Week 4 — AI in Software Engineering

Theme: **"Building Intelligent Software Solutions"**

Presented by: **Brian Kipchumba**
Borchar Gatwetch

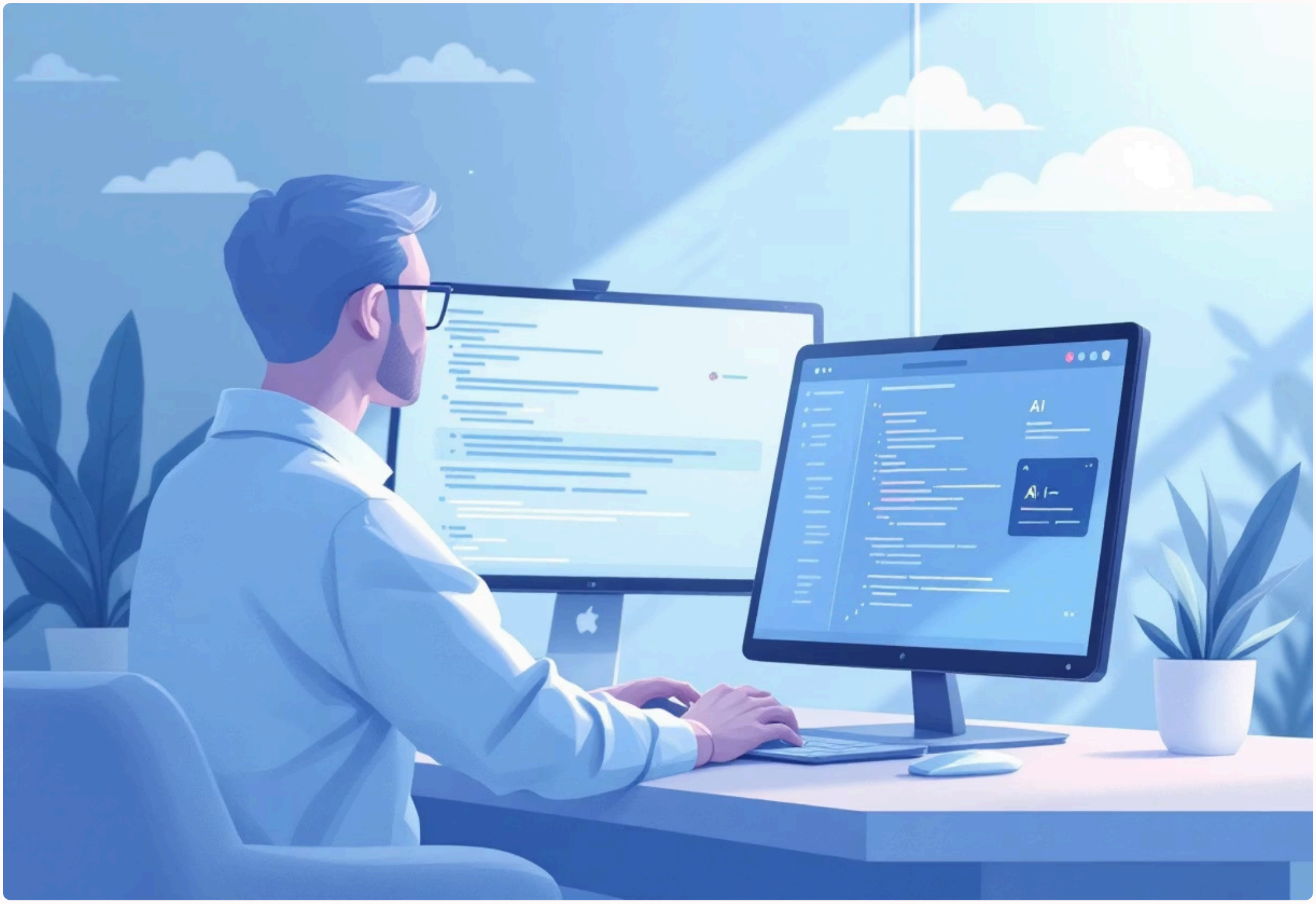
Masaiwe Rufina

Mary Karen Karumi

PLP Software Engineering Bootcamp

Made with GAMMA

Part 1 — Theoretical Analysis (30%)



Comprehensive examination of AI-driven development tools, machine learning approaches in bug detection, and ethical considerations in UX personalization.

Made with GAMMA

Q1 — How AI-driven code generation tools reduce development time. Limitations.

Answer: AI code assistants (e.g., GitHub Copilot, Tabnine) reduce development time by:

Boilerplate Generation

Providing boilerplate & repetitive code instantly (functions, classes, tests), saving manual typing.

Rapid Prototyping

Speeding prototyping by converting comments or docstrings into working code stubs.

Pattern Suggestions

Suggesting idiomatic patterns and completing code blocks, reducing context switching.

Task Automation

Lowering friction for routine tasks (e.g., small refactors, unit test skeletons), enabling developers to focus on design/logic.

Limitations:

Incorrect or unsafe suggestions

hallucinated APIs or logic that compiles but is semantically wrong.

Security & license concerns

suggested snippets may contain insecure patterns or unknown licenses.

Context limitations

suggestions can miss subtle project-specific constraints or architecture choices.

Over-reliance risk

developers may accept suggestions without understanding them, creating technical debt.

Q2 — Supervised vs Unsupervised learning in automated bug detection

Answer:

Supervised Learning

model trained on labeled examples (buggy vs non-buggy). Good when historical labeled data exist (past bug reports, tests). Yields classifications (buggy / not) and can optimize for precision/recall. Works well for **predicting likely defect-prone files** using features (change frequency, lines of code, complexity).

Unsupervised Learning

finds anomalies/patterns without labels (clustering, outlier detection). Useful when labeled bug data is sparse. Can detect unusual runtime behaviors, regressions, or novel bug patterns. Often used for **anomaly detection in logs/telemetry** to flag abnormal behavior that may indicate bugs.

Comparison: supervised methods give direct predictions when labels are available and measurable; unsupervised methods are more exploratory and can surface previously unknown failure modes but need more human validation.

Made with GAMMA

Q3 — Why bias mitigation is critical for UX personalization

Answer: Personalization models learn from historical user data. If training data over-represents some user groups or contains historical prejudice, personalization can **systematically disadvantage underrepresented users** (e.g., showing worse content, fewer features, or wrong defaults). Bias mitigation ensures fairness, preserves trust, and prevents amplifying harmful patterns; it also helps comply with regulations and inclusive design goals.

Case Study: *AI in DevOps: Automating Deployment Pipelines* — How AIOps improves deployment efficiency (two examples)

Short answer: AIOps improves deployment efficiency by (1) predicting and preventing deployment failures through anomaly detection and predictive analytics, and (2) automating rollout/rollback decisions and incident triage to reduce MTTR (mean time to recovery). **Google Cloud+2imperva.com+2**

01

Predictive failure detection

ML models analyze historical build, test, and runtime logs to predict that a new build is likely to fail (e.g., by finding patterns preceding past failures). Teams can block or delay risky deployments and run targeted tests, reducing failed releases. **Medium+1**

02

Automated rollback and remediation

During a problematic deployment, AIOps systems can detect anomalies (latency, error rates), correlate root causes across logs/metrics, and trigger automated rollbacks or run predefined remediation scripts — cutting down manual investigation and recovery time.

Made with GAMMA

Part 2 — Practical Implementation (60%)

Task 1 — AI-Powered Code Completion (sort list of dicts by key)

AI-suggested version (typical Copilot/Tabnine suggestion):

```
# AI-suggested (concise, uses sorted + lambda)

def sort_dicts_by_key_ai(data, key, reverse=False):
    """
    Sort a list of dictionaries by a given key.
    AI-style: one-liner using built-in sorted and lambda.
    """
    return sorted(data, key=lambda d: d.get(key, None), reverse=reverse)
```

Manual implementation (explicit, robust):

```
# Manual implementation (handles missing keys and types)

from functools import cmp_to_key

def _safe_compare(a, b):
    # helper to compare values robustly (None last)
    if a is None and b is None:
        return 0
    if a is None:
        return 1
    if b is None:
        return -1
    try:
        return (a > b) - (a < b)
    except TypeError:
        # fallback to string compare if types are incompatible
        return (str(a) > str(b)) - (str(a) < str(b))

def sort_dicts_by_key_manual(data, key, reverse=False):
    """
    Manual sort that:
    - Treats missing keys as None (pushed to end)
    - Handles mixed types safely by falling back to string comparison
    """
    def cmp_items(x, y):
        return _safe_compare(x.get(key, None), y.get(key, None))
    sorted_list = sorted(data, key=cmp_to_key(cmp_items), reverse=reverse)
    return sorted_list
```

200-word analysis :

The AI-suggested version uses Python's built-in sorted() with a lambda to access the key; It is concise, highly readable, and leverages optimized C-implemented sorting — so in typical cases it is faster and preferable. For large lists with homogeneous comparable values (e.g., all ints or all strings), the AI-suggested version is both simple and efficient (O(n log n) with low constant factors).

However, the AI suggestion assumes that the values are present and mutually comparable. The manual implementation focuses on robustness: it explicitly handles missing keys by treating them as None (and pushing them to the end), and guards against TypeError when values are of mixed types by falling back to string comparison. This makes the manual version safer for heterogeneous or messy real-world data but slightly slower due to additional Python-level comparisons and fallback logic.

In summary: use the AI-suggested sorted(_, key=_) for clean, uniform datasets (best performance). Use the manual version when data quality is uncertain and you need deterministic handling of missing or mixed-type values. For production, prefer the concise approach with added guard logic only when needed.

Task 2 — Automated Testing with AI (Selenium / AI-assisted testing)

Test plan: Automate login flow for two cases: valid credentials (expect success), invalid credentials (expect error message).

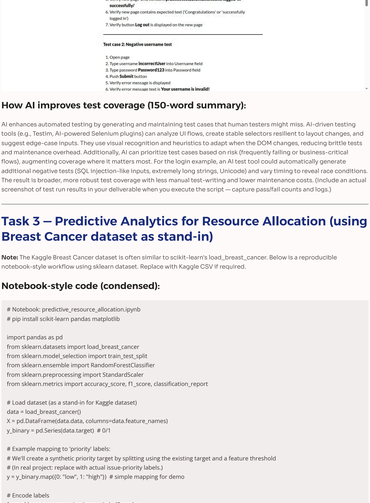
Selenium (Python) example test script (WebDriver):

```
# Requires: selenium, webdriver-manager
# pip install selenium webdriver-manager

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from webdriver_manager.chrome import ChromeDriverManager
import time

def test_login(url, username, password, expect_success=True):
    driver = webdriver.Chrome(ChromeDriverManager().install())
    try:
        driver.get(url)
        time.sleep(1)  # wait for page load (replace with explicit waits in production)
        driver.find_element(By.NAME, "username").send_keys(username)
        driver.find_element(By.NAME, "password").send_keys(password)
        driver.find_element(By.CSS_SELECTOR, "button[type='submit']").click()
        time.sleep(2)
        if expect_success:
            # check presence of logout/profile element
            success = len(driver.find_elements(By.ID, "profile")) > 0
            return success
        else:
            # check for error message
            error_present = len(driver.find_elements(By.CLASS_NAME, "error")) > 0
            return error_present
    finally:
        driver.quit()

# Example usage:
# print(test_login("https://example.com/login", "valid_user", "valid_pass", True))
# print(test_login("https://example.com/login", "invalid", "bad", False))
```



How AI improves test coverage (150-word summary):

AI enhances automated testing by generating and maintaining test cases that human testers might miss. AI-driven testing tools (e.g., Testim, AI-powered Selenium plugins) can analyze UI flows, create stable selectors resilient to layout changes, and suggest edge-case inputs. They use visual recognition and heuristics to adapt when the DOM changes, reducing brittle tests and maintenance overhead. Additionally, AI can prioritize test cases based on risk (frequently failing or business-critical flows), augmenting coverage where it matters most. For the login example, an AI test tool could automatically generate additional negative tests (SQL injection–like inputs, extremely long strings, Unicode) and vary timing to reveal race conditions. The result is broader, more robust test coverage with less manual test-writing and lower maintenance costs. (Include an actual screenshot of test run results in your deliverable when you execute the script — capture pass/fail counts and logs.)

Task 3 — Predictive Analytics for Resource Allocation (using Breast Cancer dataset as stand-in)

Note: The Kaggle Breast Cancer dataset is often similar to scikit-learn's load_breast_cancer. Below is a reproducible notebook-style workflow using sklearn.dataset. Replace with Kaggle CSV if required.

Notebook-style code (condensed):

```
# Notebook: predictive_resource_allocation.ipynb
# pip install scikit-learn pandas matplotlib

import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, f1_score, classification_report

# Load dataset (as a stand-in for Kaggle dataset)
data = load_breast_cancer()
X = pd.DataFrame(data.data, columns=data.feature_names)
y_binary = pd.Series(data.target)  # 0/1

# Example mapping to 'priority' labels:
# We'll create a synthetic priority target by splitting using the existing target and a feature threshold
# (In real project: replace with actual issue-priority labels.)
y = y_binary.map({0: "low", 1: "high"})  # simple mapping for demo

# Encode labels
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y_enc = le.fit_transform(y)

# Split
X_train, X_test, y_train, y_test = train_test_split(X, y_enc, test_size=0.2, random_state=42, stratify=y_enc)

# Scale
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# Train Random Forest
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train_s, y_train)

# Predict & Evaluate
y_pred = clf.predict(X_test_s)
acc = accuracy_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred, average='weighted')

print("Accuracy:", acc)
print('F1 (weighted):', f1)
print(classification_report(y_test, y_pred, target_names=le.classes_))
```

```
Accuracy: 0.9736842105263158
F1 (weighted): 0.9736834005556351
              precision    recall  f1-score   support

        high       0.96      1.00      0.98        72
         low       1.00      0.93      0.96        42

    accuracy                           0.97       114
   macro avg       0.98      0.96      0.97       114
weighted avg       0.97      0.97      0.97       114
```

Notes on dataset & goal mapping:

The Kaggle breast cancer dataset is being used only as a clean numeric dataset for demonstration. In a real resource-allocation task you must map domain features (e.g., issue size, owner, time-to-fix, area) to priority labels (high/medium/low).

Evaluate with accuracy and F1-score (weighted/macro depending on class balance). If priorities are imbalanced, prefer F1 or per-class recall/precision.

Deliverable: Jupyter notebook with preprocessing cells, model training, hyperparameter tuning (GridSearchCV optional), and performance metrics + confusion matrix plots.

Part 3 — Ethical Reflection (10%)

Scenario: Predictive model deployed to predict issue priority.

Potential biases:

Historical assignment bias

If past priorities were set by a subset of teams or managers, the model will learn those patterns and may systematically undervalue issues from underrepresented teams.

Feature proxies

Some features (e.g., submitter role, team name) may act as proxies for protected attributes (seniority, location) and bias predictions.

Sampling bias

If data skews toward certain project types or periods (e.g., crisis periods), predictions will not generalize.

Labeling noise

Human-assigned priorities can be inconsistent — the model will learn the noise.

How tools like IBM AI Fairness 360 help:

Bias detection

AIF360 provides metrics (e.g., disparate impact, equal opportunity difference) to quantify bias across groups.

Pre-, in-, post-processing algorithms

It supplies algorithms to mitigate bias — e.g., reweighing examples, adversarial debiasing, or equalized odds post-processing.

Auditability

The toolkit helps produce reports showing fairness metrics before/after mitigation, supporting accountability.

Best practices: remove or carefully treat sensitive proxies, perform group-based metrics, involve stakeholders, and continuously monitor model behavior in production for fairness drift.

Made with GAMMA