WEEK 4: AI IN SOFTWARE ENGINEERING

MORAA ROBERT

**PART 1: THEORETICAL ANALYSIS**

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

**AI driven code generation tools reduce development time as they speed things up through various ways:**

* **They suggest full lines or codes of block such as functions and this reduces the time taken to type out the code.**
* **They have faster prototyping by generating working scaffolding so developers can iterate faster.**
* **They also use the surrounding cod to produce relevant snippets.**
* **They can also reduce cognitive load by off-loading syntax so engineers focus on design and architecture.**

**But despite of all these pros they also have several limitations which include incorrect suggestions, over-reliance on the developer’s side, license and IP ambiguity and bias toward common patterns.**

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

In supervised learning, labeled data is needed in order to commit labeled bugfix, and this type of learning optimizes for prediction and gives clear metrics for precisions and recall. Supervised learning is mostly used for predicting whether a pull request introduces a bug and classifying commit messages and triaging bug reports but despite of all these uses this type of learning has limitations, it needs labeled examples which may be costly and it may not generalize to unseen bug types.  
Unsupervised learning on the other side uses unlabeled data, it detects anomalies.  
it has several strengths: it finds unusual behavior without labels and it is good for early detection of novel bugs. It is usually used in detection in logs, clustering similar failure traces and detecting outlier commits.

**3.Why is bias mitigation critical when using AI for user experience personalization?**

* Mismatched experiences: Biased personalization can give different quality of UX to different user groups (e.g., showing lower-quality recommendations to underrepresented groups).
* User harm & exclusion: Bias may exclude or disadvantage minority users, impacting fairness and business reach.
* Reputational and legal risk: Biased systems can violate regulations or lead to user distrust and PR damage.
* Feedback loops: Personalized systems that are biased can reinforce the bias (e.g., fewer recommendations → less engagement → less data → persistent underrepresentation).
* Ethical product design: Ensures product inclusivity and aligns with accessibility and anti-discrimination principles.

Case Study: AI in DevOps - Automating Deployment Pipelines

How AIOps improves deployment efficiency

AIOps integrates ML/AI into monitoring, CI/CD, and incident workflows to reduce manual toil, prioritize meaningful alerts, predict failures, and automate remediation; yielding faster, more reliable deployments.

**Example 1: Predictive failure detection & pre-deployment checks**  
AI models analyze historical build, test, and runtime data to predict build/test failures before they run. This allows the pipeline to:

* Block risky deployments,
* Trigger targeted extra tests,
* Save resources by avoiding doomed builds.  
  (Outcome: fewer failed deployments and faster mean time to recovery.)

**Example 2 :Automated root-cause analysis & remediation**  
When a deployment causes anomalies, AIOps correlates logs,metrics and traces across services to surface the likely root causes and can trigger automated remediation (e.g., rollback, scale up/down, reconfigure), reducing human intervention and downtime.

ANALYSIS OF THE AI POWERED CODE COMPLETION

The AI-suggested solution (sorted with key=lambda d: d.get(key, None)) is concise, leverages Python’s highly optimized TimSort, and is O(n log n) in the average/worst case with low constant factors. It’s also stable and avoids mutating the input. The manual insertion-sort implementation is clearer pedagogically but is O(n²), which becomes impractical for medium or large lists. For production code the AI suggestion is superior for performance and readability. However, the manual implementation is useful for explaining stable sort behaviour or customizing comparison semantics that sorted can't express easily. A recommended pattern is to use sorted for correctness and performance, and add explicit input validation (e.g., consistent types, None handling) when needed. If keys may be missing or of mixed types, add a normalization step (cast values to a comparable form) before sorting. Finally, always add unit tests (edge cases: empty list, missing keys, mixed types) — Copilot can generate those tests too, but review them

SUMMARY OF AUTOMATED TESTING WITH AI

Automated UI tests run by Selenium reliably exercised the login flow for both valid and invalid credentials, capturing pass/fail outcome and screenshots. AI-augmented test platforms (e.g., Testim, Selenium IDE with AI plugins) speed test case creation by suggesting locators, generating wait strategies, and proposing alternative selectors when DOM changes. This increases coverage (more input combinations, edge cases) and reduces brittle tests. The automated tests can be scheduled in CI for regression checks, and AI helps maintain tests by automatically adjusting locators and suggesting retries on flakiness. Compared to manual testing, automation reduces repetitive effort, runs more permutations, and catches regressions early; however, human review remains necessary for UX and exploratory testing.

INNOVATION CHALLENGE: PROPOSE AN AI TOOL TO SOLVE A SOFTWARE ENGINEERING PROBLEM NOT COVERED IN CLASS (E.G., AUTOMATED DOCUMENTATION GENERATION).  
**Tool name:** DocuGen-AI — Automated, Contextual Developer Documentation Generator

**Purpose**  
Automatically generate feature-level and API documentation, including example snippets and integration notes, directly from code, tests, and CI pipelines to reduce documentation debt.

**Workflow**

1. **Code intake:** Scan repo, tests, and commit messages.
2. **Context extraction:** Parse function signatures, types, docstrings, unit tests, and example usages.
3. **Knowledge enrichment:** Use project-specific README, CHANGELOG, and linked issues to add rationale/context.
4. **Doc generation:** Produce Markdown docs and API reference pages, complete with example inputs/outputs derived from unit tests (sanitized).
5. **Continuous sync:** Run on PRs to update docs; block merge if public API changes without doc updates (policy).
6. **Human review UI:** Allow maintainers to accept/adjust generated docs before publishing.

**Impact**

* Reduced onboarding time for new developers.
* Docs synced to actual code & tests → fewer stale docs.
* Enables non-technical stakeholders to read feature summaries.
* Boosts code maintainability and lowers knowledge silo risk.