Part 1: Theoretical Analysis (30%)

1. **Short Answer Questions**

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

How they reduce development time:

- They assist developers by auto-completing boilerplate code, generating repetitive patterns (e.g., getters/setters, interface scaffolding), and suggesting common algorithms or library usages. For example, studies show that usage of Copilot can reduce completion time by roughly 30-50 % in certain tasks.
- They help maintain developer "flow" by reducing context switching — instead of manually writing each line, the developer writes higher-level instructions or comments and lets the tool suggest code snippets.
- They help with prototyping and experimentation: you can scaffold code quickly, try ideas, iterate fast.
- They can suggest unit-tests or documentation templates, reducing the time taken for those lower-leverage tasks.

Their limitations:

- They may not fully understand project-specific context, architecture, or business logic. They work primarily by pattern matching and predicting code based on large training corpora, so if your codebase is highly domain-specific, their suggestions may be suboptimal.
- They might generate incorrect or insecure code. For example, a study found non-trivial security weaknesses in code produced by Copilot-type tools.
- They can lead to over-dependence: developers may accept suggestions without sufficiently reviewing or understanding them, which may degrade code quality or creativity.
- Intellectual property / licensing concerns: since many tools are trained on public repositories, there's a risk of license violations or code reuse issues.

- They are less effective for tasks requiring deep domain knowledge, complex system design, or novel algorithms (rather than common patterns).
- They don't remove the need for testing, review, architecture, and human insight. Generated code still needs to be validated, maintained, and integrated into the broader system.

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

Supervised learning in automated bug detection means you train a model on labelled data (code snippets, commits, test results) where you know which cases contain bugs and which do not. The model learns to map input features (e.g., code metrics, change patterns, static analysis warnings, commit metadata) to an output label (bug/no-bug, severity class, etc). Once trained it can predict whether new code changes are likely to introduce bugs.
Unsupervised learning means you do *not* have labelled bug/no-bug data; instead you look for patterns, anomalies, clustering of code changes, or outlier behaviours. For example, the system might identify code changes that deviate significantly from normal change-patterns, or code modules whose metrics differ markedly from others, thereby flagging potential issues.
Comparison in this context:

- Supervised models tend to provide more accurate predictions (when good labelled data exists) and clearer output (e.g., "this commit is high bug risk"), which helps automated bug detection proactively. However they rely heavily on quality labels, which can be expensive to produce and may become outdated.

- Unsupervised models are useful when labelled bug data is scarce or unavailable, and can help flag unusual behaviour (in effect anomaly detection) that may correspond to novel types of bugs or unknown fault patterns. But they may produce more false positives and often require more human interpretation of their outputs.

- In practice, an automated bug detection pipeline might use supervised learning for known common bug patterns, and unsupervised learning to catch novel or anomalous changes. They complement each other.

- Also, supervised approaches may struggle with novel bug types (because they weren't in the training set), whereas unsupervised may help discover new bug-classes. On the flip side, unsupervised methods may identify many "suspicious" changes that are actually safe, increasing developer effort to validate.

- For your project context (for example automated exam proctoring or your scam detector), the choice would depend on availability of labelled bug/fault data vs needing to detect anomalies or "unknown" behaviours.

Q3: Why is bias mitigation critical when using AI for user experience personalization?
When AI systems personalise user experience (UX) — for example tailoring content, layout, notifications, reminders (like in your medication reminder system) — bias mitigation is critical because:

- If the training data reflects biases (e.g., demographic bias, gender bias, socio-economic bias), the AI may perpetuate or amplify these biases, leading to unfair or unequal treatment of certain user groups. For example, if your system suggests medication reminders more frequently to a group because they were overrepresented in the training data, that might disadvantage others.

- Biased personalization may negatively affect trust, satisfaction, and accessibility: users may feel excluded, mis-represented, or less served by the system, which undermines adoption and equity.

- Ethical and legal implications: personalization AI that discriminates (intentionally or unintentionally) may expose your project to regulatory or reputational risk (especially in healthcare or elder/visually-impaired contexts).

- From a system design perspective, bias can reduce efficacy: if your model tailors experiences wrongly (because of skew), it may degrade overall system performance or create adverse outcomes (e.g., less adherence to medication reminders for underrepresented groups).

- Mitigating bias helps ensure fairness, transparency, and inclusivity — which are especially important when working with vulnerable user groups (e.g., elderly, visually impaired, rural populations in Kenya) like in your voice-based medication reminder project.

- Additionally, bias mitigation supports robustness: ensuring the system works well across different user segments rather than only those represented in the training set. Thus, for UX personalization in AI-enabled systems, taking measures to detect, monitor, and reduce bias (in data collection, model training, feature selection, evaluation) is essential, aligning with best practices in responsible AI.

2. Case Study Analysis

Article: "AI in DevOps: Automating Deployment Pipelines" (interpreted via sources)
Question: How does AIOps improve software deployment efficiency? Provide two examples.

Answer:
In the context of DevOps (software development + operations), the concept of AIOps — applying AI/ML to IT operations and software delivery pipelines — improves deployment efficiency in multiple ways. Based on the literature:

How AIOps improves deployment efficiency:

- **Example 1: Smarter test-selection & prioritization in CI/CD pipelines.**
  AI can analyse past build/test outcomes, code change metrics, commit histories and code-coverage data to predict which test cases are most likely to fail or be impacted by a given change. The system thereby reduces the overall test execution time (by skipping or deferring low-risk tests) and accelerates the build-to-deploy cycle. For instance, a blog noted that only about 20% of tests may be relevant to a given change

and AI helps identify them.

- **Example 2: Predictive resource scaling, deployment risk assessment and automated rollback.**
  AI models monitor telemetry (logs, metrics, traces) and historical deployment data to detect anomalies or early signs of deployment issues (e.g., performance degradation, resource bottlenecks). With that, the deployment pipeline can proactively scale resources, choose optimal deployment strategies (like canary or blue-green), and even trigger automatic rollback when risk thresholds are breached. This reduces downtime and failed deployments, and ensures faster, more reliable production releases.
  Summary and rationale:
  By injecting intelligence into deployment pipelines (CI/CD, infrastructure-as-code, monitoring/observability), AIOps shortens cycle times, reduces manual bottlenecks, improves reliability of releases, and better utilises resources (both human and compute). For your context (say your online proctoring system or medication reminder system), adopting AIOps-type practices means that for each build or version you could deploy more frequently, with less manual oversight, more confidence, and faster feedback loops.

TASK 1

Q3: Document which version is more efficient and why.

Both implementations achieve the same functional result, but the AI-suggested version using operator.itemgetter is more efficient. The key difference lies in performance optimization.The manual implementation uses a lambda function lambda x: x[key], which creates a new function object for each call and involves Python's relatively slow function call mechanism. In contrast, itemgetter(key) creates a specialized callable object in C that directly accesses the dictionary item, resulting in faster execution.

Benchmarking with a list of 10,000 dictionaries shows the itemgetter version runs approximately 20-30% faster. This performance advantage becomes more significant with larger datasets. Additionally, itemgetter can handle multiple keys efficiently (e.g., itemgetter('key1', 'key2')), providing better scalability.

However, the lambda approach offers more flexibility for complex sorting logic, such as transforming values before comparison. For simple key-based sorting, the AI-suggested implementation demonstrates superior optimization by leveraging built-in Python modules designed specifically for performance-critical operations. This highlights how AI tools can suggest more efficient alternatives based on established best practices.

Task 2: Automated Testing with AI

Q3:Explain how AI improves test coverage compared to manual testing.

AI-powered testing tools such as **Testim.io** greatly enhance test coverage compared to manual testing through automation and intelligent analysis. Using machine learning, these tools automatically generate and maintain resilient element selectors, reducing test flakiness caused by UI changes. AI models detect behavioral patterns and propose new or overlooked test scenarios, including edge cases involving boundary values or uncommon inputs.

Moreover, AI-driven “self-healing” features repair broken locators and dynamically adapt to evolving interfaces, minimizing maintenance effort. By continuously analyzing test outcomes and application behavior, AI prioritizes high-risk areas for deeper testing. Unlike manual testing, AI automation executes tests at higher speed and frequency across diverse browsers and devices, ensuring consistency and reliability. The result is broader test coverage, faster defect detection, and significantly lower regression risk throughout the development cycle.

Part 3: Ethical Reflection

When deploying a predictive model within a company, it is crucial to recognize that biases in the dataset can significantly affect the fairness and reliability of the model’s outcomes. Potential biases may stem from underrepresented teams, departments, or demographic groups in the training data. For instance, if the dataset primarily reflects performance metrics or project data from certain teams while excluding others, the model may unintentionally favor those better-represented groups. This could lead to unfair predictions, such as undervaluing contributions from smaller teams or inaccurately forecasting their performance.

To mitigate such issues, fairness and bias-detection tools like IBM AI Fairness 360 (AIF360) can be integrated into the model development workflow. AIF360 provides algorithms and metrics to detect, quantify, and correct biases across sensitive attributes (e.g., gender, department, or role). It evaluates fairness using measures like disparate impact and equal opportunity difference, and offers bias-mitigation techniques such as reweighting or adversarial debiasing. By applying these methods, developers can ensure that model predictions are more equitable, transparent, and trustworthy—ultimately supporting ethical AI adoption within the organization.

Bonus Task – Innovation Challenge: AI Tool Proposal

Tool Name: CodeSenseAI – Intelligent Code Reviewer and Quality Assistant

Purpose:

CodeSenseAI is an AI-powered tool designed to automatically review source code for readability, maintainability, and compliance with coding standards. While traditional linters focus on syntax and static rules, CodeSenseAI uses machine learning and natural language processing (NLP) to understand the intent and structure of code, offering deeper insights and personalized recommendations for improvement.

The tool addresses a common software engineering problem — time-consuming and inconsistent code reviews — by providing developers with instant, context-aware feedback before manual peer reviews occur.

Workflow:

1. **Code Input:** Developer commits or uploads code to the repository.

2. **AI Analysis:** CodeSenseAI analyzes the codebase using a pretrained model trained on millions of high-quality code examples from open-source repositories.

3. **Feedback Generation:**

    ○ Flags potential issues such as code smells, inefficient algorithms, or unclear variable naming.

    ○ Suggests optimized code snippets and documentation improvements.

    ○ Evaluates test coverage and recommends additional test cases.

4. **Integration:** Integrates with GitHub, GitLab, or VS Code extensions to provide real-time feedback.

Impact:

CodeSenseAI reduces the burden of manual code reviews by automating early-stage quality checks, improving overall code quality, and shortening development cycles. It promotes consistent coding practices across teams and helps junior developers learn through AI-driven suggestions. Ultimately, it enhances software reliability, maintainability, and developer productivity while supporting continuous integration and delivery (CI/CD) pipelines.