Đặc tính chất lượng	Lý do quan trọng	Công cụ hỗ trợ	Các bước kiểm tra
Scalability (Khả năng mở rộng)	- Xử lý tải lớn từ nhiều nguồn sự kiện - Hỗ trợ mở rộng ngang dễ dàng - Đảm bảo hiệu suất cao với xử lý không đồng bộ	- **Message Brokers**: Apache Kafka, RabbitMQ, AWS SQS/SNS, Google Pub/Sub, Azure Event Hubs - **Container Orchestration**: Kubernetes, Docker Swarm - **Cloud Services**: AWS Lambda, Google Cloud Functions, Azure Functions - **Monitoring**: Prometheus, Grafana, ELK Stack - **Load Testing**: JMeter, Locust, k6	1. Xác định tải dự kiến (số sự kiện/giây) 2. Cấu hình message broker và consumer (Kafka partition, Kubernetes pod) 3. Mô phỏng tải lớn bằng JMeter/Locust 4. Giám sát độ trễ, thông lượng bằng Prometheus/Grafana 5. Thêm consumer instance để kiểm tra autoscaling 6. Đánh giá hiệu suất và xác nhận không mất sự kiện
Reliability (Độ tin cậy)	- Đảm bảo xử lý sự kiện ít nhất một lần hoặc chính xác một lần - Xử lý lỗi tạm thời và không khắc phục được - Hỗ trợ phục hồi sau sự cố	- **Message Brokers**: Kafka (exactly-once), RabbitMQ (ACK/NACK), AWS SQS (DLQ) - **Databases**: DynamoDB, MongoDB, Redis - **Monitoring**: Prometheus, Grafana, AWS CloudWatch, Datadog - **Chaos Engineering**: Chaos Monkey, LitmusChaos - **Retry Libraries**: Spring Retry, Polly	1. Xác định yêu cầu (at-least-once/exactly-once) 2. Cấu hình DLQ, retry với exponential backoff, Idempotency Key 3. Mô phỏng lỗi (tắt consumer, ngắt mạng) 4. Giám sát số lần retry, sự kiện thất bại bằng Grafana 5. Kiểm tra phục hồi sau khi consumer khởi động lại 6. Xác nhận exactly-once delivery qua cơ sở dữ liệu

Đặc tính chất lượng	Lý do quan trọng	Công cụ hỗ trợ	Các bước kiểm tra
Maintainability (Khả năng bảo trì)	- Dễ sửa đổi, mở rộng hệ thống - Hỗ trợ phát triển song song - Giảm rủi ro lỗi nhờ hợp đồng sự kiện rõ ràng	- **Schema Management**: Confluent Schema Registry, JSON Schema, Avro - **Containerization**: Docker, Kubernetes - **CI/CD**: Jenkins, GitHub Actions, GitLab CI - **Code Quality**: SonarQube, ESLint, Checkstyle - **Documentation**: Swagger, OpenAPI, Confluence - **Monitoring**: Prometheus, Grafana	1. Kiểm tra loose coupling qua message broker 2. Đánh giá single responsibility qua mã nguồn và unit test 3. Thử triển khai độc lập một service bằng Kubernetes 4. Kiểm tra schema sự kiện trong Schema Registry 5. Phân tích mã bằng SonarQube để tìm nợ kỹ thuật 6. Giám sát lỗi runtime và hiệu suất bằng Grafana

Phân Tích Đặc Tính Chất Lượng Event-Driven Architecture

Ba Đặc Tính Chất Lượng Mong Muốn Nhất

1. Scalability (Khả năng Mở Rộng)

Tại sao đây là đặc tính quan trọng nhất:

- **Horizontal Scaling**: Mỗi microservice có thể scale độc lập theo nhu cầu cụ thể
- **Event-driven nature**: Kafka cho phép xử lý hàng triệu events/giây với partition scaling
- **Microservice Independence**: Có thể thêm nhiều instance của một service mà không ảnh hưởng khác
- **Future-proof**: Dễ dàng thêm service mới vào hệ sinh thái mà không cần chỉnh sửa code hiện có

Thể hiện trong dự án:

```
# Có thể scale bất kỳ service nào độc lập
docker-compose up --scale user-service=3 --scale notification-service=5
```

Các bước tạo ra đặc tính Scalability:

Bước 1: Thiết kế Stateless Services

```
// user-service/src/controllers/userController.js
module.exports = {
  createUser: async (request, reply) => {
    // Service không lưu trữ state, mọi thông tin đều từ request
    const { username, email, password } = request.body;

    // Xử lý business logic và lưu vào database
    const user = await userService.createUser({ username, email, password
});

    // Phát event thay vì gọi trực tiếp service khác
    await sendUserCreatedEvent(user);

    reply.code(201).send(user);
  }
};
```

Bước 2: Cấu hình Kafka Partitioning

```
# docker-compose.yml - Kafka config cho scaling
kafka:
  environment:
    KAFKA_NUM_PARTITIONS: 3  # Cho phép 3 consumer cùng xử lý
    KAFKA_DEFAULT_REPLICATION_FACTOR: 1
    KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
```

```
// shared/utils/kafkaClient.js - Producer config
const producer = kafka.producer({
  // Partition key để phân phối tải
  partitioner: Partitioners.LegacyPartitioner
});

await producer.send({
  topic: 'user.created',
  messages: [{
    // Sử dụng userId làm partition key để đảm bảo order
    key: String(userId),
    value: JSON.stringify(eventData)
  }]
});
```

Bước 3: Container Health Checks cho Auto-scaling

```
# docker-compose.yml
user-service:
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:3001/health"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 40s
```

```
// user-service/src/routes/health.js
module.exports = async function (fastify, opts) {
  fastify.get('/health', async (request, reply) => {
    try {
      // Kiểm tra database connection
      await sequelize.authenticate();

      // Kiểm tra Kafka connection
      const admin = kafka.admin();
      await admin.listTopics();

      return { status: 'healthy', timestamp: new Date().toISOString() };
    } catch (error) {
      reply.code(503);
      return { status: 'unhealthy', error: error.message };
    }
  });
};
```

Bước 4: Service Discovery cho Dynamic Scaling

Service Registry Implementation:

```
// shared/service-discovery/service-registry.js
const Redis = require('redis');

class ServiceRegistry {
  constructor() {
    this.client = Redis.createClient({
      url: process.env.REDIS_URL || 'redis://localhost:6379'
    });
    this.client.connect();
  }

  // Đăng ký service khi khởi động
  async registerService(serviceName, serviceInfo) {
    const serviceKey =
`services:${serviceName}:${serviceInfo.instanceId}`;
    const serviceData = {
      ...serviceInfo,
      registeredAt: new Date().toISOString(),
      lastHeartbeat: new Date().toISOString()
    };

    await this.client.setEx(serviceKey, 30, JSON.stringify(serviceData));
// TTL 30 seconds
    console.log(`✅ Service registered: ${serviceName} at
${serviceInfo.host}:${serviceInfo.port}`);

    // Start heartbeat
    this.startHeartbeat(serviceName, serviceInfo);
  }

  // Heartbeat để duy trì registration
  startHeartbeat(serviceName, serviceInfo) {
    const heartbeatInterval = setInterval(async () => {
      try {
        const serviceKey =
`services:${serviceName}:${serviceInfo.instanceId}`;
        const existingData = await this.client.get(serviceKey);

        if (existingData) {
          const serviceData = JSON.parse(existingData);
          serviceData.lastHeartbeat = new Date().toISOString();
          await this.client.setEx(serviceKey, 30,
JSON.stringify(serviceData));
        } else {
          // Re-register if key expired
          await this.registerService(serviceName, serviceInfo);
        }
      } catch (error) {
        console.error('❌ Heartbeat failed:', error);
      }
    }, 10000); // Every 10 seconds

    // Cleanup on process termination
    process.on('SIGTERM', () => {
      clearInterval(heartbeatInterval);
      this.deregisterService(serviceName, serviceInfo.instanceId);
    });
  }

  // Tìm tất cả instances của một service
  async discoverService(serviceName) {
    const pattern = `services:${serviceName}:*`;
    const keys = await this.client.keys(pattern);

    const services = [];
    for (const key of keys) {
      const serviceData = await this.client.get(key);
      if (serviceData) {
        services.push(JSON.parse(serviceData));
      }
    }

    return services.filter(service => {
      // Filter out stale services (no heartbeat for 60 seconds)
      const lastHeartbeat = new Date(service.lastHeartbeat);
      const now = new Date();
      return (now - lastHeartbeat) < 60000;
    });
  }

  // Load balancer - chọn service instance
  async getServiceInstance(serviceName, strategy = 'round-robin') {
    const instances = await this.discoverService(serviceName);

    if (instances.length === 0) {
      throw new Error(`No healthy instances found for service:
${serviceName}`);
    }

    switch (strategy) {
      case 'round-robin':
        return this.roundRobinSelection(serviceName, instances);
      case 'random':
        return instances[Math.floor(Math.random() * instances.length)];
      case 'least-connections':
        return this.leastConnectionsSelection(instances);
      default:
        return instances[0];
    }
  }

  roundRobinSelection(serviceName, instances) {
    const counterKey = `lb:${serviceName}:counter`;
    return this.client.incr(counterKey).then(counter => {
      return instances[(counter - 1) % instances.length];
    });
  }

  leastConnectionsSelection(instances) {
    // Simplified: chọn instance với ít connection nhất
    return instances.reduce((least, current) =>
      (current.connections || 0) < (least.connections || 0) ? current :
least
    );
  }

  async deregisterService(serviceName, instanceId) {
    const serviceKey = `services:${serviceName}:${instanceId}`;
    await this.client.del(serviceKey);
    console.log(`🗑️ Service deregistered: ${serviceName}:${instanceId}`);
  }
}

module.exports = ServiceRegistry;
```

Service Registration trong từng service:

```javascript
// user-service/src/index.js
const ServiceRegistry = require('../shared/service-discovery/service-registry');
const { v4: uuidv4 } = require('uuid');

const serviceRegistry = new ServiceRegistry();
const instanceId = uuidv4();

const start = async () => {
  try {
    await fastify.register(app);

    const port = process.env.PORT || 3001;
    const host = process.env.HOST || '0.0.0.0';

    await fastify.listen({ port, host });

    // Đăng ký service với service discovery
    await serviceRegistry.registerService('user-service', {
      instanceId,
      host: process.env.PUBLIC_HOST || 'user-service',
      port,
      version: '1.0.0',
      metadata: {
        capabilities: ['user-management', 'authentication'],
        region: process.env.REGION || 'default'
      }
    });

    fastify.log.info(`User Service running on port ${port}`);
  } catch (err) {
    fastify.log.error(err);
    process.exit(1);
  }
};

start();
```

Dynamic Service Discovery trong Gateway:

```javascript
// gateway/src/services/service-discovery-client.js
const ServiceRegistry = require('../../shared/service-discovery/service-registry');

class ServiceDiscoveryClient {
  constructor() {
    this.serviceRegistry = new ServiceRegistry();
    this.serviceCache = new Map();
    this.cacheExpiry = new Map();
  }
  async getServiceUrl(serviceName) {
    // Check cache first
    if (this.isServiceCached(serviceName)) {
      return this.serviceCache.get(serviceName);
    }

    // Discover service instance
    const instance = await
this.serviceRegistry.getServiceInstance(serviceName);
    const serviceUrl = `http://${instance.host}:${instance.port}`;

    // Cache for 30 seconds
    this.serviceCache.set(serviceName, serviceUrl);
    this.cacheExpiry.set(serviceName, Date.now() + 30000);

    return serviceUrl;
  }

  isServiceCached(serviceName) {
    const expiry = this.cacheExpiry.get(serviceName);
    if (expiry && Date.now() < expiry) {
      return true;
    }

    // Cleanup expired cache
    this.serviceCache.delete(serviceName);
    this.cacheExpiry.delete(serviceName);
    return false;
  }

  // Circuit breaker pattern
  async callService(serviceName, path, options = {}) {
    const maxRetries = 3;
    let lastError;

    for (let retry = 0; retry < maxRetries; retry++) {
      try {
        const serviceUrl = await this.getServiceUrl(serviceName);
        const response = await fetch(`${serviceUrl}${path}`, options);

        if (!response.ok) {
          throw new Error(`HTTP ${response.status}:
${response.statusText}`);
        }

        return response;
      } catch (error) {
        lastError = error;
        console.warn(`⚠ Service call failed (attempt ${retry + 1}):`,
error.message);

        // Invalidate cache on error
        this.serviceCache.delete(serviceName);
        this.cacheExpiry.delete(serviceName);

        // Exponential backoff
        if (retry < maxRetries - 1) {
          await new Promise(resolve => setTimeout(resolve, Math.pow(2, retry) * 1000));
        }
      }
    }

    throw new Error(`Service ${serviceName} unavailable after ${maxRetries} retries: ${lastError.message}`);
  }
}

module.exports = ServiceDiscoveryClient;
```

Gateway sử dụng Service Discovery:

```
// gateway/src/routes/users.js
const ServiceDiscoveryClient = require('../services/service-discovery-client');

module.exports = async function (fastify, opts) {
  const serviceDiscovery = new ServiceDiscoveryClient();

  fastify.post('/users', async (request, reply) => {
    try {
      // Dynamic service discovery thay vì hardcode URL
      const response = await serviceDiscovery.callService('user-service', '/users', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(request.body)
      });

      const userData = await response.json();
      return userData;
    } catch (error) {
      reply.code(503);
      return {
        error: 'User service unavailable',
        message: error.message,
        timestamp: new Date().toISOString()
      };
    }
  });

  fastify.post('/registrations', async (request, reply) => {
    try {
      const response = await serviceDiscovery.callService('registration-service', '/registrations', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(request.body)
      });

      const registrationData = await response.json();
      return registrationData;
    } catch (error) {
      reply.code(503);
      return { error: 'Registration service unavailable', message:
error.message };
    }
  });
};
```

Docker Compose với Service Discovery:

```
# docker-compose.yml
version: '3.8'

services:
  # Service Discovery Registry
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    command: redis-server --appendonly yes
    volumes:
      - redis-data:/data
    networks:
      - eventflow-network

  # Gateway với service discovery
  gateway:
    build: ./gateway
    environment:
      - REDIS_URL=redis://redis:6379
      - PUBLIC_HOST=gateway
    depends_on:
      - redis
      - kafka
    ports:
      - "3007:3000"
    networks:
      - eventflow-network

  # User service instances (có thể scale)
  user-service:
    build: ./user-service
    environment:
      - REDIS_URL=redis://redis:6379
      - PUBLIC_HOST=user-service
      - DATABASE_URL=postgres://postgres:admin@postgres:5432/userdb
      - KAFKA_BROKERS=kafka:9092
    depends_on:
      - redis
      - postgres
      - kafka
    # Không expose port cố định, để auto-assign
    networks:
      - eventflow-network

  # Registration service instances
  registration-service:
    build: ./registration-service
    environment:
      - REDIS_URL=redis://redis:6379
      - PUBLIC_HOST=registration-service
      - DATABASE_URL=postgres://postgres:admin@postgres:5432/registrationdb
      - KAFKA_BROKERS=kafka:9092
    depends_on:
      - redis
      - postgres
      - kafka
    networks:
      - eventflow-network

volumes:
  redis-data:

networks:
  eventflow-network:
    driver: bridge
```

Scaling với Service Discovery:

```
# Scale services dynamically
docker-compose up --scale user-service=5 --scale registration-service=3

# Gateway sẽ tự động phát hiện và load balance giữa các instances
# Redis sẽ track tất cả service instances và health status
```

2. Reliability (Độ Tin Cậy)

Tại sao quan trọng:

- **Message Durability**: Kafka đảm bảo events không bị mất với persistent storage
- **At-least-once delivery**: Mỗi event được đảm bảo xử lý ít nhất 1 lần
- **Fault Isolation**: Lỗi ở một service không làm crash toàn bộ hệ thống
- **Retry Mechanisms**: Consumer có thể retry khi xử lý thất bại

Thể hiện trong dự án:

```
// Audit service đảm bảo mọi event đều được ghi lại
await logAudit({
  eventType: EVENT_TOPICS.USER_CREATED,
  data: { userId, username, userEmail, timestamp },
});
```

Các bước tạo ra đặc tính Reliability:

Bước 1: Cấu hình Kafka Message Durability

```
# docker-compose.yml - Kafka persistence config
kafka:
  environment:
    KAFKA_LOG_RETENTION_HOURS: 168  # 7 days retention
    KAFKA_LOG_RETENTION_BYTES: 1073741824  # 1GB per partition
    KAFKA_LOG_SEGMENT_BYTES: 1073741824
    KAFKA_FLUSH_MESSAGES: 1000  # Flush every 1000 messages
```

Bước 2: Implement Consumer Retry Logic

```
// auditlog-service/src/consumers/userCreated.js
module.exports = async () => {
  const consumer = await createConsumer('audit-user-created');

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      let retries = 0;
      const maxRetries = 3;

      while (retries < maxRetries) {
        try {
          const data = JSON.parse(message.value?.toString() || '{}');

          // Validate required fields
          if (!data.userId) {
            console.warn('⚠ Missing userId in message:', data);
            return; // Skip invalid messages
          }

          // Trigger audit logging với idempotency check
          await logAuditWithRetry(data);
          console.log(`✅ Audit log created for ${topic}`);
          break; // Success, exit retry loop

        } catch (error) {
          retries++;
          console.error(`❌ Attempt ${retries} failed:`, error.message);

          if (retries >= maxRetries) {
            // Send to Dead Letter Queue after max retries
            await sendToDeadLetterQueue(topic, message, error);
            console.error(`💀 Message sent to DLQ after ${maxRetries} retries`);
          } else {
            // Exponential backoff delay
            const delay = Math.pow(2, retries) * 1000;
            await new Promise(resolve => setTimeout(resolve, delay));
          }
        }
      }
    },
  });
};

async function logAuditWithRetry(data) {
  // Idempotent check - avoid duplicate logs
  const existingLog = await AuditLog.findOne({
    where: {
      eventType: data.eventType,
      'data.userId': data.userId,
      'data.timestamp': data.timestamp
    }
  });

  if (existingLog) {
    console.log('⚠ Audit log already exists, skipping');
    return existingLog;
  }

  return await logAudit(data);
}
```

Bước 3: Circuit Breaker Pattern cho External Services

```
// notification-service/src/services/emailService.js
const CircuitBreaker = require('opossum');

const emailOptions = {
  timeout: 10000, // 10 seconds
  errorThresholdPercentage: 50, // Open circuit if 50% of requests fail
  resetTimeout: 30000, // Try again after 30 seconds
  rollingCountTimeout: 60000, // 1 minute window
  rollingCountBuckets: 6
};

const emailCircuitBreaker = new CircuitBreaker(sendEmailInternal,
emailOptions);

// Fallback when circuit is open
emailCircuitBreaker.fallback(() => {
  console.log('⚡ Email service circuit breaker is open, using fallback');
  // Log to queue for later retry
  return { success: false, reason: 'circuit_breaker_open' };
});

async function sendEmail(to, subject, body) {
  try {
    return await emailCircuitBreaker.fire(to, subject, body);
  } catch (error) {
    // Even if email fails, system continues to work
    console.error('📧 Email service unavailable:', error.message);
    return { success: false, error: error.message };
  }
}
```

Bước 4: Database Transaction với Event Publishing

```
// user-service/src/services/userService.js - Transactional Outbox Pattern
async function createUser(userData) {
  const transaction = await sequelize.transaction();

  try {
    // 1. Save user to database
    const user = await User.create(userData, { transaction });

    // 2. Save event to outbox table (same transaction)
    const eventData = {
      userId: user.id,
      username: user.username,
      userEmail: user.email,
      timestamp: new Date().toISOString()
    };

    await OutboxEvent.create({
      eventType: 'user.created',
      aggregateId: user.id,
      eventData: JSON.stringify(eventData),
      processed: false
    }, { transaction });

    // 3. Commit transaction
    await transaction.commit();

    // 4. Publish event after successful commit
    await publishUserCreatedEvent(eventData);

    return user;
  } catch (error) {
    await transaction.rollback();
    throw error;
  }
}

// Background process to handle failed event publishing
setInterval(async () => {
  const unprocessedEvents = await OutboxEvent.findAll({
    where: { processed: false },
    limit: 100
  });

  for (const event of unprocessedEvents) {
    try {
      await publishEvent(event.eventType, JSON.parse(event.eventData));
      event.processed = true;
      await event.save();
    } catch (error) {
      console.error('Failed to publish outbox event:', error);
    }
  }
}, 30000); // Every 30 seconds
```

3. Maintainability (Khả năng Bảo Trì)

Tại sao quan trọng:

- **Loose Coupling**: Services chỉ biết về events, không biết về nhau
- **Single Responsibility**: Mỗi service có một trách nhiệm rõ ràng
- **Independent Deployment**: Deploy từng service riêng biệt không ảnh hưởng khác
- **Clear Event Contracts**: Event schema rõ ràng, dễ hiểu và maintain

Thể hiện trong dự án:

```
// Mỗi service chỉ cần biết về event schema
{
  "eventType": "registration.created",
  "data": {
    "userId": 123,
    "eventId": "uuid-456",
    "userEmail": "user@example.com"
  }
}
```

Các bước tạo ra đặc tính Maintainability:

Bước 1: Implement Loose Coupling qua Event-Driven Communication

```javascript
// ❌ TIGHT COUPLING - Cách cũ (không tốt)
// user-service/src/controllers/userController.js
async createUser(userData) {
  const user = await userService.createUser(userData);

  // Direct HTTP calls tạo tight coupling
  await axios.post('http://notification-service:3004/send-welcome-email',
{
    userId: user.id,
    email: user.email
  });

  await axios.post('http://audit-service:3005/log-action', {
    action: 'user_created',
    userId: user.id
  });

  return user;
}

// ✅ LOOSE COUPLING - Cách mới (Event-Driven)
// user-service/src/controllers/userController.js
async createUser(userData) {
  const user = await userService.createUser(userData);

  // Chỉ phát event, không biết ai sẽ xử lý
  await sendUserCreatedEvent({
    userId: user.id,
    username: user.username,
    userEmail: user.email,
    timestamp: new Date().toISOString()
  });

  return user; // Service không phụ thuộc vào consumer
}
```

Bước 2: Centralized Event Types Definition

```javascript
// shared/event-types.js - Single source of truth cho events
module.exports = {
  EVENT_TOPICS: {
    // User domain events
    USER_CREATED: 'user.created',
    USER_LOGGED_IN: 'user.logged_in',
    USER_UPDATED: 'user.updated',

    // Registration domain events
    REGISTRATION_CREATED: 'registration.created',
    REGISTRATION_CANCELLED: 'registration.cancelled',
    // Notification domain events
    NOTIFICATION_SENT: 'notification.sent',
    NOTIFICATION_FAILED: 'notification.failed',

    // Audit domain events
    AUDIT_LOGGED: 'audit.logged',
    AUDIT_FAILED: 'audit.failed',
  },
};

// shared/schema/userEvents.js - Event schema validation
const Joi = require('joi');

const UserCreatedSchema = Joi.object({
  userId: Joi.number().integer().positive().required(),
  username: Joi.string().min(3).max(50).required(),
  userEmail: Joi.string().email().required(),
  timestamp: Joi.string().isoDate().required(),
  version: Joi.number().integer().default(1) // Schema versioning
});

module.exports = { UserCreatedSchema };
```

Bước 3: Service Independence với Separate Databases

```
// user-service/src/models/user.js - Own database
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize(process.env.USER_DATABASE_URL);

const User = sequelize.define('User', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  username: { type: DataTypes.STRING, unique: true, allowNull: false },
  email: { type: DataTypes.STRING, unique: true, allowNull: false },
  password: { type: DataTypes.STRING, allowNull: false }
});

// event-service/src/models/event.js - Own database
const sequelize = new Sequelize(process.env.EVENT_DATABASE_URL);

const Event = sequelize.define('Event', {
  id: { type: DataTypes.UUID, primaryKey: true, defaultValue:
DataTypes.UUIDV4 },
  name: { type: DataTypes.STRING, allowNull: false },
  capacity: { type: DataTypes.INTEGER, allowNull: false },
  registered: { type: DataTypes.INTEGER, defaultValue: 0 }
});

// auditlog-service/src/models/auditLog.js - Own MongoDB database
const mongoose = require('mongoose');

const auditLogSchema = new mongoose.Schema({
  eventType: { type: String, required: true },
  data: { type: mongoose.Schema.Types.Mixed, required: true },
  timestamp: { type: Date, default: Date.now },
  serviceSource: { type: String, required: true }
});

const AuditLog = mongoose.model('AuditLog', auditLogSchema);
```

Bước 4: Consumer-Driven Contract Testing

```
// notification-service/src/consumers/registrationCreated.js
// Consumer chỉ cần biết về event contract, không biết về producer
const { createConsumer } = require('../../shared/utils/kafkaClient');
const { EVENT_TOPICS } = require('../../shared/event-types');

module.exports = async () => {
  const consumer = await createConsumer('notification-group');

  await consumer.subscribe({
    topic: EVENT_TOPICS.REGISTRATION_CREATED,
    fromBeginning: true,
  });

  await consumer.run({
    eachMessage: async ({ message }) => {
      try {
        const eventData = JSON.parse(message.value?.toString() || '{}');

        // Contract validation - consumer defines what it needs
        const requiredFields = ['eventId', 'userId', 'userEmail'];
        for (const field of requiredFields) {
          if (!eventData[field]) {
            console.warn(`⚠ Missing required field: ${field}`);
            return; // Skip processing
          }
        }

        // Consumer có thể xử lý độc lập
        await sendRegistrationConfirmationEmail(eventData);

      } catch (error) {
        console.error('❌ Error processing registration event:', error);
      }
    },
  });
};

// Consumer không cần biết producer là gì, chỉ cần event format
async function sendRegistrationConfirmationEmail({ eventId, userId,
userEmail }) {
  // Fetch additional data if needed (loose coupling)
  const eventDetails = await fetchEventDetails(eventId);

  const emailContent = {
    to: userEmail,
    subject: `Registration Confirmed: ${eventDetails.name}`,
    body: `Your registration for ${eventDetails.name} has been confirmed.`
  };

  await sendEmail(emailContent);
}
```

Bước 5: API Gateway Pattern cho Service Abstraction

```
// gateway/src/routes/users.js - Single entry point
module.exports = async function (fastify, opts) {
  // Gateway routes requests, but services remain decoupled
  fastify.post('/users', async (request, reply) => {
    try {
      // Forward to user-service
      const response = await fastify.httpClient.post(
        'http://user-service:3001/users',
        request.body
      );

      return response.data;
    } catch (error) {
      reply.code(error.response?.status || 500);
      return { error: error.message };
    }
  });

  fastify.post('/registrations', async (request, reply) => {
    try {
      // Forward to registration-service
      const response = await fastify.httpClient.post(
        'http://registration-service:3003/registrations',
        request.body
      );

      return response.data;
    } catch (error) {
      reply.code(error.response?.status || 500);
      return { error: error.message };
    }
  });
};

// Services don't know about each other, only about events
// Gateway provides unified interface but services stay decoupled
```

Bước 6: Independent Deployment Configuration

```yaml
# docker-compose.yml - Each service can be deployed independently
version: '3.8'
services:
  user-service:
    build: ./user-service
    environment:
      - DATABASE_URL=postgres://postgres:admin@postgres:5432/userdb
      - KAFKA_BROKERS=kafka:9092
    depends_on:
      - postgres
      - kafka
    # Can be scaled independently

  notification-service:
    build: ./notification-service
    environment:
      - KAFKA_BROKERS=kafka:9092
      - SMTP_HOST=${SMTP_HOST}
    depends_on:
      - kafka
    # No database dependency - truly independent

  auditlog-service:
    build: ./auditlog-service
    environment:
      - MONGODB_URI=mongodb://mongo:27017/auditdb
      - KAFKA_BROKERS=kafka:9092
    depends_on:
      - mongo
      - kafka
    # Different database technology - independent choice
```

So Sánh: Tight Coupling vs Loose Coupling

❌ Tight Coupling (Cách cũ - Monolithic)

```javascript
// Tất cả logic trong 1 service - tight coupling
class UserController {
  async createUser(userData) {
    // 1. Tạo user
    const user = await this.userService.create(userData);

    // 2. Gửi email trực tiếp - tight coupling với email service
    await this.emailService.sendWelcomeEmail(user.email, user.name);

    // 3. Log audit trực tiếp - tight coupling với audit service
    await this.auditService.log('USER_CREATED', user.id);
    // 4. Cập nhật thống kê - tight coupling với stats service
    await this.statsService.incrementUserCount();

    return user;
  }
}

// Vấn đề của cách này:
// - Nếu email service down → toàn bộ tạo user thất bại
// - Thêm service mới → phải sửa code UserController
// - Khó test, khó scale, khó maintain
```

✅ Loose Coupling (Event-Driven Architecture)

```
// user-service: Chỉ lo tạo user và phát event
class UserController {
  async createUser(userData) {
    // 1. Tạo user - single responsibility
    const user = await this.userService.create(userData);

    // 2. Phát event - không biết ai sẽ xử lý
    await this.eventPublisher.publish('user.created', {
      userId: user.id,
      username: user.username,
      userEmail: user.email,
      timestamp: new Date().toISOString()
    });

    return user; // Xong việc, không cần biết gì khác
  }
}

// notification-service: Consumer độc lập
class NotificationConsumer {
  async handleUserCreated(event) {
    // Chỉ quan tâm đến event data, không biết producer
    await this.emailService.sendWelcomeEmail(
      event.userEmail,
      event.username
    );
  }
}

// auditlog-service: Consumer độc lập khác
class AuditConsumer {
  async handleUserCreated(event) {
    await this.auditRepo.log({
      eventType: 'USER_CREATED',
      userId: event.userId,
      timestamp: event.timestamp
    });
  }
}

// Lợi ích:
// - Email service down → user vẫn tạo được
// - Thêm service mới → chỉ cần listen event, không sửa code cũ
// - Dễ test từng phần riêng biệt
// - Scale độc lập từng service
```

Công Cụ và Bước Kiểm Tra Chất Lượng

1. Kiểm tra Scalability

Công cụ:

- **K6**: Load testing tool
- **Apache JMeter**: Performance testing
- **Kafka Consumer Lag Monitoring**: Theo dõi throughput
- **Docker Compose Scaling**: Horizontal scaling test

Bước thực hiện:

Bước 1: Cài đặt K6 và tạo test script

```
# Cài đặt K6
npm install -g k6

# Tạo file test scalability
cat > scalability-test.js << 'EOF'
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '2m', target: 100 }, // Ramp up to 100 users
    { duration: '5m', target: 100 }, // Stay at 100 users
    { duration: '2m', target: 200 }, // Ramp up to 200 users
    { duration: '5m', target: 200 }, // Stay at 200 users
    { duration: '2m', target: 0 },   // Ramp down to 0 users
  ],
};

export default function() {
  // Test user creation
  let userPayload = JSON.stringify({
    username: `testuser_${Math.random()}`,
    email: `test_${Math.random()}@example.com`,
    password: 'password123'
  });
  let userResponse = http.post('http://localhost:3007/users', userPayload,
{
    headers: { 'Content-Type': 'application/json' },
  });

  check(userResponse, {
    'user creation status is 201': (r) => r.status === 201,
    'user creation response time < 500ms': (r) => r.timings.duration <
500,
  });

  sleep(1);

  // Test event registration if user created successfully
  if (userResponse.status === 201) {
    let regPayload = JSON.stringify({
      eventId: '123e4567-e89b-12d3-a456-426614174000',
      userId: JSON.parse(userResponse.body).id,
      userEmail: JSON.parse(userResponse.body).email
    });

    let regResponse = http.post('http://localhost:3007/registrations',
regPayload, {
      headers: { 'Content-Type': 'application/json' },
    });

    check(regResponse, {
      'registration status is 201': (r) => r.status === 201,
      'registration response time < 300ms': (r) => r.timings.duration <
300,
    });
  }
}
EOF
```

Bước 2: Chạy test và scale services

```
# Chạy test
k6 run scalability-test.js

# Scale services trong khi test đang chạy
docker-compose up --scale user-service=3 --scale registration-service=2

# Monitoring Kafka consumer lag
docker exec kafka kafka-consumer-groups.sh --bootstrap-server
localhost:9092 --describe --all-groups
```

Bước 3: Monitoring throughput code

```
// monitoring/throughput-monitor.js
const { kafka } = require('../shared/utils/kafkaClient');

async function monitorConsumerLag() {
  const admin = kafka.admin();

  try {
    await admin.connect();

    // Get consumer groups
    const groups = await admin.listGroups();

    for (const group of groups.groups) {
      const groupDescription = await
admin.describeGroups([group.groupId]);
      console.log(`Consumer Group: ${group.groupId}`);

      // Get group offsets
      const offsets = await admin.fetchOffsets({
        groupId: group.groupId,
        topics: ['user.created', 'registration.created',
'notification.sent']
      });

      console.log('Current offsets:', offsets);

      // Calculate lag (simplified)
      for (const topicOffset of offsets) {
        console.log(`Topic: ${topicOffset.topic}`);
        for (const partitionOffset of topicOffset.partitions) {
          console.log(`Partition ${partitionOffset.partition}: Offset
${partitionOffset.offset}`);
        }
      }
    }
  } finally {
    await admin.disconnect();
  }
}

// Run every 10 seconds
setInterval(monitorConsumerLag, 10000);
```

2. Kiểm tra Reliability

Công cụ:

- **Chaos Engineering**: Chaos Monkey simulation
- **Message Verification**: Kiểm tra message persistence
- **Dead Letter Queue Testing**: Test error handling
- **Health Check Monitoring**: Service availability

Bước thực hiện:

Bước 1: Tạo script Chaos Testing

```bash
# Tạo script Chaos Testing
cat > chaos-test.sh << 'EOF'
#!/bin/bash

echo "Starting Chaos Engineering Test..."

# Kill random service
SERVICES=("user-service" "event-service" "registration-service"
"notification-service" "auditlog-service")
RANDOM_SERVICE=${SERVICES[$RANDOM % ${#SERVICES[@]}]}

echo "Killing service: $RANDOM_SERVICE"
docker-compose kill $RANDOM_SERVICE

# Wait and observe
echo "Waiting 30 seconds to observe system behavior..."
sleep 30

# Check if other services still working
echo "Checking remaining services health..."
curl -f http://localhost:3007/health || echo "Gateway down"
curl -f http://localhost:3001/health || echo "User service down"
curl -f http://localhost:3002/health || echo "Event service down"

# Restart the killed service
echo "Restarting $RANDOM_SERVICE"
docker-compose up -d $RANDOM_SERVICE

echo "Chaos test completed"
EOF

chmod +x chaos-test.sh
./chaos-test.sh
```

Bước 2: Test Message Persistence

```javascript
// test-message-persistence.js
const { kafka } = require('./shared/utils/kafkaClient');

async function testMessagePersistence() {
  const producer = kafka.producer();
  const consumer = kafka.consumer({ groupId: 'test-persistence' });

  try {
    await producer.connect();
    await consumer.connect();

    // Send test message
    const testMessage = {
      userId: 999,
      eventId: 'test-event',
      timestamp: new Date().toISOString()
    };

    await producer.send({
      topic: 'registration.created',
      messages: [{ value: JSON.stringify(testMessage) }]
    });

    console.log('Message sent');

    // Verify message still exists
    await consumer.subscribe({ topic: 'registration.created',
fromBeginning: true });

    let messageFound = false;
    await consumer.run({
      eachMessage: async ({ message }) => {
        const data = JSON.parse(message.value.toString());
        if (data.userId === 999) {
          messageFound = true;
          console.log('Message persisted successfully:', data);
        }
      }
    });

    setTimeout(() => {
      if (messageFound) {
        console.log('Reliability test PASSED');
      } else {
        console.log('Reliability test FAILED');
      }
      process.exit(0);
    }, 5000);

  } catch (error) {
    console.error('Error in persistence test:', error);
  }
}

testMessagePersistence();
```

Bước 3: Test Idempotency

```
// reliability/idempotency-test.js
const axios = require('axios');
```

```
async function testIdempotency() {
  console.log('Testing Idempotency...');

  // Create user multiple times with same data
  const userData = {
    username: 'idempotency-test-user',
    email: 'idempotency@test.com',
    password: 'password123'
  };

  const requests = [];
  for (let i = 0; i < 5; i++) {
    requests.push(
      axios.post('http://localhost:3007/users', userData)
        .catch(error => error.response)
    );
  }

  const responses = await Promise.all(requests);

  // First should succeed, others should fail gracefully
  const successCount = responses.filter(r => r.status === 201).length;
  const duplicateCount = responses.filter(r => r.status === 400 &&
    r.data.error.includes('already exists')).length;

  console.log(`Success responses: ${successCount}`);
  console.log(`Duplicate responses: ${duplicateCount}`);

  if (successCount === 1 && duplicateCount === 4) {
    console.log('Idempotency test PASSED');
  } else {
    console.log('Idempotency test FAILED');
  }
}

testIdempotency();
```

3. Kiểm tra Maintainability

Công cụ:

- **ESLint**: Code quality analysis
- **Dependency Graph Analysis**: Service coupling analysis
- **API Contract Testing**: Event schema validation
- **Documentation Coverage**: API documentation completeness

Bước thực hiện:

Bước 1: Code Quality Analysis

```bash
# Tạo ESLint config
cat > .eslintrc.js << 'EOF'
module.exports = {
  env: {
    commonjs: true,
    es2021: true,
    node: true,
  },
  extends: ['eslint:recommended'],
  parserOptions: {
    ecmaVersion: 12,
  },
  rules: {
    'complexity': ['warn', 10],
    'max-depth': ['warn', 3],
    'max-lines-per-function': ['warn', 50],
    'no-console': 'off',
  },
};
EOF

# Run ESLint on all services
for service in user-service event-service registration-service notification-service auditlog-service gateway; do
  echo "Analyzing $service..."
  npx eslint $service/src --ext .js
done
```

Bước 2: Service Coupling Analysis

```javascript
// analyze-coupling.js
const fs = require('fs');
const path = require('path');

function analyzeCoupling() {
  const services = ['user-service', 'event-service', 'registration-service', 'notification-service', 'auditlog-service', 'gateway'];
  const coupling = {};

  services.forEach(service => {
    coupling[service] = {
      directCalls: [],
      eventDependencies: [],
      sharedModules: []
    };

    // Analyze source files
    const srcDir = path.join(service, 'src');
    if (fs.existsSync(srcDir)) {
      scanDirectory(srcDir, service, coupling);
    }
  });

  console.log('Service Coupling Analysis:');
  console.log(JSON.stringify(coupling, null, 2));

  // Calculate coupling metrics
  let totalDirectCalls = 0;
  let totalEventCalls = 0;

  Object.values(coupling).forEach(serviceData => {
    totalDirectCalls += serviceData.directCalls.length;
    totalEventCalls += serviceData.eventDependencies.length;
  });

  console.log(`\nMetrics:`);
  console.log(`Direct service calls: ${totalDirectCalls}`);
  console.log(`Event-based communications: ${totalEventCalls}`);
  console.log(`Coupling ratio: ${totalDirectCalls/(totalDirectCalls + totalEventCalls)}`);
}

function scanDirectory(dir, service, coupling) {
  const files = fs.readdirSync(dir);

  files.forEach(file => {
    const filePath = path.join(dir, file);
    const stat = fs.statSync(filePath);

    if (stat.isDirectory()) {
      scanDirectory(filePath, service, coupling);
    } else if (file.endsWith('.js')) {
      const content = fs.readFileSync(filePath, 'utf8');

      // Look for direct HTTP calls to other services
      const httpCallMatches = content.match(/http:\/\/[\w-]+:\d+/g);
      if (httpCallMatches) {
        coupling[service].directCalls.push(...httpCallMatches);
      }

      // Look for Kafka event usage
      const eventMatches = content.match(/EVENT_TOPICS\.\w+/g);
      if (eventMatches) {
        coupling[service].eventDependencies.push(...eventMatches);
      }
    }
  });
}

analyzeCoupling();
```

Bước 3: Event Schema Validation Test

```
// test-event-schemas.js
const Joi = require('joi');
const { EVENT_TOPICS } = require('./shared/event-types');

// Define expected schemas for each event type
const eventSchemas = {
  [EVENT_TOPICS.USER_CREATED]: Joi.object({
    userId: Joi.number().required(),
    username: Joi.string().required(),
    userEmail: Joi.string().email().required(),
    timestamp: Joi.string().isoDate().required()
  }),

  [EVENT_TOPICS.REGISTRATION_CREATED]: Joi.object({
    userId: Joi.number().required(),
    eventId: Joi.string().uuid().required(),
    userEmail: Joi.string().email().required(),
    timestamp: Joi.string().isoDate().required()
  }),

  [EVENT_TOPICS.NOTIFICATION_SENT]: Joi.object({
    userId: Joi.number().required(),
    eventId: Joi.string().uuid().required(),
    recipientEmail: Joi.string().email().required(),
    subject: Joi.string().required(),
    timestamp: Joi.string().isoDate().required()
  })
};

function validateEventSchemas() {
  console.log('Validating Event Schemas...');

  // Sample events to validate
  const sampleEvents = {
    [EVENT_TOPICS.USER_CREATED]: {
      userId: 123,
      username: 'testuser',
      userEmail: 'test@example.com',
      timestamp: '2024-03-15T10:30:00Z'
    },

    [EVENT_TOPICS.REGISTRATION_CREATED]: {
      userId: 123,
      eventId: '123e4567-e89b-12d3-a456-426614174000',
      userEmail: 'test@example.com',
      timestamp: '2024-03-15T10:30:00Z'
    },

    [EVENT_TOPICS.NOTIFICATION_SENT]: {
      userId: 123,
      eventId: '123e4567-e89b-12d3-a456-426614174000',
      recipientEmail: 'test@example.com',
      subject: 'Event Registration Confirmation',
      timestamp: '2024-03-15T10:30:00Z'
    }
  };

  let validSchemas = 0;
  let totalSchemas = Object.keys(eventSchemas).length;

  Object.entries(eventSchemas).forEach(([eventType, schema]) => {
    const sampleData = sampleEvents[eventType];

    const { error } = schema.validate(sampleData);
    if (error) {
      console.log(`❌ ${eventType}: ${error.message}`);
    } else {
      console.log(`✅ ${eventType}: Schema valid`);
      validSchemas++;
    }
  });

  console.log(`\nSchema Validation Results:`);
  console.log(`Valid schemas: ${validSchemas}/${totalSchemas}`);
  console.log(`Schema compliance:
${(validSchemas/totalSchemas*100).toFixed(1)}%`);
}

validateEventSchemas();
```

Metrics và KPIs để Đánh Giá

Scalability Metrics

- **Throughput**: > 1000 events/second
- **Response Time**: P95 < 200ms
- **Resource Utilization**: CPU < 70%, Memory < 80%
- **Horizontal Scale Factor**: Có thể scale 5x mà không degradation

Reliability Metrics

- **Message Delivery**: 99.9% success rate
- **System Uptime**: > 99.5%
- **Error Rate**: < 0.1%
- **Recovery Time**: < 30 seconds sau khi service restart

Maintainability Metrics

- **Code Complexity**: Cyclomatic complexity < 10
- **Coupling Ratio**: < 20% direct calls, > 80% event-driven
- **Documentation Coverage**: > 90%
- **Test Coverage**: > 80%

Kết Luận

Event-Driven Architecture trong dự án này tối ưu hóa cho:

1. **Scalability**: Kafka partitioning + microservice independence
2. **Reliability**: Message persistence + fault isolation + audit logging
3. **Maintainability**: Loose coupling + clear event contracts + service independence

Các công cụ testing được thiết kế để đảm bảo những đặc tính này được duy trì qua các chu kỳ phát triển và deployment.