# Sesión 4: Pruebas de regresión y pruebas de carga

**Duración:** 45 minutos

---

## 🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

- Implementar estrategias efectivas de pruebas de regresión
- Diseñar y ejecutar pruebas de carga con Locust
- Configurar pipelines de regresión automatizada
- Identificar y prevenir degradaciones de rendimiento
- Crear suites de golden tests y snapshot testing
- Monitorear métricas de rendimiento a lo largo del tiempo

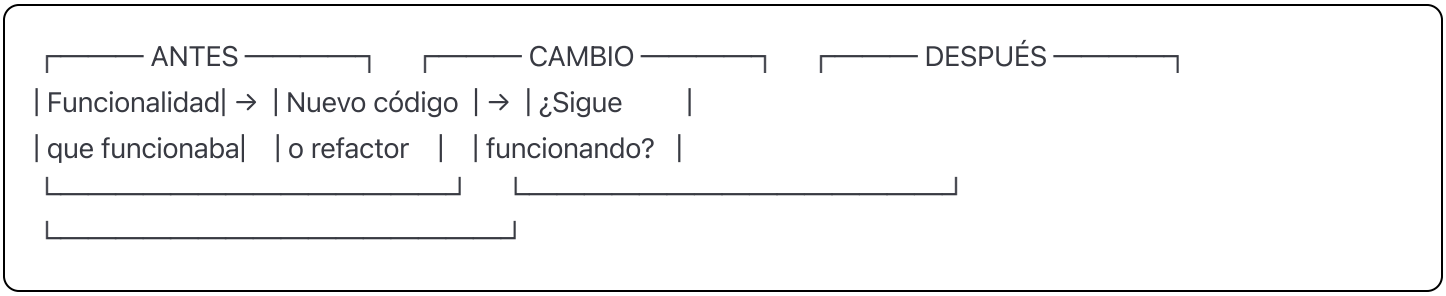
---

## 📋 Contenido

### 1. Pruebas de Regresión - Definición y Propósito (8 min)

**Analogía:** Las pruebas de regresión son como un sistema de alarma en tu casa. No previenen que alguien entre, pero te alertan inmediatamente cuando algo cambia sin autorización.

**¿Qué es regresión?**




**Tipos de regresión:**

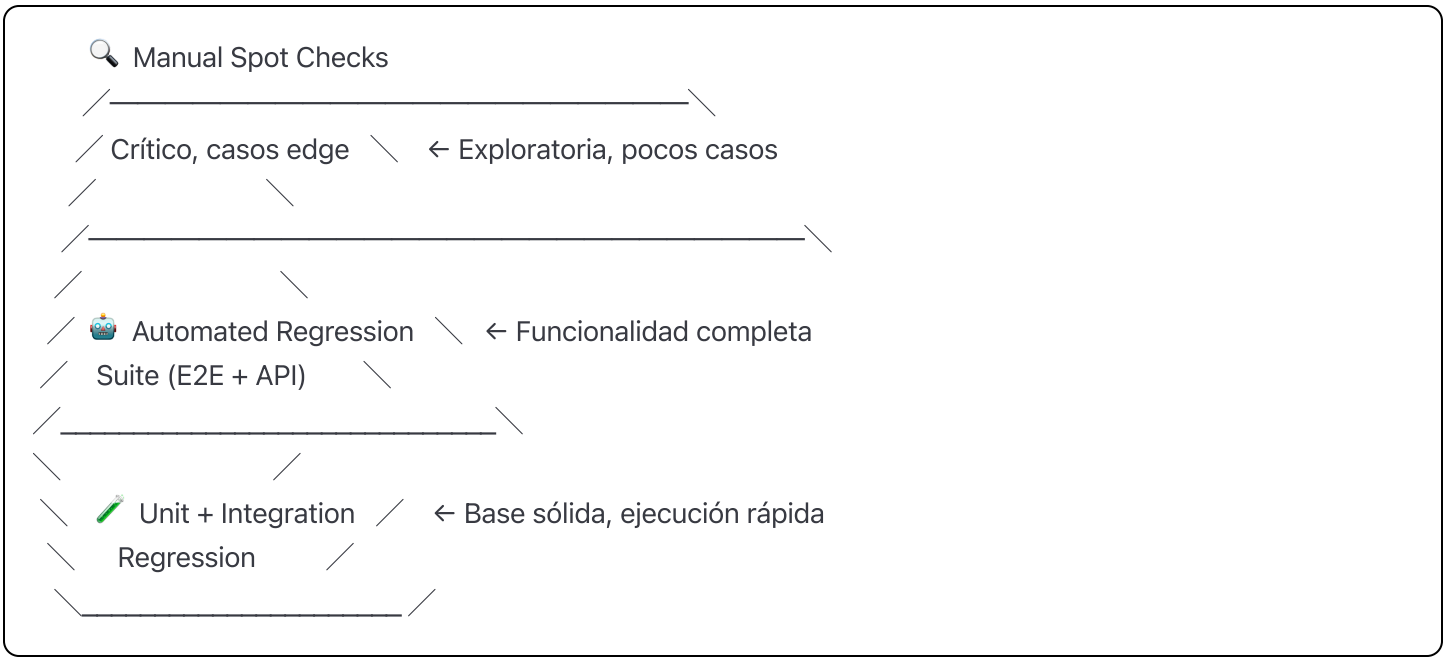
- **Regresión local:** Nueva funcionalidad rompe código existente
- **Regresión remota:** Cambio afecta módulos aparentemente no relacionados
- **Regresión de rendimiento:** Funcionalidad se vuelve más lenta
- **Regresión de configuración:** Cambios de entorno afectan comportamiento

**Cuándo ocurren regresiones:**

🔧 Refactoring → Cambios internos pueden romper contratos
🆕 Nuevas features → Interacciones imprevistas con código existente
🐛 Bug fixes → "Arreglar" un bug puede crear otros
📦 Dependency updates → Nuevas versiones cambian comportamiento
🔧 Infrastructure → Cambios de servidor/BD/red

## 2. Estrategias de Pruebas de Regresión (10 min)

### Pirámide de regresión:




### 1. Smoke Tests (Pruebas de humo):

- Verifican funcionalidad **básica crítica**
- Se ejecutan **primero** en cada deploy
- **Rápidas** (< 5 minutos)
- **Falla rápido** si algo está muy roto

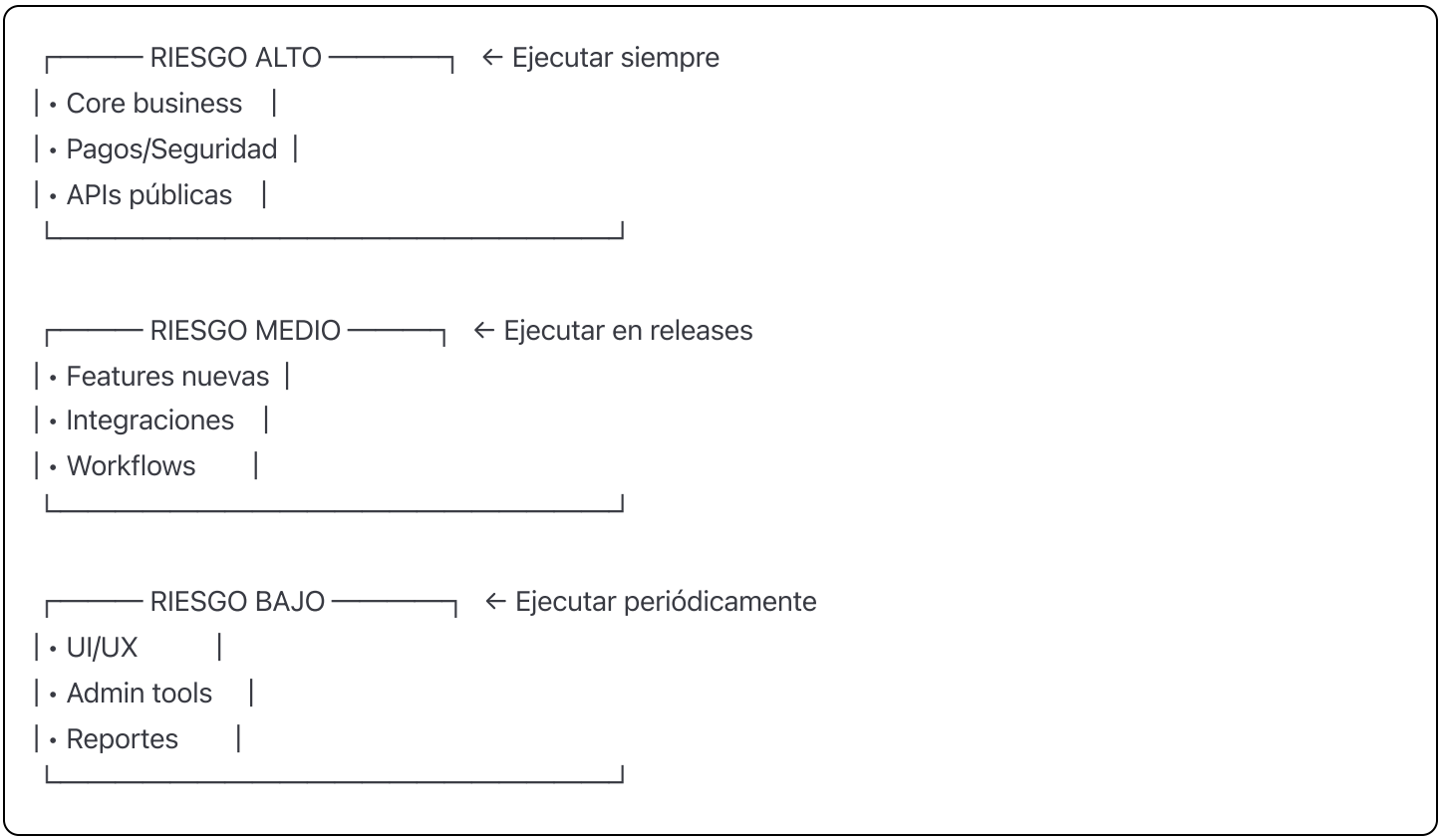
### 2. Golden Tests / Snapshot Testing:

```python
# Golden test: compara output actual vs output "dorado"
def test_user_report_golden():
    actual_report = generate_user_report(test_data)
    expected_report = load_golden_file("user_report.json")
    assert actual_report == expected_report
```

**3. Regression Test Suite:**

- **Casos históricos:** Bugs que se arreglaron antes
- **Edge cases críticos:** Casos límite importantes
- **Happy path completo:** Flujos principales
- **Configuraciones múltiples:** Diferentes entornos

**4. Estrategias de selección:**




## 3. Implementación de Regresión Automatizada (7 min)

**Configuración de pipeline:**

yaml

```yaml
# .github/workflows/regression.yml
name: Regression Testing
on:
  pull_request:
    branches: [main]
  schedule:
    - cron: "0 2 * * *"  # Diario a las 2 AM

jobs:
  smoke-tests:
    runs-on: ubuntu-latest
    steps:
      - name: Smoke Tests
        run: pytest tests/smoke/ --maxfail=1
        timeout-minutes: 5

  regression-suite:
    needs: smoke-tests
    runs-on: ubuntu-latest
    steps:
      - name: Full Regression
        run: pytest tests/regression/ -v
        timeout-minutes: 30
```

**Organización de tests:**

```
tests/
├── smoke/              ← Críticos, rápidos
│   ├── test_core_api.py
│   └── test_basic_auth.py
├── regression/         ← Suite completa
│   ├── historical/     ← Bugs del pasado
│   ├── golden/         ← Snapshot tests
│   └── scenarios/      ← Casos complejos
└── data/
    ├── golden_outputs/ ← Archivos de referencia
    └── regression_data/ ← Datos de prueba
```

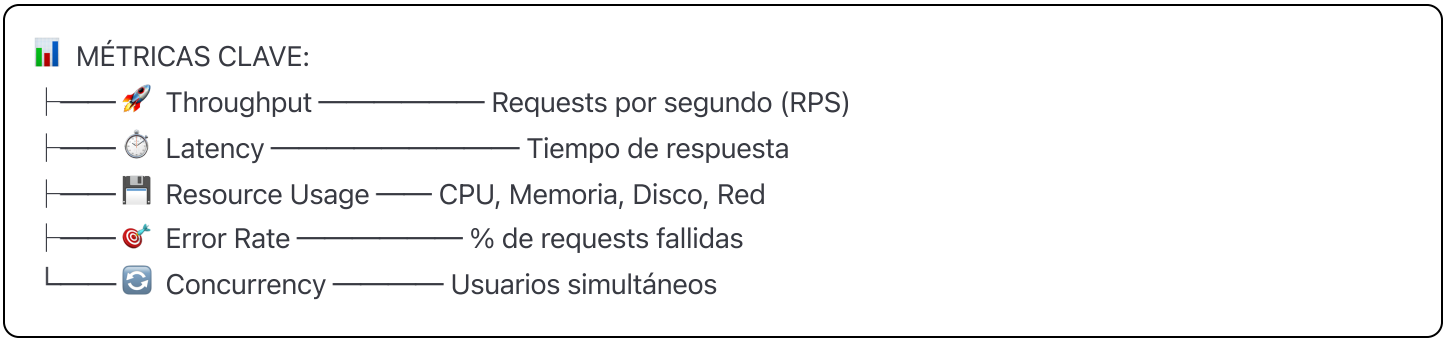
**Métricas de regresión:**

- **Test failure rate:** % de tests que fallan
- **Flaky test detection:** Tests que fallan aleatoriamente

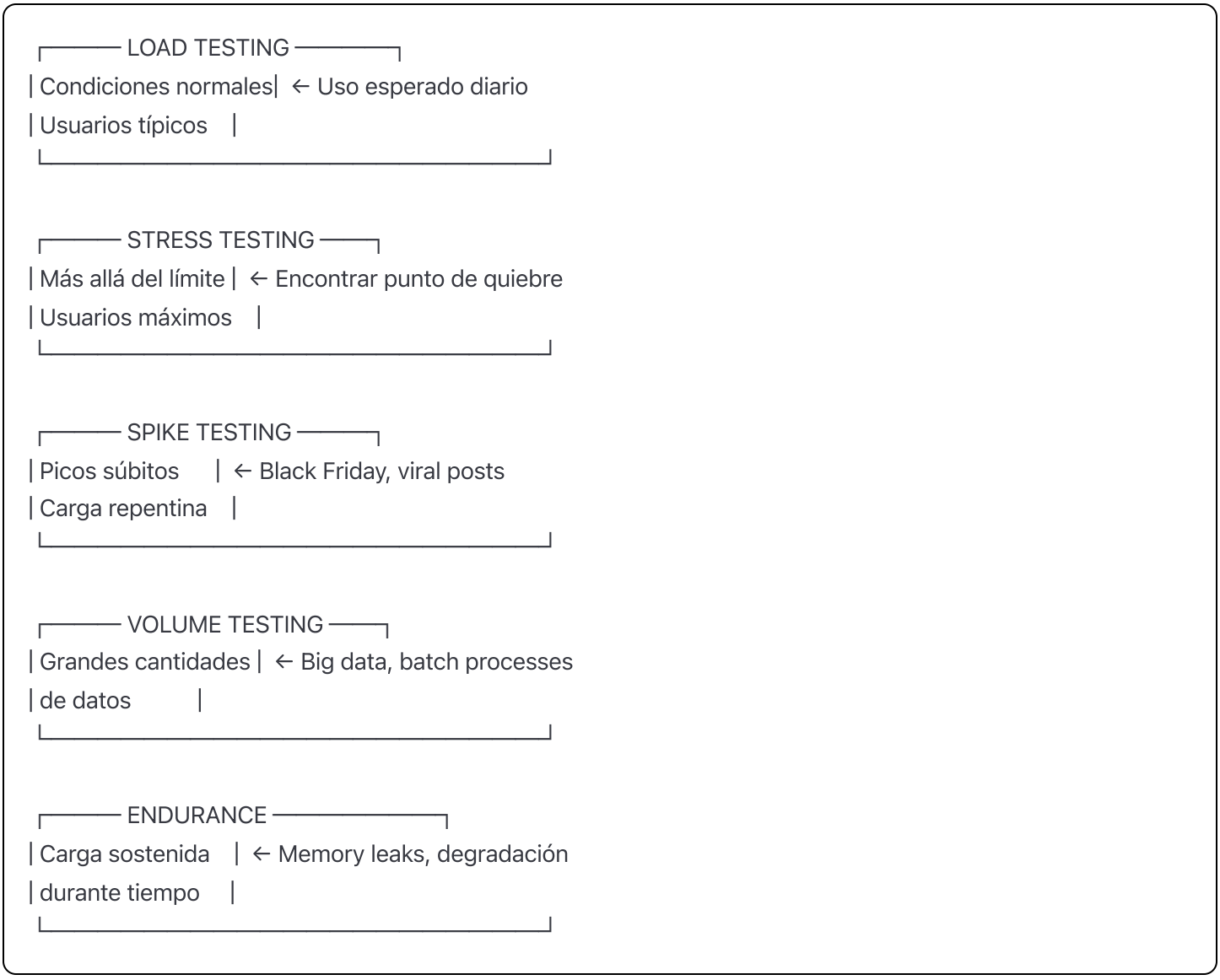
- **Coverage drift:** Pérdida de cobertura con el tiempo
- **Execution time trends:** Crecimiento del tiempo de ejecución

## 4. Pruebas de Carga - Fundamentos (10 min)

### ¿Qué miden las pruebas de carga?




### Tipos de pruebas de rendimiento:




### Herramientas populares:

- **Locust** 🐍 - Python, flexible, distribuido
- **JMeter** ☕ - Java, GUI, reportes ricos
- **k6** 🚀 - JavaScript, cloud-native
- **Artillery** 🎯 - Node.js, real-time
- **wrk** ⚡ - C, ultra-rápido, simple

## 5. Locust - Implementación Práctica (8 min)

**Ventajas de Locust:**

- **Escrito en Python** → Fácil integración con el ecosistema
- **Distribución automática** → Escalar a múltiples máquinas
- **Web UI interactiva** → Monitoreo en tiempo real
- **Scripting flexible** → Comportamientos complejos de usuario
- **Reportes integrados** → CSV, HTML, tiempo real

**Anatomía de un test de Locust:**

```python
```

```python
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(1, 3)  # Pausa entre requests

    def on_start(self):
        """Setup inicial para cada usuario"""
        self.login()

    @task(3)  # Peso 3: se ejecuta 3x más frecuente
    def view_homepage(self):
        self.client.get("/")

    @task(1)  # Peso 1: menos frecuente
    def create_post(self):
        self.client.post("/posts", json={
            "title": "Test Post",
            "content": "Content"
        })

    def login(self):
        response = self.client.post("/login", {
            "username": "testuser",
            "password": "password"
        })
        # Guardar token para requests subsecuentes
```

**Patrones de carga realistas:**

```python
```

```python
# Patrón escalonado
class SteppedLoadUser(HttpUser):
    wait_time = between(0.5, 2)

    @task
    def realistic_user_journey(self):
        # 1. Llegar al sitio
        self.client.get("/")

        # 2. Buscar algo
        self.client.get("/search?q=python")

        # 3. Ver resultado
        self.client.get("/products/123")

        # 4. Añadir al carrito (menos frecuente)
        if random.random() < 0.3:  # 30% de usuarios
            self.client.post("/cart/add", {"product_id": 123})
```

**Ejecución y escalabilidad:**

```bash
# Ejecución local
locust -f loadtest.py --host=http://localhost:8000

# Ejecución distribuida (master)
locust -f loadtest.py --master --host=http://production.com

# Workers (en diferentes máquinas)
locust -f loadtest.py --worker --master-host=master-ip
```

## 6. Análisis e Interpretación de Resultados (2 min)

**Métricas críticas a monitorear:**

🎯 Latencia (ms):
├── P50 (Median) ──── 50% de requests más rápidas que X
├── P90 ──────────────── 90% de requests más rápidas que X
├── P99 ──────────────── 99% de requests más rápidas que X
└── Max ──────────────── Request más lenta

📈 Throughput:
├── RPS promedio ── Requests por segundo sostenible
├── RPS máximo ──── Pico alcanzado
└── Degradación ─── Caída bajo carga

🚨 Errores:
├── Rate ──────────── % de requests fallidas
├── Types ─────────── HTTP 500, timeouts, conexiones
└── Patterns ──────── ¿Aumentan con la carga?

---

## ⚠️ Antipatrones y Errores Comunes

### ❌ En Regresión:

- **Solo happy path** → Faltan edge cases críticos
- **Tests demasiado frágiles** → Fallan por cambios menores de UI
- **Sin priorización** → Ejecutar todo siempre es insostenible
- **Ignorar tests flakey** → Reducen confianza del equipo

### ❌ En Load Testing:

- **Datos sintéticos irreales** → No reflejan producción
- **Entorno de prueba diferente** → Resultados no extrapolables
- **Una sola métrica** → Optimizar solo RPS ignorando latencia
- **Sin baseline** → No hay referencia para comparar

### ✅ Mejores prácticas:

- **Test data realista** → Usar datos de producción (anonimizados)
- **Entorno similar a producción** → Misma configuración de infraestructura
- **Métricas holísticas** → RPS + Latencia + Recursos + Errores
- **Automatización continua** → Ejecutar en cada release

---

## 🔧 Herramientas del Ecosistema

**Para Regresión:**

- `pytest-regressions` - Snapshot testing automático
- `pytest-benchmark` - Benchmarking de funciones
- `pytest-html` - Reportes HTML bonitos
- `allure-pytest` - Reportes ejecutivos avanzados

**Para Load Testing:**

- `locust` - Framework principal
- `pytest-benchmark` - Microbenchmarks de funciones
- `memory-profiler` - Análisis de memoria
- `py-spy` - Profiling de aplicaciones en producción

**Monitoreo continuo:**

- `Grafana` + `Prometheus` - Dashboards de métricas
- `New Relic` / `DataDog` - APM comerciales
- `Jaeger` - Distributed tracing
- `cProfile` + `snakeviz` - Profiling local

---

## 📝 Resumen Final

1. **Pruebas de regresión** previenen que nuevos cambios rompan funcionalidad existente
2. **Estrategia en capas**: Smoke tests → Regression suite → Manual spot checks
3. **Golden tests** permiten detectar cambios no intencionales en outputs
4. **Automatización** es crítica para ejecutar regresión continuamente
5. **Pruebas de carga** miden rendimiento bajo diferentes condiciones
6. **Locust** ofrece scripting flexible para patrones realistas de usuario
7. **Métricas múltiples** (latencia, throughput, errores) dan visión completa
8. **Entorno realista** es esencial para resultados útiles

---

## 🏃 Actividad Práctica Sugerida

**Ejercicio 1 - Regresión:**

1. Crear 3-5 golden tests para outputs de tu aplicación
2. Configurar pipeline que ejecute smoke tests en <5 min
3. Simular una regresión y verificar que se detecte

**Ejercicio 2 - Load Testing:**

1. Escribir script de Locust para un endpoint de tu API
2. Ejecutar prueba de carga con 10-50 usuarios concurrentes
3. Analizar resultados y identificar bottlenecks

**Tiempo estimado:** 30-35 minutos adicionales