Acción: Kubernetes reschedula pods en otros nodos
    - Tiempo de recuperación: < 5 minutos
3. **Desastre Mayor** (Región completa caída):
    - Tiempo de detección: < 5 minutos
    - Acción: Failover a región secundaria (manual)
    - Tiempo de recuperación: < 15 minutos
    - Requiere: Multi-region deployment (recomendado para futuro)

11.4 Monitoreo y Alertas

Sistema de Alertas Proactivo:

Alerta	Umbral	Acción	Canal
CPU > 80%	5 minutos consecutivos	Escalar pods	Slack + Email
Error Rate > 1%	2 minutos	Investigación inmediata	PagerDuty
Latencia > 500ms	p95 por 3 minutos	Revisar cuellos de botella	Slack
Disco > 85%	Instantáneo	Limpiar logs antiguos	Email
Pod restart loop	3 restarts en 5 min	Alerta crítica	PagerDuty
Backup fallido	Cualquier fallo	Acción inmediata	Email + SMS

Dashboard de Operaciones:

Grafana con paneles para:

- Estado de salud de todos los microservicios
- Métricas de negocio (transacciones/hora, monto total)
- Latencias por endpoint
- Tasa de errores
- Uso de recursos (CPU, RAM, Disco)
- Salud de bases de datos y cache

---

12. CUMPLIMIENTO NORMATIVO

12.1 Marcos Normativos Aplicables

El sistema bancario debe cumplir con múltiples regulaciones internacionales y locales:

ISO 27001 - Seguridad de la Información

Requisitos Implementados:

- **A.9 Control de Acceso**: RBAC implementado en todos los microservicios
- **A.10 Criptografía**: TLS 1.3 en tránsito, AES-256 en reposo
- **A.12 Seguridad de Operaciones**: Logs de auditoría completos, monitoreo 24/7
- **A.14 Seguridad en Adquisición**: Validación de proveedores cloud
- **A.18 Cumplimiento**: Auditorías trimestrales, revisión de políticas

Evidencias:

- Política de seguridad documentada
- Matriz de riesgos actualizada
- Registros de auditoría (mínimo 7 años)
- Plan de continuidad de negocio (BCP)

ISO 9001 - Gestión de la Calidad

Implementación:

- **Procesos documentados**: Diagramas de arquitectura, procedimientos operativos
- **Mejora continua**: Sprint retrospectives, análisis de incidentes
- **Satisfacción del cliente**: NPS tracking, análisis de feedback
- **Métricas de calidad**: SLA compliance, code coverage >80%

ISO 27002 - Guía para la Seguridad de la Información

Complementa ISO 27001 con controles adicionales:

- Clasificación de información (Pública, Interna, Confidencial, Restringida)
- Segregación de ambientes (Dev/QA/Prod)
- Gestión de cambios con aprobación
- Protección contra malware

ISO 27701 - Sistema de Gestión de Privacidad de Información (SGPI)

Controles de Privacidad:

- **Minimización de datos**: Solo se recopilan datos necesarios
- **Propósito limitado**: Datos usados solo para el fin declarado
- **Precisión de datos**: Mecanismos de actualización de datos personales

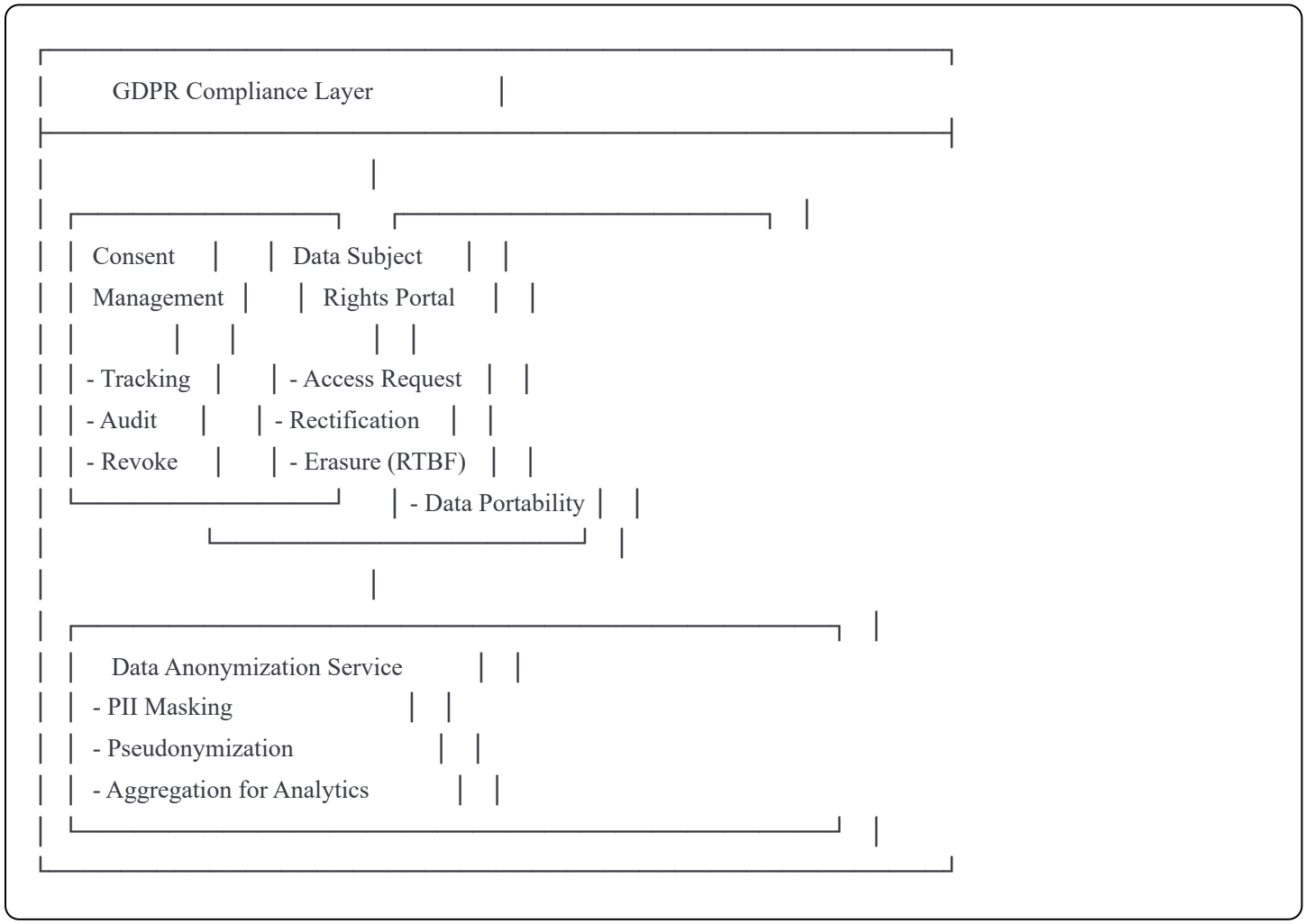
- **Limitación de almacenamiento**: Retención según tabla de retención
- **Derechos del titular**:
  - Derecho de acceso (API /my-data)
  - Derecho de rectificación (API /update-profile)
  - Derecho de portabilidad (Export data en JSON/PDF)
  - Derecho al olvido (Anonimización tras solicitud)

GDPR (Reglamento General de Protección de Datos) - Europa

Requisitos Clave:

1. **Base Legal para Procesamiento**: Consentimiento explícito del usuario
2. **Notificación de Brechas**: Dentro de 72 horas a la autoridad
3. **Privacy by Design**: Privacidad desde el diseño de arquitectura
4. **DPO (Data Protection Officer)**: Designado y contacto publicado
5. **Transferencias Internacionales**: Solo a países con adecuación o cláusulas contractuales

Implementación Técnica:



GLBA (Gramm-Leach-Bliley Act) - Estados Unidos

Requisitos Financieros:

- **Safeguards Rule**: Protección de información financiera del cliente
- **Privacy Rule**: Notificar a clientes sobre prácticas de privacidad
- **Pretexting Provisions**: Protección contra obtención fraudulenta de información

Implementación:

- Annual privacy notice a todos los clientes
- Opt-out mechanism para compartir información con terceros
- Cifrado obligatorio para datos financieros sensibles
- Vendor management (auditoría de terceros que procesan datos)

PCI DSS (Payment Card Industry Data Security Standard)

12 Requisitos Principales:

Requisito	Implementación en la Arquitectura
1. Firewall	WAF + Network Policies en K8s
2. Contraseñas por defecto	Secretos en Vault, sin defaults
3. Proteger datos almacenados	AES-256, tokenización de tarjetas
4. Cifrado en transmisión	TLS 1.3 obligatorio
5. Antivirus	ClamAV en contenedores
6. Sistemas seguros	Hardened container images
7. Acceso por need-to-know	RBAC estricto
8. Identificación única	OAuth 2.0 + MFA
9. Acceso físico restringido	Cloud provider certified datacenter
10. Logs de acceso	ELK Stack, retención 1 año
11. Testing de seguridad	Pentesting trimestral
12. Política de seguridad	Documento actualizado anualmente

Alcance PCI DSS:

- **Cardholder Data Environment (CDE)**: Segmentado en red propia
- **Tokenización**: Números de tarjeta reemplazados por tokens
- **No almacenamiento de CVV**: Nunca se guarda CVV post-autorización
- **Quarterly ASV Scans**: Vulnerability scans trimestrales
- **Annual Penetration Testing**: Por empresa certificada QSA

12.2 Tabla de Cumplimiento Consolidada

Normativa	Aspecto Clave	Evidencia en Arquitectura
ISO 27001	Seguridad integral	WAF, Vault, cifrado, auditoría
ISO 9001	Calidad de procesos	Documentación, métricas, CI/CD
ISO 27002	Controles detallados	114 controles implementados
ISO 27701	Privacidad	Portal de derechos, minimización
GDPR	Protección de datos EU	Consentimiento, RTBF, DPO
GLBA	Finanzas USA	Privacy notices, safeguards
PCI DSS	Datos de tarjetas	Tokenización, cifrado, segmentación

12.3 Proceso de Auditoría

Auditorías Planificadas:

- **Interna**: Trimestral - Equipo de Compliance
- **Externa ISO 27001**: Anual - Certificadora acreditada
- **PCI DSS QSA**: Anual - Qualified Security Assessor
- **Penetration Testing**: Trimestral - Empresa de seguridad externa

Registro de Auditoría:

Todos los eventos relevantes para compliance se almacenan en Elasticsearch con:

- Usuario que ejecutó la acción
- Timestamp preciso (UTC)
- Acción realizada (CREATE, READ, UPDATE, DELETE)
- Datos afectados (ID de entidades, no el contenido sensible)
- IP de origen
- Resultado de la operación (éxito/fallo)
- Razón de negocio (cuando aplique)

---

13. GESTIÓN DE COSTOS Y ESCALABILIDAD

13.1 Estimación de Costos Mensuales

Infraestructura Cloud (Ejemplo AWS):

Componente	Especificación	Costo Mensual (USD)
EKS Control Plane	1 cluster	$73
Nodos EC2 Producción	2x t3.xlarge (4vCPU, 16GB)	$300
Nodos EC2 Pre-Prod	2x t3.large (2vCPU, 8GB)	$150
Nodos EC2 Dev	1x t3.medium (2vCPU, 4GB)	$40
RDS PostgreSQL	db.r5.large Multi-AZ	$350
RDS Read Replicas	2x db.r5.large	$350
ElastiCache Redis	cache.r5.large	$200
MSK (Kafka)	3 nodos kafka.m5.large	$450
S3 Storage	500 GB + requests	$50
CloudWatch Logs	50 GB/mes	$30
Load Balancer	Application LB	$25
NAT Gateway	2 AZ	$90
Data Transfer	1 TB egress	$90
Backup Storage	1 TB S3 Glacier	$15
WAF	100M requests	$60
Route 53	Hosted zone + queries	$10
Elasticsearch	3 nodos r5.large.search	$400
Total Estimado		**~$2,700/mes**

Servicios Externos:

Servicio	Costo Mensual (USD)
Firebase (SMS + Push)	$50 - $200 (según volumen)
Sentry (Error Tracking)	$26 (Team plan)
HashiCorp Vault	$50 (self-hosted)
Dominio + SSL	$15
Total Servicios	**~$150/mes**

Costo Total Estimado: $2,850 - $3,000/mes (~$35,000/año)

13.2 Estrategias de Optimización de Costos

A) Autoescalado Inteligente:

- Scale down nocturno (2am-6am) a 50% de réplicas
- Scale down fines de semana (Sábado/Domingo)
- Ahorro estimado: 20-30% en costos de compute

B) Spot Instances para Dev/QA:

- Usar EC2 Spot Instances (hasta 90% descuento)
- Solo para ambientes no-productivos
- Ahorro estimado: $80/mes

C) S3 Lifecycle Policies:

- Logs > 30 días → S3 Infrequent Access (50% ahorro)
- Logs > 90 días → S3 Glacier (80% ahorro)
- Backups > 1 año → S3 Deep Archive (90% ahorro)

D) Reserved Instances para Producción:

- Commitment de 1-3 años para nodos estables
- Descuento: 40-60% vs On-Demand
- Aplicable a: RDS, EC2, ElastiCache
- Ahorro estimado: $800/mes

E) Right-Sizing:

- Análisis mensual de uso de CPU/RAM
- Downsizing de instancias sobredimensionadas
- Ahorro estimado: 15-25%

Costo Optimizado: $2,000 - $2,200/mes (~$25,000/año)

13.3 Escalabilidad Proyectada

Escenario de Crecimiento:

Métrica	Año 1	Año 2	Año 3
Usuarios activos	50,000	200,000	500,000
Transacciones/día	100,000	500,000	1,500,000
Requests/segundo (peak)	200	1,000	3,000
Almacenamiento (TB)	0.5	2	8
Nodos K8s requeridos	4	8	16
Costo mensual estimado	$3,000	$6,000	$12,000

Capacidad de Escalamiento:

La arquitectura propuesta puede escalar hasta:

- **10M+ usuarios** con la misma arquitectura base
- **100M+ transacciones/día** agregando nodos horizontalmente
- Sin necesidad de re-arquitectura (solo infraestructura)

13.4 ROI y Justificación de Inversión

Comparación con Alternativa Monolítica:

Aspecto	Microservicios (Propuesto)	Monolito Tradicional
Costo inicial	Más alto	Más bajo
Time-to-market	4-6 meses	8-12 meses
Costo de escalamiento	Lineal	Exponencial
Costo de mantenimiento	Bajo (aislado)	Alto (todo acoplado)
Downtime en deploy	0 (rolling updates)	5-15 minutos
Flexibilidad tecnológica	Alta	Baja
Costo a 3 años	$90,000	$150,000+

ROI Estimado:

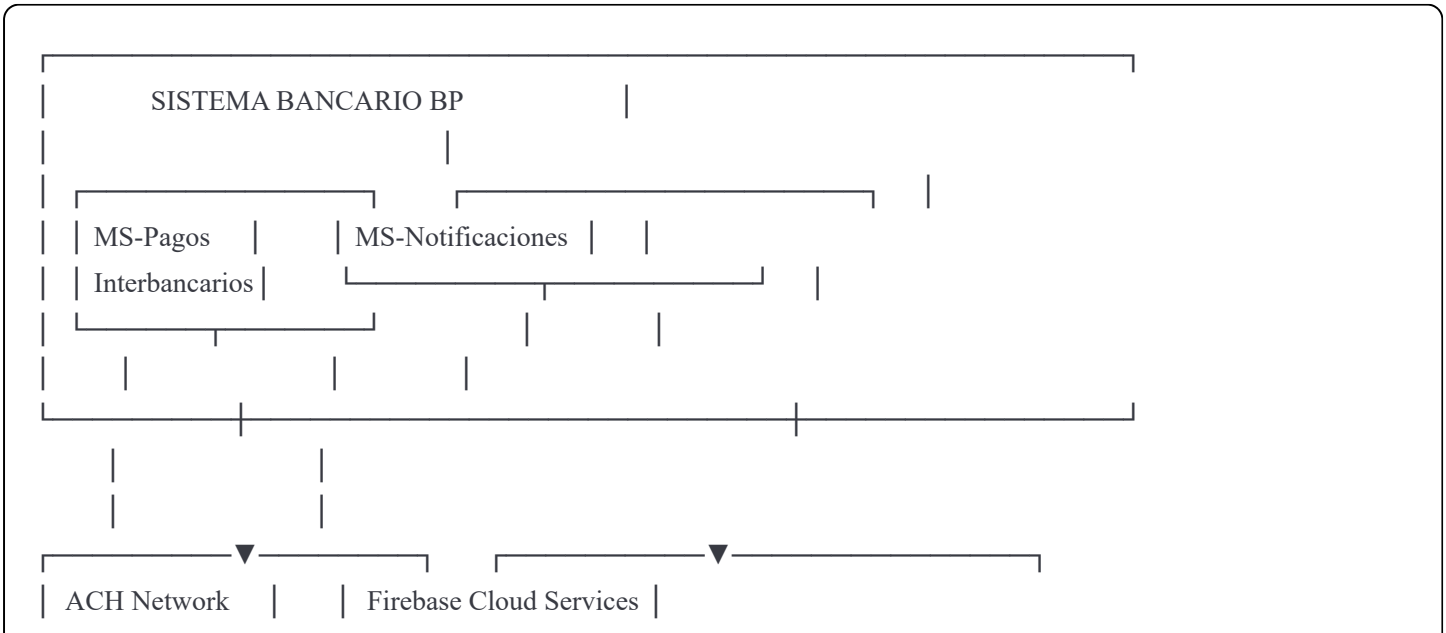
- Reducción de 50% en time-to-market = $200,000 en revenue anticipado
- Reducción de 70% en downtime = $50,000/año en pérdidas evitadas
- Mejora de 30% en conversión (mejor UX) = $500,000/año en revenue adicional

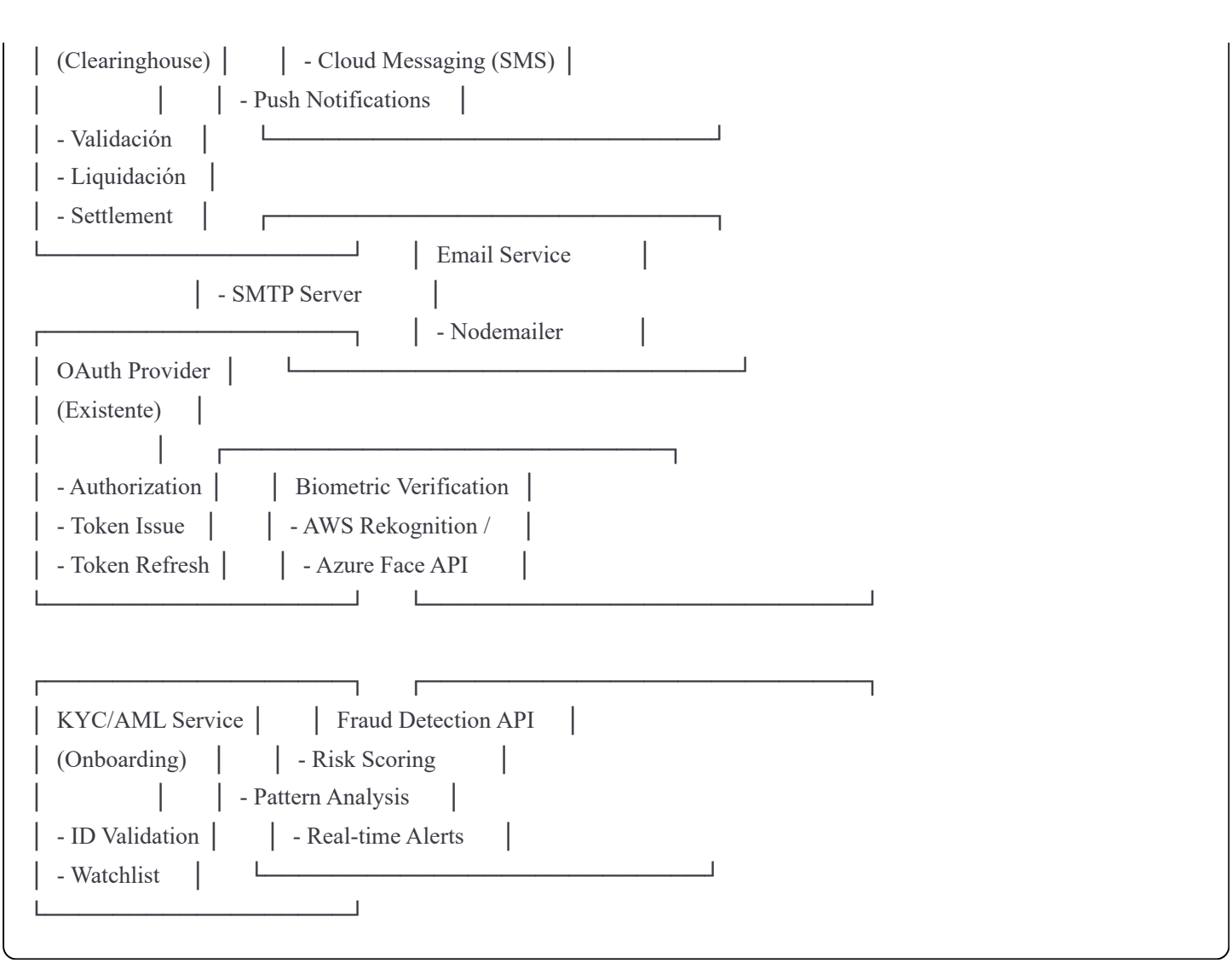
ROI Total: 600% en 3 años

---

14. DIAGRAMAS DE INTEGRACIÓN Y FLUJOS

14.1 Diagrama de Integración con Servicios Externos





14.2 Flujo Completo de Transferencia Interbancaria

1. Usuario inicia transferencia en App Móvil
   │
   ├─► [Validación local] Formato de cuenta destino
   │
   └─► POST /api/v1/transfers/interbank
      │
      ▼
2. API Gateway valida JWT y rate limit
   │
   ├─► [Si falla] → 401 Unauthorized / 429 Too Many Requests
   │
   └─► [Si OK] → Enruta a MS-Pagos-Interbancarios
      │
      ▼
3. MS-Pagos-Interbancarios procesa
   │
   ├─► Valida saldo suficiente (consulta MS-Datos-Cliente)
   ├─► Valida límites diarios (Redis cache)
   ├─► Valida cuenta destino (API ACH Network)
   │


├─► [Si falla validación] → 400 Bad Request
└─► Event: transfer.failed → Kafka
└─► Notificación a usuario
└─► [Si OK] → Procede con transacción
4. Transacción en Base de Datos
├─► BEGIN TRANSACTION
├─► Debita cuenta origen (PostgreSQL Write DB)
├─► Crea registro de transferencia (status: PENDING)
├─► COMMIT TRANSACTION
└─► Event: transfer.created → Kafka
5. Procesamiento Asíncrono
├─► MS-Auditoría consume evento → Log en Elasticsearch
├─► MS-Notificaciones consume evento
├─► Email: "Transferencia iniciada"
├─► SMS: "Transferencia de $X realizada"
└─► Push: "Transferencia exitosa"
└─► Sincronización CQRS
└─► Actualiza MongoDB (Read DB) con nuevo movimiento
6. Integración con ACH Network (Asíncrono)
├─► Worker envía lote de transferencias cada 30 min
├─► ACH procesa y responde (puede tomar horas/días)
└─► Webhook callback → Actualiza status
├─► COMPLETED: Marca como exitosa
├─► REJECTED: Reversa transacción
└─► Acredita cuenta origen
└─► Event: transfer.status_updated → Kafka
└─► Notificación final a usuario

14.3 Flujo de Onboarding con Reconocimiento Facial

└─► [Si OK] → Status: PENDING_BIOMETRIC

PASO 3: Registro Biométrico
─────────────────────────────────────

App Móvil solicita selfie video
  │
  ├─► Usuario graba video corto (3-5 segundos)
  │   - Mueve la cabeza
  │   - Parpadea
  │   - Dice una frase (liveness detection)
  │
  └─► POST /api/v1/biometric/register-face
      │
      ▼
MS-Biometría (AWS Rekognition / Azure Face)
  ├─► Extrae frames del video
  ├─► Liveness detection (no es foto/video grabado)
  ├─► Extrae vectores faciales (embeddings)
  ├─► Compara rostro con foto del documento
  │   └─► Similarity score debe ser >95%
  │
  ├─► [Si falla] → Solicita nuevo intento
  │
  └─► [Si OK] → Almacena embeddings en Vault (cifrado)
      │
      └─► Status: ACTIVE
          │
          ▼
Usuario habilitado
  ├─► Puede usar Face ID / Touch ID en dispositivo
  ├─► Puede hacer login con credenciales
  └─► Cuenta bancaria activada

PASO 4: Primer Login Post-Onboarding
──

Usuario abre app
  │
  ├─► Opción A: Login con usuario/contraseña
  │   └─► OAuth 2.0 flow (descrito anteriormente)
  │
  ├─► Opción B: Login con Face ID
  │   ├─► Device captura biometría local
  │   ├─► POST /api/v1/auth/biometric-login
  │   ├─► MS-Biometría compara con embeddings almacenados
  │   └─► [Si match >95%] → Emite JWT
  │

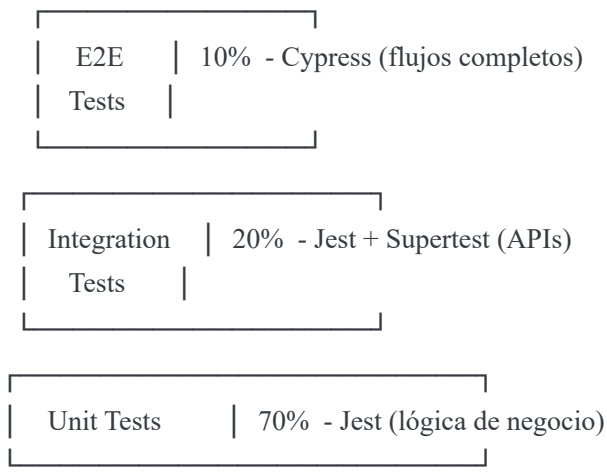
```
└─► Opción C: Login con Touch ID / Fingerprint
  └─► Similar a Face ID
```

---

15. CONSIDERACIONES ADICIONALES

15.1 Estrategia de Testing

Pirámide de Testing:



Cobertura Requerida:

- Unit Tests: >80%
- Integration Tests: >60%
- E2E Tests: Flujos críticos (login, transferencia, consulta)

Herramientas:

- Unit & Integration: Jest + Supertest
- E2E: Cypress o Playwright
- Load Testing: k6 o Artillery
- Security Testing: OWASP ZAP, Burp Suite

15.2 CI/CD Pipeline

Pipeline Automatizado:

1. Developer push a Git
   │
   ├─► Trigger CI Pipeline (GitHub Actions / GitLab CI)
   │
   ▼
2. Build Stage
   ├─► Install dependencies
   ├─► Run linter (ESLint)
   ├─► Run unit tests
   ├─► Build Docker image
   │
   └─► [Si falla] → Notifica developer, bloquea merge
   │
   ▼
3. Test Stage
   ├─► Run integration tests
   ├─► Run security scan (Snyk, Trivy)
   ├─► Code quality analysis (SonarQube)
   │
   └─► [Si falla] → Notifica team, bloquea merge
   │
   ▼
4. Deploy to Dev (automático)
   ├─► Push image a registry (ECR, Docker Hub)
   ├─► Update K8s manifest
   ├─► kubectl apply
   │
   ▼
5. Deploy to QA (manual approval)
   ├─► Run smoke tests
   ├─► Manual QA validation
   │
   ▼
6. Deploy to Pre-Prod (manual approval)
   ├─► Run full E2E test suite
   ├─► Performance testing
   │
   ▼
7. Deploy to Production (manual approval)
   ├─► Blue-Green deployment / Canary release
   ├─► Monitor errors (5 min window)
   │
   └─► [Si error rate >1%] → Automatic rollback

15.3 Documentación Requerida

Documentos Técnicos:

1. **Architecture Decision Records (ADR)**
   - Registro de todas las decisiones arquitectónicas
   - Justificación y alternativas consideradas
2. **API Documentation**
   - OpenAPI/Swagger specs para todos los endpoints
   - Ejemplos de request/response
   - Códigos de error documentados
3. **Runbooks Operacionales**
   - Procedimientos de deploy
   - Troubleshooting guides
   - Disaster recovery procedures
4. **Security Documentation**
   - Threat model
   - Security policies
   - Incident response plan
5. **Compliance Documentation**
   - Políticas de privacidad
   - Términos y condiciones
   - Mapeo de controles vs. normativas

15.4 Roadmap de Mejoras Futuras

Fase 2 (6-12 meses):

- **Service Mesh**: Istio o Linkerd para mejor observabilidad
- **GraphQL Federation**: API Gateway unificada con GraphQL
- **Machine Learning**: Detección de fraude con ML models
- **Multi-Region**: Despliegue en múltiples regiones para DR
- **Chaos Engineering**: Netflix Chaos Monkey para resilience testing

Fase 3 (12-24 meses):

- **Blockchain**: Para trazabilidad inmutable de transacciones
- **Open Banking**: APIs públicas para integraciones con fintechs

- **AI Chatbot**: Asistente virtual con NLP para soporte
- **Real-time Analytics**: Apache Flink para analytics en tiempo real
- **Edge Computing**: CDN con compute para latencia ultra-baja

---

16. CONCLUSIONES Y RECOMENDACIONES

16.1 Resumen de Decisiones Arquitectónicas Clave

1. Arquitectura de Microservicios

Se seleccionó una arquitectura de microservicios por su escalabilidad, mantenibilidad y flexibilidad tecnológica. Cada microservicio es independiente, con su propia base de datos y ciclo de despliegue, permitiendo evolucionar el sistema sin acoplamiento.

2. Patrón CQRS para Alta Performance

La separación de bases de datos para lectura y escritura (CQRS) optimiza el rendimiento en consultas frecuentes (histórico de movimientos) sin afectar las transacciones críticas. La consistencia eventual es aceptable dado que la sincronización ocurre en menos de 500ms.

3. React Native para Máxima Reutilización

React Native permite compartir código y conocimiento entre web y móvil, reduciendo time-to-market y costos de desarrollo en 30-40%. El ecosistema maduro y la abundancia de talento JavaScript justifican esta elección sobre Flutter o Kotlin Multiplatform.

4. OAuth 2.0 + Biometría para Seguridad Robusta

La integración del servicio OAuth 2.0 existente con autenticación biométrica (Face ID, Touch ID) proporciona seguridad de múltiples factores con excelente experiencia de usuario. El flujo Authorization Code with PKCE es el estándar recomendado para aplicaciones móviles.

5. Kubernetes para Orquestación y Autoescalado

Kubernetes con autoescalado horizontal y vertical permite eficiencia en costos (solo se pagan recursos usados) y alta disponibilidad (99.9% uptime). La configuración multi