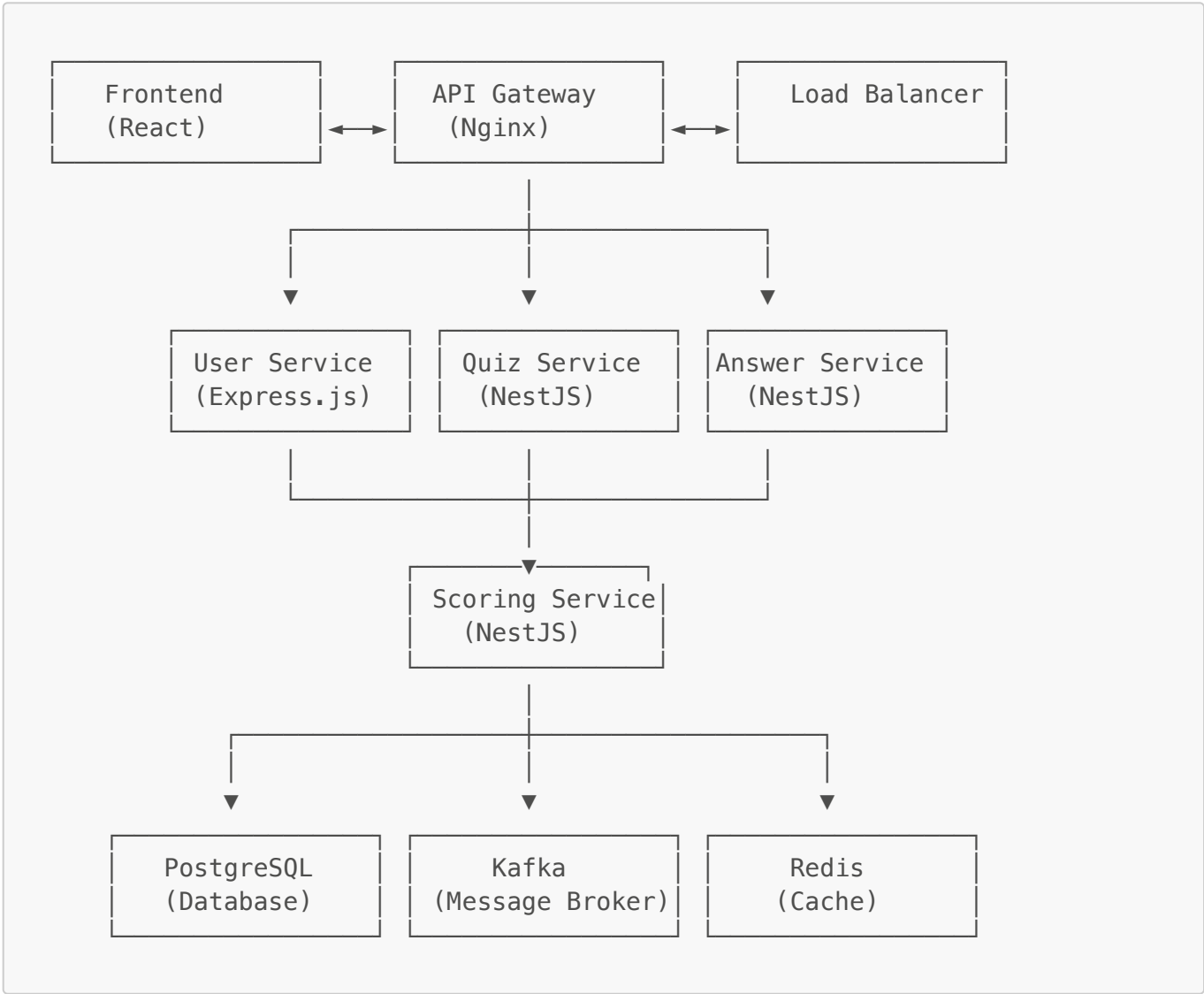
Vẽ và giải thích góc nhìn logic và góc nhìn process của Event-driven Architecture được đề xuất trong project này ? Giải thích các công cụ sử dụng và từng bước để viết mã nguồn cho một tính năng kiểm tra tính hợp lệ của dữ liệu đầu vào và ghi dữ liệu vào hệ thống nếu hợp lệ, theo góc nhìn logic và process đề xuất.

1. Góc nhìn Logic của Event-driven Architecture

1.1 Kiến trúc Logic Tổng quan



1.2 Logic Components

A. Services Logic:

- **User Service**: Xác thực, quản lý user, tham gia quiz
- **Quiz Service**: Quản lý lifecycle quiz, WebSocket gateway

- **Answer Service**: Xử lý câu trả lời, validation logic
- **Scoring Service**: Tính điểm, ranking, leaderboard

B. Data Flow Logic:

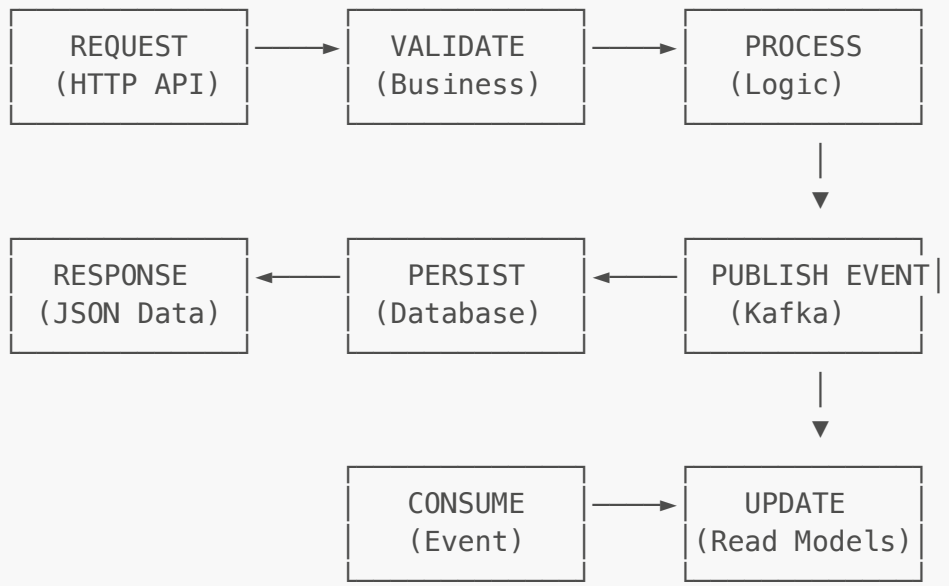
- **Synchronous**: Request/Response cho CRUD operations
- **Asynchronous**: Events cho business process
- **Real-time**: WebSocket cho live updates

C. Event Types:

```
Events:
  - player.joined: User tham gia quiz
  - quiz.started: Quiz bắt đầu
  - answer.submitted: Gửi câu trả lời
  - score.updated: Cập nhật điểm số
```

2. Góc nhìn Process của Event-driven Architecture

2.1 Process Flow Diagram



2.2 Process Steps

Step 1: Request Processing

- Client gửi HTTP request
- API Gateway route đến service phù hợp
- Service validate request data

Step 2: Business Logic

- Service thực hiện business rules

- Validate dữ liệu theo domain logic
- Kiểm tra constraints và permissions

Step 3: Event Publishing

- Nếu valid → Persist data + Publish event
- Event được gửi qua Kafka topics
- Other services consume events asynchronously

Step 4: Read Model Updates

- Services cập nhật read models
- Cache được refresh (Redis)
- WebSocket notify real-time clients

3. Công cụ Sử dụng trong Architecture

3.1 Infrastructure Tools

A. Container & Orchestration:

```
Docker & Docker Compose:
  - Containerize từng microservice
  - Định nghĩa dependencies trong docker-compose.yml
  - Isolation và reproducible environments
  - Health checks và auto-restart
```

B. Message Broker:

```
Apache Kafka:
  - Event streaming platform
  - Topics: quiz-events, user-events, answer-events
  - Partitions để horizontal scaling
  - Consumer groups cho load balancing

Zookeeper:
  - Kafka cluster coordination
  - Leader election cho partitions
  - Configuration management
```

C. Databases:

```
PostgreSQL:
  - Primary data store
  - ACID transactions
  - Relational integrity
  - Separate databases cho mỗi service
```

```
Redis:
  - Session storage
  - Real-time quiz state
  - Pub/Sub cho WebSocket
  - Caching frequently accessed data
```

3.2 Development Tools

A. Backend Frameworks:

```
NestJS (Quiz, Answer, Scoring):
  - TypeScript-first framework
  - Dependency injection
  - Built-in Kafka integration
  - WebSocket gateway support

Express.js (User Service):
  - Lightweight HTTP server
  - Middleware support
  - JWT authentication
  - Simple REST API development
```

B. ORM & Database Tools:

```
TypeORM:
  - Entity modeling
  - Migration management
  - Query builder
  - Connection pooling

Prisma (User Service):
  - Type-safe database client
  - Schema migration
  - Auto-generated types
```

C. Communication Tools:

```
Socket.IO:
  - Real-time WebSocket communication
  - Room-based messaging
  - Automatic fallbacks
  - Cross-browser compatibility

KafkaJS:
  - Node.js Kafka client
  - Producer/Consumer APIs
```

```
  - Transaction support
  - Retry mechanisms
```

4. Implementation Steps cho Data Validation & Persistence

4.1 Step 1: Thiết lập Input Validation

A. DTO (Data Transfer Object) Definition:

```
// services/answer-service/src/dto/submit-answer.dto.ts
import { IsUUID, IsNotEmpty, IsNumber, Min, Max } from 'class-validator';

export class SubmitAnswerDto {
  @IsUUID()
  @IsNotEmpty()
  quizId: string;

  @IsUUID()
  @IsNotEmpty()
  questionId: string;

  @IsUUID()
  @IsNotEmpty()
  playerId: string;

  @IsNotEmpty()
  submittedAnswer: any; // JSON data

  @IsNumber()
  @Min(0)
  @Max(300000) // 5 minutes max
  responseTime: number;
}
```

B. Validation Pipe Setup:

```
// services/answer-service/src/main.ts
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Global validation pipe
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true,        // Strip không-defined properties
    forbidNonWhitelisted: true, // Throw error nếu có extra properties
    transform: true,        // Auto-transform types
  }));
```

```
  await app.listen(3002);
}
```

4.2 Step 2: Business Logic Validation

A. Service-level Validation:

```
// services/answer-service/src/services/answer.service.ts
@Injectable()
export class AnswerService {
  async submitAnswer(submitAnswerDto: SubmitAnswerDto): Promise<Answer> {
    // 1. Validate business rules
    await this.validateAnswerSubmission(submitAnswerDto);

    // 2. Process if valid
    const answer = await this.processValidAnswer(submitAnswerDto);

    // 3. Publish event
    await this.publishAnswerEvent(answer);

    return answer;
  }

  private async validateAnswerSubmission(dto: SubmitAnswerDto):
Promise<void> {
    // Check duplicate submission
    const existingAnswer = await this.answerRepository.findOne({
      where: {
        quizId: dto.quizId,
        questionId: dto.questionId,
        playerId: dto.playerId
      }
    });

    if (existingAnswer) {
      throw new ConflictException('Answer already submitted for this
question');
    }

    // Validate quiz state
    const quiz = await this.quizService.getQuizById(dto.quizId);
    if (quiz.status !== 'ACTIVE') {
      throw new BadRequestException('Quiz is not active');
    }

    // Validate question timing
    const currentQuestion = await
this.quizService.getCurrentQuestion(dto.quizId);
    if (currentQuestion.id !== dto.questionId) {
      throw new BadRequestException('Question is not currently active');
    }
```

```
    // Validate response time
    if (dto.responseTime > currentQuestion.timeLimit * 1000) {
      throw new BadRequestException('Response time exceeded question time
limit');
    }
  }
}
```

4.3 Step 3: Data Persistence với Transaction

A. Database Transaction:

```
// services/answer-service/src/services/answer.service.ts
import { DataSource } from 'typeorm';

@Injectable()
export class AnswerService {
  constructor(
    private dataSource: DataSource,
    private answerRepository: Repository<Answer>
  ) {}

  private async processValidAnswer(dto: SubmitAnswerDto): Promise<Answer>
{
    return await this.dataSource.transaction(async manager => {
      // 1. Create answer entity
      const answer = manager.create(Answer, {
        ...dto,
        submittedAt: new Date(),
        isCorrect: await this.checkAnswerCorrectness(dto)
      });

      // 2. Save to database
      const savedAnswer = await manager.save(Answer, answer);

      // 3. Update quiz statistics (trong cùng transaction)
      await manager.increment(
        QuizStats,
        { quizId: dto.quizId },
        'totalAnswers',
        1
      );

      return savedAnswer;
    });
  }

  private async checkAnswerCorrectness(dto: SubmitAnswerDto):
Promise<boolean> {
    const question = await this.quizService.getQuestion(dto.questionId);
```

```
    // Compare submitted answer với correct answer
    return JSON.stringify(dto.submittedAnswer) ===
           JSON.stringify(question.correctAnswer);
  }
}
```

4.4 Step 4: Event Publishing

A. Kafka Event Publishing:

```
// services/answer-service/src/services/answer.service.ts
@Injectable()
export class AnswerService {
  constructor(
    private kafkaService: KafkaService
  ) {}

  private async publishAnswerEvent(answer: Answer): Promise<void> {
    const event = {
      eventId: randomUUID(),
      eventType: 'answer.submitted',
      aggregateId: answer.quizId,
      aggregateType: 'quiz',
      payload: {
        answerId: answer.id,
        playerId: answer.playerId,
        questionId: answer.questionId,
        isCorrect: answer.isCorrect,
        responseTime: answer.responseTime,
        submittedAt: answer.submittedAt
      },
      timestamp: new Date().toISOString()
    };

    await this.kafkaService.emit('answer.submitted', event);

    this.logger.log(`Published answer.submitted event for quiz
${answer.quizId}`);
  }
}
```

4.5 Step 5: Error Handling & Rollback

A. Global Exception Filter:

```
// shared/filters/all-exceptions.filter.ts
@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
```

```
  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();

    let status = 500;
    let message = 'Internal server error';

    if (exception instanceof HttpException) {
      status = exception.getStatus();
      message = exception.message;
    } else if (exception instanceof QueryFailedError) {
      status = 400;
      message = 'Database constraint violation';
    }

    // Log error
    this.logger.error(`${request.method} ${request.url}`, exception);

    response.status(status).json({
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      message,
    });
  }
}
```

5. Complete Flow Example

5.1 Answer Submission Flow

```
// Complete flow từ HTTP request đến event publishing

1. HTTP Request:
POST /api/answers/submit
{
  "quizId": "123e4567-e89b-12d3-a456-426614174000",
  "questionId": "123e4567-e89b-12d3-a456-426614174001",
  "playerId": "123e4567-e89b-12d3-a456-426614174002",
  "submittedAnswer": { "option": "A" },
  "responseTime": 15000
}

2. Validation Process:
- DTO validation (format, types, constraints)
- Business logic validation (duplicate, timing, state)
- Permission checks

3. Data Processing:
- Calculate correctness
```

```
- Save với transaction
- Update related entities

4. Event Publishing:
- Create event object
- Publish to Kafka topic 'answer-events'
- Log success

5. Response:
{
  "id": "123e4567-e89b-12d3-a456-426614174003",
  "isCorrect": true,
  "submittedAt": "2024-01-15T10:30:00Z"
}

6. Async Processing:
- Scoring service consumes event
- Calculate và update player score
- Update leaderboard
- Notify via WebSocket
```

6. Monitoring & Observability

6.1 Health Checks

```
// Health check endpoints
@Controller('health')
export class HealthController {
  @Get()
  checkHealth() {
    return {
      status: 'ok',
      timestamp: new Date().toISOString(),
      database: 'connected',
      kafka: 'connected'
    };
  }
}
```

6.2 Metrics Collection

```
// Metrics cho monitoring
- Request count và response time
- Database connection pool status
- Kafka producer/consumer lag
- Event processing time
- Error rates by service
```

Kiến trúc này đảm bảo **data integrity**, **scalability**, và **fault tolerance** thông qua event-driven patterns, validation layers, và robust error handling.