

# WK 4 ASSIGNMENT

## Part 1: Theoretical Analysis

### 1. Short Answer Questions

***Q1: Explain how AI-driven code generation tools (e.g., GitHub Copilot) reduce development time. What are their limitations?***

#### ★ How AI Reduces Development Time

AI tools like GitHub Copilot act like an AI pair programmer. They reduce development time primarily by automating the most routine, repetitive, and context-switching-heavy parts of coding.

Copilot is trained on billions of lines of public code, allowing it to understand a programmer's intent from comments or the surrounding code. It then provides real-time code suggestions, ranging from single lines to entire functions. This automation saves time by:

- **Minimising Context Switching:** Developers spend less time searching the internet for syntax, API documentation, or common solutions, allowing them to stay "in the flow" of coding.
- **Automating Boilerplate:** It quickly writes standardised or repetitive code sections, freeing the developer to focus on the unique business logic of the application.
- **Accelerating Task Completion:** Studies show developers complete tasks significantly faster (some reports cite over 50% faster) when using Copilot because the AI shoulders the mechanical writing.

#### ★ Concrete Tasks Copilot Accelerates

Here are some specific tasks where Copilot dramatically speeds up a software engineer's workflow:

- **Generating Boilerplate Code:** Quickly writing repetitive structures, like setting up a new class, creating common data models, or adding standard headers/footers to files.
- **Writing Unit Tests:** Generating test cases and mock objects based on the function code it's meant to test, which is a critical but often time-consuming task.

- **Translating Code:** Automatically suggesting implementations in a different language or framework based on the logic of existing code (e.g., converting a function from Python to JavaScript).
- **API and Library Usage:** Suggesting the correct methods and arguments for libraries or frameworks, even if the developer isn't perfectly familiar with the API structure.
- **Adding Documentation and Comments:** Generating clear and descriptive comments or docstrings for functions based on the code's logic.
- **Refactoring and Code Improvement:** Proposing cleaner or more idiomatic ways to write a section of code, helping to improve quality as it's being written.

### ★ Important Limitations Engineers Should Be Aware Of

While incredibly helpful, Copilot is a tool, not a replacement for a developer. Engineers must maintain critical oversight. Here are three key limitations:

- **Code Quality and Correctness aren't Guaranteed:**
  - **The Limitation:** Copilot sometimes suggests code that is subtly incorrect, inefficient, or does not fully account for edge cases or the broader project architecture. Since it's trained on vast amounts of public code, it can also suggest suboptimal or dated patterns.
  - **The Awareness:** Engineers must review and validate every suggestion, treating it as a first draft rather than a final solution. Relying too heavily on it can lead to bugs or poor performance.
- **Intellectual Property and Licensing Concerns:**
  - **The Limitation:** Because Copilot is trained on a massive dataset including public open-source code, there's a small but real risk that a suggestion could closely resemble copyrighted or licensed code without proper attribution.
  - **The Awareness:** Organisations should use tools and policies to manage this risk, and developers should be cautious about accepting long, identical code blocks, especially in proprietary or commercial projects.
- **Risk of Over-Reliance and Skill Erosion:**
  - **The Limitation:** For junior developers, or for anyone using the tool uncritically, constantly accepting suggestions for even basic syntax or

common patterns can lead to a dependency that erodes their fundamental programming skills and critical thinking.

- **The Awareness:** Engineers must intentionally use Copilot as a learning aid—by asking why it made a suggestion—and still devote mental energy to designing complex logic rather than blindly delegating the task.

**Q2: Compare supervised and unsupervised learning in the context of automated bug detection.**

★ Differences

Feature	Supervised Learning	Unsupervised Learning
Training Data	<b>Labelled data</b> (Code samples explicitly tagged as "buggy" or "clean," or with a specific bug type.)	<b>Unlabeled data</b> (Raw code samples without predefined labels or classifications.)
Goal	<b>Prediction/Classification</b> (Learn a mapping from code features to known bug labels.)	<b>Pattern Discovery</b> (Find inherent structure, groupings, or anomalies in the code.)
Bugs Found	Known, previously identified, or characterised types of bugs.	Anomalous patterns that deviate from the norm, potentially indicating new or unknown bugs.
Common Tasks	Bug/No-Bug Classification, Severity Prediction, Bug Location Prediction.	Anomaly Detection, Code Clustering.

★ Real-World Examples

Approach	Concise Real-World Example
Supervised Learning	<b>Buggy Module Prediction:</b> A model is trained on historical data from a codebase, where each code module is labelled with a feature vector (e.g., code complexity metrics, change frequency) and a label ('bug-prone' or 'not bug-prone'). The model is then used to predict which new modules are likely to contain bugs.
Unsupervised Learning	<b>Anomaly Detection in Execution Traces:</b> An algorithm is used to analyse millions of execution traces or logs. It learns the 'normal' patterns of execution (e.g., function call sequences, resource usage) and flags any execution trace that significantly deviates from the learned norm as an anomaly, which may indicate a crash or a novel bug.

### ★ Discovering New Types of Bugs

The unsupervised learning approach is generally better for discovering new types of bugs, and here's why:

- **No Pre-existing Knowledge Required:** Supervised learning is limited to finding bug types that are present and explicitly labelled in its training data. It learns to recognise what it has *already been shown*.
- **Focus on Deviation:** Unsupervised learning, particularly through anomaly detection or clustering techniques, does not rely on a list of known bug signatures. Instead, it models what constitutes *normal, healthy* code or execution behaviour.
- **Novelty Detection:** Anything that falls outside the boundary of this learned 'normal' pattern is flagged as an anomaly. Since new, or "zero-day," bugs often manifest as unexpected and unique behaviours, they are precisely the kind of outliers that unsupervised models excel at detecting. They discover

unknown, hidden structures in the data that are unconstrained by human-defined labels.

### ***Q3: Why is bias mitigation critical when using AI for user experience personalisation?***

#### **★ Why Bias Mitigation is Crucial for Personalised User Experiences**

Bias mitigation is crucial when using AI for personalised user experiences because unchecked biases can lead to unfair, discriminatory, and harmful outcomes, which severely erode user trust and limit the software's effectiveness.

AI personalisation aims to tailor content, recommendations, and services to individual needs. However, if the underlying machine learning model is trained on data that reflects historical or societal biases (like gender, race, or socioeconomic status imbalances), the AI will learn and amplify those biases. Instead of providing an equitable and optimised experience for everyone, a biased system can:

- Systematically disadvantage certain user groups.
- Reinforce harmful stereotypes and cultural norms.
- Create filter bubbles that restrict the exposure of users to diverse or necessary information, thus limiting opportunities or viewpoints.

Essentially, without bias mitigation, personalised AI risks creating an inequitable digital experience that alienates users and can cause real-world harm.

#### **★ Two Examples of Potential Harms**

Failure to address bias in personalised AI can result in significant harm, including:

- **Reinforced Economic Inequality in Job Recommendations:** A job search platform's personalised recommendation engine, trained on historical hiring data that favoured men for high-paying technical roles and women for administrative roles, continues to learn and reinforce this pattern.
  - **Harm:** The system will disproportionately show high-value job advertisements to men and lower-value ones to women with similar qualifications. This limits the career opportunities for women, perpetuating the gender pay gap and economic inequality by restricting their access to information about lucrative positions.
- **Exclusion and Disenfranchisement in Content Delivery:** A social media or news feed algorithm is personalised based on regional engagement data that

historically underrepresents certain dialects, non-standard languages, or minority viewpoints.

- **Harm:** The algorithm will prioritise content and discussions from the overrepresented majority, leading to the near-invisibility of content and concerns relevant to the minority group. This silences minority voices, restricts their ability to connect and organise, and ultimately provides a significantly poorer, less relevant, and exclusionary user experience.

### ★ Common Approach to Reducing Bias

A common and highly effective approach to reducing bias in machine learning models, particularly those used for personalisation, is Data Preprocessing and Augmentation.

This approach focuses on fixing the root cause of many biases: the training data. It involves techniques applied before the model is trained, such as:

- Collecting more diverse and representative data to ensure all relevant demographic groups and contexts are included.
- Re-weighting or oversampling data from underrepresented groups to balance the dataset.
- Correcting or sanitising the data by identifying and removing features (or proxies for features) that are directly linked to protected attributes (like race or gender) but are not necessary for the core task.

By actively ensuring the training data is fair and representative, the resulting personalised AI model is less likely to learn and amplify societal inequities.

Would you like to know more about the different stages where bias can be mitigated in the AI development lifecycle (e.g., during model training or post-deployment)?

## 2. Case Study Analysis

***Read the article: AI in DevOps: Automating Deployment Pipelines.***

***Answer: How does AIOps improve software deployment efficiency? Provide two examples.***

AIOps significantly improves software deployment efficiency by enabling smarter decision-making, predictive analysis, and self-healing capabilities throughout the software lifecycle, thus automating repetitive tasks and increasing system reliability.

### ★ Core Ways AIOps Improves Deployment Efficiency

The article "AI-Powered DevOps: Automating Software Development and Deployment" highlights the following core improvements to deployment efficiency:

Area of Improvement	Core AIOps Mechanism	Impact on Efficiency
<b>Continuous Integration/Deployment (CI/CD)</b>	Predicts potential build failures and optimizes test case execution using historical data.	Ensures successful deployments and <b>reduces the risk of downtime</b> when systems go live.
<b>Automated Monitoring &amp; Incident Management</b>	Uses AI-enabled observability to analyze logs, metrics, and traces in real time, detecting patterns that indicate an <b>imminent failure</b> (anomaly detection).	Enables <b>proactive prevention</b> of issues before they affect users and automates remediation, <b>reducing downtime</b> from hours to seconds.
<b>Infrastructure as Code (IaC) Optimization</b>	Analyzes IaC configurations to recommend configurations that appropriately <b>balance performance and cost</b> .	Enhances settings optimization and detects misconfigurations that could lead to security exploits.

### ★ Practical Examples

Drawing from the provided article, here are two practical examples illustrating how AIOps speeds up deployment, reduces errors, or saves developer time:

- **Speeding Up Deployment and Saving Developer Time (CircleCI)**
  - **Mechanism:** CircleCI uses AI to analyze the historical success and failure rates of various test cases within the CI/CD workflow.
  - **Result:** The AI determines the optimal execution model and prioritizes running the test cases that offer the best efficiency savings first. This ensures developers receive feedback quicker, allowing them to iterate

more effectively and speed up the overall deployment process by eliminating unnecessary or time-consuming test runs that aren't providing value.

- **Reduced Errors and Faster Remediation (Harness & Facebook)**
  - **Harness:** Uses AI to automatically roll back failed deployments, minimizing the need for human intervention. This directly reduces human error and operational mistakes that can occur during manual fixes, ensuring that system stability is restored immediately.
  - **Facebook:** Leverages an AI-powered testing framework that can anticipate which tests are likely to "flake out" or fail despite code changes. By predicting these unreliable tests, the system focuses only on the most critical tests, making the deployment process run faster and more smoothly while reducing the number of errors introduced.