Shift Left with Generative AI: Automating Library Replacement

FEBRUARY 26, 2025

Ipek Ozkaya, ozkaya@sei.cmu.edu
Technical Director, Engineering Intelligent Software Systems

Carnegie Mellon University
Software Engineering Institute

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM25-0159

Generative AI – The Landscape

People who talk about it

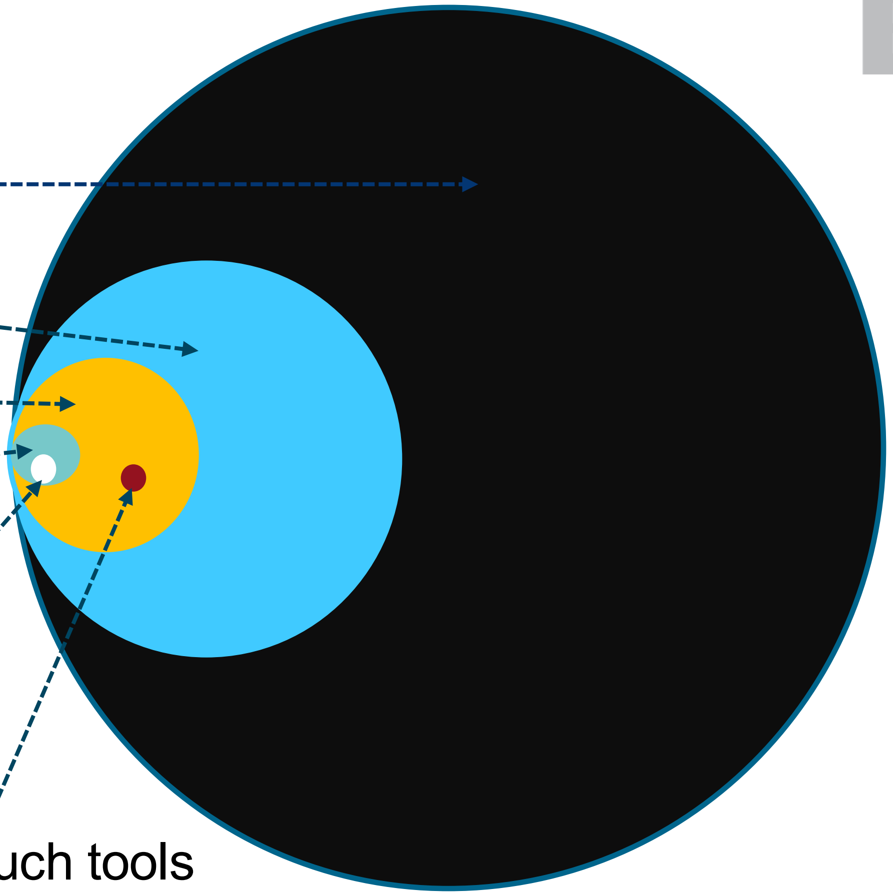
People who understand it

People who research it

People who implement solutions for it

People who develop next generation AI algorithms and hardware solutions

People who consider the impact of use of such tools on system development and its structure and behavior



Automating Library Upgrades

Most organizations rely on error prone and resource intensive manual processes to replace libraries at a tempo that does not always match needs.

- NDAA Section 835 Technical Debt study* uncovered multiple teams continuing to deploy systems using out-of-date libraries because of challenges in replacing or updating them.
- Because we lack tools to accelerate this process, teams are unable to keep up with supply chain changes and other upgrades with architectural implications.

*I. Ozkaya, F. Shull, J. Cohen, and B. O'Hearn, "Report to the Congressional Defense Committees on National Defense Authorization Act (NDAA) for Fiscal Year 2022 Section 835 Independent Study on Technical Debt in Software-Intensive Systems," *Carnegie Mellon University, Software Engineering Institute's Digital Library*. Software Engineering Institute, Technical Report CMU/SEI-2023-TR-003, 7-Dec-2023 [Online]. Available: https://doi.org/10.1184/R1/24043392. [Accessed: 1-Jun-2024].

Different Forms of Library Replacement

Replace Library During Translation

Replace library *Foo* used in language A with library *Bar* in language B

Replace Library

Replace library *Foo* with library *Bar*

Major Library Upgrades

Update library Foo from v1.1.x to v2.0.0

Minor Library Upgrades

Update library Foo from v1.1.x to v1.1.y

Our focus

- Library is used loosely to include libraries and frameworks
- Existing tools such as RevAPI (https://revapi.org/) and OpenRewrite (https://docs.openrewrite.org/) assist the workflows.

Tools like Dependabot (https://github.com/dependabot) already help

Decomposing Library Replacement

An example: Apache HTTP Server 2.3 introduces a new heartbeat feature. Our project uses a custom heartbeat library, and we would like to replace that with Apache's version when upgrading to 2.3.

Possible sub-tasks for this replacement

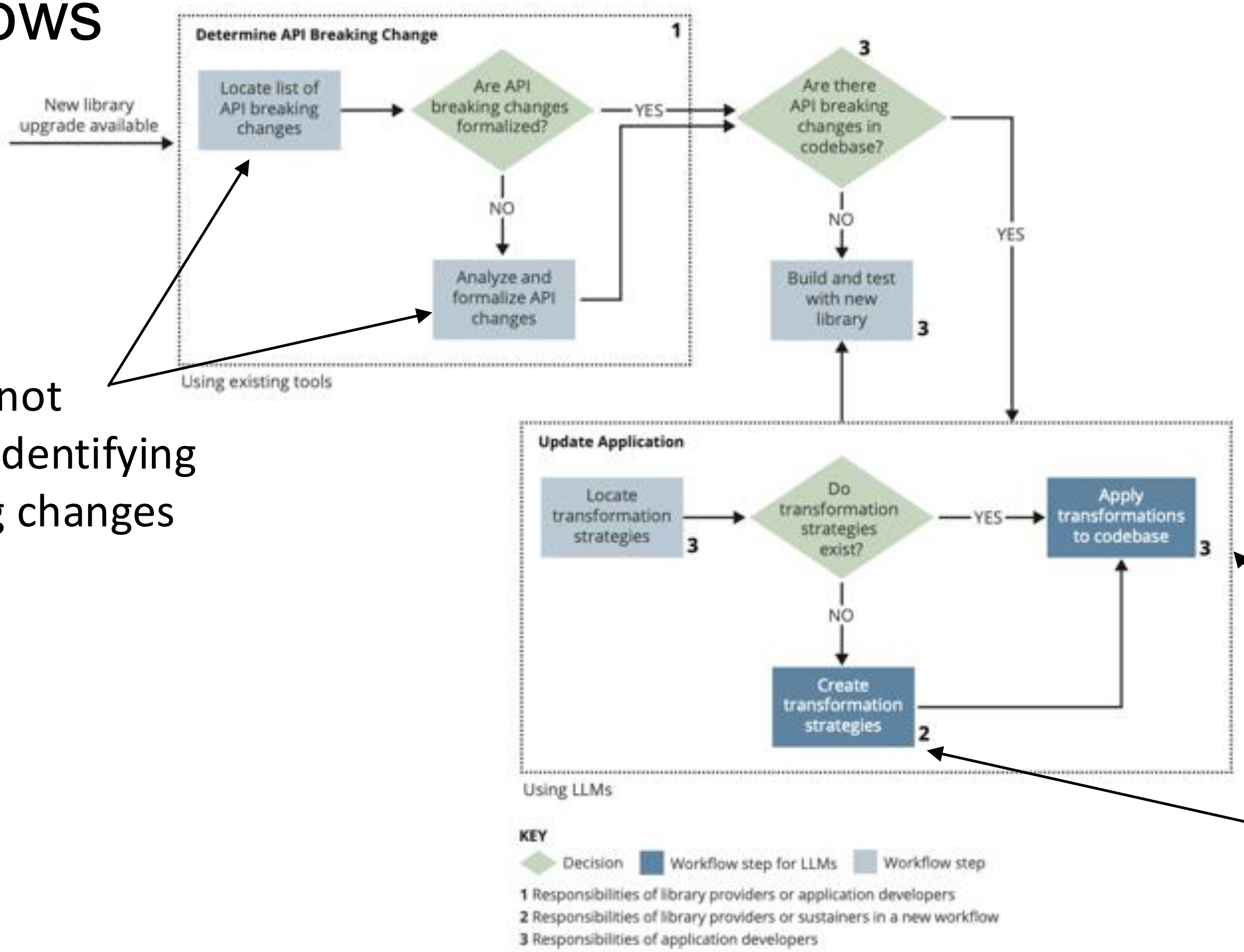
- Discovery sub-tasks
  - Find all files that use <custom_hearbeat>
  - Enumerate specific elements of the <custom_hearbeat> API that are used (e.g., 4 of 20 elements are actually used)
  - Separate elements into dependent subsets (e.g., 2 elements are part of a protocol such as open before use)
- Strategy sub-tasks
  - For each subset of API elements, discover a transformation from <custom_heartbeat> to <Apache 2.3 mod_hearbeat>
- Solution sub-tasks
  - For each file using <custom_heartbeat>, modify code to realize relevant transformation strategies

Integration of non-LLM approaches

LLM focus

OpenRewrite or LLM?

Workflows



1: LLMs are not effective in identifying API breaking changes

2: While our experiments show mixed results, LLMs have potential for improving the overall library upgrade flow

[1] Analyzing API Breaking Changes with LLMs

Carnegie Mellon University Software Engineering Institute

Experiments to answer :

- Can the LLM/other existing tools identify API breaking changes in an upgrade?
- Can the LLM/other existing tools find both syntactic and semantic changes?
- For an API breaking change, can the LLM describe how the API changed?
- Can the LLM provide the list in the format we specify?
- Can the LLM/other existing tools combine different sources of information?

Approach:

- Baseline API breaking changes for upgrading to Mockito 2.1.
  - 220 API breaking changes using RevAPI (including all internal APIs)
- Create prompt variation strategies.
  - Invariants: role (Java developer), approach (step-by-step), output format (baseline format)
  - Eight variations
- Run each prompt, compare with the baseline

Analyzing API Breaking Changes - Results

Carnegie Mellon University Software Engineering Institute

Prompt	Total LLM responses	Correct	Partially Correct	Incorrect	Missing (Baseline total – (Correct +Partially Correct))	
Basic	46	1	0	45	219	
Prompt with example_type: Different_lib	69	1	4	64	215	
Prompt with example_type: Same_lib	38	0	0	38	220	
Prompt with source_of_truth: Uploaded_document	95	0	0	95	220	
Prompt asking class_by_class	40	0	0	40	220	
Prompt with class_by_class Iterative [cumulative]	122	15	15	92	190	Works better for class removals than for other change types
Prompt with definition_of_change_types	121	1	0	120	219	8 unique, the last 3 changes repeated 30+ times until token limit reached

Conclusion: Existing tools, such as RevAPI, are more reliable sources for generating a formal list of syntactic API breaking changes in library upgrades. Our goal was to evaluate if LLMs could add missing semantic changes; however, they fall short in this area, as well as in fact-checking syntactic changes.

Create Transformation Strategies

A transformation strategy is a set of instructions for changing your application code to adapt to the API breaking change. Experiments aimed to address:

- What does an OpenRewrite recipe (rewrite rule) look like?
- What information do we need to feed the LLM to generate either the fix or the recipe?
- What strategies do developers resort to?
- When should an OpenRewrite recipe be used versus a prompt?

Approach:

- Conduct our own experiments to understand LLM strengths/weaknesses.
- Design two developer exercises* (one without any use of LLMs, one using LLMs + OpenRewrite) and compare time and correctness.

*41 masters of software engineering graduate students enrolled in CMU Fall 2024 18664 - Software Refactoring class

Can LLMs assist in Generating Fixes?

Goal: Identify how to fix code

- From Mockito 1.x to 2.1
- 73 RevAPI reported external API breaking changes
- Generated prompt variations
- Used successful variations to create prompt templates to generate recommended fixes.

RevAPI Code	Prompt Template
java.class.noLongerImplementsInterface	In Mockito verson 2.1 the class[KEY] no longer implements the [IMPLEMENTS_OLD] interface. Can a different class be used in its place? Reply with only the final suggestion.
java.class.noLongerInheritsFromClass	In Mockito verson 2.1 the class[KEY] no longer inherits from the class [INHERIT_OLD].How would you change an existing codebase to adapt to this? Reply with only the final suggestion.
java.class.removed	The Mockito class[KEY] has been removed. Suggest a different class that can be used instead. Reply with only the final suggestion.
java.class.visibilityReduced	The Mockito class[KEY] had its visibility reduced to [VISIBILITY_NEW].Suggest a different class that can be used instead. Reply with only the final suggestion.
java.generics.formalTypeParameterAdded	In Mockito verson 2.1 the method[SUB] had its signature changed. How would you change an existing codebase to adapt to this?Reply with only the final suggestion.
java.method.addedToInterface	In Mockito verson 2.1 the interface[KEY] now requires the new method [SUB].How would you change an existing codebase to adapt to this? Reply only with the final suggestion.
java.method.removed	The Mockito method[KEY].[SUB] was removed. How would you change an existing codebase to adapt to this?[NULL]
java.method.returnTypeTypeParametersChanged	The Mockito method[KEY].[SUB] had its return type changed to [RETURN_NEW].How would you change an existing codebase to adapt to this? Reply with only the final suggestion.
java.method.visibilityReduced	In Mockito verson 2.1 the method[KEY].[SUB] had its visibility reduced from [VISIBILITY_OLD] to [VISIBILITY_NEW].What Mockito method can be used in its place? Reply with only the final suggestion.
java.class.kindChanged	In Mockito verson 2.1 the class[KEY] was changed to an interface. How would you change an existing codebase to adapt to this? Reply with only the final suggestion.

Mixed Results

For some change types, e.g. java.method.addedToInterface, there is no general solution and so *incomplete* is the best possible result.

Such cases typically rely on knowledge of application-specific use of the API.

Change Type	Grand Total	Correct	Incomplete	Incorrect
java.class.removed	13	7		6
java.generics.formalTypeParameterAdded	9	4	1	4
java.class.noLongerInheritsFromClass	4	4		
java.method.removed	12	3	2	7
java.method.returnTypeTypeParametersChanged	9	3	6	
java.class.noLongerImplementsInterface	3	2		1
java.class.kindChanged	1	1		
java.class.visibilityReduced	2	1		1
java.method.visibilityReduced	1			1
java.method.addedToInterface	19		9	10
Grand Total	**73**	**25**	**18**	**30**

Developer Experiments – Time Spent

Which approach (LLM-prompt + OpenRewrite or traditional) took less time end-to-end (discovering and applying fixes)?

Case Study	Traditional Faster	LLM + Open Rewrite Faster
Vysper	14	7
Ari-toolkit	12	8

Reported in hours:

Overall students struggled more with the ari-toolkit and LLM combination as there was one change that could not be fixed with a recipe.

Case Study	Traditional (in avg. hrs)	LLM + Open Rewrite (in avg. hrs)
Vysper	6	8.7 hrs (3 hrs to 20 hrs)
Ari-toolkit	6	10 hrs (3 hrs to >25hrs)

Developer Experiments – Prompting Approaches

Naïve approaches

- some directly put in the assignment text and asked for the solution
- some blindly generated code and pasted errors into the next prompt
- some did not follow instructions (e.g., not separating solution identification from creating a recipe)

Evolving approaches – some changed direction mid-way

Sophisticated approaches

- provided detailed requirements of context and tools being used
- enumerated the task change-by-change

The Right Tool for the Right Task

The LLMs must be targeted narrowly on tasks they are good at.

- LLM-based approaches need to take the risk of **developer naiveté** into account.
  - Developers often use naïve approaches and fail to decompose tasks for better LLM utility.
- Effective LLM use **requires** integrating them into workflows with complementary tools and approaches.
  - LLMs struggle with fact-finding and analytical tasks, a limitation now confirmed by other research.

Image generated by Dall-E

Maturing the Use of AI for Software Engineering

CHALLENGES

Scale
degrading effectiveness

Errors
catching vs. avoiding errors

Abstractions
design concepts, models, …

Consistency
propagating changes

Data
when to train or fine tune

APPROACHES

Decompose problems using architectural insights

Enforce consistency across multiple smaller sub-problems

Pair with static analyses

Discover use of abstractions from artifacts

Rely on existing code search and static analysis tools

A fool with a tool is still a fool!

Grady Booch

Team

CMU SEI

Ipek Ozkaya

James Ivers

Mena Kostial

Tapajit Day

Robert Edman

CMU ECE

Prof. Leo Sousa

Celina Cywinska

Vicky Xie

THANK YOU!