drop-load-testing-spec

Drop Load Testing & Performance Benchmarks Specification

Version: 1.0 **Date:** 2026-02-17 **Author:** architect (Sonnet 4.5) **Status:** Draft **MC Task:** #1200

---

1. Executive Summary

This specification defines the load testing strategy, performance benchmarks, and capacity planning for the Drop fintech application. The goal is to establish baseline performance metrics, identify bottlenecks, and ensure the system can handle realistic user load before production scale deployment.

Tool Choice: k6 (Grafana k6) — open-source, JavaScript-based, excellent Grafana ecosystem integration, active community.

Performance Target: Support 100 concurrent users (baseline), 500 concurrent users (peak), 1000 concurrent users (stress test) with P95 API response times < 300ms and P99 < 500ms.

Timeline: Phased implementation over 5 days (see Section 11).

---

2. Tool Selection: k6 vs Artillery vs Gatling

2.1 Comparison Matrix

Criterion	k6	Artillery	Gatling	Winner

Language	JavaScript (Go runtime)	Node.js/YAML/JS	Scala DSL	k6
Learning Curve	Medium (JS familiarity)	Low (YAML config)	High (Scala)	Artillery
Performance	Excellent (Go runtime)	Good (Node.js)	Excellent (JVM)	k6/Gatling
Scripting Flexibility	High (JS API)	High (JS + YAML)	Medium (Scala DSL)	k6
CI/CD Integration	Native support	Native support	Native support	Tie
Reporting	Grafana, JSON, HTML	JSON, HTML, plugins	HTML, Gatling Cloud	k6
Open Banking/PSD2 Testing	Full control (HTTP/headers)	Full control	Full control	Tie
Next.js 16 Compatibility	Excellent (HTTP/REST)	Excellent	Excellent	Tie
Local + Cloud	Yes (k6 + k6 Cloud)	Yes (Artillery + Cloud)	Yes	Tie
Community & Docs	Strong (Grafana Labs)	Strong (Artillery.io)	Strong (Gatling Corp)	Tie
Cost	Free (OSS)	Free (OSS)	Free (OSS)	Tie

2.2 Recommendation: **k6**

Rationale:

1. **JavaScript-based scripting** — Drop team already uses JS/TS (Next.js, React), no new language learning required
2. **Grafana ecosystem** — Drop will use Grafana for production monitoring (future), k6 integrates natively
3. **Performance** — Go runtime provides excellent performance for simulating 1000+ concurrent users on a single machine
4. **Flexibility** — Full HTTP control for PSD2 API testing (custom headers, SCA flows, OAuth)
5. **Active development** — Grafana Labs maintains k6, frequent updates, strong community
6. **CI/CD ready** — GitHub Actions integration out of the box

Artillery is a close second (easier YAML config), but k6's Grafana integration and Go performance edge win for a fintech app that will scale.

Gatling is excellent but requires Scala knowledge (team uses JS/TS stack).

Sources:

- Load Testing PoC: k6 vs Artillery vs Locust vs Gatling
- Artillery vs k6 - Fork My Brain
- k6 Documentation

---

3. Performance Requirements (SLA Targets)

3.1 API Response Time Targets

Based on PSD2 best practices and fintech industry standards:

Endpoint Type	P50	P95	P99	Rationale
Auth (login, register, logout)	< 150ms	< 250ms	< 400ms	Critical path, user expects instant
Payments (remittance, QR payment)	< 200ms	< 300ms	< 500ms	PSD2 real-time payment expectation
Account Info (AISP balance read)	< 100ms	< 200ms	< 350ms	Cached data, should be fast
Transaction History	< 150ms	< 250ms	< 400ms	Read-heavy, paginated
Admin (audit log, KYC review)	< 300ms	< 500ms	< 800ms	Lower priority, internal use
Health Check	< 50ms	< 100ms	< 150ms	Uptime monitoring

Note on PSD2 Latency:

- PSD3 (upcoming) emphasizes API uptime, latency, and error rate standards (PSD2 had API performance issues)
- Real-time payment initiation expected under PSD3
- Target: **95% of payment initiations complete within 300ms** (excluding external bank processing time)

Sources:

- What Should You Expect from the New PSD3 Rules?
- PSD2 Compliance: Impacts & Solutions

3.2 Page Load Time Targets

Metric	Target	Tool
FCP (First Contentful Paint)	< 1.5s	Lighthouse
LCP (Largest Contentful Paint)	< 2.5s	Lighthouse
TTFB (Time to First Byte)	< 200ms	k6, Lighthouse
TTI (Time to Interactive)	< 3.5s	Lighthouse
CLS (Cumulative Layout Shift)	< 0.1	Lighthouse

Next.js 16 Real-World Benchmark:

- Mobile LCP reduced from 26.4s to 0.9s after Next.js 16 migration (218% boost)
- Turbopack dev server: instant HMR

Sources:

- I Migrated to Next.js 16 and Got 218% Performance Boost
- Next.js Performance Optimization Guide

3.3 Concurrent User Capacity

Load Profile	Concurrent Users	Duration	Success Criteria
Baseline	100	10 min	P95 < 300ms, 0% errors
Peak	500	10 min	P95 < 300ms, < 1% errors
Stress	1000	5 min	P95 < 500ms, < 5% errors
Spike	0→500 in 30s	5 min	P95 < 400ms, < 2% errors

Norwegian Market Context:

- Population: ~5.5M
- Realistic Year 1 target: 10K-100K active users
- Peak concurrent (0.5% of active): 50-500 users
- **100 concurrent = realistic baseline, 500 = peak, 1000 = stress test**

3.4 Database Performance Targets

Metric	SQLite (Demo)	PostgreSQL (Prod)
Query P95	< 10ms	< 5ms

Metric	SQLite (Demo)	PostgreSQL (Prod)
Concurrent Writes	1 (serialized)	100+ (MVCC)
Max Connections	1	100 (pgBouncer pool)
Throughput	~1K writes/sec	~10K writes/sec

SQLite Limitation:

- SQLite uses file-level locking → **only 1 write at a time** (WAL mode allows concurrent reads)
- Under load: write contention causes SQLITE_BUSY errors
- **Conclusion:** SQLite is fine for demo/MVP (< 50 concurrent users), PostgreSQL required for production (> 100 concurrent users)

Sources:

- SQLite vs PostgreSQL Performance
- PostgreSQL vs SQLite Concurrency

---

4. Load Test Scenarios

4.1 Scenario 1: User Registration & Login Flow

User Journey:

1. User visits `/register`
2. Submits phone + PIN + name + DOB
3. Receives OTP (mocked)
4. Verifies OTP → account created
5. Redirected to `/login`
6. Logs in with phone + PIN
7. Receives JWT in httpOnly cookie
8. Redirected to `/dashboard`

k6 Script: `tests/load/scenarios/auth-flow.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');
```

```
export const options = {
  stages: [
    { duration: '2m', target: 100 }, // Ramp up to 100 users
    { duration: '5m', target: 100 }, // Stay at 100 users
    { duration: '2m', target: 0 },   // Ramp down
  ],
  thresholds: {
    http_req_duration: ['p(95)<250'], // 95% under 250ms
    errors: ['rate<0.01'],             // Error rate < 1%
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

export default function () {
  const phone = `+4740${Math.floor(Math.random() * 1000000).toString().padStart(6, '0')}`;
  const pin = '1234';

  // Step 1: Register
  let registerRes = http.post(`${BASE_URL}/api/auth/register`, JSON.stringify({
    phone,
    pin,
    firstName: 'Load',
    lastName: 'Test',
    dateOfBirth: '1990-01-01',
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  check(registerRes, {
    'register status 200': (r) => r.status === 200,
  }) || errorRate.add(1);

  sleep(1);

  // Step 2: Verify OTP (mocked — in demo, OTP is auto-verified)
  // Skip for demo

  // Step 3: Login
```

```
  let loginRes = http.post(`${BASE_URL}/api/auth/login`, JSON.stringify({
    phone,
    pin,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  check(loginRes, {
    'login status 200': (r) => r.status === 200,
    'has JWT cookie': (r) => r.cookies.token !== undefined,
  }) || errorRate.add(1);

  sleep(2);
}
```

Expected Results:

- **100 users:** P95 < 250ms, 0% errors
- **500 users:** P95 < 300ms, < 1% errors (SQLite may show write contention)
- **1000 users:** Expect SQLITE_BUSY errors (write serialization)

4.2 Scenario 2: Send Money (Remittance) Flow

User Journey:

1. User logged in (JWT cookie)
2. Visits `/send`
3. Selects recipient (or creates new)
4. Enters amount + currency (NOK → RSD, BAM, EUR, etc.)
5. Reviews exchange rate quote
6. Confirms transfer (PISP initiated)
7. Receives transaction confirmation
8. Redirected to `/transactions`

k6 Script: `tests/load/scenarios/send-money-flow.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');
```

```
export const options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '5m', target: 100 },
    { duration: '1m', target: 0 },
  ],
  thresholds: {
    http_req_duration: ['p(95)<300'],
    errors: ['rate<0.01'],
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

export function setup() {
  // Create a test user and get JWT
  const phone = '+4740999999';
  const pin = '1234';

  const registerRes = http.post(`${BASE_URL}/api/auth/register`, JSON.stringify({
    phone,
    pin,
    firstName: 'Load',
    lastName: 'Test',
    dateOfBirth: '1990-01-01',
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  const loginRes = http.post(`${BASE_URL}/api/auth/login`, JSON.stringify({
    phone,
    pin,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  const token = loginRes.cookies.token[0].value;
  return { token };
}
```

```
export default function (data) {
  const { token } = data;

  // Step 1: Get exchange rate quote
  let rateRes = http.get(`${BASE_URL}/api/rates/RSD`, {
    headers: { Cookie: `token=${token}` },
  });

  check(rateRes, {
    'rate status 200': (r) => r.status === 200,
  }) || errorRate.add(1);

  const rate = JSON.parse(rateRes.body).rate;

  sleep(1);

  // Step 2: Create recipient (or reuse existing)
  let recipientRes = http.post(`${BASE_URL}/api/recipients`, JSON.stringify({
    name: 'Test Recipient',
    country: 'RS',
    currency: 'RSD',
    bankAccount: '160-123456-78',
    bankName: 'Banca Intesa',
  }), {
    headers: {
      'Content-Type': 'application/json',
      Cookie: `token=${token}`,
    },
  });

  check(recipientRes, {
    'recipient created': (r) => r.status === 200 || r.status === 201,
  }) || errorRate.add(1);

  const recipientId = JSON.parse(recipientRes.body).id;

  sleep(1);

  // Step 3: Initiate transfer (PISP)
```

```
  let transferRes = http.post(`${BASE_URL}/api/transactions/remittance`, JSON.stringify({
    recipientId,
    sendAmount: 1000, // NOK
    sendCurrency: 'NOK',
    receiveCurrency: 'RSD',
    exchangeRate: rate,
  }), {
    headers: {
      'Content-Type': 'application/json',
      Cookie: `token=${token}`,
    },
  });

  check(transferRes, {
    'transfer status 200': (r) => r.status === 200,
    'transaction created': (r) => JSON.parse(r.body).id !== undefined,
  }) || errorRate.add(1);

  sleep(2);
}
```

Expected Results:

- **100 users:** P95 < 300ms, 0% errors
- **500 users:** P95 < 400ms, < 1% errors (SQLite write contention)
- **1000 users:** High error rate (SQLITE_BUSY), P95 > 500ms

4.3 Scenario 3: QR Payment Flow

User Journey:

1. User logged in
2. Merchant generates QR code (merchant dashboard)
3. User visits `/scan`
4. Scans QR code (camera permission)
5. Reviews payment amount
6. Confirms payment (PISP initiated)
7. Receives confirmation
8. Merchant dashboard updates (real-time)

k6 Script: `tests/load/scenarios/qr-payment-flow.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');

export const options = {
  stages: [
    { duration: '1m', target: 50 },
    { duration: '5m', target: 100 },
    { duration: '1m', target: 0 },
  ],
  thresholds: {
    http_req_duration: ['p(95)<300'],
    errors: ['rate<0.01'],
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

export function setup() {
  // Create merchant account + user account
  // Merchant generates QR code
  // Return { userToken, merchantQrCode }

  // Simplified for demo: assume QR code exists
  const phone = '+4740888888';
  const pin = '1234';

  const loginRes = http.post(`${BASE_URL}/api/auth/login`, JSON.stringify({
    phone,
    pin,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  const token = loginRes.cookies.token[0].value;

  // Mock QR code (merchant ID + amount)
  const qrCode = 'merchant123:amount500:currency:NOK';
```

```
  return { token, qrCode };
}

export default function (data) {
  const { token, qrCode } = data;

  // Step 1: Decode QR (client-side in real app, server validation here)
  const [merchantId, amountStr, , currency] = qrCode.split(':');
  const amount = parseFloat(amountStr.replace('amount', ''));

  sleep(1);

  // Step 2: Initiate QR payment (PISP)
  let paymentRes = http.post(`${BASE_URL}/api/transactions/qr-payment`, JSON.stringify({
    merchantId,
    amount,
    currency,
  }), {
    headers: {
      'Content-Type': 'application/json',
      Cookie: `token=${token}`,
    },
  });

  check(paymentRes, {
    'payment status 200': (r) => r.status === 200,
    'transaction created': (r) => JSON.parse(r.body).id !== undefined,
  }) || errorRate.add(1);

  sleep(2);
}
```

Expected Results:

- **100 users:** P95 < 300ms, 0% errors
- **500 users:** P95 < 400ms, < 2% errors
- **1000 users:** High error rate (SQLite limit)

4.4 Scenario 4: Bank Account Sync (AISP Call Simulation)

User Journey:

1. User logged in
2. Visits `/accounts`
3. App triggers AISP balance sync (background)
4. Balance updated in `bank_accounts` table
5. Dashboard shows fresh balance

k6 Script: `tests/load/scenarios/bank-sync-flow.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');

export const options = {
  stages: [
    { duration: '2m', target: 200 },
    { duration: '5m', target: 200 },
    { duration: '2m', target: 0 },
  ],
  thresholds: {
    http_req_duration: ['p(95)<200'], // Cached reads should be fast
    errors: ['rate<0.01'],
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

export function setup() {
  // Login user
  const phone = '+4740777777';
  const pin = '1234';

  const loginRes = http.post(`${BASE_URL}/api/auth/login`, JSON.stringify({
    phone,
```

```
    pin,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  const token = loginRes.cookies.token[0].value;
  return { token };
}

export default function (data) {
  const { token } = data;

  // Step 1: GET bank accounts (triggers sync in background)
  let accountsRes = http.get(`${BASE_URL}/api/user/account`, {
    headers: { Cookie: `token=${token}` },
  });

  check(accountsRes, {
    'accounts status 200': (r) => r.status === 200,
    'has balance': (r) => JSON.parse(r.body).balance !== undefined,
  }) || errorRate.add(1);

  sleep(5); // User stays on page
}
```

Expected Results:

- **200 users:** P95 < 200ms, 0% errors (reads are concurrent in SQLite WAL mode)
- **500 users:** P95 < 250ms, < 0.5% errors
- **1000 users:** P95 < 300ms, < 1% errors

4.5 Scenario 5: Dashboard Load (Mixed Read Endpoints)

User Journey:

1. User logs in
2. Dashboard loads:
   - User account info (`GET /api/auth/me`)
   - Bank account balance (`GET /api/user/account`)

- Last 5 transactions (`GET /api/transactions?limit=5`)
- Notifications (`GET /api/notifications?limit=10`)

3. All parallel requests (simulates real dashboard load)

k6 Script: `tests/load/scenarios/dashboard-load.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate } from 'k6/metrics';

const errorRate = new Rate('errors');

export const options = {
  stages: [
    { duration: '2m', target: 300 },
    { duration: '5m', target: 300 },
    { duration: '2m', target: 0 },
  ],
  thresholds: {
    http_req_duration: ['p(95)<250'],
    errors: ['rate<0.01'],
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

export function setup() {
  const phone = '+4740666666';
  const pin = '1234';

  const loginRes = http.post(`${BASE_URL}/api/auth/login`, JSON.stringify({
    phone,
    pin,
  }), {
    headers: { 'Content-Type': 'application/json' },
  });

  const token = loginRes.cookies.token[0].value;
  return { token };
}
```

```
export default function (data) {
  const { token } = data;
  const headers = { Cookie: `token=${token}` };

  // Parallel requests (batch)
  const responses = http.batch([
    ['GET', `${BASE_URL}/api/auth/me`, null, { headers }],
    ['GET', `${BASE_URL}/api/user/account`, null, { headers }],
    ['GET', `${BASE_URL}/api/transactions?limit=5`, null, { headers }],
    ['GET', `${BASE_URL}/api/notifications?limit=10`, null, { headers }],
  ]);

  check(responses[0], { 'me status 200': (r) => r.status === 200 }) || errorRate.add(1);
  check(responses[1], { 'account status 200': (r) => r.status === 200 }) || errorRate.add(1);
  check(responses[2], { 'transactions status 200': (r) => r.status === 200 }) ||
errorRate.add(1);
  check(responses[3], { 'notifications status 200': (r) => r.status === 200 }) ||
errorRate.add(1);

  sleep(10); // User stays on dashboard
}
```

Expected Results:

- **300 users:** P95 < 250ms, 0% errors (all reads, SQLite handles well)
- **500 users:** P95 < 300ms, < 0.5% errors
- **1000 users:** P95 < 400ms, < 1% errors

---

5. Load Profiles (Realistic Traffic Patterns)

5.1 Baseline Load (100 Concurrent Users)

Scenario: Normal weekday traffic **Duration:** 30 minutes **Ramp:** Linear over 5 minutes

```
export const options = {
  stages: [
```

```
    { duration: '5m', target: 100 },  // Ramp up
    { duration: '20m', target: 100 }, // Sustain
    { duration: '5m', target: 0 },    // Ramp down
  ],
};
```

Expected Behavior:

- All endpoints P95 < 300ms
- 0% error rate
- CPU < 50%, Memory < 2GB
- Database: no contention

5.2 Peak Load (500 Concurrent Users)

Scenario: Weekend evening peak (users sending remittances) **Duration:** 15 minutes **Ramp:** Linear over 3 minutes

```
export const options = {
  stages: [
    { duration: '3m', target: 500 },
    { duration: '10m', target: 500 },
    { duration: '2m', target: 0 },
  ],
};
```

Expected Behavior:

- API endpoints P95 < 300ms
- Write-heavy endpoints (remittance, QR payment) may see P95 < 400ms (SQLite write contention)
- Error rate < 1%
- CPU 60-80%, Memory 3-4GB
- Database: SQLite write serialization causes occasional SQLITE_BUSY (PostgreSQL recommended)

5.3 Stress Test (1000 Concurrent Users)

Scenario: Black Friday sale (merchant QR payments spike) **Duration:** 10 minutes **Ramp:** Linear over 2 minutes

```
export const options = {
  stages: [
    { duration: '2m', target: 1000 },
    { duration: '5m', target: 1000 },
    { duration: '3m', target: 0 },
  ],
};
```

Expected Behavior:

- **SQLite will break** — expect high error rate (> 5%)
- API endpoints P95 > 500ms
- Database: SQLITE_BUSY errors on writes
- **Conclusion:** PostgreSQL required for 1000+ concurrent users

---

5.4 Spike Test (0 ? 500 in 30 Seconds)

Scenario: Marketing campaign goes viral (sudden traffic surge) **Duration:** 10 minutes

```
export const options = {
  stages: [
    { duration: '30s', target: 500 }, // Spike
    { duration: '5m', target: 500 },  // Sustain
    { duration: '2m', target: 0 },
  ],
};
```

Expected Behavior:

- Initial spike: P95 may exceed 400ms (cold start, cache warming)
- After 1 minute: P95 stabilizes < 300ms
- Error rate < 2% during spike, < 1% after stabilization
- **Key metric:** How quickly does the system recover from sudden load?

---

6. Database Performance Considerations

6.1 SQLite Limitations

Aspect	SQLite Behavior	Impact at Scale
Write Concurrency	1 write at a time (file lock)	SQLITE_BUSY under load
Read Concurrency	Unlimited (WAL mode)	Good read performance
Connection Pool	N/A (file-based)	No connection overhead
Transactions	Serialized	Bottleneck for payments
Max Throughput	~1K writes/sec (SSD)	Insufficient for 500+ users

Recommendation:

- **Demo/MVP (< 50 concurrent users):** SQLite is fine
- **Production (> 100 concurrent users):** Migrate to PostgreSQL

6.2 PostgreSQL Optimizations

Optimization	Configuration	Impact
Connection Pooling	pgBouncer (100 connections)	Reduces connection overhead
MVCC	Built-in (PostgreSQL default)	Concurrent reads + writes
Indexes	Composite indexes on (user_id, created_at)	Faster transaction queries
WAL	Enabled by default	Crash recovery + replication
Vacuum	Autovacuum enabled	Prevents table bloat

Expected Performance:

- **100 users:** P95 < 200ms, 0% errors
- **500 users:** P95 < 250ms, 0% errors
- **1000 users:** P95 < 300ms, < 0.5% errors

Connection Pool Config (pgBouncer):

```
[databases]
drop = host=localhost port=5432 dbname=drop

[pgbouncer]
pool_mode = transaction
max_client_conn = 1000
```

```
default_pool_size = 20
reserve_pool_size = 5
```

---

7. Bottleneck Identification Strategy

7.1 What to Measure

Layer	Metrics	Tools
HTTP	Request rate, response time, error rate	k6
Application	CPU, memory, event loop lag	Node.js metrics, PM2
Database	Query time, lock wait time, connection count	SQLite EXPLAIN QUERY PLAN, PostgreSQL pg_stat_statements
Network	Bandwidth, latency, packet loss	k6, Docker stats

7.2 Common Bottlenecks & Symptoms

Bottleneck	Symptom	Solution
Database Write Lock	SQLITE_BUSY errors, high P95 on writes	Migrate to PostgreSQL
CPU Bound	100% CPU, slow response times	Horizontal scaling, optimize hot paths
Memory Leak	OOM crashes, gradual memory increase	Profile with Node.js heap snapshots
Event Loop Blocking	High event loop lag, slow all endpoints	Move heavy computation to background jobs
Connection Pool Exhausted	"No connections available" errors	Increase pool size, use pgBouncer
Slow Queries	High database query time	Add indexes, optimize JOIN queries

---

7.3 Profiling Commands

SQLite Query Analysis:

```
sqlite3 data/drop.db
sqlite> EXPLAIN QUERY PLAN SELECT * FROM transactions WHERE user_id = ? ORDER BY created_at
```

```
DESC LIMIT 10;
```

PostgreSQL Query Analysis:

```
-- Enable query stats
CREATE EXTENSION pg_stat_statements;

-- View slow queries
SELECT query, calls, total_exec_time, mean_exec_time
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;
```

Node.js CPU Profiling:

```
node --cpu-prof src/drop-app/server.js
# Generates CPU profile → analyze with Chrome DevTools
```

Memory Profiling:

```
node --inspect src/drop-app/server.js
# Open chrome://inspect → take heap snapshot
```

---

8. Monitoring During Load Tests

8.1 Real-Time Metrics (k6 Dashboard)

k6 Built-in Metrics:

- `http_req_duration` — Response time (P50, P95, P99)
- `http_req_failed` — Error rate (%)
- `http_reqs` — Request rate (req/s)
- `vus` — Virtual users (current)
- `iterations` — Total iterations completed

Custom Metrics (add to scripts):

```
import { Trend, Counter } from 'k6/metrics';
```

```
const authDuration = new Trend('auth_duration');
const paymentErrors = new Counter('payment_errors');

// In test:
authDuration.add(loginRes.timings.duration);
paymentErrors.add(transferRes.status !== 200 ? 1 : 0);
```

8.2 System Metrics (Parallel Monitoring)

During k6 test, run in parallel:

```
# Terminal 1: k6 test
k6 run --out json=results.json tests/load/scenarios/send-money-flow.js

# Terminal 2: Docker stats (if running in container)
docker stats drop-app

# Terminal 3: SQLite database monitoring
watch -n 1 "sqlite3 data/drop.db 'SELECT COUNT(*) FROM transactions'"

# Terminal 4: Application logs
tail -f logs/app.log | grep ERROR
```

Grafana Integration (Future):

- k6 can export metrics to InfluxDB → Grafana dashboards
- Real-time visualization of load test results
- Alerting on threshold breaches

8.3 Key Metrics to Track

Metric	Warning Threshold	Critical Threshold
P95 Response Time	> 300ms	> 500ms
Error Rate	> 1%	> 5%
CPU Usage	> 70%	> 90%
Memory Usage	> 80%	> 95%
Database Query Time	> 50ms (P95)	> 100ms

Metric	Warning Threshold	Critical Threshold
Database Connections	> 80% pool	100% pool

---

9. CI Integration (Performance Regression Tests)

9.1 GitHub Actions Workflow

File: `.github/workflows/load-test.yml`

```
name: Load Test

on:
  pull_request:
    branches: [main, staging]
  schedule:
    - cron: '0 2 * * 1' # Weekly Monday 2 AM

jobs:
  load-test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'

      - name: Install dependencies
        run: npm ci
        working-directory: src/drop-app

      - name: Build app
        run: npm run build
```

```
        working-directory: src/drop-app

      - name: Start app (background)
        run: |
          npm run start &
          sleep 10
        working-directory: src/drop-app

      - name: Install k6
        run: |
          sudo gpg -k
          sudo gpg --no-default-keyring --keyring /usr/share/keyrings/k6-archive-keyring.gpg
--keyserver hkp://keyserver.ubuntu.com:80 --recv-keys C5AD17C747E3415A3642D57D77C6C491D6AC1D69
          echo "deb [signed-by=/usr/share/keyrings/k6-archive-keyring.gpg]
https://dl.k6.io/deb stable main" | sudo tee /etc/apt/sources.list.d/k6.list
          sudo apt-get update
          sudo apt-get install k6

      - name: Run load test (baseline)
        run: k6 run --out json=results.json tests/load/scenarios/dashboard-load.js
        env:
          BASE_URL: http://localhost:3000

      - name: Check thresholds
        run: |
          # k6 exits with code 99 if thresholds fail
          if [ $? -eq 99 ]; then
            echo "Load test thresholds FAILED"
            exit 1
          fi

      - name: Upload results
        uses: actions/upload-artifact@v4
        with:
          name: load-test-results
          path: results.json
```

Trigger: PR to main/staging, or weekly schedule **Purpose:** Catch performance regressions before merge **Threshold:** If P95 > 300ms or error rate > 1%, fail the build

---

9.2 Baseline Results Storage

First run (before optimization):

```
k6 run --out json=baseline-results.json tests/load/scenarios/dashboard-load.js
```

Store in repo:

```
tests/load/baselines/
├── dashboard-load-baseline.json
├── send-money-baseline.json
├── qr-payment-baseline.json
└── auth-flow-baseline.json
```

Regression detection:

- Compare current test results to baseline
- Alert if P95 increases > 20%
- Fail CI if P95 increases > 50%

---

10. Next.js 16 Performance Optimizations

10.1 Server Components (Default in Next.js 16)

Impact:

- Reduced client-side JavaScript bundle size
- Faster initial page load (no React hydration for server components)
- Data fetching happens server-side (lower TTFB)

Usage:

```
// app/dashboard/page.tsx (Server Component by default)
export default async function DashboardPage() {
  const user = await getUser(); // Server-side fetch
  const transactions = await getTransactions(); // Parallel fetch

  return (
```

```
    <div>
      <UserInfo user={user} /> {/* Server Component */}
      <TransactionList transactions={transactions} /> {/* Server Component */}
    </div>
  );
}
```

Load Test Impact:

- Dashboard load: P95 reduced by 30-40% (less client-side rendering)
- FCP improved by 50%

10.2 Caching with "use cache" Directive (New in Next.js 16)

Impact:

- Explicit cache control for pages, components, and functions
- Reduced API calls (cached responses)

Usage:

```
// app/transactions/page.tsx
'use cache';

export default async function TransactionsPage() {
  const transactions = await getTransactions();
  return <TransactionList transactions={transactions} />;
}
```

Load Test Impact:

- Repeated requests: P95 reduced by 70-80% (cache hit)
- Database load: 90% reduction (cached queries)

Sources:

- Next.js 16 Features
- Next.js Performance Optimization Guide

10.3 React Compiler (Stable in Next.js 16)

Impact:

- Automatic memoization of components
- Reduced re-renders
- Better runtime performance

Configuration:

```
// next.config.ts
export default {
  experimental: {
    reactCompiler: true,
  },
};
```

Load Test Impact:

- Client-side rendering: 20-30% faster
- Lighthouse Performance score: +10 points

Sources:

- Next.js 16 Performance Boost

10.4 Turbopack (Fast Dev Server)

Impact:

- Instant HMR (Hot Module Replacement)
- Faster builds

Usage:

```
npm run dev -- --turbo
```

Load Test Impact:

- Development iteration speed: 3x faster
- Build time: 2x faster

---

11. Capacity Planning (Norwegian Market Projections)

11.1 User Growth Projections

Year	Active Users	Peak Concurrent (0.5%)	Required Capacity
Year 1 (MVP)	10K	50	100 (2x buffer)
Year 2	50K	250	500 (2x buffer)
Year 3	100K	500	1000 (2x buffer)
Year 5	500K	2500	5000 (2x buffer)

Norwegian Market Context:

- Population: 5.5M
- Digital payment penetration: 95% (Vipps, BankID)
- Realistic market share Year 1: 0.2% → 10K users
- Realistic market share Year 3: 2% → 100K users

11.2 Infrastructure Scaling Plan

Users	Database	App Instances	Load Balancer
< 50	SQLite (demo)	1x Next.js	None
50-500	PostgreSQL (single)	2x Next.js	Nginx
500-5K	PostgreSQL (primary + replica)	4x Next.js	AWS ALB
5K-50K	PostgreSQL (RDS Multi-AZ)	8x Next.js	AWS ALB + Auto Scaling
50K+	PostgreSQL (Aurora)	16x Next.js	AWS ALB + Auto Scaling + CDN

11.3 Cost Estimation (Vercel + Supabase/Neon)

Scenario: 10K active users, 50 concurrent peak

Service	Plan	Cost/Month
Vercel (Next.js hosting)	Pro	$20

Service	Plan	Cost/Month
Neon (PostgreSQL)	Launch	$20
Sentry (Error tracking)	Team	$26
Total		**$66/month**

Scenario: 100K active users, 500 concurrent peak

Service	Plan	Cost/Month
Vercel (Next.js hosting)	Enterprise	$500+
AWS RDS (PostgreSQL Multi-AZ)	db.r6g.large	$300
AWS ALB	Standard	$20
Sentry	Business	$80
Total		**$900/month**

---

12. Implementation Plan (Phased Approach)

Phase 1: Setup & Baseline (Day 1)

Tasks:

1. Install k6: `brew install k6` (macOS) or `apt install k6` (Linux)
2. Create test directory: `src/drop-app/tests/load/`
3. Write baseline script: `tests/load/scenarios/dashboard-load.js`
4. Run first test: `k6 run tests/load/scenarios/dashboard-load.js`
5. Document baseline results: `tests/load/baselines/dashboard-load-baseline.json`

Deliverable: Baseline performance metrics (P50, P95, P99) for dashboard load

---

Phase 2: Core Flow Scripts (Day 2-3)

Tasks:

1. Write auth flow script (register + login)
2. Write send money script (remittance)
3. Write QR payment script

4. Write bank sync script
5. Run all scripts @ 100 concurrent users
6. Document results

Deliverable: 5 load test scripts covering all critical user journeys

---

Phase 3: Load Profiles (Day 4)

Tasks:

1. Run baseline load (100 users, 30 min)
2. Run peak load (500 users, 15 min)
3. Run stress test (1000 users, 10 min)
4. Run spike test (0→500 in 30s)
5. Identify bottlenecks (likely: SQLite write contention)

Deliverable: Load profile results + bottleneck analysis report

---

Phase 4: Optimization & Retest (Day 5)

Tasks:

1. Implement optimizations:
   - Add database indexes
   - Enable Next.js caching
   - Optimize slow queries
2. Rerun all load tests
3. Compare before/after results
4. Document performance improvements

Deliverable: Optimization report (before/after metrics)

Phase 5: CI Integration (Day 6)

Tasks:

1. Create GitHub Actions workflow (`.github/workflows/load-test.yml`)
2. Add baseline threshold checks
3. Test workflow on PR
4. Store results as artifacts

Deliverable: Automated load testing in CI pipeline

---

13. Success Criteria

Metric	Target	Actual	Status
Baseline (100 users)	P95 < 300ms, 0% errors	TBD	□
Peak (500 users)	P95 < 400ms, < 1% errors	TBD	□
Stress (1000 users)	P95 < 500ms, < 5% errors	TBD	□
Dashboard Load	P95 < 250ms	TBD	□
Auth Flow	P95 < 250ms	TBD	□
Send Money	P95 < 300ms	TBD	□
QR Payment	P95 < 300ms	TBD	□
Bank Sync	P95 < 200ms	TBD	□

---

14. Appendix: k6 Script Template

File: `tests/load/template.js`

```
import http from 'k6/http';
import { check, sleep } from 'k6';
import { Rate, Trend } from 'k6/metrics';

// Custom metrics
const errorRate = new Rate('errors');
const customDuration = new Trend('custom_duration');

// Test configuration
export const options = {
  stages: [
    { duration: '2m', target: 100 }, // Ramp up
    { duration: '5m', target: 100 }, // Sustain
    { duration: '2m', target: 0 },   // Ramp down
  ],
  thresholds: {
```

```
    http_req_duration: ['p(95)<300'], // 95% under 300ms
    errors: ['rate<0.01'],             // Error rate < 1%
  },
};

const BASE_URL = __ENV.BASE_URL || 'http://localhost:3000';

// Setup (runs once before test)
export function setup() {
  // Login, create test data, etc.
  return { token: 'example-jwt' };
}

// Main test function (runs for each VU iteration)
export default function (data) {
  const { token } = data;

  // HTTP request
  const res = http.get(`${BASE_URL}/api/endpoint`, {
    headers: { Cookie: `token=${token}` },
  });

  // Validation
  const success = check(res, {
    'status 200': (r) => r.status === 200,
    'has data': (r) => JSON.parse(r.body).data !== undefined,
  });

  if (!success) {
    errorRate.add(1);
  }

  customDuration.add(res.timings.duration);

  // Think time
  sleep(1);
}

// Teardown (runs once after test)
export function teardown(data) {
```

```
  // Cleanup test data
}
```

---

15. Sources & References

Load Testing Tools

- Load Testing PoC: k6 vs Artillery vs Locust vs Gatling
- Artillery vs k6 Comparison
- k6 Official Documentation
- k6 Examples Repository
- Synthetic Testing Next.js with k6 Cloud
- GitHub: PM2 + Next.js + k6 Example

PSD2 Compliance & Performance

- What Should You Expect from PSD3 Rules?
- PSD2 Compliance Guide
- What is PSD2? - Nordea
- PSD2 Technical Security Requirements

Next.js 16 Performance

- Next.js 16 Migration: 218% Performance Boost
- Next.js 16 Features Overview
- Next.js Performance Optimization Guide
- Next.js Performance Best Practices
- Next.js Production Checklist

Database Performance

- SQLite vs PostgreSQL Performance
- PostgreSQL vs SQLite Comparison
- SQLite vs PostgreSQL - Airbyte

- PostgreSQL vs SQLite 2026 Comparison

16. Approval

Reviewed by: Alem (CEO) **Status:** Pending **Next Steps:** Implement Phase 1 (Setup & Baseline)

Revision #4
Created 2026-02-18 08:44:44 UTC by John
Updated 2026-05-31 20:02:15 UTC by John