

# Quality Attributes Analysis

## Overview

For this project we selected three quality attributes - Performance, Scalability and Reliability. This document highlights how these were achieved through design decision and implementation.

## 1. Performance

### Target Metrics

- API response time: < 100ms
- Event processing time: < 50ms per service
- End-to-end click processing: < 300ms

### Design Decisions for Performance

#### 1.1 Asynchronous Event Processing

**Implementation:** Used pub-sub pattern with event bus

- Click Ingestion publishes events without waiting for processing
- Each subscriber processes independently
- No blocking operations

**Source Code** (`src/services/click-ingestion.js`):

```
// Publish to event bus – non-blocking
eventBus.publish('click-events', clickEvent);

// Immediately return response
res.json({ success: true, message: 'Click received' });
```

#### 1.2 In-Memory Event Bus

**Implementation:** Used Node.js EventEmitter for fast message passing

- No network latency
- No disk I/O
- Direct function calls between services

**Source Code** (`src/config/event-bus.js`):

```
publish(topic, data) {
  this.emit(topic, data); // Immediate in-memory event
}
```

### 1.3 Minimal Processing Per Service

**Implementation:** Each service performs only the specified job

- Fraud Detection: Simple scoring algorithm
- Billing: Basic calculations
- Analytics: Counter updates

### Performance Test Results

From demo run with 20 clicks:

- Average API response: ~50ms
- Total processing time: ~300ms per click
- System handled 20 clicks in 10 seconds = 2 clicks/second

**Conclusion:** Performance targets met for academic demonstration.

---

## 2. Scalability

### Target Metrics

- Support 10,000 clicks/second (design goal)
- Horizontal scaling of services
- No single point of failure

### Design Decisions for Scalability

#### 2.1 Pub-Sub Architecture

**Implementation:** Loose coupling between services

- Services don't call each other directly
- Can run multiple instances of any service
- Load distributed automatically

**Source Code ([src/app.js](#)):**

```
// Each service subscribes independently
fraudDetection.start(); // Can run multiple instances
billing.start(); // Can run multiple instances
analytics.start(); // Can run multiple instances
```

#### 2.2 Stateless Services

**Implementation:** Services don't store state between requests

- Fraud Detection: Stateless scoring
- Billing: Budget stored separately (would be in DB)
- Analytics: Aggregates can be distributed

## 2.3 Topic-Based Routing

**Implementation:** Multiple services can subscribe to same topic

- All fraud detection instances get click-events
- All billing instances get validated-clicks
- Load balanced automatically

**Source Code (`src/config/event-bus.js`):**

```
subscribe(topic, callback) {  
  this.on(topic, callback); // Multiple subscribers allowed  
}
```

## Scalability Analysis

**Current Implementation:**

- 1 publisher, 3 subscribers
- In-memory event bus (single machine)

**Production Scaling Path:**

1. Replace EventEmitter with Apache Kafka
2. Run multiple instances of each service
3. Kafka partitions distribute load
4. Can scale to 10,000+ clicks/second

**Horizontal Scaling Example:**

```
Click Ingestion: 3 instances (load balanced)  
Fraud Detection: 5 instances (Kafka consumer group)  
Billing: 3 instances (Kafka consumer group)  
Analytics: 2 instances (Kafka consumer group)
```

**Conclusion:** Architecture designed for horizontal scalability.

---

## 3. Reliability

Target Metrics

- 99.9% uptime
- No data loss

- Error Handling

## Design Decisions for Reliability

### 3.1 Budget Validation

**Implementation:** We check advertiser budget, and if there is a balance we charge the click. The charge never goes beyond remaining advertiser budget.

- Protects advertisers against overspending
- If the budget is exhausted the clicks are rejected and not charged
- Maintains integrity of billing system.

**Source Code** (`src/services/billing-service.js`):

```
// Check if advertiser has enough budget
if (budget.remaining < cost) {
    logger.log('BillingService', `BUDGET EXCEEDED for ${advertiserId}`);
    return; // Don't process this click
}
```

### 3.2 Fraud Detection

**Implementation:** Validate clicks before we charge. Following checks are performed

- User agent
- IP address
- A fraud score is calculated
- As a result, suspicious clicks are blocked

**Source Code** (`src/services/fraud-detection.js`):

```
if (fraudScore >= 0.7) {
    // This click needs to be ignored
    eventBus.publish('fraud-alerts', {...});
    // Don't publish to validated-clicks
}
```

### 3.3 Error Handling

**Implementation:** Following checks are performed

- API Input parameters are validated
- Error logging is implemented
- Graceful handling of errors

**Source Code** (`src/services/click-ingestion.js`):

```

try {
  // Validate required fields
  if (!clickData.ad_id || !clickData.campaign_id) {
    return res.status(400).json({ error: 'Missing required fields' });
  }
  // Process click
} catch (err) {
  logger.error('ClickIngestion', err.message);
  res.status(500).json({ error: 'Internal error' });
}

```

### 3.4 Service Independence

**Implementation:** Services are implemented such that each of them operate independently

- For example, if fraud detection fails, billing can continue to work
- For example, If the analytics service fails, the billing still continues to run
- Failures are not cascaded, they are isolated to the service that failed.

### Reliability Test Scenarios

#### Scenario 1: Budget Exhaustion

- Initial budget: \$100
- After spending \$100, all further clicks are rejected
- Result: System prevents overspending and hence overcharging to employers

#### Scenario 2: Fraud Detection

- IF Bot user agent detected
- AND Fraud score > 0.7
- Result: Click is rejected and excluded from billing

#### Scenario 3: Invalid Input

- Missing required fields
- Result: 400 error is returned and system continues to function

**Conclusion:** System reliably handles the error and edge cases

---

## Summary

Quality Attribute	Target	Implementation	Status
Performance	<100ms API response	Async pub-sub, in-memory events	Met
Scalability	10K clicks/sec	Horizontal scaling design	Designed
Reliability	99.9% uptime	Error handling, validation	Implemented

## Key Takeaways

1. **Performance:** High performance is achieved through asynchronous processing and minimal per-service work
2. **Scalability:** Scalability of system is achieved by pub-sub architecture and stateless services
3. **Reliability:** System reliability is ensured through validation, error handling, and service independence

## Future Improvements

1. **Performance:** Add caching layer (Redis) for frequently accessed data
  2. **Scalability:** Implement with Apache Kafka for production-level throughput
  3. **Reliability:** Add database persistence and message replay capabilities
-