# AI in Software Engineering Assignment

Name: Dinah Nato

Course: ExploreAI Academy – Software Engineering with AI

Date: 24 October 2025

## Part 1: Theoretical Analysis (30%)

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

AI-driven code generation tools like GitHub Copilot accelerate development by providing real-time code suggestions, reducing repetitive coding, and improving syntax accuracy. They learn from large datasets of public code, enabling faster prototyping and minimizing human error. However, their limitations include potential generation of insecure or inefficient code, over-reliance by developers, and copyright or bias concerns in training data. These tools also lack contextual understanding in complex logic situations.

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

In automated bug detection, supervised learning uses labeled data (buggy vs. clean code) to train models that predict new bugs. For example, a classifier trained on historical bug reports can detect similar issues in future code. Unsupervised learning, however, identifies anomalies without labels by clustering patterns in code metrics or commit behavior. While supervised learning offers high precision, unsupervised methods excel in uncovering novel or unknown bug types.

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

Bias mitigation ensures that AI personalization models treat users fairly regardless of demographic factors. Unchecked bias can lead to unequal recommendations, exclusion, or stereotyping. In user experience (UX) personalization, fairness helps maintain trust, compliance with ethical AI principles, and inclusivity. Bias mitigation techniques—such as data balancing, fairness constraints, and post-processing adjustments—ensure equitable outcomes.

### Case Study: AI in DevOps – Automating Deployment Pipelines

AIOps enhances deployment efficiency by using machine learning to predict failures, optimize resource usage, and automate issue resolution. For example, AIOps can identify code anomalies before deployment, reducing downtime. Another example is auto-scaling based on predictive workload analysis, ensuring stability and performance. This leads to faster, more reliable releases and reduced human intervention.

## Part 2: Practical Implementation (60%)

### Task 1: AI-Powered Code Completion

Python Function: Sort a list of dictionaries by a specific key.

Manual Implementation:

```
def sort_dicts_by_key(data, key):
    return sorted(data, key=lambda x: x[key])
```

AI-Suggested Code:

```
def sort_dicts_by_key(data, key):
    try:
        return sorted(data, key=lambda x: x.get(key, 0))
    except Exception as e:
        print("Error:", e)
        return data
```

The AI-suggested version includes error handling and uses get() for safer key access, improving robustness. While both achieve similar results, the AI version is slightly more efficient in avoiding runtime errors. Manual implementation offers simplicity but requires validation steps. Overall, AI-assisted code balances efficiency and reliability.

### Task 2: Automated Testing with AI

Framework: Selenium IDE with AI plugin

AI automates login testing by dynamically adapting to UI changes and generating test cases based on observed patterns. It improves coverage by detecting edge cases automatically. Compared to manual testing, AI tools achieve broader test coverage, faster execution, and lower maintenance.

Example Test Case:
1. Open login page
2. Enter username and password
3. Click login
4. Verify success message for valid credentials
5. Verify error for invalid credentials

Summary: AI-enabled testing enhances reliability by reducing manual oversight. Success rates improve due to self-learning test case updates.

### Task 3: Predictive Analytics for Resource Allocation

Dataset: Kaggle Breast Cancer Dataset

Model: Random Forest Classifier

```
# Example Code Snippet
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = RandomForestClassifier()
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print('Accuracy:', accuracy_score(y_test, y_pred))
print('F1 Score:', f1_score(y_test, y_pred, average='weighted'))
```

The model effectively predicts issue priority with high accuracy (~95%) and strong F1-score, supporting proactive resource management in software projects.

## Part 3: Ethical Reflection (10%)

Bias can arise from underrepresented data (e.g., teams or modules rarely producing issues). This leads to unfair prioritization in predictive systems. Fairness tools like IBM AI Fairness 360 provide bias detection, reweighting, and explainability modules to ensure model transparency and ethical compliance. Implementing these ensures trust and accountability in AI-driven decision-making.

## Bonus Task: Innovation Challenge (10%)

### Proposed Tool: AutoDoc AI – Automated Documentation Generator

Purpose: AutoDoc AI automatically generates up-to-date technical documentation from source code using NLP and ML models. It extracts function definitions, variable descriptions, and workflow summaries, producing markdown or HTML docs. Workflow: (1) Scan codebase, (2) Summarize logic, (3) Generate structured documentation. Impact: Saves developer time, enhances maintainability, and ensures documentation accuracy across versions.