5.9 Punto de Vista de Comportamiento

Propósito del Punto de Vista de Dinámica

Este punto analiza el comportamiento del sistema en tiempo de ejecución, enfocándose en:

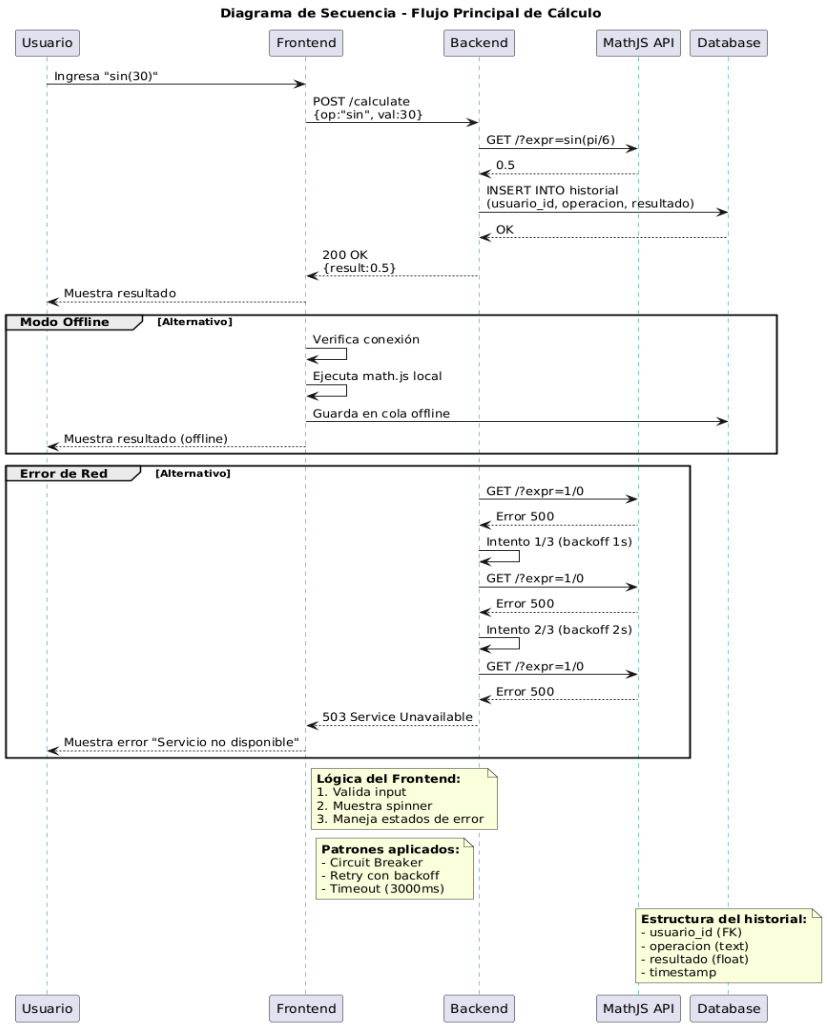
- Secuencias de interacción entre componentes
- Gestión de estados críticos
- Flujos de eventos complejos
- Patrones de concurrencia y paralelismo

Objetivos clave:

1. Modelar escenarios de uso realistas
2. Identificar condiciones de carrera potenciales
3. Optimizar el flujo de operaciones concurrentes

Diagramas de Comportamiento

Diagrama de Secuencia Principal

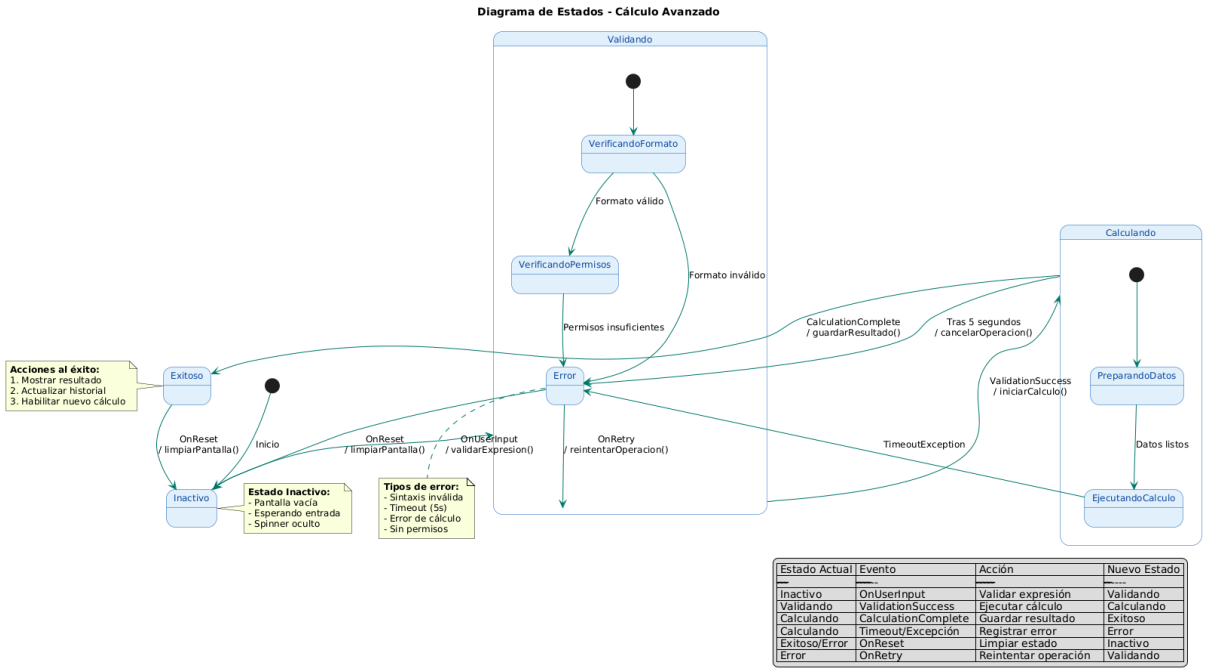


DatabaseMathJSBackendFrontendUsuarioDatabaseMathJSBackendFrontendUsuarioIngres a "sin(30)"POST /calculate {op:"sin", val:30}GET /?expr=sin(pi/6)0.5INSERT historialOK200 OK {result:0.5}Muestra resultado

Variantes:

1. Modo offline: Frontend usa cálculo local
2. Error de red: Reintentos con backoff exponencial

Diagrama de Estados (Cálculo Avanzado)



Input recibido

Input válido

Input inválido

Resultado OK

Timeout/Excepción

Reset

Tras 5 segundos

Inactivo

Validando

Calculando

Error

Exitoso

Tabla de Transiciones:

Estado Actual	Evento	Acción	Nuevo Estado

Inactivo	OnUserInput	Validar expresión	Validando
Validando	ValidationSuccess	Ejecutar cálculo	Calculando
Calculando	CalculationComplete	Guardar resultado	Exitoso

Patrones de Concurrencia

Las secuencias entre componentes clave siguen un flujo típico como:

1. Usuario genera evento (clic, teclado).
2. Interfaz detecta evento y lo pasa al gestor.
3. El gestor de eventos determina el tipo de acción a ejecutar.
4. La lógica de cálculo es invocada.
5. El resultado es devuelto y mostrado en pantalla.
6. Si aplica, se guarda en el historial.

Este patrón se repite y varía según el tipo de operación o estado actual de la aplicación.

Pool de Conexiones (Backend)

java
// Ejemplo HikariCP
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql:///calculator");
config.setMaximumPoolSize(20);
HikariDataSource ds = new HikariDataSource(config);
Métricas de Pool:

- Tamaño activo: 5-15 conexiones
- Tiempo máximo de espera: 500ms
- Tiempo de vida: 30 minutos

Web Workers (Frontend)

javascript
// Cálculo en segundo plano

```
const worker = new Worker('calc-worker.js');
worker.postMessage({op: 'fibonacci', n: 30});
worker.onmessage = (e) => {
  console.log('Resultado:', e.data);
};
```

Ventajas:

- No bloquea UI durante cálculos largos
- Aislamiento de fallos

Gestión de Errores

El sistema pasa por una serie de estados lógicos, que pueden representarse como una máquina de estados finita:

- Esperando entrada: Estado inicial del sistema.
- Validando: La entrada es revisada por el sistema.
- Calculando: El motor de operaciones procesa la expresión.
- Mostrando resultado: Se presenta el resultado al usuario.
- Error: El sistema muestra un mensaje de error y permite la corrección.

Cada transición está desencadenada por un evento.

Jerarquía de Excepciones

python

```
class CalculatorError(Exception):
  pass

class InvalidInputError(CalculatorError):
  pass

class OverflowError(CalculatorError):
  pass
```

Estrategia de Reintentos

typescript

```
async function calculateWithRetry(
    operation: string,
    maxRetries = 3,
    baseDelay = 1000
): Promise<number> {
    for (let i = 0; i < maxRetries; i++) {
        try {
            return await api.calculate(operation);
        } catch (err) {
            if (i === maxRetries - 1) throw err;
            await new Promise(r => setTimeout(r, baseDelay * (i + 1)));
        }
    }
}
```

Modelado de Cargas

Existen condiciones clave que afectan el flujo del sistema:

¿La expresión es válida? → Si no, ir al estado de error.

¿Está el sistema en modo offline? → Si sí, almacenar solo en local.

¿La operación es avanzada? → Enviar al backend, de lo contrario, procesar localmente.

Estas decisiones controlan qué componente se activa y cómo se encadena el siguiente paso del proceso.

Escenario: Hora Pico

bash

```
# Datos de carga simulada (Locust)
$ locust -f load_test.py --users 1000 --spawn-rate 100
```

Resultados Esperados:

Throughput: 500 ops/seg

Latencia p95: < 1s

Tasa de error: < 0.1%

Estrategia de Throttling

nginx

```
# Configuración NGINX
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=10r/s;

server {
    location /api/ {
        limit_req zone=api_limit burst=20;
    }
}
```

Flujos Complejos

Cálculo en Cadena

"2 + 3 * sin(30)":
1. Calcular sin(30) → 0.5
2. Calcular 3 * 0.5 → 1.5
3. Calcular 2 + 1.5 → 3.5

Diagrama:

```
plantuml
@startuml
start
:Calcular sin(30);
:Multiplicar por 3;
:Sumar 2;
stop
@enduml
```

Historial con Paginación

```
http
GET /api/history?page=2&size=10
```

Estados Involucrados:

1. Carga inicial
2. Scroll infinito
3. Actualización en tiempo real

Optimización de Rendimiento

Memoización

```
javascript
const memoizedSin = memoize((angle) => {
    return Math.sin(angle * Math.PI / 180);
```

});
Cache LRU:
Tamaño máximo: 100 entradas
TTL: 1 hora

Precalentamiento

bash
# Script de precalentamiento
for op in "sin(30)" "sqrt(16)"; do
curl -X POST "http://localhost/api/calculate" \
-H "Content-Type: application/json" \
-d "{\"operation\":\"$op\"}"
done

Seguridad en Tiempo de Ejecución

Sandboxing

javascript
const safeEval = new vm.Script(`
Math.${operation}(${value})
`, { timeout: 1000 });

Rate Limiting Dinámico

func adaptiveRateLimit() {
if systemLoad > 80% {
reduceRateBy(50%)
}
}

Monitorización en Producción

Métricas Clave

Métrica	Fuente	Umbral Crítico
Tiempo de CPU/op	Prometheus	> 300ms
Memoria heap	Grafana	> 80%

Cola de tareas pendientes	RabbitMQ	> 100

Dashboard Ejemplo

```
json
{
  "panels": [
    {
      "title": "Estado Cálculos",
      "type": "heatmap",
      "metrics": [
        "rate(api_calculations_total[5m])"
      ]
    }
  ]
}
```

5.10 Propósito del Punto de Vista de Recursos

Este punto analiza los requerimientos y gestión de recursos hardware/software, incluyendo:

- Asignación de infraestructura física y lógica
- Planificación de capacidad
- Estrategias de optimización
- Modelos de escalamiento

Objetivos clave:

1. Garantizar disponibilidad bajo carga máxima
2. Minimizar costos operacionales
3. Cumplir SLAs de rendimiento

Requerimientos de Infraestructura

El código del sistema está organizado en un repositorio estructurado por capas funcionales:

/frontend: Contiene todos los archivos relacionados con la interfaz de usuario.

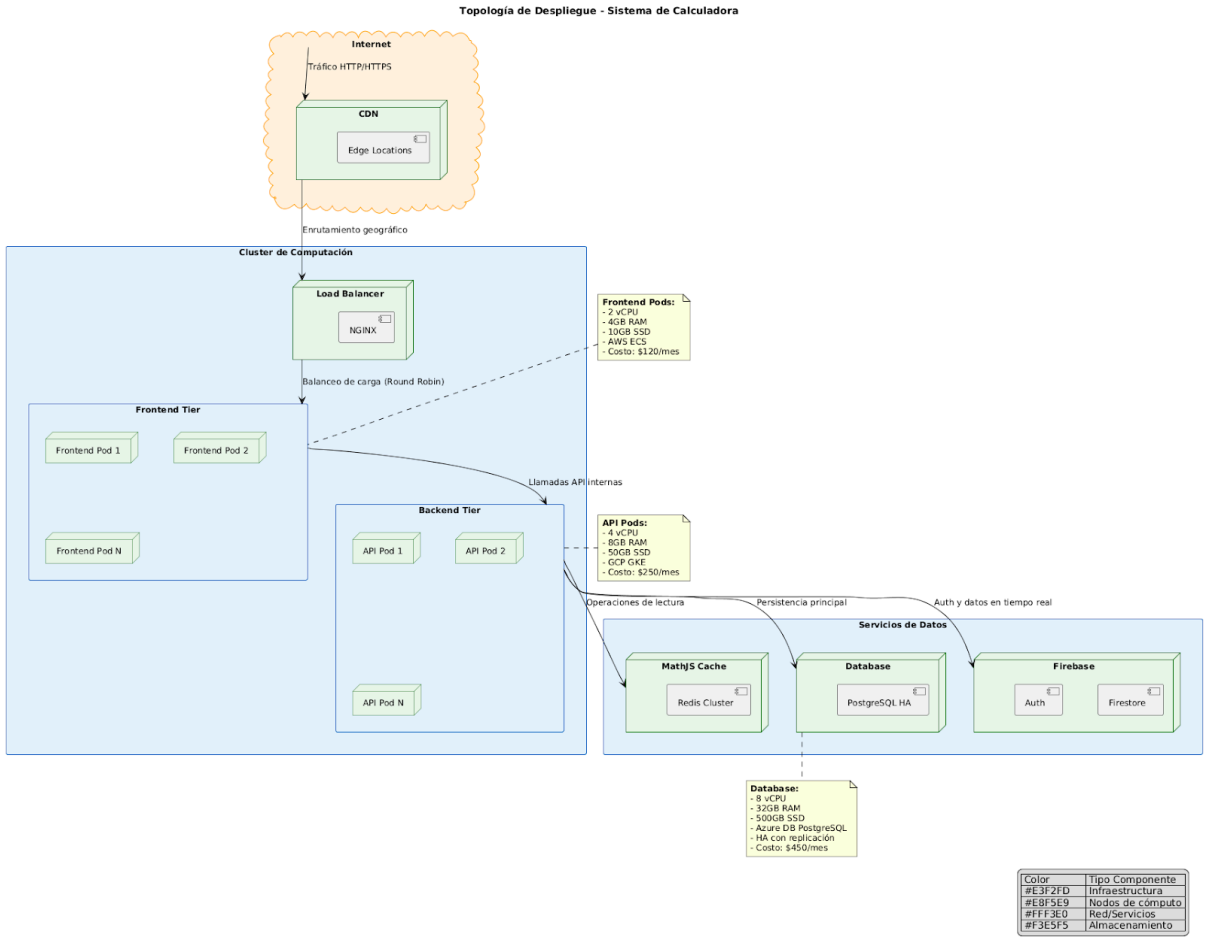
/backend: Incluye los servicios y la lógica de negocio, como los cálculos matemáticos y el historial.

/shared: Agrupa funciones y estructuras reutilizables entre frontend y backend (por ejemplo, validaciones o constantes).

/tests: Contiene las pruebas automatizadas unitarias e integradas.

Cada carpeta tiene submódulos claros y documentados, siguiendo una convención de nombres estándar.

Topología de Despliegue



CDN
Load Balancer
Frontend Pods
API Pods
MathJS Cache
PostgreSQL
Firebase

Especificaciones Técnicas

Componente	Requerimientos	Proveedor	Costo Mensual

Frontend	2 vCPU, 4GB RAM, 10GB SSD	AWS ECS	$120
Backend API	4 vCPU, 8GB RAM, 50GB SSD	GCP GKE	$250
Database	8 vCPU, 32GB RAM, 500GB SSD	Azure DB PostgreSQL	$450

Modelo de Escalamiento

Escalado Horizontal

bash

```
# Configuración Kubernetes HPA
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-scaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Umbrales:

- Escalar arriba: CPU > 70% por 5 min
- Escalar abajo: CPU < 30% por 30 min

Escalado Vertical

```
# Plan de escalamiento progresivo
```

```
resource "google_compute_instance" "api" {
  machine_type = "e2-standard-4" # 4vCPU, 16GB RAM
  allow_stopping_for_update = true

  lifecycle {
    replace_triggered_by = [
      google_monitoring_alert_policy.high_cpu.instance
    ]
  }
}
```

Optimización de Recursos

Perfilado de Rendimiento

python

```
# Script de profiling (Python)
import cProfile
import math

def calculate():
    [math.sin(x) for x in range(100000)]

cProfile.run('calculate()', sort='cumtime')
```

Resultados típicos:

- 75% tiempo en operaciones math.sin
- 20% en manejo de memoria

Estrategias de Cache

Capa	Tecnología	TTL	Hit Rate
CDN	Cloudflare	5 min	92%
Memoria	Redis	30 seg	85%
Disco	SQL Cache	1 hora	78%

Balanceo de Carga

Algoritmos Implementados

Algoritmo	Uso	Configuración
Round Robin	Tráfico general	peso igual por pod
Least Connections	Operaciones largas	preferir instancias menos cargadas
IP Hash	Sesiones persistentes	basado en IP cliente

Configuración NGINX

```
nginx
upstream backend {
    least_conn;
    server api1.example.com;
    server api2.example.com;
    keepalive 32;
}

server {
    location /api/ {
        proxy_pass http://backend;
    }
}
```

Planificación de Capacidad

Proyección de Crecimiento

Año	Usuarios Diarios	Operaciones/Día	Requerimientos CPU
2024	50,000	500,000	16 vCPU
2025	120,000	1,200,000	32 vCPU
2026	300,000	3,000,000	64 vCPU + GPU

Modelo de Costos

```
# Fórmula costo mensual
Costo = (vCPU * $0.04/h) + (RAM_GB * $0.01/h) + (Storage_GB * $0.10)
```

Ejemplo:
(16 * $0.04 * 720) + (32 * $0.01 * 720) + (500 * $0.10) = $460.80 + $230.40 + $50 = $741.20

Estrategia de Alta Disponibilidad

Distribución Geográfica

Región	AZs	Réplicas DB	Latencia
us-east-1	3	2	<50ms
eu-central-1	2	1	<100ms
ap-southeast-1	2	1	<150ms

Política de Backup

terraform
```
resource "aws_db_instance" "main" {
  backup_retention_period = 35 # días
  backup_window           = "07:00-09:00"
  maintenance_window      = "Sun:03:00-Sun:05:00"
}
```

Monitorización de Recursos

Métricas Clave

Métrica	Herramienta	Umbral Crítico
Uso CPU	Prometheus	>80% por 5 min
Memoria disponible	Grafana	<20% libre
Latencia red	CloudWatch	>500ms

Dashboard Ejemplo

json
```
{
```

```
  "title": "Estado Recursos",
  "panels": [
    {
      "type": "gauge",
      "title": "Uso CPU",
      "targets": [{
        "expr": "100 - (avg by(instance) (rate(node_cpu_seconds_total{mode='idle'}[5m])) *
100"
      }]
    }
  ]
}
```

Estrategia Verde-Azul

Implementación

bash

```
# Despliegue en Kubernetes
kubectl apply -f api-green.yaml
kubectl patch svc api -p '{"spec":{"selector":{"version":"green"}}}'
```

Ventajas:

- Cero downtime durante actualizaciones
- Rollback inmediato si hay fallos