**PLP ACADEMY**

**FEBRUARY COHORT VII 2025**

**AI IN SOFTWARE ENGINEERING**

GROUP 100

GROUP MEMBERS:

1. STEVE ASUMBA
2. EMMANUELLA AIMALOHI ILEOGBEN

**WEEK 4: AI IN SOFTWARE ENGINEERING** (**THEME**:**"BUILDING INTELLIGENT SOFTWARE SOLUTIONS" 💻🤖 )**

**ASSIGNMENT DOCUMENTATION WHICH CONSISTS OF PART ONE: THEORETICAL ANALYSIS, PART TWO: PRACTICAL IMPLEMENTATION, PART THREE: ETHICAL REFLECTION AND THE BONUS TASK**

**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?
* **Q2**: Compare supervised and unsupervised learning in the context of automated bug detection.
* **Q3**: Why is bias mitigation critical when using AI for user experience personalization?

**2. Case Study Analysis**

* Read the article: [*AI in DevOps: Automating Deployment Pipelines*.](https://azati.ai/blog/ai-powered-devops-automation/)
* Answer: How does AIOps improve software deployment efficiency? Provide two examples.

**Solutions:**

**Q1: AI-Driven Code Generation Tools**

AI-driven code generation tools like GitHub Copilot, Amazon CodeWhisperer, and Tabnine significantly reduce development time by automating repetitive tasks and assisting developers in writing code more efficiently. Here are how those tools achieve this:

1. *Boilerplate Code Generation:* This automates the creation of repetitive code (e.g. class definitions, API endpoints, database queries), saving time on mundane tasks.

2. *Context-Aware Suggestions:* This analyze the surrounding code to provide relevant suggestions, improving workflow efficiency. For example, when writing a React component, it may trigger AI to suggest appropriate state management logic.

3. *Code Autocompletion:* These tools predict and suggest code snippets in real-time as developers’ type, reducing manual typing and syntax errors. For example, when writing a function, Copilot can auto-complete the entire block based on context.

4. *Documentation & Comment Generation:* Some tools auto-generate docstrings and comments, improving code maintainability.

5. *Faster Prototyping & Experimentation*: Developers can quickly generate POC (Proof of concept) code, accelerating early-stage development.

6. *Error Reduction & Faster Debugging:* These AI tools detect potential bugs and suggest fixes, reducing debugging time. For example, Copilot may recommend a missing “*try-catch”* block for error handling.

7. *Learning from Open-Source Repositories:* These tools provide best-practice implementations without requiring manual searches while been trained on vast public codebases.

**Limitations of Ai Code Generation Tools**

1. *Lack of Deep Understanding:* These AI models don’t truly “understand” logic; they predict patterns, leading to incorrect or nonsensical suggestions.

2. *Legal & Licensing Issues:* The AI-generated code may inadvertently replicate licensed or proprietary code, leading to compliance risks.

3. *Code Quality & Security Risks:* AI-generated code may contain vulnerabilities or inefficiencies if trained on flawed public repositories.

4. *Performance Overhead:* Some tools require constant internet access or high computational resources, slowing down workflows.

5. *Over-Reliance & Skill Erosion:* Junior developers may become dependent on Ai tools, hindering their ability to learn core programming concepts.

6. *Limited Context Awareness:* AI tools struggle with large-scale architectural decisions or complex business logic, while good at short snippets.

7. *Bias in Training Data:* The models trained on open-source data may reflect biases or outdated practices present in those repositories.

AI-driven code generation tools significantly speed up development by automating repetitive tasks, reducing errors, and improving code suggestions. However, they should not replace human oversight, developers must review AI-generated code for correctness, security, and efficiency. Used wisely, these tools act as powerful assistants, not replacements, in the software development lifecycle.

**Q2: Supervised vs. Unsupervised Learning for Automated Bug Detection**

Automated bug detection can leverage both supervised and unsupervised learning techniques, each with distinct advantages and limitations. Below is the comparison of the two approaches in this context:

1. **Supervised Learning for Bug Detection:** This relies on labelled datasets where bugs (or vulnerabilities) are pre-identified, and the model learns to classify or predict defects based on these examples.

**How it Works:**

a. It requires a dataset of code snippets labelled as “buggy” or “clean”.

b. The common techniques are classified into logistic regression, decision trees, and neural networks.

c. It also includes a sequence model which can be RNNs, Transformers for code analysis, etc.

d. Example tools/applications, includes DeepCode or Snyk Code which uses historical bug data to predict vulnerabilities.

e. It predicts bug in models using IDEs (e.g., IntelliJ Inspect Code)

**Advantages**

a. *High accuracy if well-trained:* It works well when sufficient labelled data exists.

b. Good for known bug patterns, it is good at detecting recurring issues, for example, null pointer exceptions, SQLi.

c. Many models (e.g., decision trees) provide reasoning for bug detection.

**Limitations**

a. *Bias in training data:* It may miss bugs if underrepresented in the dataset.

b. *Requires labelled data:* It manually annotates bugs which is usually time-consuming and expensive.

c. It struggles with novel or rare bugs not present in training data.

2. **Unsupervised Learning for Bug Detection:** This does not require labelled data and instead identified anomalies or patterns in code that deviate from the norm.

**How It Works:**

a. It trains on raw and unlabeled (data) code (e.g. GitHub repositories).

b. Its common techniques include clustering (e.g., k-means for grouping similar code patterns);

c. Furthermore, it detects anomalies (e.g., autoencoders, isolation forests);

d. Also includes statistical analysis which allows deviation from typical code metrics.

e. The examples of tools/applications include rule-based + anomaly detection (Semgrep partially). It finds unusual code patterns (CodeBERT).

**Advantages**

a. It is easier to apply to large, diverse codebases.

b. It works in raw codebases without manual annotation.

c. It can detects novel bugs with unusual patterns that supervised models miss.

**Limitations**

a. It is harder to interpret why an anomaly was detected.

b. It depends on future engineering and needs good representations (e.g., ASTs, embeddings).

c. It may flag correct code as false positives.

**Comparison Summary**

|  |  |  |
| --- | --- | --- |
| Aspect | Supervised Learning | Unsupervised Learning |
| 1.Data Requirement | Needs labeled bug/non-bug examples. | Works on raw, unlabeled code |
| 2. Scalability | Limited by labeled dataset availability. | More scalable (no labeling needed). |
| 3. Explainability | Easier to interpret (e.g., decision rules) | Harder to explain (e.g., anomaly scores). |
| 4.Detection Capability | Best for known, recurring bugs. | Can detect novel, unseen anomalies. |
| 5.Accuracy | High if training data is good. | Prone to false positives. |

**Best Use Cases**

1. Supervised Learning:

a. Well-defined bug patterns (e.g., buffer overflows, SQL injection).

b. Projects with historical bug data (e.g., enterprise software).

2. Unsupervised Learning:

a. New or undocumented vulnerabilities.

b. Large-scale code analysis where labeling is impractical.

**Hybrid Approaches**

Many modern tools (e.g., GitHub CodeQl, SonarQube) combine both:

a. Supervised for known vulnerabilities.

b. Unsupervised for anomaly detection.

In Conclusion, supervised learning is precise but data-hungry, its ideal when labeled datasets exist. Unsupervised learning is flexible but noisy, better for exploratory bug hunting. Combining both often yields best results in automated bug detection.

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

AI-powered personalization (e.g., recommendation systems, dynamic content, targeted ads) tailors user experiences based on data-driven insights. However, unchecked AI bias can lead to harmful, unfair, or exclusionary outcomes, making bias mitigation essential.

1. *It Prevents Discrimination and Exclusion:* AI models learn from data. If the training data reflects existing societal biases (e.g., historical discrimination, underrepresentation of certain groups), the Ai will inevitably learn and perpetuate these biases. This can lead to personalized experiences that reinforce harmful stereotypes, discriminate against specific demographics (based on race, gender, age, socioeconomic status, etc.), and exclude marginalized users. For instance, a job-search platform’s AI recommends higher-paying roles only to male users due to historical hiring biases.

2. *Unfair Treatment:* AI can unintentionally privilege or disadvantage certain groups. This can manifest in content selection (favoring certain types of content or sources), consumer profiling (stereotyping users based on skewed features), or even in the pricing and availability of services.

3. *Alienation and Offense:* When personalization is biased, it can lead to irrelevant, offensive, or frustrating experiences for users. Imagine a clothing retailer’s Ai consistently suggesting outfits that reinforce outdated gender norms, or a travel platform primarily catering to a high-income user, alienating budget-conscious travelers. This can make users feel misunderstood, disrespected, or even targeted.

4. *Reduced Trust in the Brand*: Users are increasingly aware of AI’s capabilities and potential for bias. If a brand’s personalized experiences are perceived as unfair or discriminatory, it will significantly erode user trust and credibility, leading to negative brand perception and potential public backlash.

5. *It Decreases User Engagement:* Users are less likely to engage with products or services that don’t genuinely cater to their needs or that make them feel marginalized. This directly impacts key UX metrics like time spent, conversion rates, and repeat usage.

6. *It Damages Brand Reputation:* In today’s interconnected world, instances of AI bias can quickly go viral, leading to severe reputational damage. Brands that deploy biased AI risk public backlash, boycotts, and long-term harm to their image.

7. *Legal and Regulatory Risks:* Governments and regulatory bodies are increasingly focusing on AI ethics and fairness. Biased AI algorithms that violate anti-discrimination or data protection laws can lead to hefty fines, legal disputes, and regulatory scrutiny.

8. *Operational Inefficiencies:* Biased AI can lead to inefficient resource allocation and sub-optimal strategies, impacting overall business performance and effectiveness.

9. *Missed Market Potential:* Biased AI can inadvertently exclude valuable consumer segments. By failing to understand and cater to diverse user groups, businesses miss out on significant market opportunities and potential revenue.

**How Bias Arises in AI Personalization**

1. *Feedback Loops:* If a biased recommendation is reinforced by user engagement (even if that engagement is due to limited choices), it can create a positive feedback loop that amplifies the initial bias over time.

2. *Data Bias:* This is the most common source, arising from unrepresentative, incomplete, or historically prejudiced training data.

3. *Algorithmic Bias:* This occurs when the design of the algorithm itself, or the features it prioritizes, inadvertently introduces bias, even with unbiased data.

4. *Human Decision Bias:* Biases from developers, data labelers, or other human inputs during the AI lifecycle can seep into the system.

**Mitigation Strategies are Essential**

To counteract these risks, bias mitigation involves a multifaceted approach, including:

a. Continuously checking AI outputs for signs of bias.

b. Establishing clear principles for responsible AI development and deployment.

c. Making AI decisions transparent and understandable to both developers and users.

d. To ensure training data accurately reflects the diversity of the user base.

e. Incorporating human review and intervention in AI-driven decisions.

f. User Feedback Mechanisms: Allowing users to report biased or irrelevant experiences.

In conclusion, bias mitigation is not just an ethical nice-to-have, but a fundamental requirement for creating truly effective, inclusive, and trustworthy AI-powered user experience personalization. Failure to address bias can lead to significant negative consequences for users, brands, and society as a whole.

**Case Study Analysis**

- Read the article: [*AI in DevOps: Automating Deployment Pipelines*.](https://azati.ai/blog/ai-powered-devops-automation/)

- Answer: How does AIOps improve software deployment efficiency? Provide two examples.

**Solution:**

**AI in DevOps: Automating Deployment Pipelines**

How AIOps Improve Software Deployment Efficiency

AiOps (Artificial Intelligence for IT Operations) enhances software deployment by automating repetitive tasks, predicting failures, and optimizing workflows using AI/ML. Below are two key ways it improves efficiency, as highlighted in the Azati article:

1. Automated Anomaly Detection & Incident Resolution

**How it works:**

AIOps tools analyze logs, metrics, and traces in real-time to detect anomalies (e.g., sudden latency spikes, deployment failures). Machine learning models predict issues before they escalate and suggest or auto-implement fixes.

For Example, if a deployment causes memory leaks, AIOps can auto-rollback the release and alert engineers. Tools like Datadog or Dynatrace use AI to correlate incidents across systems, reducing mean time to resolution (MTTR).

**The Impact:**

a. It enhances faster deployments with fewer rollbacks.

b. It reduces downtime from undetected failures.

2. Intelligent Continuous Integration/Continuous Deployment (CI/CD) Optimization

**How it works:**

AIOps optimizes CI/CD pipelines by predicting build failures, prioritizing test cases, and allocating resources dynamically.

For example, AI skips redundant tests by running passed tests on unchanged code, which speeds up pipelines. AI predicts peak load during deployment and auto-scales cloud infrastructure using AWS Auto Scaling + AI-driven triggers.

**Impact:**

a. It executes faster pipeline execution by reducing CI runtime between 30-50%.

b. It lowers infrastructure costs via efficient resource use.

**Key Benefits from the Article**

1. *Proactive Risk Mitigation:* AI predicts deployment risks (e.g., compatibility issues)

2. *Data-Driven Decisions:* Recommends optimal deployment windows based on historical success rates.

3. *Self-Healing Systems:* Automatically resolves common deployment errors (e.g., container crashes).

Tools Mentioned/Related

Azure AIOps, Google Cloud’s Operations Suite, Splunk IT Service Intelligence.

Conclusion: AIOps boosts deployment efficiency by replacing manual oversight with AI-driven automation, ensuring faster, more reliable software releases. Companies adopting AIOps see fewer outages, lower costs, and accelerated DevOps cycles.

### ****Part 2: Practical Implementation**** (60%)

#### ****Task 1: AI-Powered Code Completion****

**- Tool**: Use a code completion tool like GitHub Copilot or Tabnine.

**- Task**:

a. Write a Python function to sort a list of dictionaries by a specific key.

b. Compare the AI-suggested code with your manual implementation.

c. Document which version is more efficient and why.

**- Deliverable**: Code snippets + 200-word analysis.

**Assignment Solution:** Submitted on a Jupyter Notebook which would be attached and uploaded to the GitHub repository. The Jupyter Notebook is titled Week 4-Part 2

#### ****Task 2: Automated Testing with AI****

**- Framework**: Use Selenium IDE with AI plugins or Testim.io.

**- Task**:

a. Automate a test case for a login page (valid/invalid credentials).

b. Run the test and capture results (success/failure rates).

c. Explain how AI improves test coverage compared to manual testing.

**- Deliverable**: Test script + screenshot of results + 150-word summary.

**Solution:**

Selenium IDE with AI plugins for Login Page Testing

Test Script (Selenium IDE format) – File uploaded as Week4-Login Tests.side

**Screen recording of Results Include:**

1. Two Test cases (valid (Part4) and invalid login) with green checkmarks indicating success

2. Execution time for each test

3. Summary showing 100% pass rate

4. AI plugin notifications showing element locator adjustments made during execution.

**150-Word Summary: How AI Improves Test Coverage**

AI-enhanced testing tools like Selenium IDE with AI plugins or Testim.io significantly improve test coverage compared to manual testing. AI algorithms can automatically:

1. Generate additional test cases by analyzing application behavior and identifying edge cases humans might miss.

2. Self-heal tests by adjusting locators when UI elements change, reducing maintenance.

3. Analyze Dom structures to detect potential vulnerabilities in untested areas.

4. Prioritize tests based on risk analysis and code changes.

5. Perform visual validation by comparing screenshots with baselines.

While manual testing is limited by human speed and attention span, AI can execute thousands of test variations rapidly. AI tools learn from each execution, continuously improving test coverage. They detect subtle patterns in test failures that humans might overlook, and can test across data combinations automatically. This results in more robust test coverage with less effort, especially for regression testing in agile environments.

#### ****Task 3: Predictive Analytics for Resource Allocation****

**- Dataset**: Use [Kaggle Breast Cancer Dataset.](https://www.kaggle.com/competitions/iuss-23-24-automatic-diagnosis-breast-cancer/data)

**- Goal**:

a. Preprocess data (clean, label, split).

b. Train a model (e.g., Random Forest) to predict issue priority (high/medium/low).

c. Evaluate using accuracy and F1-score.

**- Deliverable**: Jupyter Notebook + performance metrics.

**Solution:**

Predictive Analysis for Resource Allocation - Breast Cancer Dataset

**Key Points About This Solution:**

- *Data Preprocessing:* The breast cancer dataset is clean (no missing values).

1. Created synthetic priority labels (high/medium/low) based on tumor malignancy and size.

2. Used stratified train-test split to maintain class distribution.

- *Model Training:* Implemented Random Forest with hyperparameter tuning using GridSearchCV.

1. Optimized for F1-score (weighted) to handle class imbalance.

2. Included class weight adjustments to improve minority class performance.

- *Evaluation Metrics:* Reported both accuracy and F1-score (weighted).

1. Included a confusion matrix and classification report.

2. Added feature importance analysis to understand model decisions.

*Performance:* The model should achieve high accuracy (> 95%) on this dataset.

1. F1-score will be slightly lower due to the class imbalance.

2. The confusion matrix show how well it distinguishes between priority levels.

### ****Part 3: Ethical Reflection**** (10%)

**- Prompt**: Your predictive model from Task 3 is deployed in a company. Discuss:

a. Potential biases in the dataset (e.g., underrepresented teams).

b. How fairness tools like IBM AI Fairness 360 could address these biases.

**Solution:**

**Addressing Bias and Fairness in the Breast Cancer Predictive Model Deployment**

Potential Biases in the Dataset: While the Breast Cancer Wisconsin dataset is widely used in machine learning, several potential biases could impact real-world deployment, they are as follows:

1. *Demographic Representation Bias*: The dataset lacks demographic information (age, race, ethnicity, socioeconomic status).

- Most samples like come from a similar geographic region (Wisconsin).

- Potential underrepresentation of minority populations in the training data.

2. *Measurement Bias*: All tumor measurements were collected using the same imaging technology.

- Diagnostic criteria may very across different hospitals/pathologists.

- Thresholds for “high priority” (mean radius > 17.5) may not generalize globally.

3. *Labeling Bias*: Our synthetic priority labels (high/medium/low) are based solely on tumor size and malignancy.

- Real clinical priorities might consider additional factors (patient age, comorbidities).

- The label creation process may introduce subjective bias.

4. *Temporal Bias*: The data was collected in the 1990s.

- Modern diagnostic techniques and treatment protocols may differ

- Cancer characteristics may have evolved over time.

5. *Access Bias:* The dataset only includes patients who had access to healthcare and diagnostic services.

- May not represent populations with limited healthcare access.

**Addressing Biases with IBM AI Farness 360 (AIF360)**

The IBM AI Fairness 360 toolkit provides several approaches to mitigate these biases:

1. **Bias Detection**:

Python

from aif360.datasets import BinaryLabelDataset

from aif360.metrics import BinaryLabelDatasetMetric

# Convert our data to AIF360 format (would need to add demographic attributes)

privileged\_group = [{'race': 1}] # Assuming white patients are privileged

unprivileged\_group = [{'race': 0}]

dataset = BinaryLabelDataset(df=pd.concat([X, y], axis=1),

label\_names=['priority'],

protected\_attribute\_names=['race'])

metric = BinaryLabelDatasetMetric(dataset,

unprivileged\_groups=unprivileged\_group,

privileged\_groups=privileged\_group)

print("Disparate Impact Ratio:", metric.disparate\_impact())

print("Statistical Parity Difference:", metric.statistical\_parity\_difference())

2. **Mitigation Strategies:**

Pre-processing:

Python

**from aif360.algorithms.preprocessing import Reweighing**

**# Apply reweighting to balance groups**

**RW = Reweighing(unprivileged\_groups=unprivileged\_group,**

**privileged\_groups=privileged\_group)**

**dataset\_transf = RW.fit\_transform(dataset)**

**In-processing:**

**Python**

**from aif360.algorithms.inprocessing import AdversarialDebiasing**

**# Use adversarial debiasing during model training**

**debiased\_model = AdversarialDebiasing(privileged\_groups=privileged\_group,**

**unprivileged\_groups=unprivileged\_group)**

**debiased\_model.fit(dataset)**

**Post-processing:**

**Python**

**from aif360.algorithms.postprocessing import EqOddsPostprocessing**

**# Calibrate predictions to equalize odds**

**postprocessor = EqOddsPostprocessing(privileged\_groups=privileged\_group,**

**unprivileged\_groups=unprivileged\_group)**

**postprocessor.fit(dataset, dataset\_pred)**

**3. Comprehensive Fairness Assessment**

**Python**

**from aif360.metrics import ClassificationMetric**

**classification\_metric = ClassificationMetric(**

**dataset, dataset\_pred,**

**unprivileged\_groups=unprivileged\_group,**

**privileged\_groups=privileged\_group)**

**print("Average Odds Difference:", classification\_metric.average\_odds\_difference())**

**print("Equal Opportunity Difference:", classification\_metric.equal\_opportunity\_difference())**

**print("Theil Index:", classification\_metric.theil\_index())**

**Implementation Recommendations for the Company**

**1. Data Collection Enhancement:**

**a. Augment dataset with demographic information.**

**b. Include samples from diverse geographic regions.**

**c. Partner with multiple healthcare institutions.**

**2. Continuous Monitoring:**

**a. Implement fairness metrics in production monitoring**

**b. Track performance across patient subgroups.**

**c. Set up alert thresholds for fairness metrics.**

**3. Model Retraining Protocol:**

**a. Establish regular retraining cycles with new data.**

**b. Include fairness as an optimization objective.**

**c. Maintain multiple model versions for audit purposes.**

**4. Clinical Validation:**

**a. Have medical professionals validate priority labels.**

**b. Incorporate additional clinical factors into priority determination.**

**c. Establish an ethics review board for model updates.**

**5. Transparency Measures:**

**a. Document all data sources and preprocessing decisions.**

**b. Provide model cards with fairness characteristics.**

**c. Implement explainability tools for clinical users.**

**By implementing these fairness measures, the company can ensure the predictive model allocates resources more equitably while maintaining clinical effectiveness. The AIF360 toolkit provides concrete technical solutions to complement organizational policies focused on equitable healthcare delivery.**

### ****Bonus Task (Extra 10%)****

**- Innovation Challenge**: Propose an AI tool to solve a software engineering problem not covered in class (e.g., automated documentation generation).

**- Deliverable**: 1-page proposal outlining the tool’s purpose, workflow, and impact.

**Solution:**

**AI-Powered “Code Context Assistant” for Legacy System Modernization**

Problem Statement

Many organizations struggle with legacy codebases that lack proper documentation, have outdated dependencies, or use deprecated patterns. Developers waste countless hours:

a. Deciphering undocumented code

b. Identifying safe modernization paths.

c. Avoiding breaking changes during refactoring

Proposed Solution: Code Context Assistant (CoCA)

An AI-Powered IDE plugin that:

1. Recommends modernization paths.

2. Detects breaking change risks.

3. Provides interactive code evolution guide.

4. Automatically generates contextual documentation.

**Key Features**

|  |  |  |
| --- | --- | --- |
| *Feature* | *Description* | *AI Technique Used* |
| Smart Code Annotations | Generates inline comments explaining complex logic | Transformer-based NLP (CodeTS) |
| Dependency Modernization Advisor | Recommends library upgrades with mitigation steps | Dependency Graph Analysis + LLMs |
| Change Impact Simulator | Predicts which tests/files might break from a change | Code Embeddings + Graph Neural Networks |
| Architecture Visualizer | Generates interactive diagrams of system flow | Code2Vec + D3.js Integration |
| Temporal Code Analysis | Shows how a code block evolved over time | Git History Mining + Time-Series Modeling |

Technical Implementation

Python

class CodeContextAssistant:

def \_\_init\_\_(self, repo\_path):

self.repo = Repository(repo\_path)

self.embedder = CodeBERTEmbedder()

self.llm = FineTunedGPT4ForCode()

def generate\_docs(self, code\_block):

"""Generates contextual documentation"""

embeddings = self.embedder.encode(code\_block)

return self.llm.generate(

f"Explain this code for maintenance:\n{code\_block}\n"

f"Code embeddings: {embeddings[:5]}..."

)

def assess\_refactor\_risk(self, file\_path):

"""Predicts breakage risk score (0-1)"""

dependency\_graph = build\_call\_graph(file\_path)

risk\_score = GraphNeuralNetwork.predict\_breakage(dependency\_graph)

return risk\_score

**Innovation Differentiators**

1. Temporal Intelligence: Analyses git history to explain why code evolved a certain way.

2. Change Simulation Sandbox: “What-if” testing environment for proposed changes.

3. Pattern Preservation Mode: Detects intentional ant-patterns (like performance hacks) before “fixing” them.

4. Multi-Language Support: Works across Java, COBOl, Python, and legacy DSLs.

**Business Value**:

|  |  |
| --- | --- |
| Metric | Improvement |
| Onboarding Time | | (Arrow down) 65% |
| Modernization Errors | | (Arrow down) 80% |
| Documentation Coverage | | (Arrow up) 90% |
| Mean Time to Refactor | | (Arrow down) 70% |

**Deployment Options**:

1. VS Code/IntelliJ Plugin

2. CI/CD Integration

3. Standalone Web App for Architects

**Ethical Considerations**

1. *Code Privacy:* Local processing option for sensitive repos.

2. *Bias Mitigation:* Regular audits for modernization recommendations.

3. *Transparency:* “Show your work” toggle for AI-generated suggestions.

The tool bridges the gap between understanding legacy systems and executing safe modernization – a pain point not adequately addressed by current AI coding assistants.

**Deliverables:**

1-page proposal outlining the tool’s purpose, workflow, and impact.

**Project Proposal: Code Context Assistant (CoCA):**

An AI-Powered Tool for legacy System Modernization

**Purpose**

The Code Context Assistant (CoCA) is an AI-driven Ide plugin designed to accelerate and de-risk legacy system modernization. It also helps developers to:

1. predict breaking changes before refactoring.

2. visualize code evolution to understand historical context.

3. automatically document complex, undocumented codebases.

4. recommend safe upgrade paths for dependencies and architecture.

Target users include software engineers, architects, and DevOps teams working with ageing systems.

**Workflow**

1. Code Analysis & Documentation Generation: IT scans the codebase using CodeBERT and GPT-4 to generate, inline comments for complex logic, high-level module summaries, and call-graph visualizations.

2. Change Impact Simulation: Before refactoring, developers run a “What-if” analysis to predict test failures, highlight dependent modules at risk, estimate effort for proposed changes.

3. Modernization Advisory: It identifies outdated dependencies and suggests upgrades, It flags deprecated patterns (for example, unsafe memory operations in C++), and ir provides step-by-step migration scripts.

4. Interactive Code Evolution Explorer: It mines Git history to show why a code block was modified, the past bugs linked to changes, and the ownership and expertise mapping.

**Expected Impact**

|  |  |
| --- | --- |
| Metric | Improvement |
| Onboarding Time | | (Arrow down) 65% |
| Modernization Errors | | (Arrow down) 80% |
| Documentation Coverage | | (Arrow up) 90% |
| Mean Time to Refactor | | (Arrow down) 70% |

Business Value

a. It preserves institutional knowledge as teams change.

b. It accelerates cloud migration for legacy applications.

c. It reduces modernization risks in critical systems.

Deployment Options

a. IDE Plugin (VS Code, IntelliJ)

b. CI/CD Integration (GitHub Actions, Jenkins)

c. Standalone Web App for architects

Next Steps

1. Pilot Phase – Test with 3-5 enterprise codebases.

2. Feedback Loop – Incorporate developer UX improvements.

3. General Availability – Launch on JetBrains Marketplace & VS Code Extensions.

**Project Lead: Emmanuella Aimalohi Ileogben**

**Estimated Timeline: 6-9 months to MVP**

**CoCA bridges the gap between understanding legacy systems and executing safe modernization which is a critical need in today’s fast-evolving tech landscape.**