Design Document

Compute

Would you run it as Lambda or containers (ECS/Fargate)? Why?

Option A: AWS Lambda (Serverless)

- **Pros**:
  - No server management
  - Auto-scaling
  - Pay per request
  - Good for variable traffic
- **Cons**:
  - Cold starts
  - 15-minute timeout limit
  - Limited control over runtime
- **Use Case**: Low to moderate traffic, event-driven processing

Option B: ECS/Fargate (Containers)

- **Pros**:
  - More control
  - No cold starts
  - Long-running processes
  - Better for high throughput
- **Cons**:
  - More operational overhead
  - Cost even when idle
- **Use Case**: High traffic, predictable load, need for long-running processes

Recommendation: Use **Lambda** for feed handlers and API endpoints (stateless, request-driven). Use **ECS/Fargate** for order/error processors (long-running, continuous processing).

Streaming

What would replace valid_orders / error_orders (SQS, EventBridge, Kinesis, MSK)?

Current Implementation: In-memory queues with subscriber pattern.

Production Design Options:

Option A: Amazon SQS

- **Pros**: Simple, managed, built-in retry, dead-letter queues
- **Cons**: Not ideal for high-throughput, limited ordering guarantees
- **Use Case**: Good for moderate volume (< 1000 messages/sec per partner)

Option B: Amazon EventBridge

- **Pros**: Event-driven, rule-based routing, integration with many AWS services
- **Cons**: Less control over message ordering
- **Use Case**: Good for event-driven architecture with multiple consumers

Option C: Amazon Kinesis Data Streams

- **Pros**: High throughput, ordered processing, replay capability
- **Cons**: More complex, higher cost, requires shard management
- **Use Case**: High-volume, real-time processing needs

Option D: Amazon MSK (Managed Kafka)

- **Pros**: Industry standard, high throughput, strong ordering, multiple consumers
- **Cons**: Most complex, higher operational overhead
- **Use Case**: Enterprise-grade, high-volume, multiple consumer groups

Recommendation: Start with **SQS** for simplicity and cost-effectiveness. Migrate to **Kinesis** or **MSK** if volume exceeds SQS limits or ordering becomes critical.

Storage

What would you use for orders and errors (DynamoDB vs RDS)? Why?

Current Implementation: In-memory Map and arrays.

Production Design Options:

Option A: Amazon DynamoDB

- **Pros**:
    - Serverless, auto-scaling
    - Fast queries with GSI
    - Built-in TTL for error cleanup
    - Strong consistency for sequence numbers
- **Cons**:
    - Query patterns limited by key design
    - Cost at scale
- **Use Case**: High-throughput writes, simple queries, serverless architecture

Option B: Amazon RDS (PostgreSQL/MySQL)

- **Pros**:
    - SQL queries, complex aggregations
    - ACID transactions
    - Familiar for most developers
- **Cons**:
    - Requires capacity planning
    - Scaling is more complex
- **Use Case**: Complex queries, relational data, existing SQL expertise

Recommendation: Use **DynamoDB** for orders (high write volume, simple queries) and **RDS** if complex analytics are needed. For errors, use **DynamoDB** with TTL.

Table Design (DynamoDB):

- **Orders Table**:
    - Partition Key: `partnerId`
    - Sort Key: `sequenceNumber`
    - GSI: `eventTime-index` (for time-range queries)
- **Errors Table**:
    - Partition Key: `partnerId`
    - Sort Key: `receivedTime`
    - TTL: `expirationTime` (30 days)

Frontend hosting

How to host the React app (S3 + CloudFront recommended)

Current Implementation: React app served by Vite dev server.

Production Design:

- **Hosting**: S3 + CloudFront
    - Build React app: `npm run build`
    - Upload `dist/` to S3 bucket

    - Configure CloudFront distribution
    - Custom domain with Route 53
- **API Integration**:
    - API Gateway endpoint
    - CORS configured
    - Environment variables for API URL

Configuration & secrets

env vars, parameter store, secrets manager

Environment Variables

- Store in **AWS Systems Manager Parameter Store** (for non-sensitive config)
- Store in **AWS Secrets Manager** (for API keys, database credentials)
- Use IAM roles for service-to-service authentication

Configuration Examples

- Partner configurations (API keys, endpoints)
- Validation rules
- Stream names
- Database connection strings

Observability

logs/metrics/alarms you'd set up

Logging

- **CloudWatch Logs**: Centralized logging for all services
- Structured logging (JSON format)
- Log retention: 30 days (configurable)

Metrics

- **CloudWatch Metrics**:
    - Request count, latency (API Gateway)
    - Processing time (Lambda/ECS)

    - Queue depth (SQS)
    - Error rate
    - Order volume per partner
- **Custom Metrics**:
    - Orders processed per minute
    - Validation failure rate
    - Average processing time

Alarms

- High error rate (> 5% of requests)
- Queue depth threshold (SQS > 1000 messages)
- API latency > 1 second (p95)
- Processing failures
- Storage capacity warnings

Tracing

- **AWS X-Ray**: Distributed tracing for request flows
- Track requests from API Gateway through Lambda/ECS to DynamoDB

Dashboards

- **CloudWatch Dashboards**:
    - Real-time order processing rate
    - Error rate by partner
    - API performance
    - Queue metrics

CI/CD

pipeline stages (lint/test/build/package/deploy), and how you'd roll out safely

Pipeline Stages

1. **Source**: GitHub/GitLab/Bitbucket
2. **Build**:
    - Install dependencies
    - Run linters (ESLint, Prettier)
    - Run unit tests

    - Build TypeScript
3. **Test**:
    - Integration tests
    - End-to-end tests (optional)
4. **Package**:
    - Create Docker images (if using ECS)
    - Package Lambda functions
5. **Deploy**:
    - Deploy to staging environment
    - Run smoke tests
    - Deploy to production (with approval)

Deployment Strategy

Blue/Green Deployment (ECS):

- Deploy new version alongside old
- Route traffic gradually (10% → 50% → 100%)
- Rollback by switching traffic back

Canary Deployment (Lambda):

- Deploy new version to 10% of traffic
- Monitor metrics
- Gradually increase to 100%

Infrastructure as Code:

- **AWS CDK** or **Terraform** for infrastructure
- Version controlled
- Repeatable deployments

Pipeline Tools

- **AWS CodePipeline** + **CodeBuild** + **CodeDeploy**
- Or **GitHub Actions** / **GitLab CI** with AWS deployment