Microservices Questions and Practical Exercises with SOLID Principles

This document provides a comprehensive set of microservices-related questions and practical exercises, categorized by difficulty (Basic, Intermediate, Advanced) and topic, with detailed answers and solutions. The content integrates the SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to emphasize best practices in microservices design. Python is used for coding exercises, and tasks include Docker and Kubernetes configurations to align with modern deployment practices. The questions are designed for interview preparation or certifications, covering architecture, communication, deployment, and SOLID principles.

Basic Microservices Concepts

Theory Questions

1. **What is a microservices architecture?**
   - **Answer**: Microservices architecture is a design approach where an application is built as a collection of small, loosely coupled services, each responsible for a specific function. Each service runs as an independent process, communicates via well-defined APIs (e.g., REST, gRPC), and can be developed, deployed, and scaled independently. Unlike monolithic architectures, microservices enable modularity, flexibility, and easier maintenance but introduce complexity in distributed systems management.
2. **What is the Single Responsibility Principle (SRP) in the context of microservices?**
   - **Answer**: SRP, part of SOLID, states that a class or module should have only one reason to change, meaning it should have a single responsibility. In microservices, this translates to each service handling a specific business capability (e.g., a user service manages user data, not payments). This ensures services are focused, easier to maintain, and independently deployable.
3. **How do microservices differ from a monolithic architecture?**
   - **Answer**: In a monolithic architecture, all application components (UI, business logic, database access) are tightly coupled in a single codebase and deployed as one unit. Microservices break the application into independent services, each with its own codebase, database, and deployment pipeline. Monoliths are simpler to develop initially but harder to scale; microservices offer scalability and flexibility but require complex coordination.
4. **What are the benefits of microservices?**
   - **Answer**:
     - **Scalability**: Each service can be scaled independently based on demand.
     - **Flexibility**: Different services can use different technologies (polyglot architecture).
     - **Resilience**: Failure in one service doesn't necessarily affect others.
     - **Deployability**: Independent deployments enable faster updates.
     - **Maintainability**: Smaller codebases are easier to understand and modify.
5. **What are the challenges of microservices?**
   - **Answer**:
     - **Distributed Systems Complexity**: Managing inter-service communication, latency, and consistency.
     - **Data Management**: Each service may have its own database, complicating data consistency.
     - **Monitoring**: Requires robust logging and tracing across services.
     - **Deployment Overhead**: Managing multiple services increases operational complexity.
     - **Testing**: End-to-end testing is harder due to distributed nature.
6. **What is the Open/Closed Principle (OCP) in microservices design?**
   - **Answer**: OCP states that software entities should be open for extension but closed for modification. In microservices, this means designing services to allow new functionality (e.g., adding endpoints) without changing existing code. For example, use API versioning or pluggable modules to extend service behavior without altering its core logic.
7. **What is an API Gateway in microservices?**
   - **Answer**: An API Gateway is a single entry point for client requests, routing them to appropriate microservices. It handles cross-cutting concerns like authentication, rate limiting, and request aggregation, reducing client complexity. Examples include AWS API Gateway and Kong.
8. **What is the difference between synchronous and asynchronous communication in microservices?**

- **Answer**: Synchronous communication (e.g., REST, gRPC) involves direct, real-time requests where the caller waits for a response. Asynchronous communication (e.g., message queues like RabbitMQ, Kafka) involves sending messages without waiting for an immediate response, improving decoupling and resilience but adding latency.

9. **What is a Service Registry in microservices?**

   - **Answer**: A Service Registry (e.g., Consul, Eureka) is a centralized database that tracks the locations (e.g., IP addresses, ports) of microservice instances. Services register themselves, and clients or other services query the registry for discovery, enabling dynamic scaling and load balancing.

10. **What is the Liskov Substitution Principle (LSP) in microservices?**

    - **Answer**: LSP states that objects of a derived class should be substitutable for objects of the base class without altering program correctness. In microservices, this applies to APIs or interfaces; for example, if a new version of a service implements the same interface, clients should use it without changes. This ensures backward compatibility in service contracts.

Practical Exercises

1. **Create a simple Python microservice**

   - **Task**: Build a Python microservice using Flask that returns a list of users, adhering to SRP (one service, one responsibility).
   - **Solution**:

```
# user_service.py
from flask import Flask, jsonify

app = Flask(__name__)

users = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]

@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_service.py .
CMD ["python", "user_service.py"]
```

```
# requirements.txt
flask==2.3.3
```

```
docker build -t user-service .
docker run -d -p 5000:5000 --name user-service user-service
```

     - **Explanation**: The Flask service handles user data (SRP: manages only users). The Dockerfile uses a slim Python image for efficiency. Test by accessing `http://localhost:5000/users`. Output: `[{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]`.

2. **Deploy a microservice to Kubernetes**

   - **Task**: Deploy the user service to Kubernetes with a Deployment and ClusterIP Service.
   - **Solution**:

```
# user-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: user-service:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: user-service
  namespace: default
spec:
  selector:
    app: user-service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
  type: ClusterIP
```

```
docker tag user-service:latest <your-registry>/user-service:latest
docker push <your-registry>/user-service:latest
kubectl apply -f user-deployment.yaml
kubectl get deployments,services,pods
```

- **Explanation**: The Deployment runs 2 replicas of the user service, and the Service exposes it internally. Push the image to a registry (e.g., Docker Hub) and apply the manifest. Verify with `kubectl get`.

3. **Implement a health check for a microservice**

- **Task**: Add a health endpoint to the user service, adhering to OCP (extendable without modifying core logic).
- **Solution**:

```
# user_service_health.py
from flask import Flask, jsonify

app = Flask(__name__)

users = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]

@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)

@app.route('/health', methods=['GET'])
def health():
    return jsonify({"status": "healthy"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_service_health.py .
CMD ["python", "user_service_health.py"]
```

```
docker build -t user-service-health .
docker run -d -p 5000:5000 --name user-service-health user-service-health
```

- **Explanation**: The `/health` endpoint extends functionality without modifying the user logic (OCP). Test with `curl http://localhost:5000/health`. Output: `{"status": "healthy"}`.

Intermediate Microservices Concepts

Theory Questions

1. **What is the Interface Segregation Principle (ISP) in microservices?**
   - **Answer**: ISP states that clients should not be forced to depend on interfaces they don't use. In microservices, this means designing APIs with specific, minimal endpoints for each client's needs. For example, a user service should expose only user-related endpoints, not unrelated features like payment processing.
2. **What is the Dependency Inversion Principle (DIP) in microservices?**
   - **Answer**: DIP states that high-level modules should not depend on low-level modules; both should depend on abstractions. In microservices, this means services depend on abstract interfaces (e.g., APIs or message contracts) rather than specific implementations. For example, a service communicates via a message queue interface, not a specific queue like RabbitMQ.
3. **What is circuit breaking in microservices?**
   - **Answer**: Circuit breaking prevents cascading failures by stopping requests to a failing service after a threshold of failures. When the circuit is "open," requests are blocked or fallback logic is executed. Tools like Resilience4j or Istio implement circuit breakers.
4. **What is the difference between orchestration and choreography in microservices?**
   - **Answer**: Orchestration involves a central controller (e.g., a conductor service) directing the workflow of microservices. Choreography uses event-driven communication, where services react to events independently. Orchestration is centralized but less flexible; choreography is decentralized and more resilient.
5. **What is a Saga pattern in microservices?**

- **Answer**: The Saga pattern manages distributed transactions across microservices. It breaks a transaction into a series of local transactions, each executed by a service. If a step fails, compensating transactions (rollbacks) are triggered. Sagas can be choreographed (event-driven) or orchestrated (centrally managed).

6. **What is the role of a message queue in microservices?**
   - **Answer**: Message queues (e.g., RabbitMQ, Kafka) enable asynchronous communication by decoupling producers and consumers. They handle message persistence, retries, and load balancing, improving scalability and fault tolerance.
7. **What is the time complexity of service discovery in a microservices architecture?**
   - **Answer**: Service discovery (e.g., via Consul, Eureka) typically uses a hash table for service lookup, resulting in O(1) average-case time complexity. However, network latency or registry updates may add practical overhead.
8. **What is eventual consistency in microservices?**
   - **Answer**: Eventual consistency means that data across microservices becomes consistent over time, not immediately. Each service updates its database independently, and changes propagate via events or replication, suitable for distributed systems where strong consistency is impractical.
9. **What are the security considerations for microservices?**
   - **Answer**:
     - Use mutual TLS for secure inter-service communication.
     - Implement API Gateway authentication (e.g., OAuth2, JWT).
     - Encrypt sensitive data in transit and at rest.
     - Use network policies in Kubernetes to restrict traffic.
     - Regularly scan container images for vulnerabilities.
     - Enforce least privilege with RBAC.
10. **How do microservices handle data consistency?**
    - **Answer**: Microservices often use separate databases per service (database-per-service pattern), leading to eventual consistency. Techniques include:
      - **Event Sourcing**: Store events as the source of truth, replaying them to reconstruct state.
      - **CQRS (Command Query Responsibility Segregation)**: Separate read and write operations for scalability.
      - **Sagas**: Coordinate distributed transactions.
      - **Compensating Transactions**: Undo failed operations.

Practical Exercises

1. **Implement a microservices system with two services**
   - **Task**: Create a user service and an order service, communicating via REST, with Dependency Inversion (both depend on an abstract API). Deploy with Docker Compose.
   - **Solution**:

```
# user_service.py
from flask import Flask, jsonify

app = Flask(__name__)

users = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]

@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    return jsonify(user) if user else (jsonify({"error": "User not found"}), 404)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```python
# order_service.py
from flask import Flask, jsonify
import requests

app = Flask(__name__)

orders = [
    {"id": 1, "user_id": 1, "item": "Book"},
    {"id": 2, "user_id": 2, "item": "Pen"}
]

@app.route('/orders/<int:user_id>', methods=['GET'])
def get_orders(user_id):
    user_response = requests.get(f'http://user-service:5000/users/{user_id}')
    if user_response.status_code != 200:
        return jsonify({"error": "User not found"}), 404
    user = user_response.json()
    user_orders = [o for o in orders if o["user_id"] == user_id]
    return jsonify({"user": user["name"], "orders": user_orders})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001)
```

```yaml
# docker-compose.yml
version: '3'
services:
  user-service:
    build: ./user-service
    ports:
      - "5000:5000"
    networks:
      - app-network
  order-service:
    build: ./order-service
    ports:
      - "5001:5001"
    depends_on:
      - user-service
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
# Directory structure
# user-service/Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_service.py .
CMD ["python", "user_service.py"]
```

```
# order-service/Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY order_service.py .
CMD ["python", "order_service.py"]
```

```
# requirements.txt (both services)
flask==2.3.3
requests==2.31.0
```

```
docker-compose up --build
```

- **Explanation**: The user service exposes a `/users/<id>` endpoint, and the order service depends on it via HTTP (DIP: depends on user service's API, not implementation). Docker Compose runs both services on a bridge network. Test with `curl http://localhost:5001/orders/1`. Output: `{"user": "Alice", "orders": [{"id": 1, "user_id": 1, "item": "Book"}]}`.

2. **Implement an asynchronous microservice with Kafka**

- **Task**: Create a Python microservice that publishes user creation events to Kafka, adhering to ISP (minimal interface for consumers).
- **Solution**:

```
# user_producer.py
from flask import Flask, request, jsonify
from kafka import KafkaProducer
import json

app = Flask(__name__)
producer = KafkaProducer(bootstrap_servers='kafka:9092',
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))

@app.route('/users', methods=['POST'])
def create_user():
    user = request.get_json()
    producer.send('user-events', user)
    return jsonify({"status": "User event published"}), 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# docker-compose.yml
version: '3'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    networks:
      - app-network
  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    networks:
      - app-network
  user-service:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - kafka
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_producer.py .
CMD ["python", "user_producer.py"]
```

```
# requirements.txt
flask==2.3.3
kafka-python==2.0.2
```

```
docker-compose up --build
curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "name": "Alice"}' http://localhost:5000/users
```

- **Explanation**: The service publishes user creation events to a Kafka topic, exposing a minimal POST endpoint (ISP). Kafka and Zookeeper run in Docker Compose for message queueing. Test by sending a user creation request. Output: `{"status": "User event published"}`.

3. **Check for valid API requests using a circuit breaker**

- **Task**: Implement a Python microservice with a circuit breaker to handle failures when calling another service.
- **Solution**:

```
# order_service_circuit.py
from flask import Flask, jsonify
import requests
from circuitbreaker import circuit

app = Flask(__name__)

@circuit(failure_threshold=5, recovery_timeout=30)
def call_user_service(user_id):
    response = requests.get(f'http://user-service:5000/users/{user_id}')
    response.raise_for_status()
    return response.json()

@app.route('/orders/<int:user_id>', methods=['GET'])
def get_orders(user_id):
    try:
        user = call_user_service(user_id)
        orders = [{"id": 1, "user_id": user_id, "item": "Book"}]
        return jsonify({"user": user["name"], "orders": orders})
    except Exception as e:
        return jsonify({"error": "Service unavailable, circuit open"}), 503

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001)
```

```
# Use same docker-compose.yml and user-service from previous task
# Dockerfile for order-service
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY order_service_circuit.py .
CMD ["python", "order_service_circuit.py"]
```

```
# requirements.txt
flask==2.3.3
requests==2.31.0
circuitbreaker==2.0.0
```

```
docker-compose up --build
curl http://localhost:5001/orders/1
```

- **Explanation**: The order service uses a circuit breaker to handle failures when calling the user service. After 5 failures, the circuit opens for 30 seconds, returning a fallback response. Output: `{"user": "Alice", "orders": [{"id": 1, "user_id": 1, "item": "Book"}]}` or `{"error": "Service unavailable, circuit open"}` if the user service fails.

Advanced Microservices Concepts

Theory Questions

1. **How do SOLID principles improve microservices design?**

   - **Answer**:
     - **SRP**: Ensures each service has a single responsibility, reducing complexity and improving scalability.
     - **OCP**: Allows services to add new features (e.g., new endpoints) without modifying existing code, supporting API versioning.
     - **LSP**: Ensures new service versions can replace older ones without breaking clients, maintaining compatibility.
     - **ISP**: Prevents services from exposing unnecessary endpoints, reducing coupling and improving security.
     - **DIP**: Enables services to depend on abstractions (e.g., APIs, message contracts), facilitating technology changes and testing.

2. **What is the time complexity of API Gateway routing?**
    - **Answer**: API Gateway routing typically involves a hash table or trie for path matching, yielding O(1) or O(m) complexity, where m is the path length. However, additional processing (e.g., authentication, rate limiting) may add O(n) overhead, where n is the number of rules or tokens.
3. **What is Event Sourcing in microservices?**
    - **Answer**: Event Sourcing stores the state of a system as a sequence of events, rather than a single state snapshot. Each event represents a state change, and the current state is reconstructed by replaying events. It's useful for auditing, debugging, and eventual consistency in microservices.
4. **What is CQRS (Command Query Responsibility Segregation)?**
    - **Answer**: CQRS separates read (query) and write (command) operations into different models. In microservices, this means separate services or databases for reads (e.g., optimized for queries) and writes (e.g., optimized for transactions), improving scalability and performance.
5. **How do you handle distributed tracing in microservices?**
    - **Answer**: Distributed tracing tracks requests across multiple services to identify performance bottlenecks or failures. Tools like Jaeger, Zipkin, or OpenTelemetry instrument services to generate trace IDs and spans, which are aggregated for analysis. Each service adds metadata to requests, enabling end-to-end visibility.
6. **What are the best practices for microservices deployment in Kubernetes?**
    - **Answer**:
        - Use Deployments for stateless services and StatefulSets for stateful ones.
        - Implement liveness and readiness probes for health checks.
        - Use Horizontal Pod Autoscalers (HPA) for dynamic scaling.
        - Apply network policies to restrict inter-service traffic.
        - Use ConfigMaps and Secrets for configuration management.
        - Leverage Ingress for external access with SSL termination.
        - Monitor with Prometheus and Grafana for metrics and alerts.
7. **What is the role of Istio in microservices?**
    - **Answer**: Istio is a service mesh that manages microservices communication, providing features like traffic routing, load balancing, security (mTLS), observability (tracing, metrics), and fault injection. It operates as a sidecar proxy (Envoy) alongside each service, simplifying complex networking tasks.
8. **What is a monotonic queue in microservices?**
    - **Answer**: A monotonic queue maintains elements in sorted order (increasing or decreasing). In microservices, it's not commonly used directly but can apply to task scheduling or event processing (e.g., prioritizing events in a Kafka consumer based on timestamps).
9. **How do you handle versioning in microservices APIs?**
    - **Answer**: Versioning strategies include:
        - **URI Versioning**: Include version in the URL (e.g., `/v1/users`).
        - **Header Versioning**: Use custom headers (e.g., `Accept: application/vnd.api.v1+json`).
        - **Query Parameter Versioning**: Use query params (e.g., `/users?version=1`).
        - Maintain backward compatibility (LSP) to avoid breaking clients and deprecate old versions gradually.
10. **What are the disadvantages of microservices?**
    - **Answer**:
        - **Complexity**: Managing distributed systems increases operational overhead.
        - **Latency**: Inter-service communication adds network latency.
        - **Data Consistency**: Eventual consistency complicates transactions.
        - **Monitoring**: Requires advanced tools for tracing and logging.
        - **Testing**: End-to-end testing is challenging due to multiple services.

Practical Exercises

1. **Implement a microservice with Event Sourcing**
    - **Task**: Create a Python microservice that stores user creation events in a list (simulating an event store) and reconstructs state, adhering to SRP.
    - **Solution**:

```
# user_event_sourcing.py
from flask import Flask, request, jsonify

app = Flask(__name__)
event_store = []

def reconstruct_user_state(user_id):
    user_events = [e for e in event_store if e["user_id"] == user_id]
    state = {"id": user_id, "name": None}
    for event in user_events:
        if event["type"] == "UserCreated":
            state["name"] = event["data"]["name"]
    return state

@app.route('/users', methods=['POST'])
def create_user():
    data = request.get_json()
    event = {"type": "UserCreated", "user_id": data["id"], "data": data}
    event_store.append(event)
    return jsonify({"status": "User created"}), 201

@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    state = reconstruct_user_state(user_id)
    return jsonify(state) if state["name"] else (jsonify({"error": "User not found"}), 404)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_event_sourcing.py .
CMD ["python", "user_event_sourcing.py"]
```

```
# requirements.txt
flask==2.3.3
```

```
docker build -t user-event-service .
docker run -d -p 5000:5000 --name user-event-service user-event-service
curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "name": "Alice"}' http://localhost:5000/users
curl http://localhost:5000/users/1
```

- **Explanation**: The service stores events in a list (SRP: manages user events) and reconstructs state by replaying them. Test by creating a user and retrieving their state. Output: `{"id": 1, "name": "Alice"}`.

2. **Implement CQRS in a microservice**

- **Task**: Create a Python microservice with separate read and write endpoints for user data, adhering to ISP (minimal interfaces).
- **Solution**:

```
# user_cqrs.py
from flask import Flask, request, jsonify

app = Flask(__name__)
write_store = {}
read_store = {}

@app.route('/users/write', methods=['POST'])
def write_user():
    data = request.get_json()
    user_id = data["id"]
    write_store[user_id] = data
    read_store[user_id] = data  # Sync for simplicity
    return jsonify({"status": "User written"}), 201

@app.route('/users/read/<int:user_id>', methods=['GET'])
def read_user(user_id):
    user = read_store.get(user_id)
    return jsonify(user) if user else (jsonify({"error": "User not found"}), 404)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_cqrs.py .
CMD ["python", "user_cqrs.py"]
```

```
# requirements.txt
flask==2.3.3
```

```
docker build -t user-cqrs .
docker run -d -p 5000:5000 --name user-cqrs user-cqrs
curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "name": "Alice"}' http://localhost:5000/users/write
curl http://localhost:5000/users/read/1
```

- **Explanation**: The service separates write (`/users/write`) and read (`/users/read`) endpoints (ISP: distinct interfaces). In production, use separate databases for read/write. Output: `{"id": 1, "name": "Alice"}`.

3. **Deploy microservices with Istio service mesh**

- **Task**: Deploy the user and order services in Kubernetes with Istio for traffic management, assuming Istio is installed.
- **Solution**:

```yaml
# user-order-istio.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: user-service:latest
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: user-service
  namespace: default
spec:
  selector:
    app: user-service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
      - name: order-service
        image: order-service:latest
        ports:
        - containerPort: 5001
---
apiVersion: v1
kind: Service
metadata:
  name: order-service
  namespace: default
spec:
```

```
  selector:
    app: order-service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5001
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: order-service-vs
  namespace: default
spec:
  hosts:
  - order-service
  http:
  - route:
    - destination:
        host: order-service
        port:
          number: 80
    retries:
      attempts: 3
      perTryTimeout: 2s
```

```
kubectl apply -f user-order-istio.yaml
kubectl get deployments,services,virtualservices
```

- **Explanation**: The user and order services are deployed with Istio's VirtualService for retry logic. Istio's sidecar proxies handle traffic. Push images to a registry, apply the manifest, and verify. Test with `kubectl port-forward svc/order-service 5001:80` and `curl http://localhost:5001/orders/1`.

4. **Implement a Saga pattern for distributed transactions**

- **Task**: Simulate a Saga by coordinating user creation and order placement across services using Kafka events.
- **Solution**:

```
# user_saga.py
from flask import Flask, request, jsonify
from kafka import KafkaProducer
import json

app = Flask(__name__)
producer = KafkaProducer(bootstrap_servers='kafka:9092',
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))

@app.route('/users', methods=['POST'])
def create_user():
    user = request.get_json()
    event = {"type": "UserCreated", "user_id": user["id"], "data": user}
    producer.send('saga-events', event)
    return jsonify({"status": "User created, event published"}), 201

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
# order_saga.py
from flask import Flask, jsonify
from kafka import KafkaConsumer
import threading

app = Flask(__name__)
orders = []

def consume_events():
    consumer = KafkaConsumer('saga-events', bootstrap_servers='kafka:9092',
                             value_deserializer=lambda x: json.loads(x.decode('utf-8')))
    for message in consumer:
        event = message.value
        if event["type"] == "UserCreated":
            orders.append({"id": len(orders) + 1, "user_id": event["user_id"], "item": "Default Item"})

@app.route('/orders/<int:user_id>', methods=['GET'])
def get_orders(user_id):
    user_orders = [o for o in orders if o["user_id"] == user_id]
    return jsonify(user_orders)

if __name__ == '__main__':
    threading.Thread(target=consume_events, daemon=True).start()
    app.run(host='0.0.0.0', port=5001)
```

```
# Use same docker-compose.yml as Kafka example
```

```
docker-compose up --build
curl -X POST -H "Content-Type: application/json" -d '{"id": 1, "name": "Alice"}' http://localhost:5000/users
curl http://localhost:5001/orders/1
```

- **Explanation**: The user service publishes a `UserCreated` event, and the order service consumes it to create an order (choreographed Saga).
  Output: `[{"id": 1, "user_id": 1, "item": "Default Item"}]`.

5. **Monitor microservices with Prometheus**

- **Task**: Instrument the user service with Prometheus metrics and deploy in Kubernetes.
- **Solution**:

```python
# user_service_metrics.py
from flask import Flask, jsonify
from prometheus_client import Counter, start_http_server

app = Flask(__name__)
request_count = Counter('user_service_requests_total', 'Total requests to user service')

users = [
    {"id": 1, "name": "Alice"},
    {"id": 2, "name": "Bob"}
]

@app.route('/users', methods=['GET'])
def get_users():
    request_count.inc()
    return jsonify(users)

@app.route('/metrics', methods=['GET'])
def metrics():
    from prometheus_client import generate_latest
    return generate_latest()

if __name__ == '__main__':
    start_http_server(8000)
    app.run(host='0.0.0.0', port=5000)
```

```
# user-prometheus.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
      - name: user-service
        image: user-service-metrics:latest
        ports:
        - containerPort: 5000
        - containerPort: 8000
---
apiVersion: v1
kind: Service
metadata:
  name: user-service
  namespace: default
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: "8000"
spec:
  selector:
    app: user-service
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_service_metrics.py .
CMD ["python", "user_service_metrics.py"]
```

```
# requirements.txt
flask==2.3.3
prometheus-client==0.20.0
```

```
docker build -t user-service-metrics .
docker push <your-registry>/user-service-metrics:latest
kubectl apply -f user-prometheus.yaml
kubectl port-forward svc/user-service 8000:80
curl http://localhost:8000/metrics
```

- **Explanation**: The service exposes a `/metrics` endpoint for Prometheus scraping. The Service annotation enables Prometheus discovery. Test by accessing metrics. Requires a Prometheus server to scrape metrics in production.

Guidelines

- **Practice Platforms**: Use Minikube, Kind, or cloud-based Kubernetes clusters (e.g., GKE, EKS) for deployment practice. Platforms like KodeKloud Labs offer microservices sandboxes.
- **Resources**: Refer to Kubernetes documentation, Martin Fowler's microservices articles, and YouTube tutorials (e.g., TechWorld with Nana) for best practices. Study SOLID principles via Robert C. Martin's books or blogs.
- **Best Practices**: Design services with SOLID principles, use API Gateway for routing, implement circuit breakers, and monitor with Prometheus/Grafana. Ensure loose coupling and high cohesion.
- **Certifications**: Prepare for AWS Certified Solutions Architect or CNCF's KCNA/CKAD with practice tests from Whizlabs or Udemy.
- **Optimization**: Optimize microservices with efficient communication (e.g., gRPC), minimal image sizes, and auto-scaling. Use Istio for advanced traffic management.
- **Security**: Secure APIs with JWT/OAuth2, use mTLS in service meshes, and scan images with tools like `docker scout`.