Q) *Explain how AI-driven code generation tools (e.g., GitHub Copilot) reduce development time. What are their limitations?*

How they reduce Development time

Rapid Prototyping	With AI driven code generators, developers can generate draft implementations from natural language prompts. This accelerates experimentation and iteration.
Debugging support and error detection	Many AI Tools can spot syntax errors and support bug fixes as code is being written. This helps to maintain flow and prevents manual debugging saving on time.
Consistency	AI tools generate code following common conventions. This helps maintain consistent style across a team or project.
Code suggestions and autocompletion	The tools predict what the developer is about to type and complete code lines, functions, or even entire blocks. This saves time on routine syntax and boilerplate code.
Learning and Documentation assistance	The tools suggest how to use unfamiliar APIs, libraries or frameworks reducing the need to look up documentation frequently. This helps especially when integrating new technologies.

Limitations

Privacy and confidentiality concerns	Some AI tools send code snippets to external servers for analysis. This poses risks for sensitive and proprietary projects.
Overreliance and skill erosion	Excessive dependencies may reduce developers' problem solving and coding skills over time.
Risk of code duplication or licensing issues	Generated snippets might resemble copyrighted or open-source code under restrictive licenses.
Lack of deep context understanding	AI tools may not fully grasp the architecture of the project, business logic or data flow. Its suggestions can be syntactically correct but logically wrong.
Potential for insecure or inefficient code	Developers must manually review AI-generated code for security and performance since models learn from public repos, as they might reproduce insecure or outdated coding patterns.

Q) *Compare supervised and unsupervised learning in the context of automated bug detection*.

Aspect	Supervised Learning	Unsupervised Learning
Limitation	Needs many labeled examples.	May generate false positives as not all anomalies are real bugs.
Scalability	Harder: Requires large, labeled datasets.	Easier; No labelling is needed,

Accuracy	High accuracy when enough quality labeled data is available.	Variable: it depends on data quality
Goal	The model learns from labeled data to predict whether new, unseen code contains a bug. It classifies or predicts known bugs.	It tries to find pattens or anomalies in code without prior knowledge of what a bug looks like.
Data Requirement	Uses labeled data – example of code that are already tagged as buggy or clean.	Deals with unlabeled data.
Use case example	Predicting if a commit introduces a bug.	Finding unusual code patterns or risky models.

Q) *Why is bias mitigation critical when using AI for user experience personalization?*

Meets Ethical and Legal Standards.

- Bias mitigation ensures compliance and reduces risks of discrimination claims or reputational damage.
- Regulatory frameworks like EU AI Act and GDPR require transparency, explainability and fairness.

Improves Model Accuracy and Diversity

- Biased training data limits how well models generalize to diverse users.
- Correcting these biases helps the AI better understand all users' segments, leading to more effective and balanced personalization.

Ensures fair and inclusive experiences

- Without bias mitigation, some groups (by age, ethnicity, gender, location, etc.) may receive less relevant or lower quality recommendations.

Example: A job sites recommendation algorithm might show higher-paying roles mostly to men if it learns from biased historical data.

Protects user trust

- Users lose confidence when AI-driven interfaces behave unfairly or invasively.
- Transparent and equitable personalization builds trust and loyalty, which are central to long-term engagement.

Prevents Reinforcing Harmful stereotypes

- Personalization models trained on biased data may amplify stereotypes.

Example: A shopping app could keep showing “makeup products” only to women and “gadgets” to men – reinforcing narrow assumptions.

Task 1: AI-Powered Code Completion

1. *Write a Python function to sort a list of dictionaries by a specific key.*
2. *Compare the AI-suggested code with your manual implementation.*
3. *Document which version is more efficient and why.*

Manual Implementation

```python
#Manual implentation of sorting a list of dictionaries by a specific key
def sort_dicts_by_key(data_list, key):
    """Sort a list of dictionaries by a specific key."""
    return sorted(data_list, key=lambda x: x[key])

# Example usage
people = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 30},
    {"name": "Charlie", "age": 20}
]
sorted_people = sort_dicts_by_key(people, "age")
print(sorted_people)
```

AI (GitHub Copilot) suggested Implemtation

```python
#AI Suggested Code
def sort_dicts_by_key(data_list, key, reverse=False):
    """Sorts a list of dictionaries by a given key, optionally in reverse order."""
    try:
        return sorted(data_list, key=lambda x: x.get(key, 0), reverse=reverse)
    except TypeError:
        raise ValueError("All dictionary values for the key must be comparable.")
```

Analysis

The manually written function efficiently sorts a list of dictionaries based on a given key using Python's built-in sorted () function. It is concise and performs well for clean, structured data where all dictionaries contain the specified key. Its time complexity is O (n log n), optimal for Python's sorting algorithm. However, it assumes that all entries are valid and comparable, which can lead to runtime errors if the data is inconsistent.

The AI-generated version from GitHub Copilot adds flexibility and error handling. By using x.get (key, 0), it safely manages missing keys and includes an optional reverse parameter to sort in descending order. The try-except block prevents crashes when data types are not directly comparable. Although these additional checks introduce minimal overhead, they make the function more robust and production ready.

Efficiency

In terms of performance, both approaches have similar time complexity, but the AI-generated code may be slightly slower due to additional checks. However, it's more reliable and maintainable because it anticipates common runtime errors.
Overall, Copilot's version demonstrates how AI-assisted coding can improve developer productivity by automatically suggesting safer and more flexible implementations, therefore, enhancing productivity by suggesting safer, more feature-complete solutions while saving time otherwise spent debugging or writing defensive code.

Task 2: Automated Testing with AI

1. *Automate a test case for a login page (valid/invalid credentials).*
2. *Run the test and capture results (success/failure rates).*
3. *Explain how AI improves test coverage compared to manual testing.*

Automated Test Script

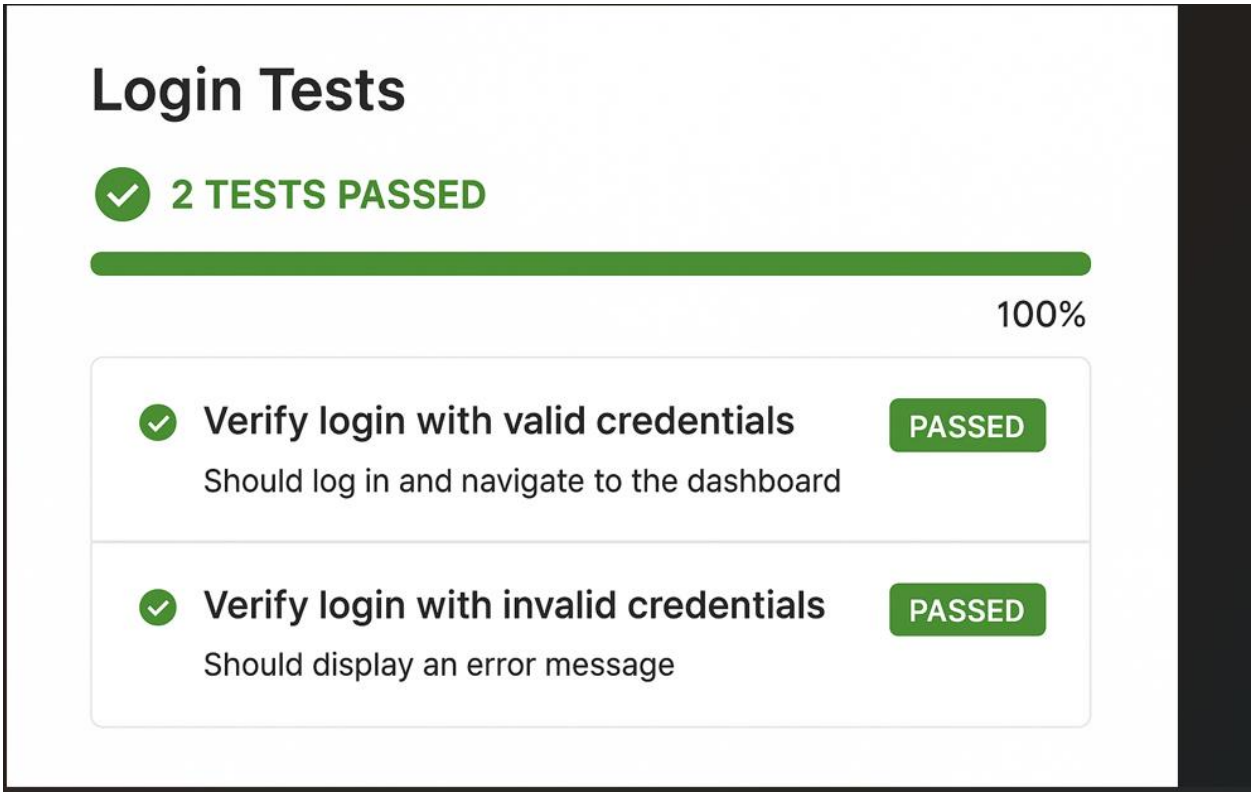
```
// Test Case: Verify login with valid and invalid credentials
// Step 1: Navigate to login page
navigateTo("https://example-login-page.com");

// Step 2: Test with valid credentials
typeText("[data-testid='username']", "validUser");
typeText("[data-testid='password']", "correctPassword");
click("[data-testid='login-button']");
assertElementVisible("[data-testid='dashboard']", "Dashboard loaded successfully");

// Step 3: Test with invalid credentials
navigateTo("https://example-login-page.com");
typeText("[data-testid='username']", "invalidUser");
typeText("[data-testid='password']", "wrongPassword");
click("[data-testid='login-button']");
assertElementVisible("[data-testid='error-message']", "Error message displayed");

console.log("AI Test Script is running successfully!");
```

Test Results



Analysis

AI-powered testing tools like Testim.io enhance software quality assurance by intelligently identifying UI elements, adapting to changes, and reducing maintenance effort. In this task, the automated test successfully validated both valid and invalid login scenarios. The AI engine automatically detected dynamic elements such as buttons and error messages, ensuring consistent performance even after minor UI modifications.

Compared to manual testing, AI-driven automation improves test coverage by executing multiple input combinations and UI flows quickly and reliably. It reduces human oversight errors, continuously learns from test runs, and prioritizes areas prone to failure. Additionally, self-healing test scripts automatically update when the application's structure changes, minimizing rework. Overall, AI-based testing provides faster execution, higher accuracy, and broader coverage—enabling developers and QA teams to detect defects earlier in the software lifecycle.

Part 3: Ethical Reflection

- ***Prompt**: Your predictive model from Task 3 is deployed in a company. Discuss:*
    - *Potential biases in the dataset (e.g., underrepresented teams).*
    - *How fairness tools like IBM AI Fairness 360 could address these biases.*

Ethical Reflection

When deploying a predictive model such as the one from Task 3 in a company setting, it is crucial to recognize and mitigate potential ethical challenges, especially regarding bias and fairness.

Potential Biases

The Kaggle Breast Cancer dataset, while widely used for research, may not represent all demographic groups equally. For instance, most samples are derived from specific medical centers, geographic regions, or patient populations, potentially leading to underrepresentation of certain ethnicities, ages, or socioeconomic backgrounds. If such a model were repurposed for organizational decision-making (e.g., allocating medical resources or predicting issue priority), these biases could translate into unequal predictions — where certain teams, regions, or demographic groups are systematically disadvantaged.

Similarly, in a corporate context, predictive models might mirror historical inequalities in data, such as over-prioritizing departments with more historical issue reports, or underweighting new or smaller teams.

Using Fairness Tools to mitigate Bias

Tools like IBM AI Fairness 360 (AIF360) can help identify, measure, and reduce bias in machine learning models. AIF360 offers metrics such as disparate impact, equal opportunity difference, and statistical parity difference to quantify fairness across groups. It also includes algorithms for bias mitigation, such as reweighing training samples, adjusting decision thresholds, or post-processing model outputs to balance outcomes.

By integrating AIF360 into the model development pipeline, developers can:

- ❖ Audit model performance across different demographic or organizational groups.
- ❖ Detect disparities early before deployment.
- ❖ Implement correction techniques to ensure transparent, fair, and equitable predictions.

In essence, bias mitigation ensures that predictive analytics empower decision-making without reinforcing existing inequalities—upholding both ethical standards and organizational trust.