ICS 2309 - Commercial Programming

CAT_01

AKECH DAU ATEM

SCT211-0535/2022

Real-World Application: Feeler as a Case Study

Feeler: AI-Powered Sentiment Analysis for Customer Feedback

Project Overview:
Feeler is an AI-driven sentiment analysis system designed to analyze and classify customer feedback from online reviews and social media. By leveraging Natural Language Processing (NLP), Feeler provides real-time insights into customer opinions, helping businesses improve their products and services.

Core Technologies:

- **Backend:** Django REST Framework for API development.
- **Frontend:** React.js for an interactive user interface.
- **Machine Learning:** TensorFlow and NLP models for sentiment classification.
- **Database:** PostgreSQL for structured storage of user feedback and results.
- **Deployment:** Docker containers and CI/CD pipelines for scalable cloud deployment.

Feeler as the Reference Project in This CAT:
Throughout this document, Feeler is used as a practical example to illustrate key commercial programming concepts, including Agile development, DevOps practices, database management, API design, and microservices architecture. The project's real-world application provides a solid foundation for understanding software engineering principles in a commercial setting.

Section A

Question 1: Software Development Paradigms

a) Software Development Methodologies

i. **Agile**

- **Definition:** Agile is an iterative software development approach emphasizing collaboration, adaptability, and incremental delivery.
- **Example in FEELER:** The project used Scrum with 2-week sprints. For instance, **Phase 4 (Frontend Development)** focused on delivering the real-time sentiment

dashboard in one sprint, with daily standups ensuring alignment between UI/UX and backend teams.

ii. **DevOps**

- **Definition:** DevOps integrates development and operations to automate workflows, ensuring rapid and reliable deployments.
- **Example in FEELER:** The **CI/CD pipeline in Phase 3** leveraged GitHub Actions to automate testing and deployment. Docker containers ensured environment consistency, while Jenkins orchestrated workflow automation.

iii. **Lean Software Development**

- **Definition:** Lean focuses on value delivery by eliminating inefficiencies such as redundant code or over-engineering.
- **Example in FEELER:** Advanced features like **custom AI model training (Phase 2)** were deferred to prioritize launching core functionalities like **real-time sentiment analysis**, accelerating time-to-market.

b) Integration of Paradigms in 21st-Century Software Development

Synergy of Agile, DevOps, and Lean in FEELER

- **Agility:** Iterative sprints facilitated rapid prototyping of the sentiment analysis dashboard (Phase 4).
- **Automation:** CI/CD pipelines minimized deployment errors during backend API integration (Phase 3).
- **Efficiency:** Lean principles ensured critical features, like NLP model training (Phase 2), were prioritized over non-essential tasks.

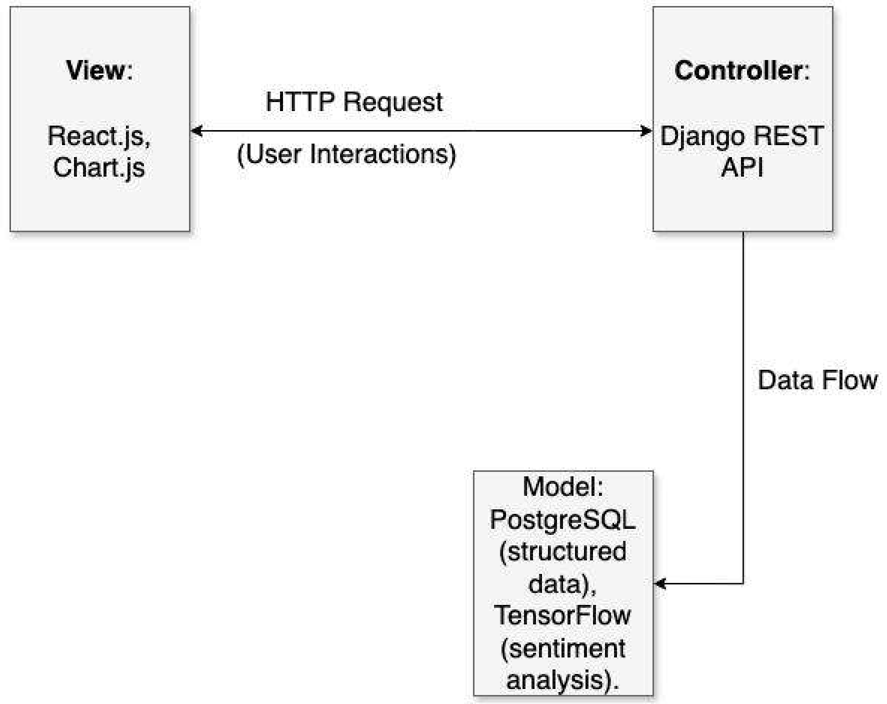
This integration aligns with modern software development practices by ensuring adaptability, speed, and resource optimization.

Question 2: Enterprise Software System Design

a) MVC Architecture Illustration and Roles

MVC Architecture in FEELER

Diagram Representation:



- **Model:** Manages data and business logic (e.g., Django models handling sentiment data and ML predictions).
- **View:** Renders UI components (e.g., React.js frontend displaying real-time sentiment graphs).
- **Controller:** Processes user requests (e.g., Django REST APIs routing data between the frontend and ML model).

b) Scalability and Security Considerations

Scalability Approaches:

Horizontal Scaling: AWS Elastic Load Balancer distributes traffic during peak sentiment analysis requests.
Microservices Architecture: Independent services (e.g., authentication, sentiment analysis) run in Docker containers, ensuring modular scaling (Phase 6).

Security Measures:

Role-Based Access Control (RBAC): Implemented in **Phase 2** to enforce user permissions (e.g., admin vs. guest).
Data Encryption: HTTPS/TLS for API communications and AES-256 encryption for stored user data (Phase 5 security audit).

Question 3: DevOps and CI/CD

a) Importance of CI/CD in Modern Development

- Automate software delivery,
- Ensuring fast updates,
- Reduced bugs,
- Reliable workflow.

CI/CD in FEELER
CI/CD pipelines enabled automated testing (e.g., PyTest for backend APIs) and seamless deployment to AWS, reducing manual errors and accelerating releases.

- Example: **GitHub Actions (Phase 3)** deployed updates to the staging environment within 10 minutes of code commits.

b) Version Control (Git & GitHub)

- Version Control: Tracks code changes.
- Branching Strategy: Feature branches prevent conflicts in main.
- Pull Requests & Code Reviews: Maintain code quality.

Example from FEELER:

- **Branching Strategy:**
  <u>main branch:</u> Stable production-ready code.
  <u>dev branch:</u> Staging environment for integration testing.
  <u>Feature branches</u> (sentiment-api, UI-updates): Isolated development of new features.
- **GitHub Workflow:**

Pull Requests (PRs): Required approvals from the ML engineer and backend lead.
Issue Tracking: Bugs and enhancements (e.g., "*Incorrect sentiment classification in Phase 2*") were logged in GitHub Issues.

c) Additional Tools

- **Jenkins: (**Manages CI/CD pipelines with flexibility.**)-** Orchestrated multi-stage pipelines (**build → test → deploy**) for FEELER's backend.
- **Docker: (**Enables containerized deployments.**)**
- **Jest:** Tested React components (e.g., sentiment graph rendering) before deployment (Phase 5).
- **GitHub Actions** – Automates testing and deployment.

Question 4: Frontend vs Backend and API Documentation

a) Frontend vs Backend

Aspect	Frontend (FEELER)	Backend (FEELER)
Technologies	React.js, Figma, Chart.js	Django, PostgreSQL, TensorFlow
Responsibilitie s	UI rendering, Real-time data visualization	API development, ML model integration
Team Roles	UI/UX Designer, Frontend Developer	Backend Developer, ML Engineer

b) API Documentation & Testing

- **Purpose:** API documentation ensures developers understand request/response formats and integration workflows.
- **Tools Used:**
    - **Swagger UI:** Generates interactive API docs. (Auto-generated API docs for Django REST endpoints.)
    - **Postman:** Documents and tests APIs. (Shared collection for testing sentiment analysis APIs.)

Section B

Question 5: GitHub and CI/CD

(a) Steps to Initialize a GitHub Repository and Branching Strategy

1. **Create a GitHub Repository**
   a. Go to GitHub and click **New Repository**.
   b. Name it (e.g., feeler for our project) and choose **Public/Private**.
   c. Initialize with a **README.md** and select .gitignore for (Python/Node.js templates).
2. **Clone Repository Locally**

```
git clone https://github.com/username/feeler.git
cd feeler
```

3. **Set Up Branching Strategy**
   a. Use **main** for stable production-ready code.
   b. Use **develop** for ongoing development.
   c. Feature branches (feature/<task-name>) for new features.

```
git checkout -b develop
git push origin develop
```

4. **Implement Feature Branch Workflow**

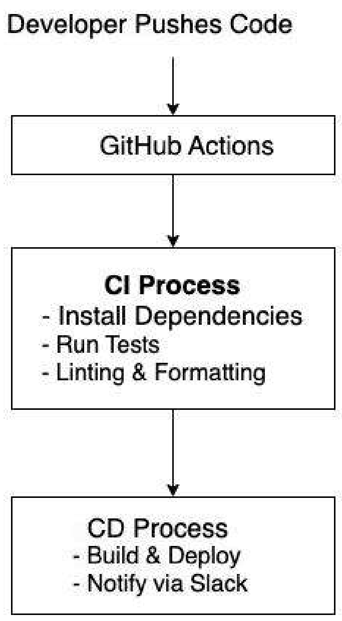
```
git checkout -b feature/new-functionality
git add .
git commit -m "Added new functionality"
git push origin feature/new-functionality
```

5. **Create a Pull Request (PR)**
   a. On GitHub, go to **Pull Requests** and compare the feature/new-functionality branch with develop.
   b. Request a review before merging.
6. **Merge and Delete Feature Branch**

```
git checkout develop
git merge feature/new-functionality
git push origin develop
git branch -d feature/new-functionality
```

(b) CI/CD Pipeline using GitHub Actions

Pipeline Diagram



A CI/CD pipeline automates testing and deployment.

1. **Developer pushes code to GitHub** (e.g., pushing changes to develop).
2. **GitHub Actions runs the CI/CD workflow**, which includes:
    a. Checking out the code
    b. Installing dependencies
    c. Running tests
3. **If tests pass, the build is deployed**, or an error is reported.

GitHub Actions Workflow

- Create a .github/workflows/ci.yml file:

```
name: CI/CD Pipeline


on:
  push:
    branches:
      - main
      - develop

jobs:
  build:
     runs-on: ubuntu-latest

     steps:
       - name: Checkout Code
         uses: actions/checkout@v3

       - name: Set up Python
         uses: actions/setup-python@v4
         with:
           python-version: 3.12
    cache: 'pip'  # Caches dependencies

       - name: Install Dependencies
         run: |
            python -m venv venv
            source venv/bin/activate
            pip install -r requirements.txt

•      - name: Run Tests
         run: |
            source venv/bin/activate
            pytest --junitxml=test-results.xml
```

(c) Common Git Mistakes and Solutions

1. **Committing sensitive data-** (e.g., API keys in .env files)
   Solution: Use .gitignore to exclude them. And environment variables. Example: export SECRET_KEY=yourkey
2. **Not using branches properly** (working directly in main)
   Solution: Always create a feature branch before making changes. Always work in feature branches and merge via pull requests.
3. **Forgetting to pull latest changes**
   Solution: Run git pull origin develop before making edits to avoid conflicts.

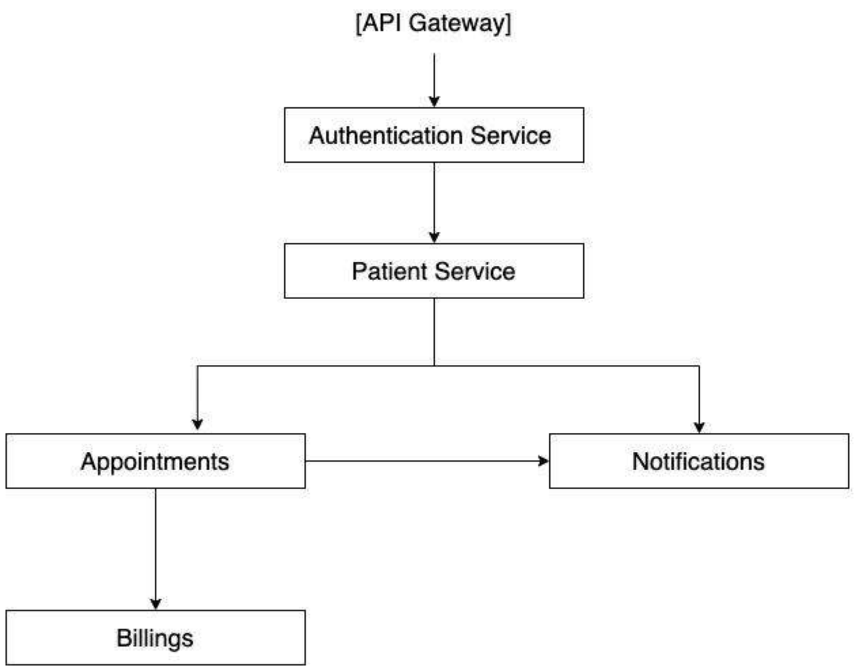
Question 6: Microservices in Healthcare

(a) Microservices Architecture for a Patient Management System

A patient management system in a hospital can be broken down into independent microservices, each handling a specific function. The key components could include:

1. **Patient Service –** Stores patient records and medical history.
2. **Appointment Service –** Manages scheduling and doctor availability.
3. **Billing Service –** Handles invoices and payments.
4. **Authentication Service –** Manages user login and permissions.
5. **Notification Service –** Sends email/SMS reminders.
   Each of these microservices runs independently and communicates via APIs. If one fails (e.g., billing), the rest of the system continues to function.

   Architecture Overview



(b) Why Microservices Instead of Monolithic Architecture?

1. **Scalability –** Individual services can be scaled separately (e.g., high demand for appointments, low for billing).
2. **Fault Isolation –** A failure in one microservice does not crash the entire system.
3. **Faster Development –** Teams can work on different services independently.
4. **Easier Maintenance –** Bug fixes and updates can be made to specific services without affecting the whole system.

For example, in our **Feeler** project, a monolithic approach would mean one large system handling everything (data storage, sentiment analysis, API). A microservices approach would separate these into:

Analysis Service (runs sentiment model)
Data Service (stores feedback)
API Service (handles external requests)
This makes it easier to scale and maintain.

(c) SQL vs. NoSQL for Performance and Scalability

Choosing between SQL and NoSQL depends on system needs:

1. **SQL (Structured Query Language)**
   a. Best for **structured data** (e.g., patient records, billing).
   b. **Example:** PostgreSQL for handling transactions securely.
   c. **Use Case:** A hospital database needs strict relationships between patients, doctors, and prescriptions.
2. **NoSQL (Not Only SQL)**
   a. Ideal for **scalable, flexible data** (e.g., storing logs, real-time feedback).
   b. **Example:** MongoDB for handling unstructured patient notes.
   c. **Use Case:** A system tracking real-time patient vitals where new data keeps arriving dynamically.

For **Feeler**, if I was logging social media sentiment data, a **NoSQL** database (like Firebase or MongoDB) would be better for fast, flexible storage. If I needed structured reports and analytics, an **SQL** database (like PostgreSQL) would be more efficient.

Question 7: Wireframing & UX Considerations in S/W Architecture

(a) Four Best Practices for Wireframing with Figma/Adobe XD

1. **Start with Low-Fidelity Wireframes**
   a. Begin with simple black-and-white layouts to define structure and functionality before adding colors and details.
   b. Helps focus on user flow rather than visual design.
2. **Use Consistent Grid and Layout**
   a. Maintain alignment using a grid system (e.g., an **8px grid** for spacing).
   b. Ensures responsive and well-structured designs across devices.

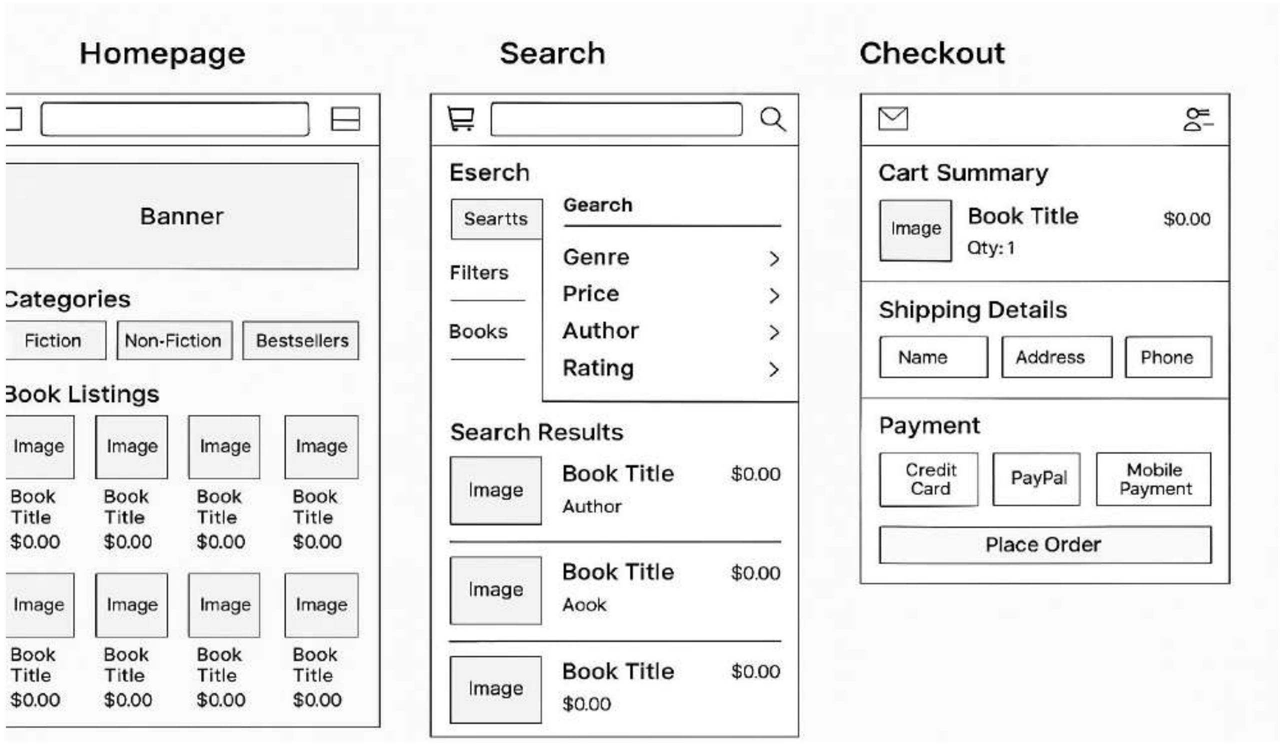
3. **Keep Navigation Intuitive**
    a. Use clear and familiar UI patterns (e.g., a top navbar or sidebar menu).
    b. Ensures users can navigate smoothly without confusion.
4. **Include Placeholder Content and Annotations**
    a. Use dummy text (e.g., “Lorem ipsum”) and icons to represent real content.
    b. Add comments or labels to explain elements for developers.

(b) UI Wireframes for an Online Bookstore (6 Marks)

Below is a description of the wireframes for three key pages:

1. **Homepage**
    a. **Header:** Logo, search bar, cart icon, and user profile.
    b. **Main Banner:** Featured books or promotions.
    c. **Categories Section:** Fiction, Non-fiction, Bestsellers, etc.
    d. **Book Listings:** Grid of books with images, titles, and prices.
2. **Search Page**
    a. **Search Bar (Sticky at the Top).**
    b. **Filters:** Genre, price range, author, ratings.
    c. **Search Results:** List of books with thumbnails, titles, author names, and prices.
3. **Checkout Page**
    a. **Cart Summary:** Books selected, quantity, and total price.
    b. **Shipping Details:** Name, address, phone number.
    c. **Payment Options:** Credit card, PayPal, mobile payment.

Wireframe Representation



(c) Role of UX Considerations in Software Architecture

UX (User Experience) considerations influence software architecture by ensuring that the system is built for**:**

- Usability,
- Efficiency,
- User satisfaction.

Example 1: Feeler (Sentiment Analysis App)

- The **system architecture** must **support real-time processing** so users get instant sentiment analysis.
- A **well-designed API structure** ensures smooth interaction between the UI and the NLP engine.

Example 2: E-Commerce Website (e.g., AMAZON)

- The backend should **support fast search queries** for a seamless shopping experience.
- Features like **autosuggestions and real-time stock updates** improve usability.

By integrating **UX principles** into software design, applications become **faster, more reliable, and user-friendly**.