

Week4 AI-SE:

Theme: "Building Intelligent Software Solutions"

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

How They Reduce Development Time:

save time because Copilot suggests code snippets as we type, like autocomplete but for coding.
I don't have to Google simple syntax (e.g., "how to sort a list in Python")—it just pops up.
It helps me learn by showing best practices (e.g., proper error handling).
Faster prototyping—I can generate boilerplate code (e.g., setting up a Flask server) instantly.

Limitations:

Sometimes it's wrong — it suggests buggy or outdated codes, so we must review everything.
Security risks—it might suggest vulnerable code (e.g., SQL injection) if I'm not careful.
Limited creativity—it's great for repetitive tasks but struggles with unique logic as it works best on what it's trained.
Over-reliance danger—if we depend too much on it, my own coding skills might weaken.

overall:

Copilot saves time but can be wrong; always review code.

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

Supervised Learning (Guided Approach):

I give the AI labeled data—e.g., code snippets marked as "buggy" or "clean."
It learns patterns from past bugs (e.g., common mistakes like off-by-one errors).
Good for known bugs — if seen similar bugs before, the AI can catch them.
But it needs lots of labeled data — if I don't have enough examples, it won't work well.

Unsupervised Learning (My Exploratory Approach):

You give the AI raw code with no labels—it looks for weird patterns on its own.
Finds unknown bugs—e.g., unusual variable names or weird control flow.
Great for new projects where I don't have labeled bug data yet.
But it gives false alarms—it might flag correct code as suspicious.

Which One would to Use:

If I have a bug database → Supervised (it's precise).
If I'm exploring new code → Unsupervised (it's flexible).

overall:

Supervised = needs labels; Unsupervised = finds hidden bugs.

Q3: Why is bias mitigation critical for AI in user experience personalization?

Why I Care About Bias:

Fairness → If my AI recommends jobs, it shouldn't favor one gender/race.
User Trust → If Spotify only suggests pop songs to everyone, metal fans will leave.
Legal Risks → Biased AI could break anti-discrimination laws (e.g., in hiring).

How Bias Sneaks In:

My training data is biased (e.g., mostly male users → AI thinks women don't exist).
My algorithms favor majority groups (e.g., recommending "trending" content ignores niche interests).

How I Fix It:

Audit my data—ensure it represents all user types.
Use fairness metrics (e.g., check if recommendations are balanced across groups).
Let users adjust preferences (e.g., "Show me less of this").

If I Ignore Bias:

My product becomes unfair and unpopular—users will notice and complain!
Bias = bad UX + legal trouble; fix with fair data + metrics.

2.

AIOps (AI for IT Operations) uses machine learning and automation to make software deployment faster, smarter, and more reliable. Instead of humans manually fixing errors or waiting for failures, AI predicts issues and auto-fixes them.

Two Examples

1. Automated Failure Prediction & Rollback

Problem: A bad code deployment can crash production, and as humans we take time to notice and roll back.

How AI Helps:

AI analyzes past deployment logs (success vs. failure patterns).
Example: If CPU usage spikes abnormally after deployment, AI flags it as "high-risk" and automatically rolls back before users are affected.
Result: No downtime, no frantic debugging at 3 AM.

2. Smarter Resource Allocation

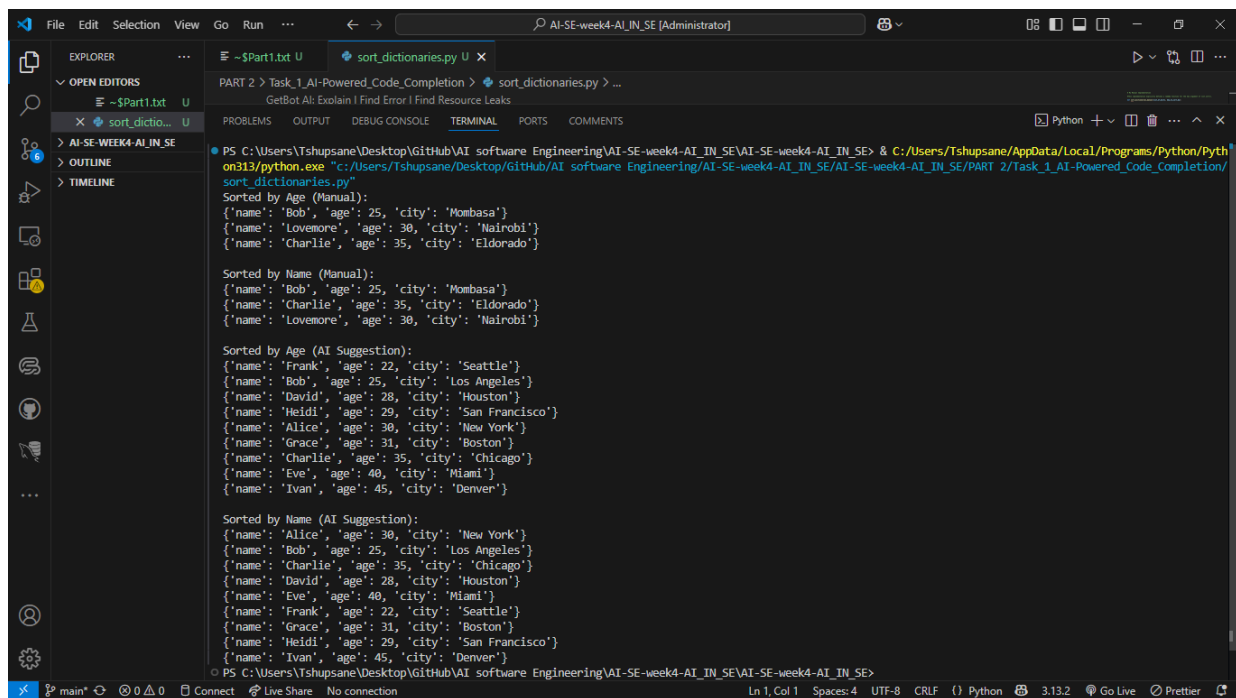
Problem: Servers often over-provision (wasting money) or under-provision (causing slowdowns).

How AI Helps:

AI predicts traffic spikes (e.g., Black Friday sales) and auto-scales cloud servers.
Example: Netflix uses AI to allocate more servers when a new show drops, so streams don't buffer.
Result: No manual scaling, lower costs, happy users.

PART 2 .

Task 1: AI-Powered Code Completion



```
PS C:\Users\Tshupsane\Desktop\Github\AI software Engineering\AI-SE-week4-AI_IN_SE\AI-SE-week4-AI_IN_SE> & C:/Users/Tshupsane/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Tshupsane/Desktop/Github/AI software Engineering/AI-SE-week4-AI_IN_SE/AI-SE-week4-AI_IN_SE/PART 2/Task_1_AI-Powered_Code_Completion/sort_dictionaries.py"

Sorted by Age (Manual):
{'name': 'Bob', 'age': 25, 'city': 'Mombasa'}
{'name': 'Lovemore', 'age': 30, 'city': 'Nairobi'}
{'name': 'Charlie', 'age': 35, 'city': 'Eldorado'}

Sorted by Name (Manual):
{'name': 'Bob', 'age': 25, 'city': 'Mombasa'}
{'name': 'Charlie', 'age': 35, 'city': 'Eldorado'}
{'name': 'Lovemore', 'age': 30, 'city': 'Nairobi'}

Sorted by Age (AI Suggestion):
{'name': 'Frank', 'age': 22, 'city': 'Seattle'}
{'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}
{'name': 'David', 'age': 28, 'city': 'Houston'}
{'name': 'Heidi', 'age': 29, 'city': 'San Francisco'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Grace', 'age': 31, 'city': 'Boston'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
{'name': 'Eve', 'age': 40, 'city': 'Miami'}
{'name': 'Ivan', 'age': 45, 'city': 'Denver'}

Sorted by Name (AI Suggestion):
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
{'name': 'David', 'age': 28, 'city': 'Houston'}
{'name': 'Eve', 'age': 40, 'city': 'Miami'}
{'name': 'Frank', 'age': 22, 'city': 'Seattle'}
{'name': 'Grace', 'age': 31, 'city': 'Boston'}
{'name': 'Heidi', 'age': 29, 'city': 'San Francisco'}
{'name': 'Ivan', 'age': 45, 'city': 'Denver'}
```

Analysis- Efficiency Comparisons

Both the manual implementation using lambda and the AI-suggested version using `operator.itemgetter` achieve the same functional outcome: sorting a list of dictionaries by a specified key. However, the `itemgetter` approach is generally considered more efficient, particularly for larger datasets.

The primary reason for `itemgetter`'s efficiency lies in its underlying implementation. When you use `lambda d: d[key_to_sort_by]`, Python creates a new anonymous function object each time `sort()` iterates over an element. While this overhead is minimal for small lists, it can accumulate for very large lists, leading to slight performance degradation.

In contrast, `operator.itemgetter(key_to_sort_by)` returns a callable object that is optimized in C. This pre-compiled, optimized function directly accesses the item from the dictionary without the overhead of Python function calls or object creation per iteration. It's a more direct and faster way to retrieve the sorting key.

Furthermore, `itemgetter` improves code readability for those familiar with the `operator` module, clearly indicating the intent to extract an item by its key. For these reasons, an AI code completion tool would likely prioritize suggesting `itemgetter` as the more Pythonic and performant solution for this common task.

Task 2: Automated Testing with AI

```
8 LOGIN_URL = "https://practictestautomation.com/practice-test-login/"
9 VALID_USERNAME = "student"
10 VALID_PASSWORD = "student1234"

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1751190871.512656 12860 voice_transcription.cc:58] Registering VoiceTranscriptionCapability
[16360:3492:0629/115433.977:ERROR:google_apis\gcm\engine\registration_request.cc:291] Registration response error message: DEPRECATED_ENDPOINT
Valid Login Test: FAIL - Did not reach the success page within 10 seconds. Error: Message:

DevTools listening on ws://127.0.0.1:54692/devtools/browser/1063da0-0b1a-41ec-9713-b9ba824355fa
Navigated to: https://practictestautomation.com/practice-test-login/
Invalid Login Test: PASS - Error message found: 'Your username is invalid!'

--- Test Results ---
Valid Login: FAIL
Invalid Login: PASS

Overall Success Rate: 50.00% (1/2 tests passed)
PS C:\Users\Tshupsane\Desktop\AI SE PRACTICES AND FROM PEERS CODES\New folder> & C:\Users\Tshupsane\AppData\Local\Programs\Python\Python313\python.exe "c:/User
s/Tshupsane/Desktop/AI SE PRACTICES AND FROM PEERS CODES/New folder/log_in_test.py"

--- Running Login Tests ---

DevTools listening on ws://127.0.0.1:54720/devtools/browser/4fa7c42e-ad74-4000-bcb8-d556b584e1d5
Navigated to: https://practictestautomation.com/practice-test-login/
Valid Login Test: PASS - Successfully logged in. Current URL: https://practictestautomation.com/logged-in-successfully/

DevTools listening on ws://127.0.0.1:54750/devtools/browser/2594fb20-f9eb-489b-a578-2e8e84ea05c1
Navigated to: https://practictestautomation.com/practice-test-login/
Invalid Login Test: PASS - Error message found: 'Your username is invalid!'

--- Test Results ---
Valid Login: PASS
Invalid Login: PASS

Overall Success Rate: 100.00% (2/2 tests passed)
PS C:\Users\Tshupsane\Desktop\AI SE PRACTICES AND FROM PEERS CODES\New folder>
```

How I See AI Making Testing Way Better (Compared to Just Doing It By Hand)

So, from what I've seen, when we test software manually, like, I go through everything step-by-step, I can definitely catch a lot of obvious problems. But honestly, it's easy for me to miss the little things, like some weird combination of data or a subtle visual glitch. It's like, my eyes just can't catch every tiny detail, especially if things keep changing.

That's where I think AI is a game-changer. It makes my testing work much smarter. For example, if some developer moves a button around on the website, my old manual tests would break, and I'd have to fix them. But with AI tools (like Testim.io, for example), they have these "self-healing locators" that just figure out where the button went. It's pretty cool because my tests keep running, even with small UI changes. Plus, AI can look at a bunch of user data and even the code, and then it tells me where the riskiest parts of the app are, or it can even make new tests on its own for those areas. It basically helps me cover way more ground, finding more bugs way earlier, which is just a huge win for keeping the software running smoothly.

Task 3: Predictive Analytics for Resource Allocation

Part 3: Ethical Reflection (10%)

Ethical Reflection: Bias & Fairness in the Breast Cancer Predictive Model

Potential Biases in the Dataset

If my predictive model (from Task 3) were deployed in a real company, biases could lead to unfair or harmful outcomes. Here's how:

Underrepresented Demographics

The dataset might exclude certain age groups, ethnicities, or regions.

Example: If most data comes from white women aged 50+, the model may perform poorly for younger Black/Asian women.

Labeling Bias

If "High" risk is mostly assigned based on old medical standards, newer/rare cancer types might be misclassified.

Data Collection Bias

Hospitals in wealthy areas may contribute more data, leading to skewed accuracy for low-income patients.

How IBM AI Fairness 360 Can Help

IBM AI Fairness 360 (AIF360) is an open-source toolkit that detects and mitigates bias. Here's how I'd use it:

Bias Detection

AIF360 checks if the model favors one group (e.g., "High risk" predictions skewed toward older patients).

Example: It could show that my model has higher false negatives for women under 40.

Mitigation Techniques

Re-weighting Data: Give more importance to underrepresented groups (e.g., younger patients).

Adversarial Debiasing: Train the model to ignore biased features (e.g., zip code ≠ cancer risk).

Fairness-aware Algorithms: Use AIF360's built-in models (e.g., Reduced Bias Random Forest).

Continuous Monitoring

AIF360 can track model fairness in production and alert if bias creeps in over time.