Q.1. A JSON file for a movie streaming platform is provided below.

Write the REST API endpoints (based on the Open API specification) for all the CRUD operations, assuming that the API is hosted at **`https://movies.com/api/streaming`**. Provide the request body where appropriate. [5 marks]

JSON File:

```json
{
  "streaming": {
    "name": "StreamFlix",
    "location": "Online",
    "movies": [
      {
        "title": "Inception",
        "director": {
          "name": "Christopher Nolan",
          "birth_year": 1970,
          "nationality": "British-American"
        },
        "genre": {
          "name": "Sci-Fi",
          "description": "Science Fiction Movies"
        },
        "price": 3.99,
        "stock": 100,
        "id": 1
      }
    ]
  }
}
```

Fundamentals (Additional Reading)

1. **Top-level container**
   The whole document is a single JSON object with one *key*, `"streaming"`. Its value is another object that holds platform metadata and the catalogue.
2. **Platform metadata**
   Inside `"streaming"`, the keys `"name"` and `"location"` describe the streaming platform itself.
   - `"name": "StreamFlix"` is the platform's brand.
   - `"location": "Online"` indicates it's an online-only service (not a physical shop).
3. **Movie catalogue**
   “`movies": []` is an array. Each element is one movie object in the catalogue and is enclosed by braces `{}`. Arrays allow zero or more movies to be listed.
4. **Movie object shape**
   Each movie contains several fields:
   `"title"`: the film's name.

   `"director"`: a **nested object** with details about the director.

   `"genre"`: a **nested object** describing the category.

   `"price"`: rental or purchase price (decimal number).

   `"stock"`: how many units/licenses are available (integer, non-negative).

   `"id"`: a unique integer identifier for the movie.

5. **Director sub-object**
   `"director"` has:

   `"name"`: director's full name.

   `"birth_year"`: four-digit year (integer).

   `"nationality"`: free-text descriptor.

6. **Genre sub-object**
   `"genre"` has:

   `"name"`: genre label (e.g., “Sci-Fi”).

   `"description"`: human-readable explanation of the genre.

7. **Example record**
   Example array has one movie: **Inception** with its director (Christopher Nolan), genre ("Sci-Fi"), price **3.99**, stock **100**, and id **1**.

8. **Typical constraints (implied, not encoded)**

   `id` should be unique per movie.

   `stock` should be ≥ 0.

   `price` should be ≥ 0.00 with two decimal places if treated as currency.

   Director/genre are embedded here; they could be separate resources in a larger system, but this JSON models them inline within each movie.

Answer

To implement CRUD operations for the `movies` resource in the given streaming platform, the endpoints can be designed as follows:

1. Create (POST)

- **Endpoint:**
  `POST https://movies.com/api/streaming/movies`
- **Request Body:**

```
{
  "title": "Interstellar",
  "director": {
    "name": "Christopher Nolan",
    "birth_year": 1970,
    "nationality": "British-American"
  },
  "genre": {
    "name": "Sci-Fi",
    "description": "Space Exploration and Science Fiction"
  },
  "price": 4.99,
  "stock": 50
}
```

Just copy and paste the section under `/movies` within the braces `{}` and change the attributes as shown above.

2. Read (GET)

- Get all movies:
  `GET https://movies.com/api/streaming/movies`
- Get a single movie by ID:
  `GET https://movies.com/api/streaming/movies/{id}`

  Example:

  ```
  GET https://movies.com/api/streaming/movies/1
  ```

3. Update (PUT/PATCH)

- **Endpoint:**
  `PUT https://movies.com/api/streaming/movies/{id}`
- **Request Body (example – updating stock and price):**

```
{
  "price": 2.99,
  "stock": 120
}
```

4. Delete (DELETE)

- **Endpoint:**
  `DELETE https://movies.com/api/streaming/movies/{id}`
  Example: `DELETE https://movies.com/api/streaming/movies/1`

Q.2 Evaluate the following statements (agree/disagree) with a brief justification. [Answer should contain a few bullet points of no more than 2–4 lines]. [4 marks]

a. A microservices-based architecture is always more efficient than

monolithic architecture.
b. Continuous deployment eliminates the need for manual intervention in the release process.

Answer

- **a. Disagree.** Microservices can improve scalability and team autonomy, but add network overhead, operational complexity, and distributed failure modes. For small, cohesive systems, a monolith can be faster to develop, simpler to operate, and more efficient in resource use. So it will depend on the type of application. In some cases microservices may be a better choice, in others monolithic may be a better option.
- **b. Disagree.** Continuous deployment automates delivery to production, but guardrails still need human input: defining approval policies, handling rollbacks for ambiguous failures, and overseeing compliance/security exceptions. Not every change is risk-free or fully automatable.

Additional Notes

a. Microservices vs Monolith

- **Large-scale e-commerce (e.g., Amazon, Netflix, Uber)** – These companies are *documented case studies* of adopting microservices because their systems need high scalability, independent team ownership, and resilience. They explicitly moved away from monoliths due to scale and complexity.
- **Small internal HR portal / university attendance system** – Industry and academic sources agree that for small, cohesive systems with limited scale, monoliths are preferable. They are easier to build, deploy, and maintain when complexity is low. This is a commonly cited best practice in software architecture literature.

b. Continuous Deployment and Human Guardrails

- **Financial applications** – In regulated domains like banking and trading, manual approvals for production deployment are mandated (e.g., PCI DSS, SOX compliance). Continuous deployment pipelines may exist, but final pushes often require human approval.

- **Ambiguous failures (performance regressions)** – This is a well-documented limitation of automated CD. Automated tests may pass while a system experiences subtle latency or throughput degradation in production. Human monitoring and intervention are required.
- **Healthcare systems** – HIPAA and other healthcare compliance frameworks often demand human review for certain types of changes, especially when patient data or safety is involved.

Q.3 Differentiate between Edge computing, Fog computing, and Cloud computing. Provide practical examples for all three approaches. [6 marks]

Answer

Edge Computing

Edge computing means the processing happens right where the data is generated, or very close to it — for example, on sensors, devices, or gateways. The main benefit is ultra-low latency, because the data doesn't have to travel far, and reduced bandwidth, since raw streams don't need to be sent to distant servers. This is useful when quick, local decisions are needed or where connectivity is patchy.

Example: A factory robot arm uses an on-device vision model to stop instantly if a hazard is detected. Similarly, a retail security camera can count customers and trigger alerts without waiting for a cloud response.

Fog Computing

Fog computing sits in the middle, between the edge and the cloud. Instead of every device working alone or everything being sent to the cloud, fog nodes (like local servers, micro-data centers, or ISP points) collect data from many edge devices. They can preprocess, filter, or coordinate the data before passing on summaries to the cloud. This reduces cloud load and allows for faster regional decisions.

Example: In a smart city, roadside units aggregate traffic data from multiple intersections. They can adjust local traffic lights in real time, while sending aggregated trends to the cloud for broader city-wide traffic planning.

Cloud Computing

Cloud computing is centralized, relying on hyperscale data centers with massive compute and storage capacity. It excels at heavy workloads, such as analytics across millions of records, large-scale machine learning training, or long-term data archiving. While it provides scalability and global reach, the trade-off is higher latency for real-time needs.

Example: A retailer uses cloud platforms to train demand-forecasting models, run recommendation engines, and maintain a historical data lake covering all regions.

Q.4 'QuickLearn' is a new application that offers short tutorials on various subjects. The platform supports categories like technology, arts, and business, which can be filtered by difficulty levels. Users can search tutorials, add new content, view others' contributions, rate or comment on the tutorials, and delete their own uploads. Discuss the role of DevOps and AIOps in supporting this application. [4 marks]

Answer

DevOps for QuickLearn

DevOps ensures that new features on QuickLearn—like category filters, search, uploads, ratings, and comments—are delivered quickly and reliably. Continuous Integration and Continuous Deployment (CI/CD) pipelines automate testing so that features such as search accuracy, user permissions, or comment posting work correctly before going live. Infrastructure as Code tools (like Terraform) allow the entire setup—web front-end, APIs, and databases—to be deployed in a consistent and repeatable way. DevOps also brings observability: logging, metrics, and tracing help monitor system health, such as error rates or slow responses. Techniques like blue-green or canary deployments make updates safer, allowing new features to be rolled out gradually with the option to roll back instantly if issues arise. Security is baked into the process, from code scanning to secrets management, ensuring the platform is both fast and safe.

AIOps for QuickLearn

AIOps builds on this by using artificial intelligence to monitor and optimise operations. It analyses logs, metrics, and events to spot unusual patterns—for example, a sudden spike in search delays, upload failures, or bursts of inappropriate comments. It can automatically connect issues across services (like database slowdowns affecting search) and even trigger fixes, such as scaling up servers or clearing caches. AIOps also predicts usage peaks to help with capacity planning and filters out false alarms by recognising patterns, reducing the workload on engineers. For troubleshooting, it compares current incidents with past ones to speed up root-cause analysis. Beyond keeping the platform stable, AIOps can also enhance user experience by supporting smarter moderation and recommendations, separating genuine engagement signals from system health data.

Q.5 Briefly explain each API approach and identify suitable scenarios: SOAP, REST, GraphQL, WebSocket, gRPC, Server-Sent Events. [6 marks]

Answer

When we look at **SOAP**, we are dealing with the most formal and heavyweight API approach. It uses XML, defines strict contracts through WSDL, and includes built-in standards for things like security, reliability, and transactions. This makes it reliable in enterprise or financial systems where guaranteed message delivery and formal protocols are essential, but it also makes it slower and harder to work with compared to newer approaches.

Use case: Banking systems, payment gateways, or legacy integrations.

Sample request (XML over HTTP):

```
<soap:Envelope>
```

```
  <soap:Body>
    <GetBalance><AccountId>12345</AccountId></GetBalance>
  </soap:Body>
</soap:Envelope>
```

REST simplifies things by being resource-oriented and stateless, using standard HTTP verbs like GET, POST, PUT, and DELETE. It usually communicates with JSON, supports caching, and is easy to scale. That is why it is the most common choice for modern web APIs such as e-commerce platforms. However, one drawback is that REST can sometimes return either more data than the client needs or not enough, leading to inefficient communication.

Use case: CRUD operations in e-commerce, social media, or content platforms.

Sample request:

```
GET /api/users/42
Response: { "id": 42, "name": "Alice" }
```

GraphQL was designed to solve that problem. Instead of letting the server decide what data to send, the client specifies exactly what it needs, and all of it can come in a single request. This reduces over-fetching and under-fetching, which is especially useful for mobile or web applications where efficiency matters. The trade-off is that GraphQL requires a well-defined schema and more complexity on the server side compared to REST.

Use case: Apps with varying UI needs like social networks or dashboards.

Sample request (query):

```
{
  user(id: 42) { name, posts { title } }
}
```

WebSocket is quite different because it moves away from the request–response model entirely. It creates a persistent, full-duplex connection between client and server, meaning both can send data to each other in real time. This makes WebSocket the natural choice for live chats, collaborative tools, or online games.

Unlike REST or GraphQL, which are request-driven, WebSocket is event-driven, and unlike Server-Sent Events, it supports two-way communication.

Use case: Real-time chat apps, multiplayer games, collaborative editing.

Sample message (over open socket):

```
Client → { "message": "Hello" }
Server → { "reply": "Hi there!" }
```

gRPC focuses on high-performance service-to-service communication. It uses HTTP/2 along with Protocol Buffers, giving it compact, binary messages that are strongly typed. This makes it much faster than REST or GraphQL and a good fit for internal microservices or edge-to-core communication pipelines where speed and strict contracts are critical. It is less suitable for public-facing APIs but excellent for backend systems.

Use case: Microservice communication in large platforms.

Sample (IDL definition):

```
rpc GetUser(UserRequest) returns (UserResponse);
```

Finally, **Server-Sent Events (SSE)** are a lighter way to provide real-time updates. They establish a one-way channel from the server to the client, which is enough when the client only needs to listen for updates, such as stock prices, live scores, or notifications. SSE is simpler than WebSockets but only supports one-way communication, so the choice depends on whether bidirectional interaction is needed.

Use case: Live dashboards, notifications, live scores.

Sample event stream:

```
data: { "score": "2-1" }
```

Q.6 A retail organization wants to develop a predictive model to determine the likelihood of product returns. The dataset includes

purchase details, customer feedback, and product ratings. Apply the CRISP-DM methodology to analyze this problem and explain each stage in detail. [5 marks]

Answer

1) Business Understanding. Define the objective: predict the probability a purchase will be returned within the return window to reduce reverse-logistics cost and improve customer experience. Establish KPIs: AUC/PR-AUC for discrimination, calibration (Brier score), expected cost saved, and operational metrics (fewer no-fault returns). Document constraints (fairness across segments, regulatory handling of customer feedback text, explanation requirements for agents).

2) Data Understanding. Inventory available data: purchase details (SKU, price, discount, channel, delivery time), customer profile (tenure, prior returns), product attributes (category, size/fit notes), and unstructured customer feedback/ratings. Explore target leakage (e.g., using post-return workflows), class imbalance (typically low positive rate), seasonality (festive returns), and channel effects (online vs store). Perform EDA: return rates by category/size, sentiment vs returns, delivery-delay correlations.

3) Data Preparation. Handle missing values (impute or "missing" category), normalise/encode categorical features (category, brand), derive features (discount depth, size mismatches, delivery SLA breach, sentiment scores from feedback, recent return streak). Address imbalance with stratified splits, class-weighted loss, or balanced sampling. Partition data temporally to respect ordering (train on past months, validate on later months).

4) Modelling. Start with explainable baselines (regularised logistic regression), then tree-based ensembles (Gradient Boosting/Random Forest/XGBoost). Include calibrated probabilities (Platt/Isotonic) for decision thresholds tied to costs (shipping, handling, restocking). Consider text features via TF-IDF or embeddings for feedback; compare models with and without text to quantify lift.

5) Evaluation. Use AUC/PR-AUC plus business metrics: expected cost saving at chosen threshold, confusion matrix by category/size, calibration plots, and fairness slices (e.g., new vs loyal customers). Perform back-testing across months and stress tests for peak seasons. Validate operationally: do high-risk orders benefit from size guidance or extra QC?

6) Deployment. Serve the model behind an API with feature store parity to training transformations. Monitor drift (input and prediction), calibration, and return-rate lift. Implement human-in-the-loop for edge cases (e.g., very high value orders). Plan for retraining cadence (monthly/quarterly), A/B tests on interventions (pre-purchase fit prompts, post-purchase follow-ups), and clear rollback procedures.

Additional Reading

CRISP-DM: An Overview

CRISP-DM stands for **Cross-Industry Standard Process for Data Mining**. It is the most widely used, structured methodology for tackling data mining and predictive modelling projects. The key idea is that data science projects are not just about building models; they require understanding the business, preparing the data, validating outcomes, and ensuring deployment fits real-world needs.

The methodology has **six phases**, which are usually iterative rather than strictly linear.

1. Business Understanding

The first stage focuses on **what the organisation wants to achieve**. This means translating business objectives into data science goals. For example: reduce customer churn, predict fraud, or classify returns. Constraints, success criteria, and risks are also identified here.

- *Output*: A clear project charter with objectives, success metrics, and constraints.

2. Data Understanding

Next, you gather and explore the data. This involves identifying what data sources are available, assessing data quality, and exploring data distributions, correlations, or anomalies. You also look for possible **target leakage** and begin forming hypotheses about patterns in the data.

- *Output*: A description of the data, quality reports, and initial insights through exploratory data analysis (EDA).

3. Data Preparation

This is where most of the effort in data projects lies. Data is cleaned, transformed, and engineered into a suitable format for modelling. Tasks may include handling missing values, encoding categorical variables, normalising numerical features, feature extraction, and partitioning datasets into training, validation, and test sets.

- *Output*: A modelling-ready dataset with selected features and appropriate splits.

4. Modelling

Here, you choose and apply machine learning algorithms. You may try multiple models (e.g., regression, decision trees, neural networks), tune hyperparameters, and experiment with different feature subsets. Importantly, modelling is iterative: the choice of model might highlight new data requirements, sending you back to the preparation phase.

- *Output*: One or more candidate models, with performance metrics on validation data.

5. Evaluation

At this stage, you step back and check if the model actually meets the **business objectives** defined earlier. It's not just about accuracy or AUC; you must consider business KPIs, cost-benefit trade-offs, fairness, and interpretability. If the model isn't fit for purpose, you may refine it or even revisit earlier stages.

- *Output*: A final model evaluation report, including technical and business measures of success.

6. Deployment

Finally, the model is integrated into business processes. This could mean deploying a web service, creating batch scoring jobs, or providing decision-support tools. Monitoring is critical: you track model drift, accuracy degradation, and business impact. Plans for retraining and retracking results are also established.

- *Output*: A production system with monitoring, maintenance plan, and documentation.