Distributes Systems Lab 4 Report

Submitted By : Shahu Abhay Kor | **Student ID :** 25203580

1. Introduction

This lab demonstrates the practical application of microservices architectural principles by designing, implementing, and deploying a distributed e-commerce order processing system. The application consists of two microservices that communicate via REST APIs and are deployed on Kubernetes.

Technology Stack:

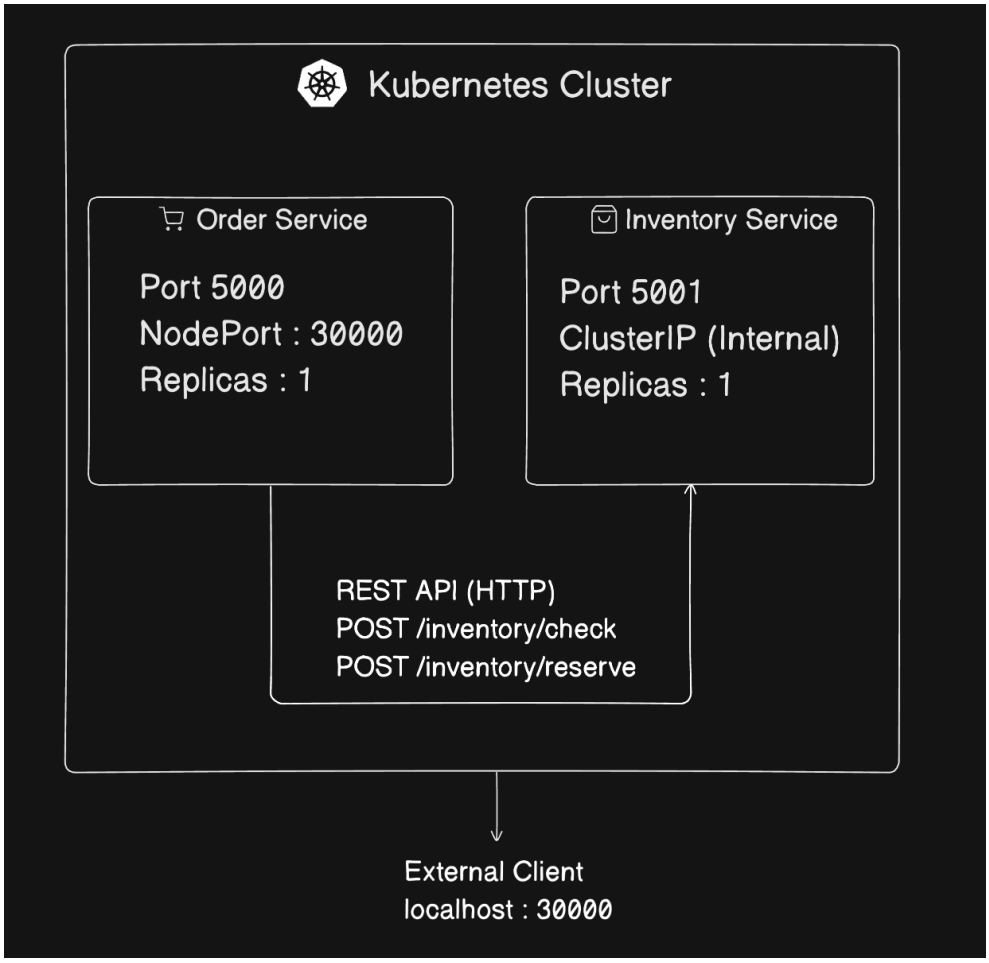
- **Language:** Python 3.12
- **Framework:** Flask
- **Containerization:** Docker
- **Orchestration:** Kubernetes (Docker Desktop)
- **Communication:** Synchronous REST

Application Overview:
The system comprises an Order Service (manages order lifecycle) and an Inventory Service (manages product stock).
These services demonstrate key distributed systems concepts including service boundaries, inter-service communication, and containerized deployment.

Github Repo : https://github.com/ShahuKor/Distributed-Systems-Lab-4

---

2. System Design & Architecture



2.2 Service Boundaries

Order Service Responsibilities:

- Order creation, retrieval, and cancellation
- Orchestrating inventory checks and reservations
- Customer-facing API

Inventory Service Responsibilities:

- Product catalog management
- Inventory tracking and availability checks
- Stock reservation and release

2.3 Docker Containerization

Each service is containerized using Docker with Python 3.12-slim base images:

```dockerfile
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 5000

CMD ["python", "app.py"]
```

2.4 Kubernetes Deployment

Inventory-service.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-service
  labels:
    app: inventory-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: inventory-service
  template:
    metadata:
      labels:
        app: inventory-service
    spec:
      containers:
        - name: inventory-service
          image: inventory-service:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 5001
              name: http
          env:
            - name: PORT
              value: "5001"
          livenessProbe:
            httpGet:
              path: /health
              port: 5001
            initialDelaySeconds: 10
            periodSeconds: 30
          readinessProbe:
            httpGet:
              path: /health
              port: 5001
            initialDelaySeconds: 5
            periodSeconds: 10
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "256Mi"
              cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: inventory-service
  labels:
    app: inventory-service
spec:
  type: ClusterIP
  selector:
    app: inventory-service
  ports:
    - port: 5001
      targetPort: 5001
      protocol: TCP
      name: http
```

Order-service.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  labels:
    app: order-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
        - name: order-service
          image: order-service:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
              name: http
          env:
            - name: PORT
              value: "5000"
            - name: INVENTORY_SERVICE_URL
              value: "http://inventory-service:5001"
          livenessProbe:
            httpGet:
              path: /health
              port: 5000
            initialDelaySeconds: 10
            periodSeconds: 30
          readinessProbe:
            httpGet:
              path: /health
              port: 5000
            initialDelaySeconds: 5
            periodSeconds: 10
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "256Mi"
              cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: order-service
  labels:
    app: order-service
spec:
  type: NodePort
  selector:
    app: order-service
  ports:
    - port: 5000
      targetPort: 5000
      nodePort: 30000
      protocol: TCP
      name: http
```

Key Configuration:

- Each service deployed as a Kubernetes Deployment
- Health probes (liveness and readiness) for monitoring
- Resource limits (256Mi memory, 500m CPU)
- Service discovery via Kubernetes DNS
- Order Service exposed via NodePort (30000) for external access
- Inventory Service uses ClusterIP (internal only)

---

3. Architectural Analysis

3.1 Microservice Granularity

Decompose the application into two distinct services.

Granularity Disintegrators (Factors Favoring Separation):

1. **Code Volatility:** Order processing logic changes more frequently than inventory management, which means it can be updated independently without affecting the other part.

2. **Scalability Requirements:** Order creation tends to experience traffic spikes during big sales events, while inventory checks stay fairly consistent. Because of that, scaling each service on its own helps use resources more efficiently.

3. **Fault Tolerance:** If the inventory service goes down, orders can still queue or fail gracefully instead of bringing down the whole system. Keeping the services isolated helps prevent cascading failures.

4. **Domain Boundaries:** Orders and inventory belong to different business domains with their own responsibilities and lifecycles, so separating them fits naturally.

Granularity Integrators (Factors Favoring Combination):

1. **Data Relationships:** Orders and inventory are closely connected, every order depends on inventory verification and reservation.

2. **Transaction Boundaries:** Ideally, order creation and inventory deduction would be atomic. Distributed systems trade ACID transactions for eventual consistency.

3. **Network Overhead:** Calling between services over the network introduces extra latency compared to just calling functions within the same process.

Justification: Even though orders and inventory are tightly linked, the advantages of splitting them up (like independent scaling, better fault isolation, and more flexible deployments) end up being more valuable for an e-commerce system that needs to handle fluctuating loads.

Using a two-phase approach (check availability first, then reserve) provides consistency that's good enough.

3.2 Inter-Service Communication

Trade-Off Analysis of using synchronous REST API calls between services:

Advantages of REST:

- **Immediate Consistency:** Users get instant order confirmation right after the inventory is verified.
- **Simplicity:** It's a well-understood protocol and has a lot of existing tooling, which makes it easier to work with.
- **Request-Response Semantics:** This method fits naturally with the whole "check then reserve" workflow.
- **Debugging:** HTTP logs make it easy to see the exact request/response flow when trying to debug issues.

Disadvantages of REST:

- **Tight Coupling:** The Order Service depends directly on the Inventory Service being available.
- **Blocking:** Threads have to wait for responses from the inventory service, which can reduce throughput when the system is under heavy load.
- **Latency:** Network round-trips add delay (check + reserve = 2 calls per order)

- **Cascading Failures:** If Inventory Service slows down, Order Service threads can exhaust

Alternative Considered: Asynchronous Messaging

Using a message broker (RabbitMQ, Kafka) would provide:

- Temporal decoupling (services don't need to be online simultaneously)
- Better fault tolerance (messages queue during outages)
- Traffic buffering (handles spikes gracefully)

However, asynchronous messaging introduces:

- Infrastructure complexity (need message broker)
- Eventual consistency (users don't get immediate confirmation)
- More difficult debugging (harder to trace message flows)

For this application, the user experience requirement for immediate order confirmation makes synchronous REST the better choice. The strong consistency guarantees and implementation simplicity outweigh coupling concerns. In production, a hybrid approach could be used: synchronous for critical path, asynchronous for notifications.

3.3 Deployment Architecture

Deployed on Kubernetes with health probes and resource limits.

Benefits:

- **High Availability:** Kubernetes automatically restarts failed pods
- **Service Discovery:** Built-in DNS resolves “inventory-service” without hardcoded IPs
- **Health Monitoring:** Liveness probes restart unhealthy pods, readiness probes remove pods from load balancing
- **Resource Management:** CPU/memory limits prevent resource exhaustion
- **Declarative Configuration:** YAML manifests define desired state

Trade-Offs:

- Operational complexity vs production-ready capabilities
- Learning curve vs industry-standard tooling
- Configuration verbosity vs powerful features

4. Key Architectural Decisions (ADRs)

ADR-1: Use Synchronous REST Communication

Context: Order Service needs to verify inventory before creating orders.

Decision: Implement synchronous HTTP REST calls.

Consequences:

- Immediate consistency and user feedback
- Simpler implementation (no message broker)
- Easier debugging
- Tight coupling between services
- Potential for cascading failures
- Higher latency from network calls

Rationale: User experience prioritized over architectural purity.

ADR-2: Separate Order and Inventory Services

Context: Application needs both order management and inventory tracking.

Decision: Decompose into two independent services.

Consequences:

- Independent scaling and deployment
- Fault isolation
- Team autonomy
- Technology flexibility
- No ACID transactions across services
- Network overhead
- Increased operational complexity

Rationale: Scalability and operational benefits outweigh coordination costs for production e-commerce.

ADR-3: Scale to Single Replica

Context: During testing with 2 replicas per service, orders were created successfully but couldn't be retrieved consistently. Inventory wasn't updating properly.

Root cause: each pod has independent in-memory storage, and Kubernetes load balancer distributed requests across different pods.

Decision: Scale both services to 1 replica for the lab demonstration.

Consequences:

- Data consistency across requests
- Simpler debugging
- Demonstrates understanding of distributed systems challenges
- No high availability (single point of failure)
- No load distribution
- Not production-ready

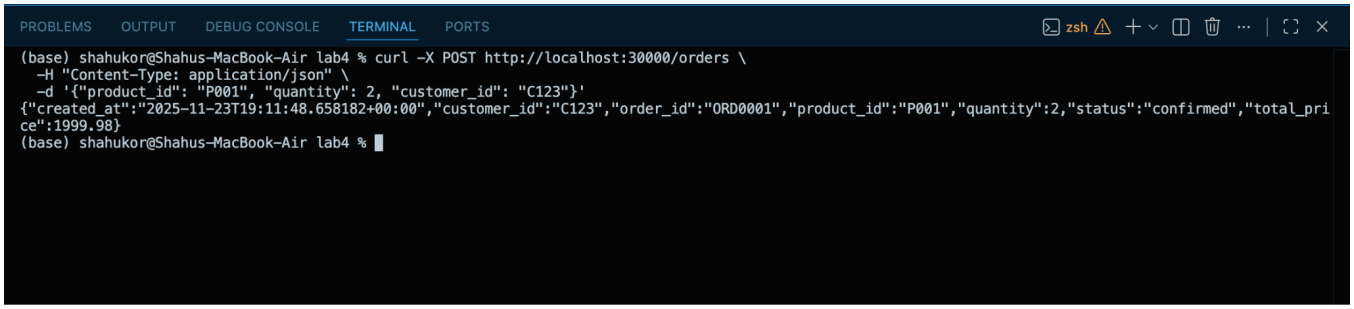
Rationale: This revealed a fundamental microservices principle: **services should be stateless with externalized state**. Production systems would use a shared database (PostgreSQL, MongoDB) instead of in-memory storage, allowing safe scaling to multiple replicas. This hands-on discovery reinforced concepts of data consistency, state management, and the CAP theorem.

---

5. Testing & Validation

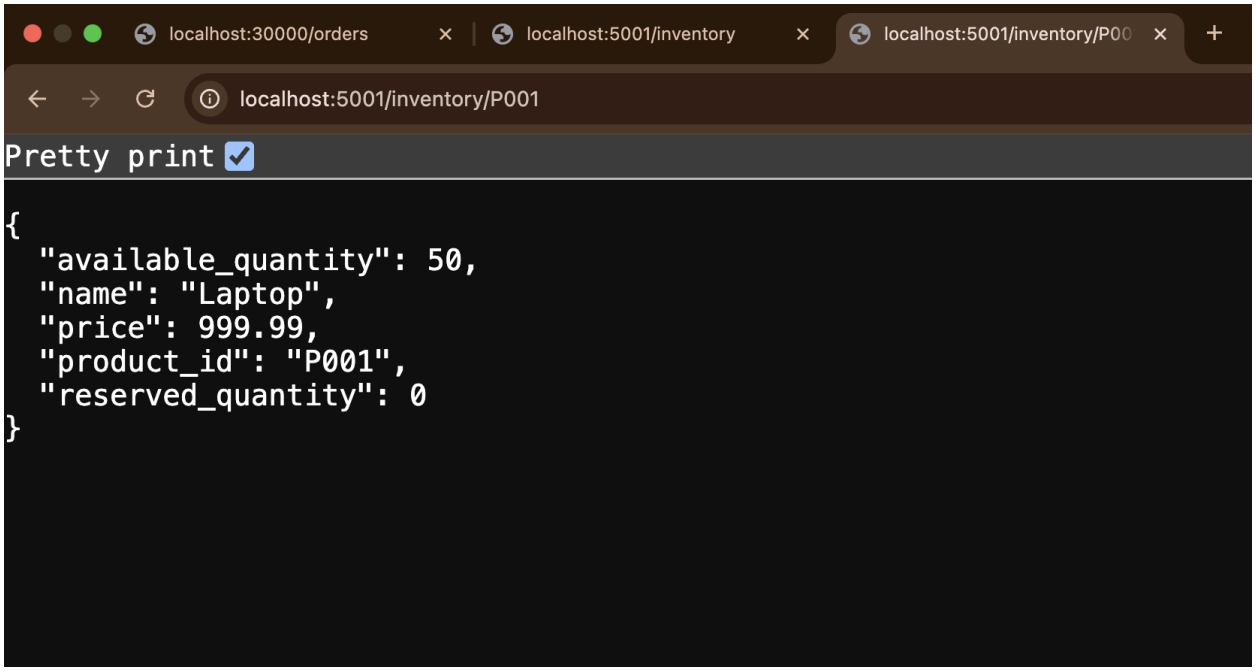
Functional Testing

Test 1: Successful Order Creation

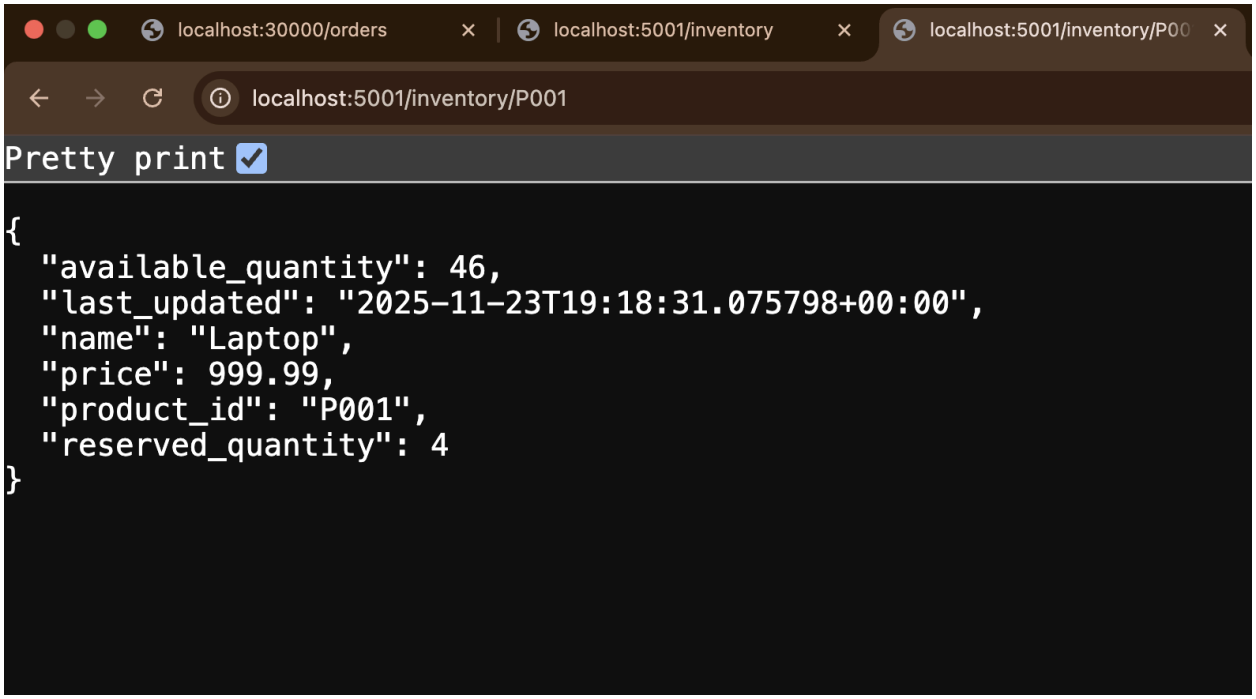


Test 2: Inventory Verification

Before order: available quantity = 50



After 2 orders of 2 units each: available quantity = 46



6. Key Learnings

1. Every architectural decision has consequences. No "best practices," only "least worst trade-offs."

2. The replica issue highlighted why distributed state is challenging and why external databases are essential.

3. K8s provides production capabilities but requires investment to understand properly.

4. In-memory storage was fine for demo but clearly showed its limitations when scaling.

5. Learning `kubectl logs`, `kubectl describe`, and understanding load balancing was crucial.

---

7. Conclusion

This lab successfully demonstrated microservices design, implementation, and deployment. The e-commerce order processing system illustrates service decomposition, REST communication, and Kubernetes orchestration while highlighting practical challenges.

I have used AI to help summarize the document and grammatically correct the sentences where needed, but nothing was changed from the original texts.