Module 19 Week 21 Assignment

Qn1:

a. **Explain how you would implement generative AI for code generation in a full stack project.**
b. **Outline specific implementation steps and discuss the potential benefits and challenges of this approach. (5 Marks)**

---

a. **First the approach would be to**

- **Gather requirements & technology selection:**
- **Define Use Cases:** Identify specific tasks for the AI , in this case code generation.
- **Choose the LLM and Tech Stack (frontend, backend (middleware, LLM)**
- **Testing and Deployment:** Test the flow with various prompts to ensure the prompt engineering is robust and the sanitization works correctly. Deploy the backend middleware to a cloud platform (e.g., AWS) and the frontend to a static host.
- **Monitor:** Monitor usage and costs, and gather feedback to iteratively improve the prompts and functionality.

b. **Implementation Steps:**

1. **Define Scope & Select Tools:** Choose the AI and the tech stack (e.g., React for frontend, Node.js/Express for backend).
2. **Design the Architecture:** Adopt a client-middleware-server pattern. The frontend client sends requests to your custom backend API (middleware), which then communicates with the external AI server.
3. **Develop the Backend Service (Node.js/Express Middleware):**

- Create an Express server.
- Set up a POST endpoint (e.g., /generate-code).
- Inside the endpoint handler:
- **Authenticate** the request (if applicable).
- **Engineer the prompt:** Combine the user's request with a system prompt (e.g., "You are an expert React developer. Output only code.").
- **Securely call the AI API** using your stored API key.
- **Sanitize the response:** Parse the output to remove any non-code text (like Markdown ``` blocks).
- **Send the clean code** back to the frontend.

4. **Develop the Frontend Client (React/Vanilla JS):**

- Build a UI with a text input and a button.
- Attach an event handler to the button that collects the input, sends a POST request to your backend endpoint, and displays the formatted response.

5. **Implement Safety Measures:**

- Never execute generated code automatically. Treat all output as a suggestion.
- Implement human-in-the-loop: require developer review and approval before code is used.

c. <u>Potential Benefits:</u>

- **Increased Productivity:** Dramatically reduces time spent writing boilerplate code (components, API routes, tests).
- **Lowered Learning Curve:** Helps junior developers learn and overcome obstacles by generating examples.
- **Enhanced Consistency:** Can promote best practices and consistent patterns across the codebase.

<u>Potential Challenges</u>:

- **Code Correctness & Security:** The AI can generate incorrect, inefficient, or insecure code. **Human review is compulsory.**
- **Vendor Lock-in & Cost:** Reliance on a third-party API can become expensive and create dependency.
- **Intellectual Property (IP) Concerns:** Sending proprietary code to an external API raises data privacy and IP ownership questions that must be addressed via terms of service or by using self-hosted models.
- **Over-reliance:** A risk that developers may lose deep understanding or the ability to code from first principles.

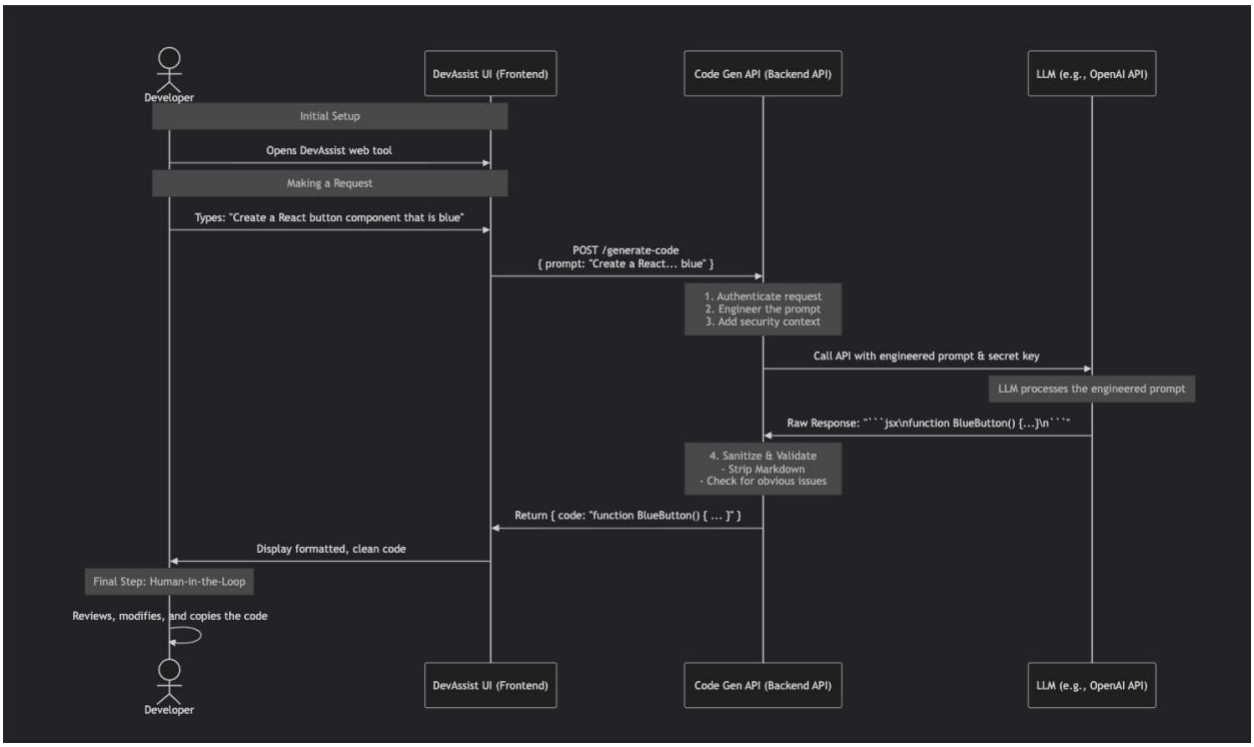
➔ How can Generative AI be used to convert natural language requests into SQL queries or API calls? Provide an example and discuss the implications for database management and API design. (5 Marks)

Converting Natural Language to SQL/API Calls with Generative AI Generative AI acts as a sophisticated **translator** or **compiler** that bridges the gap between human language and machine language (SQL/API endpoints). It understands the *intent* behind a user's question and maps it to the precise structured command needed to fulfill that request.

The implementation would follow the same pattern as Q1: a frontend for the user, a backend to handle the logic, and an external AI service.

Actors & Components:

6. **Developer:** The end-user, a programmer who wants to generate code.
7. **DevAssist UI (Frontend):** The React or Vanilla JS web application. Its job is to be the interface.
8. **Code Gen API (Backend API / Middleware):** The Node.js/Express server. This is the **brains** of the operation.
9. **LLM (e.g., OpenAI API):** The external generative AI service that provides the raw intelligence.



Phase 1: Initial Setup

➔ **What Happens:** The developer opens the DevAssist web tool in their browser.

Phase 2: Making a Request

➔ **What Happens:** The developer types a natural language command into the UI (e.g., *"Create a React button component that is blue"*) and hits a "Generate" button.

Phase 3: The Frontend Calls the Backend

➔ **What Happens:** The frontend packages the user's prompt into a POST request and sends it to a specific endpoint on backend API (e.g., https://your-api.com/generate-code). The request body contains a JSON object like { "prompt": "Create a React... blue" }.
The frontend never talks directly to the OpenAI API. It always goes through secure, controlled middleware.

Phase 4: Backend Processing & Prompt Engineering

➔ This is the most critical phase your **Node.js/Express middleware**.
Authenticate Request: The backend checks if the request is coming from an authorized user or system. This prevents unauthorized usage.
Engineer the Prompt: The backend takes the user's simple prompt and *enhances* it with crucial context and instructions. This is what turns a vague request into a high-quality result.

Prompt Engineering: The practice of crafting precise instructions for the LLM. This is done **in the backend** by templating the user's input with system rules (e.g., "You are an expert developer...") and project context (e.g., tech stack, style rules). This is the secret to getting high-quality results.
e.g. Express.js (see appendix A) Role: Express is a web framework for Node.js that makes it easy to create the backend server (middleware). It handles:

- Defining routes (endpoints like /api/generate-code).
- Parsing request data.
- Calling other services (like the OpenAI API).
- Sending responses back to the frontend.

Phase 5: The Secure Call to the AI

➔ **What Happens:** The backend uses its securely stored API key (kept in an environment variable, not the frontend code) to call the OpenAI API. It sends the expertly engineered prompt from the previous step.

Phase 6: The AI Processes the Request

➔ **What Happens:** The Large Language Model (LLM) from OpenAI or another provider analyzes the engineered prompt, understands the context ("expert React developer", "version 18"), and generates a response.

The quality of the engineered prompt directly dictates the quality of the output. A good prompt yields great code; a bad prompt yields garbage.

Phase 7: The AI Returns a Raw Response

What Happens: The AI sends its answer back to your backend. This answer is often "wrapped" in Markdown formatting for readability,
Example Raw Response: "Here's your component:\n```jsx\nfunction BlueButton() { return <button style={{ backgroundColor: 'blue' }}>Click Me</button>; }\n```"
This raw output is not ready for the developer. It needs to be cleaned.

Phase 8: Sanitization & Validation

➔ **What Happens:** Your backend middleware performs crucial "post-processing":
Strip Markdown: It uses code (e.g., a regular expression) to remove the text and the Markdown code blocks (```), extracting *only* the pure code: function BlueButton() { return <button style={{ backgroundColor: 'blue' }}>Click Me</button>; }

This step ensures the developer gets a clean, ready-to-use code snippet without

having to manually delete explanations and formatting.

Phase 9: Sending the Clean Code to Frontend

➔ **What Happens:** The backend sends the purified code in a structured JSON response back to the frontend (e.g., { "code": "function BlueButton() { ... }" }). The frontend now has the final product to display.

Phase 10: Displaying the Result

➔ **What Happens:** The frontend receives the response, stops showing a "loading" indicator, and displays the generated code in a nice, readable format (often with syntax highlighting). User see the result of their request.

Phase 11: Human-in-the-Loop

➔ **What Happens:** The developer **reviews, understands, modifies, and finally copies** the generated code into their actual project.
This final review is a critical quality and security gate.

Provide an example and discuss the implications for database management and API design.

When Generative AI is used to auto-generate backend code, SQL queries, and API endpoints, the quality of DB design and API structure directly impacts whether the generated code is correct, secure, and efficient.

In the **Code Generation** use case (e.g., "create a React button" in Question 1 answer), the primary database is **not directly involved in the AI's real-time processing flow**.

However, a database is still critical for the application itself. It would be used for:
User Authentication: Storing user accounts, hashed passwords, and API keys.
Request History: Saving a log of generated code snippets per user for later reference.
Application Data: Managing any other data your app needs (e.g., user profiles, project settings).
So, in the code generation flow, the database supports the *application*, not the *AI's core reasoning*. It's in the background, not the foreground of the AI sequence. The database is **supporting the application's functionality** (auth, history, settings). It is not the target of the AI's output.

In another following use case, this is where the database moves to the **forefront**. It is the **target** of the AI-generated command.

When a user asks: *"What was our total revenue from customers in Singapore last month?"*

The AI's job is to translate that into a SQL query that **joins** these tables, **filters** them, and **aggregates** the data. (eg by Express in app.js) **see Appendix B**

In Natural-Language-to-SQL: The database is **the source of truth and the target of the AI's output**. The AI's sole purpose is to generate commands to interact with it. This is why security (read-only access) and performance (query limits) are important in this use case.

Your backend does **NOT** know how to convert this to SQL. So, it calls the AI (e.g., the OpenAI API or SageMaker endpoint) with a carefully engineered prompt like this:

The AI analyzes the prompt, understands the schema, and generates the correct SQL code and it sends this *text* back to backend.

➔ **Backend Executes the "Normal Query":**

- Backend application now receives this SQL string from the AI.
- It uses a database library (like pg for PostgreSQL) to open a connection to the database.
- It executes the received SQL query **just like any other query** it would run.

➔ **Database Processes It:**

- The database receives the query. It doesn't know an AI wrote it. It could have been written by a human developer. It processes the JOIN, WHERE, and SUM clauses and computes the result.

➔ **Result is Returned:**

- The database sends the result (e.g., { revenue: xxxx }) back to backend, which then sends it to the user.

The AI is an **external service** (a separate API) that backend calls upon **on-the-fly** to perform a **translation task**. It is the "brain" in the middleware that handles the complex task of understanding intent and generating syntax.

Q3 Describe the process of fine-tuning a pre-trained language model for a specific task.
What are the advantages of this approach compared to training a model from scratch? (5 Marks)

Fine-tuning a pre-trained language model involves taking a model already trained on a large, general dataset and further training it on a smaller, task-specific dataset to adapt it for a particular use case. This approach offers significant advantages over training a model from scratch, including efficiency, improved performance, and reduced data and resource requirement

- ➔ Start with a **Pre-Trained Model**: Select a large language model that has already learned general language patterns from massive datasets.
- ➔ Prepare Task-Specific Data: Collect and label a smaller dataset tailored to the target task (e.g., sentiment analysis, domain-specific Q&A).
- ➔ Continue Training: Train the model further on the specific dataset, adjusting only certain layers or the entire network depending on task complexity.
- ➔ Optimize Hyperparameters: Use strategies such as lowering learning rates or freezing initial layers to retain general features while specializing deeper layers.
- ➔ Evaluate and Iterate: Assess the model's accuracy on validation data and iterate as needed for optimal performance.

Advantages Over Training from Scratch

- **Efficiency**: Fine-tuning leverages previously learned knowledge, drastically reducing the need for extensive compute power and time compared to full-scale training.
- **Better Performance with Less Data**: Pre-trained models only need a small amount of task-specific data to perform well, making them effective in low-resource environments.
- **Faster Deployment**: Fine-tuning speeds up the customization of models for new applications, enabling rapid development for specialized tasks.
- **Cost-Effective**: Using a pre-trained foundation lowers both computational and financial costs associated with large-scale model training.
- **Higher Accuracy for Specific Tasks**: Adapting models to niche domains or specific requirements often leads to superior performance versus a general model or one trained only on task data.

Q4 What is prompt engineering, and why is it important when working with large language models?
Provide an example of how you would craft an effective prompt for a text generation task. (5 Marks)

Prompt engineering is the process of designing and refining inputs to guide large language models (LLMs) like GPT-5, enabling them to generate more accurate, relevant, and controlled outputs for a given task.

It is important because the quality, clarity, and structure of the prompt directly influence the quality and usefulness of the AI's responses, making it a crucial skill when working with LLMs for real-world applications.

Why Prompt Engineering Is Important

- ➔ **Directs model behavior**: By carefully choosing prompts, it is possible to instruct the model to focus on specific tasks, styles, or constraints without changing its underlying architecture.
- ➔ **Improves output quality**: Well-crafted prompts reduce errors, avoid irrelevant information, and lead to more accurate, creative, and coherent results for tasks such as summarization, code generation, or dialogue.
- ➔ **Saves time and resources**: Rather than retraining or fine-tuning models for each new task, effective prompts allow immediate adaptation to new requirements, saving computational and human resources.
- ➔ **Enables adaptability**: Good prompt engineering can help handle different tasks, domains, or personas, often leveraging just natural language instructions.

Example: Suppose the task is to generate an Express.js API endpoint. Compare these two prompts:

- Less effective prompt:

  "Write an API for orders."

- More effective prompt:

  "You are a senior Node.js developer. Write an Express.js API endpoint /orders/top that retrieves the top 5 customers by total spend. Use parameterized SQL for security, assume a PostgreSQL database with tables customers(id, name) and orders(id, customer_id, total_amount, order_date). Return the results as JSON with fields customer_name and total_spend."

The second prompt works better because it:

- Defines role/context (“senior Node.js developer”).
- Provides schema grounding (Postgres tables).
- Sets clear requirements (endpoint path, return format).
- Enforces best practices (parameterized SQL, JSON response).

In summary, good prompt engineering reduces ambiguity and helps the model generate code (or text) that is not only correct but also aligned with best practices.

Appendix A

The endpoint is created and lives in your Express.js server.
The endpoint is not a physical thing; it's a **block of code** you write in Express application that defines:

10. **A URL path** (e.g., /generate-code)
11. **An HTTP method** (e.g., POST)
12. **A function** to run when a request arrives at that address and method.

This block of code is called **route handler**.

How it Works in Code

Here’s a simplified version of what the code in app.js

```
// This is Express route handler
app.post('/api/generate-code', async (req, res) => {
  try {
    // 1. Get the simple input from the frontend
    const userPrompt = req.body.prompt; // e.g., "a blue button"

    // 2. Authenticate the request here (e.g., check an API key)

    // 3. ENGINEER THE PROMPT
    // Express uses its hardcoded rules to improve the user's request.
    const engineeredPrompt = `
    You are an expert React developer. Return only code with no explanations.

    ${PROJECT_CONTEXT} // <-- Express "injects" the rules here

    Create a React component for: ${userPrompt}
    `;

    // 4. Now call the OpenAI API with the engineered prompt, not the user's raw prompt.
    const openaiResponse = await callOpenAIApi(engineeredPrompt); // You write this function

    // 5. Send the AI's response back to the frontend
    res.json({ code: openaiResponse });

  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Appendix B

The Role of Express.js in AI Integration
This critical step is done by the Express.js backend.
The Express server is not just a passive relay; it's an **intelligent orchestrator**. Its job is to execute the logic that makes the AI call possible. Here's a breakdown of what the Express code actually does in that step:

The Express Server's Responsibilities in This Step:
1. **It Has the Logic:** The code to build the engineered prompt is **hardcoded into the Express application**. It knows the rules, the database schema, and how to structure the request for the AI.
2. **It Has the Security Credentials:** The Express server securely stores the API key needed to call the OpenAI API (or the AWS credentials to call a SageMaker endpoint) in its environment variables. The frontend never sees these keys.
3. **It Makes the Network Call:** The Express server uses a library like axios or the AWS SDK to perform the HTTP request to the AI service's API. This is an asynchronous operation that it handles.

4. It Processes the Response: When the AI responds with the raw SQL, the Express server is responsible for parsing that response, extracting the code, and handling any errors.

Example: The Actual Express Code for This Step

Here is what the code inside your Express route (`/api/natural-language-to-sql`) would look like for this specific step:

javascript

```
// This function is called when a POST request is made to /api/natural-langua
ge-to-sql
app.post('/api/natural-language-to-sql', async (req, res) => {
  try {
    // 1. Get the user's question from the frontend request
    const userQuestion = req.body.question; // "What was our total revenue...
"

    // 2. 🔄 ENGINEER THE PROMPT (This is the intelligence) 🔄
    // This is all hardcoded logic within the Express server.
    const prompt = `
    You are a senior database administrator. Convert the following natural la
nguage request into a precise PostgreSQL query.
    Only return the SQL, no other text.

    Database Schema:
    - Table: customers (columns: id, name, email, city)
    - Table: orders (columns: id, customer_id, order_date, total_amount)

    User Request: "${userQuestion}"
    `;

    // 3. 🔄 CALL THE AI API (This is the action) 🔄
    // The Express server uses axios to call the external AI service.
    const openaiResponse = await axios.post(
      'https://api.openai.com/v1/chat/completions',
      {
        model: "gpt-4",
        messages: [{ role: "user", content: prompt }],
        temperature: 0.1 // Low temperature for less creative, more precise o
utput
      },
      {
        headers: {
          'Authorization': `Bearer ${process.env.OPENAI_API_KEY}` // Using se
cret key
```

```
        headers: {
          'Authorization': `Bearer ${process.env.OPENAI_API_KEY}` // Using se
cret key
        }
      }
    );

    // 4. Extract the generated SQL from the AI's response
    const generatedSql = openaiResponse.data.choices[0].message.content.trim
();

    // 5. (Optional but crucial) Validate the SQL before executing it!
    // ... validation logic could go here ...

    // 6. Now, the Express server can execute this SQL on the database
    // const dbResult = await db.query(generatedSql);

    // 7. Send the result back to the frontend
    res.json({ sql: generatedSql /*, data: dbResult.rows */ });

  } catch (error) {
    console.error("Error in AI SQL generation:", error);
    res.status(500).json({ error: "Failed to generate SQL" });
  }
});
```

Key Takeaway:

The app.post() function and the code inside it **are your Express server in action**. The act of building the prompt and calling the AI API is not a separate service—it's the core purpose of this specific Express **route handler**.

So, you are right to point to that step. It is the moment where the Express server leverages its programmed intelligence and secure position to use the external AI as a tool to solve the user's problem.

and epxress code in app.js?

Exactly. All of this code lives in your Express application, which is typically defined in a file like app.js, server.js, or index.js.

This file is the **heart of your backend server**. It's where you set up Express, define all the rules (middleware), and most importantly, declare all your endpoints (routes) like the one we just discussed.

Here is what a complete app.js file might look like, incorporating the natural-language-to-SQL endpoint. The comments explain each part.

app.js - The Complete Express Application File

javascript

```javascript
// 1. IMPORT LIBRARIES
// This is where you bring in the tools you need.
const express = require('express'); // The main Express framework
const axios = require('axios'); // A library to make HTTP calls to the AI API

// 2. INITIALIZE THE EXPRESS APPLICATION
// This creates your server. You can now configure it and add routes.
const app = express();

// 3. CONFIGURE MIDDLEWARE
// These are functions that process all incoming requests before they reach your routes.
app.use(express.json()); // This is crucial: it parses JSON data from request bodies and makes it available
 in `req.body`

// 4. DEFINE YOUR ROUTES (THE ENDPOINTS)
// This is where you tell your server how to handle requests to specific URLs.

// Endpoint for generating code (from our first example)
app.post('/api/generate-code', async (req, res) => {
  // ... (code from the previous example would go here)
});

// Endpoint for converting natural language to SQL (from our latest example)
app.post('/api/natural-language-to-sql', async (req, res) => {
  try {
    // 1. Get the user's question from the frontend request body
    const userQuestion = req.body.question;

    // 2. ENGINEER THE PROMPT (The intelligence hardcoded into Express)
```

```
  const prompt = `
  You are a senior database administrator. Convert the following natural language request into a precise PostgreSQL query.
  Only return the SQL, no other text.

  Database Schema:
  - Table: customers (columns: id, name, email, city)
  - Table: orders (columns: id, customer_id, order_date, total_amount)

  User Request: "${userQuestion}"
  `;

  // 3. CALL THE AI API (The external service call)
  const openaiResponse = await axios.post(
    'https://api.openai.com/v1/chat/completions',
    {
      model: "gpt-4",
      messages: [{ role: "user", content: prompt }],
      temperature: 0.1
    },
    {
      headers: {
        'Authorization': `Bearer ${process.env.OPENAI_API_KEY}` // Securely use the API key from an environment variable
      }
    }
  );

  // 4. Extract the generated SQL from the AI's response
  const generatedSql = openaiResponse.data.choices[0].message.content.trim();
  console.log("Generated SQL:", generatedSql); // For debugging

  // 5. Send the SQL back to the frontend.
```

```
    // In a real app, you might execute it on the DB here and send the data instead.
    res.json({ sql: generatedSql });

  } catch (error) {
    console.error("Error in AI SQL generation:", error);
    res.status(500).json({ error: "Failed to generate SQL" });
  }
});

// A simple health check endpoint to see if the server is alive
app.get('/health', (req, res) => {
  res.send('Server is up and running!');
});

// 5. START THE SERVER
// This tells Express to start listening for incoming requests on a specific port.
const PORT = process.env.PORT || 3000; // Use the environment variable or default to 3000
app.listen(PORT, () => {
  console.log(`Express server is running on http://localhost:${PORT}`);
});
```

Summary: What This File Does

1. **Setup:** It imports necessary libraries and creates an Express app.
2. **Configuration:** It adds essential middleware (express.json()) to understand JSON data.
3. **Logic Definition:** It defines the endpoints (/api/generate-code, /api/natural-language-to-sql). Each endpoint is a **route handler**—a function that contains the step-by-step logic for what to do when a request arrives. This is where the "AI calling" logic lives.
4. **Execution:** It finally starts the server, making it live and ready to respond to requests.

When run the command node app.js in terminal, starting this entire application. The code for calling the AI is an integral part of this application.