Program Analysis and AI for Legacy Code Modernisation

Dr. Alok Lele and Dr. Lina Ochoa
ASML, TU/e

ESI Symposium
Eindhoven, The Netherlands
October 7, 2025

Why are we here?

ASML In the News

2020 → 2023

BBC Sign in Home News Sport Reel Worklife Trav

NEWS

Home | Coronavirus | Video | World | UK | Business | Tech | Science | Stories | Entertainment & Arts | Health

Tech

Apple iPhone 12: The chip advance set to make smartphones smarter

By Leo Kelion
Technology desk editor

13 October 2020

When Apple unveils its new iPhones, expect it to make a big deal of the fact they're the first handsets in the world to be powered by a new type of chip.

The "five nanometre process" involved refers to the fact that the chip's transistors have been shrunk down - the tiny on-off switches are now only about 25 atoms wide - allowing billions more to be packed in.

Effectively it means more brain power.

Travel back just four years, and many industry insiders **doubted the advance**

That it has been, is in large part down to the ingenuity of a relatively obscure Dutch company - ASML.

But it's currently the only company making them. And they are still more cost-effective than alternative options, in part because of a low defect rate.

How ASML became Europe's most valuable tech firm

21 February

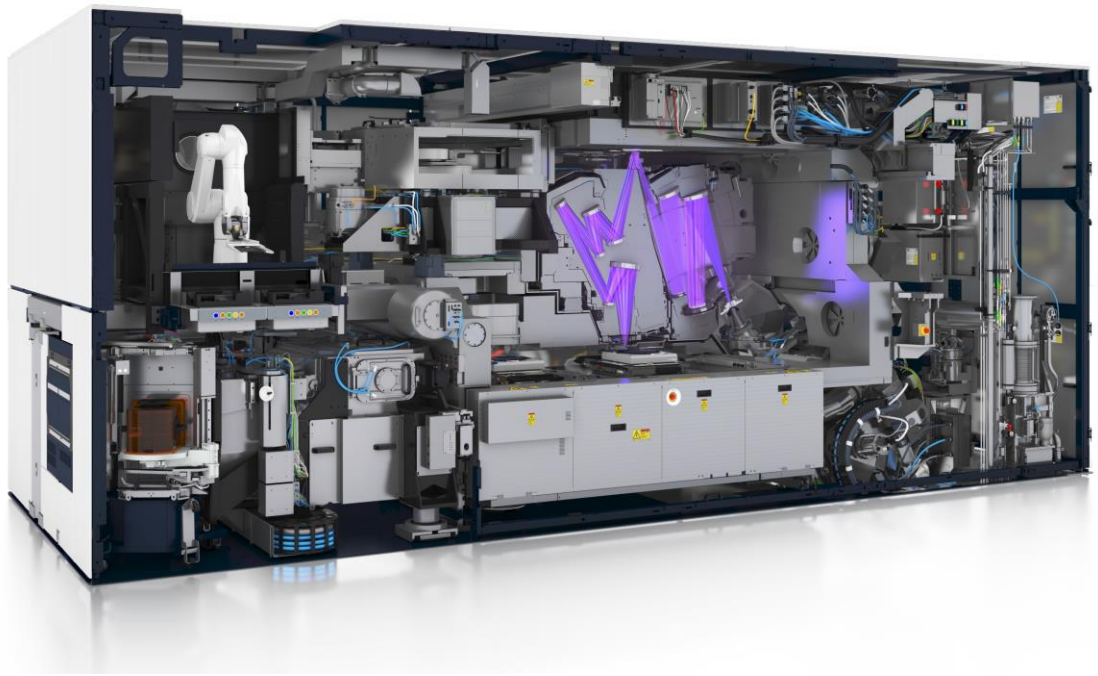
We take that back

We take that back

Complexity of a Twinscan (NXE) System

Has more than 100,000 individual parts, 3,000 cables, 40,000 bolts and 2 km of hosing

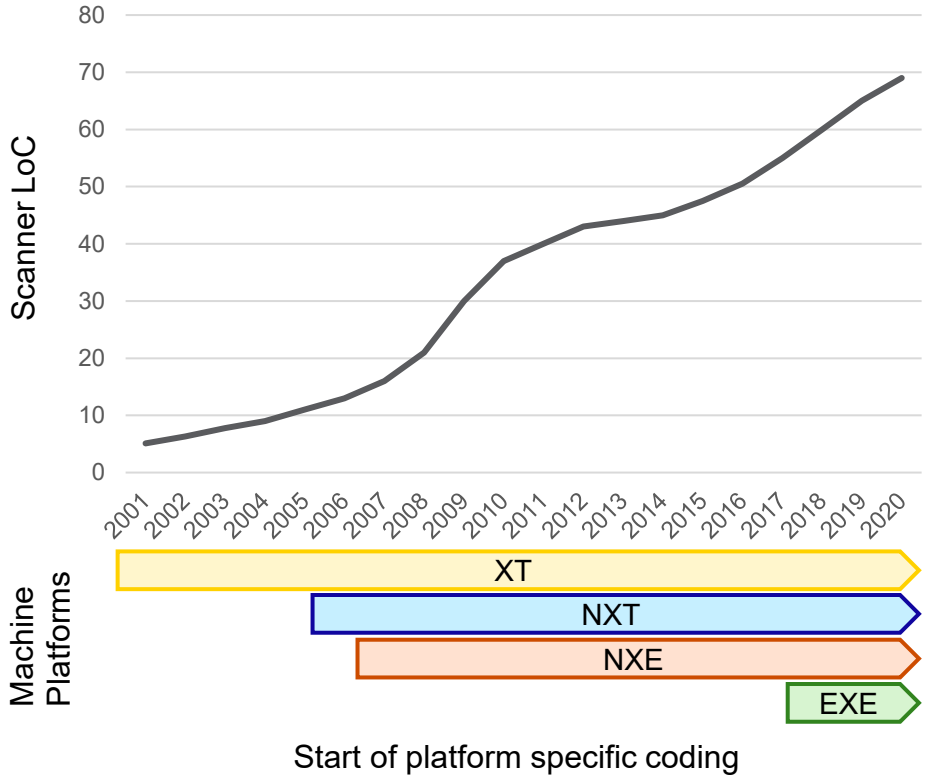
Took 20 years of sustained R&D to develop



Has about 1,500 sensors to capture imaging data

Generates about 4.5 TB of data per day

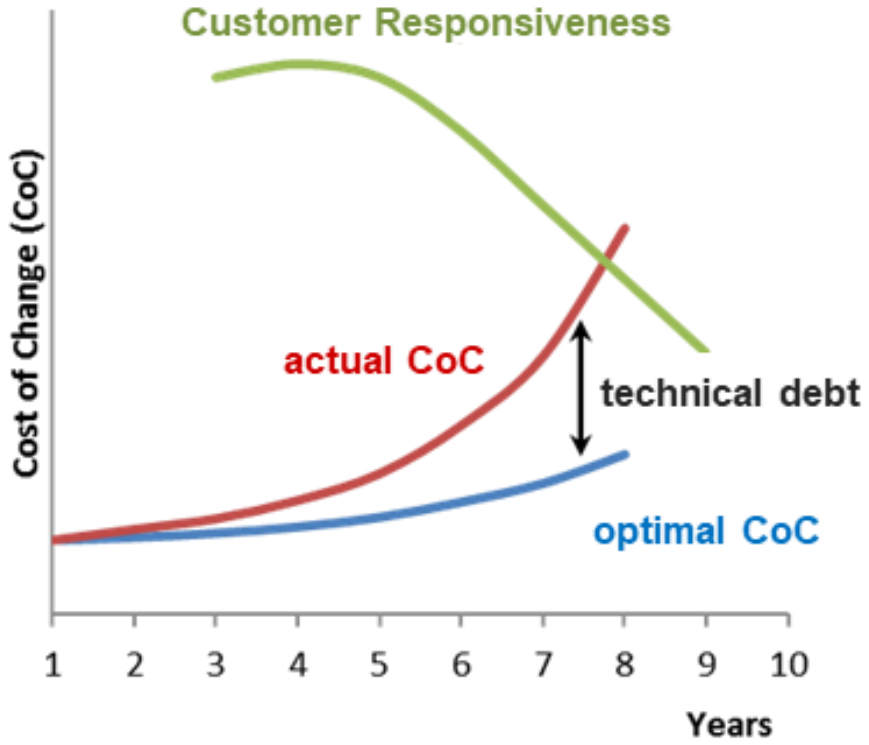
Increasing complexity in developing scanner software



Machine Platform	Machine Types	Variants	System Enhancements
EXE	EXE:5000	1	6
	EXE:5200	1	2
NXE	NXE:3350	1	4
	NXE:3400	2	24
	NXE:3600	1	13
	NXE:3800	1	8
NXT	NXT:1470	1	27
	NXT:19xx	7	295
	NXT:2xxx	4	108
	NXT:8xx	2	71
XT	XT:1000	2	92
	XT:1060	1	50
	XT:1250	2	64
	XT:14xx	6	217
	XT:4xx	10	328
	XT:8XX	13	1340

Theory vs. Reality of Code Modernization

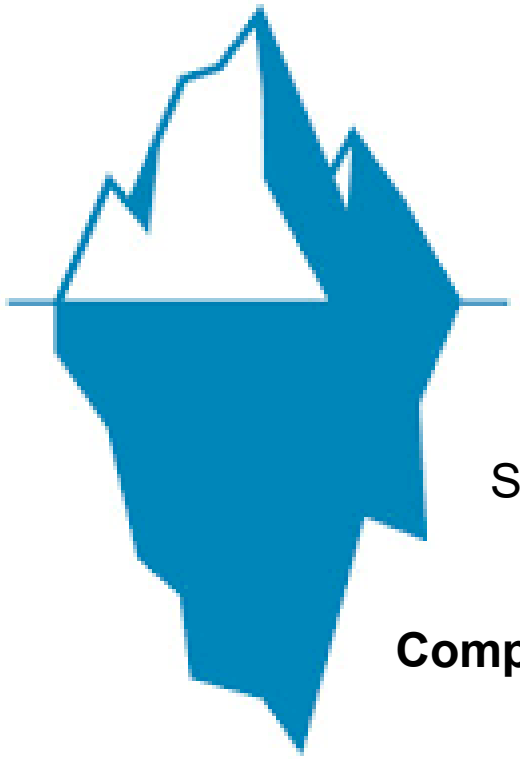
Technical Complexity Is Just One Aspect of Cost of Change



+ **Organization** complexity
+ **Development** complexity
+ **Business** complexity

Theory vs. Reality of Code Modernization

Technical Complexity Is Just One Aspect of Cost of Change



New business opportunities

Business **sustenance** (e.g., EoL migration, reorganization)

Software **maintenance**

Competence gap

The AI Belief

Generative AI demonstrates remarkable capability in understanding **context**, being **flexible** at applying complex **tasks**, and generating **comprehensible outputs** at scale.

But can they be trusted?

Early exploration

It’s a Team Efforts

ASML

Thijs Bressers

Hamza Meddeb

Erdem Alici

Edwin Roos

Capgemini

Jamel Fehri

Jacco Steegman

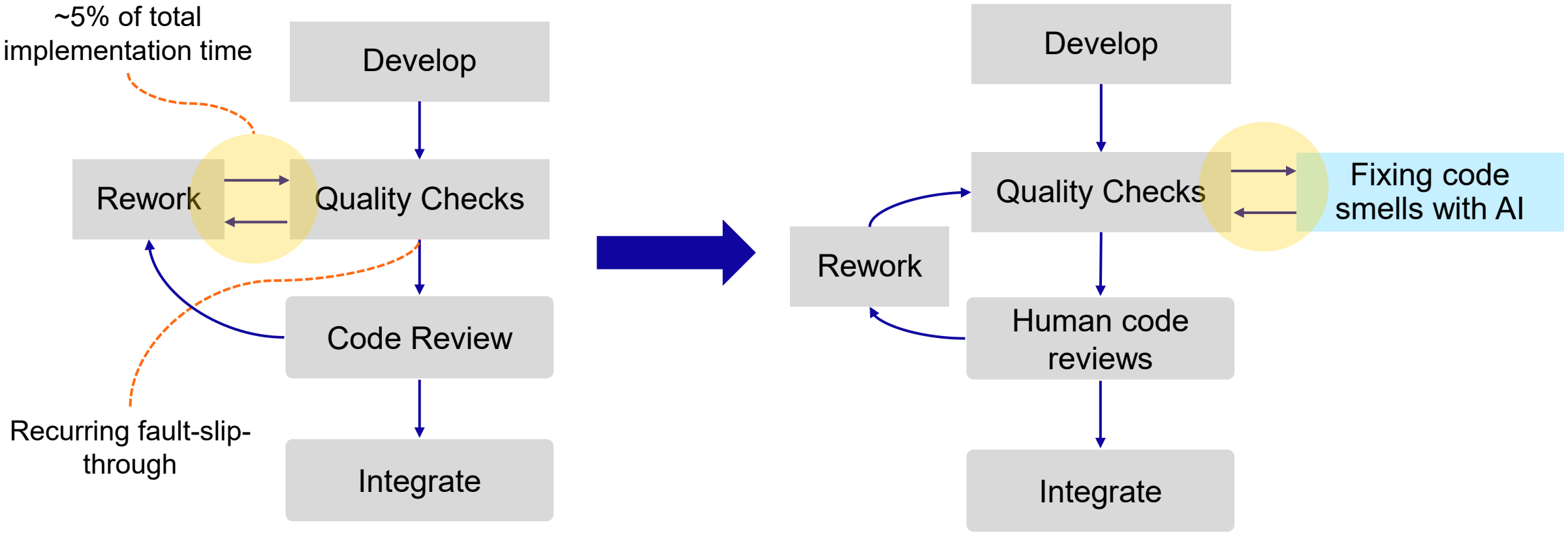
Leonardos Pastinakas

Sandeep Patil

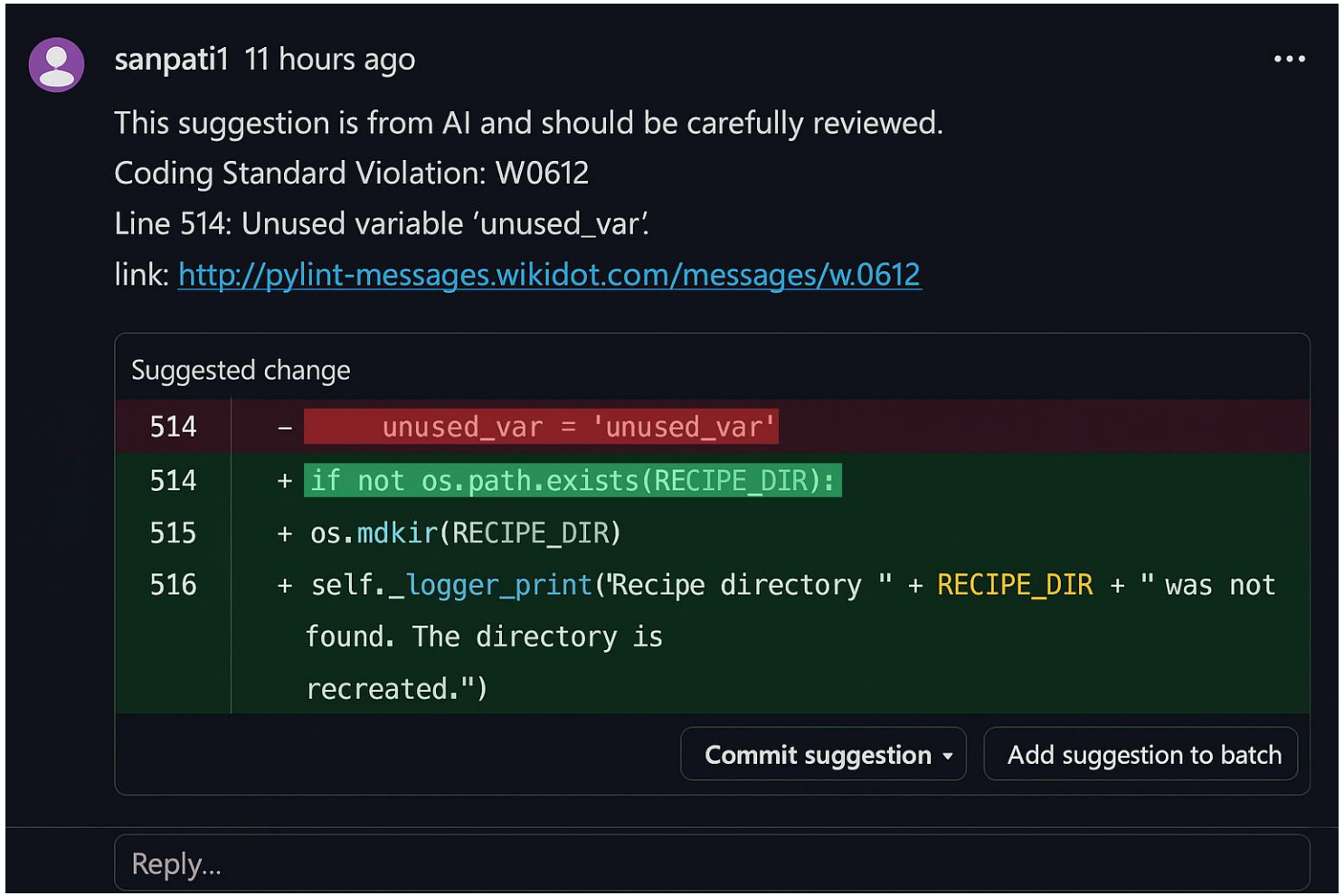
Mahsa Chitsaz

Early Exploration

Using AI to rework coding violations



Visualizing the output



Results

	Language	# Rules	# Violations	# Fixed violations	Success rate
TICS	Python	13	257	241	~94%
	C/C++	10	458	368	~80%
AMX	Python/C/C++	8	1,075	1,037	~96%

*__fixed__ = correct syntax + original violation fixed + no new violation introduced.

Takeaway 1: When AI works, it works well.

Results

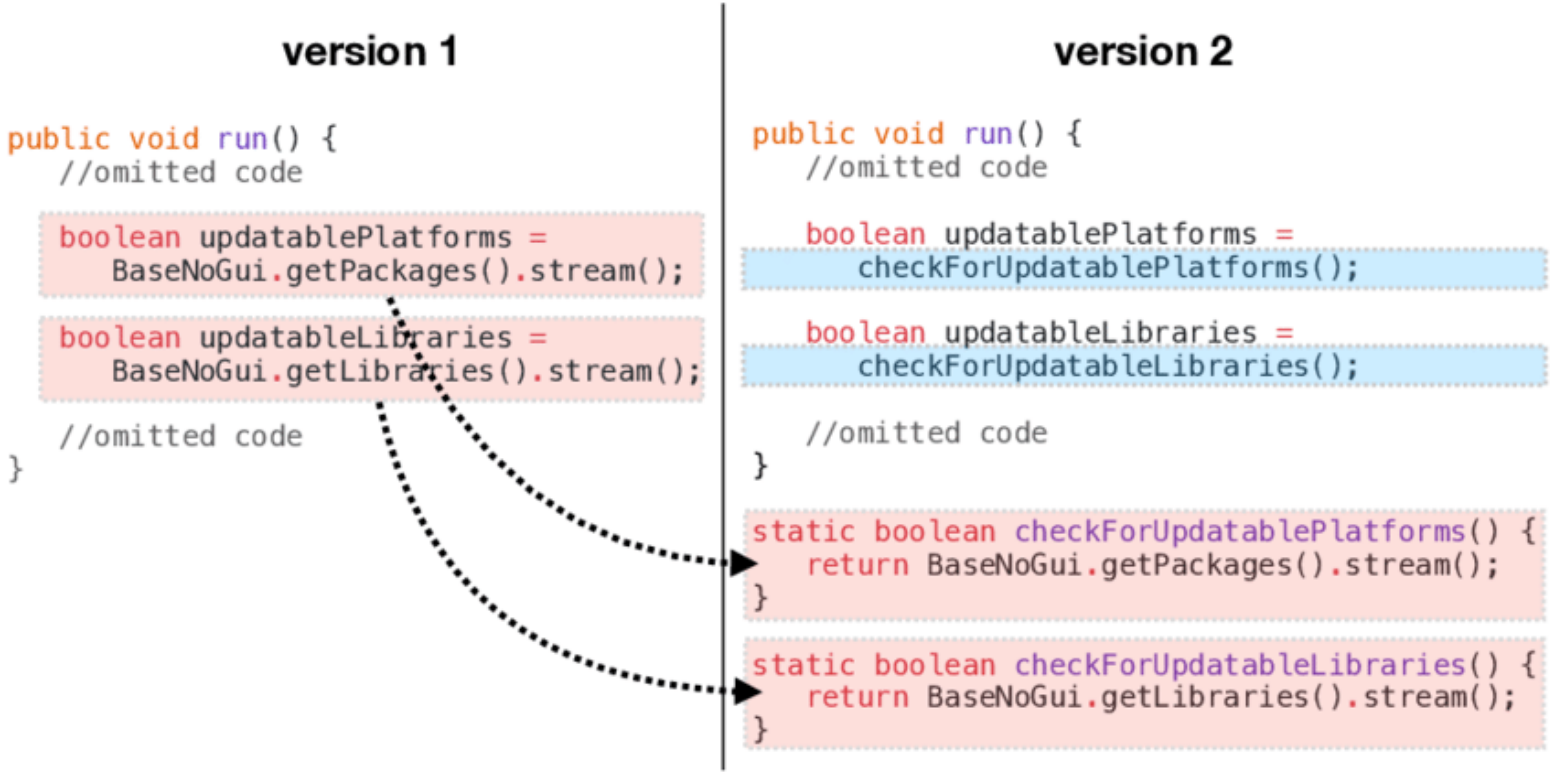
	# Total rules	# Rules addressed	# Violations	# Fixed violations	Success rate
TICS	81	23	2,869	609	~21%
AMX	18	8	2,354	1,037	~44%

*__fixed__ = correct syntax + original violation fixed + no new violation introduced.

Takeaway 2: AI does not work in many places (yet).

Where Things Start to Get Difficult

Example: Reducing Cyclomatic Complexity via Extract Method



Example taken from: *Hora, Andre & Robbes, Romain. (2020). Characteristics of method extractions in Java: a large scale empirical study. Empirical Software Engineering. 25. 10.1007/s10664-020-09809-8.*

Let’s do some research…

Research Team

Prof. Mark van den Brand

Dr. Joao Godinho Ribeiro

Dr. Jelle Piepenbrock

Prof. Mykola Pechenizkiy

Research Team

Static vs. LLM Approaches to Automatic Code Refactoring

Jeffrey Lint

Arturs Remesis

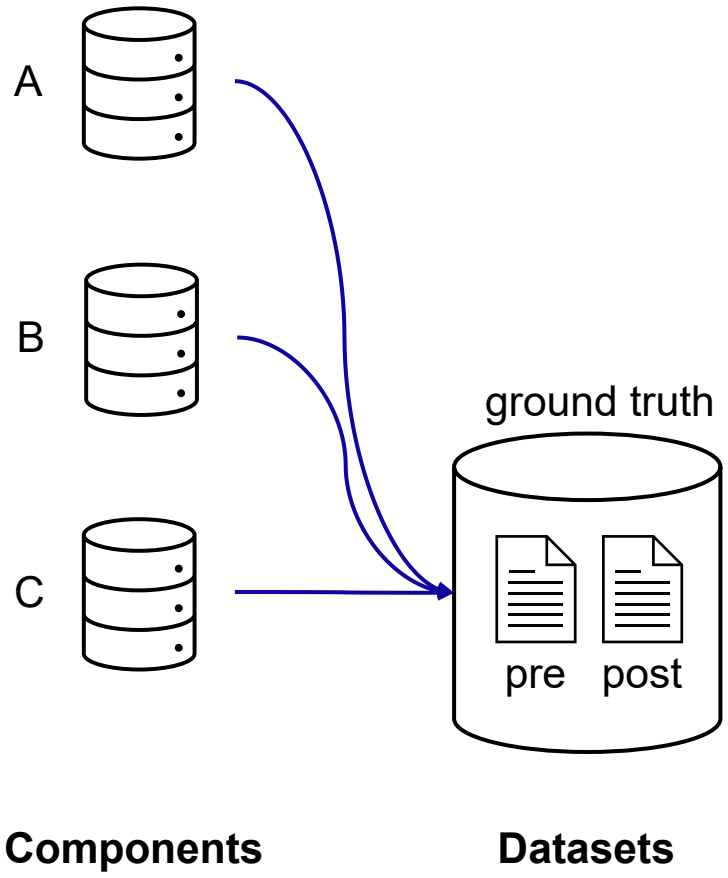
The Research Question

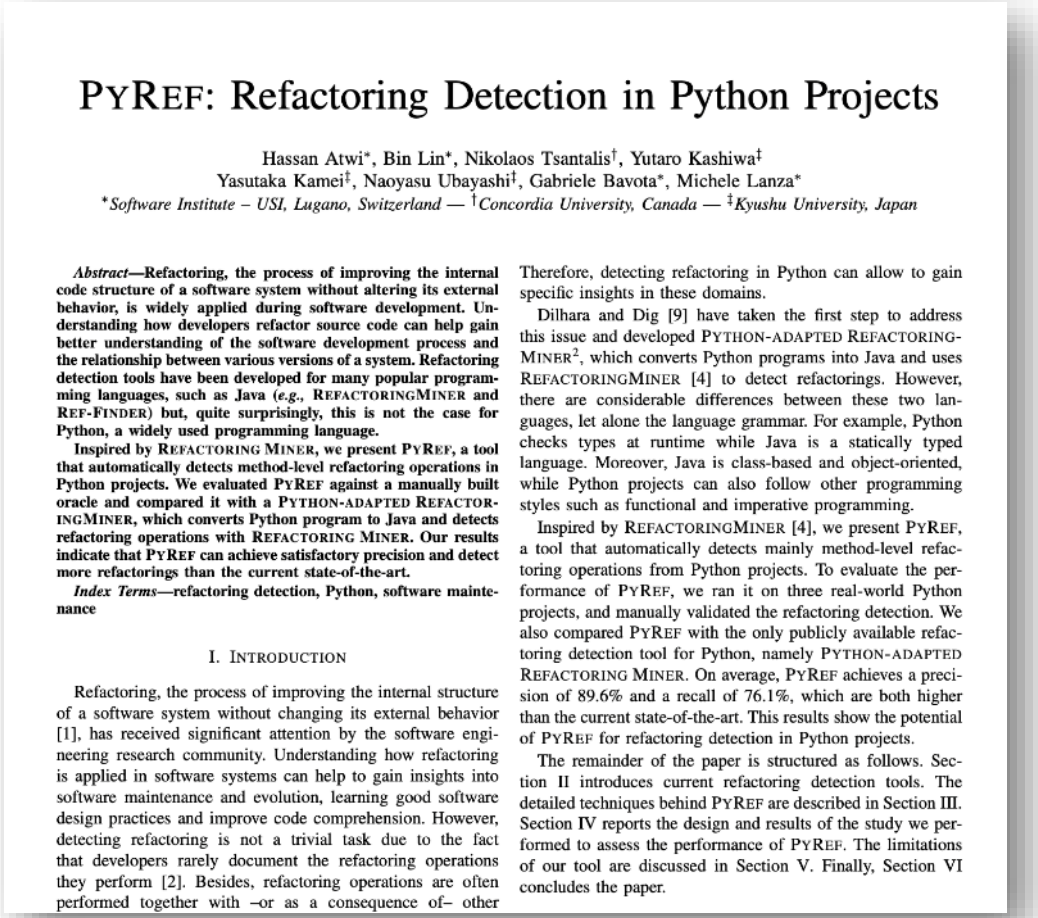
To what extent do **LLM approaches** compare to **static** ones when **extracting methods** in a large production codebase?

What did we do?

Study Design

Datasets





PyRef: Refactoring Detection in Python Projects

Hassan Atwi*, Bin Lin*, Nikolaos Tsantalis†, Yutaro Kashiwa‡
Yasutaka Kamei‡, Naoyasu Ubayashi‡, Gabriele Bavota*, Michele Lanza*
*Software Institute – USI, Lugano, Switzerland — †Concordia University, Canada — ‡Kyushu University, Japan

***Abstract*—Refactoring, the process of improving the internal code structure of a software system without altering its external behavior, is widely applied during software development. Understanding how developers refactor source code can help gain better understanding of the software development process and the relationship between various versions of a system. Refactoring detection tools have been developed for many popular programming languages, such as Java (*e.g.*, RefactoringMiner and Ref-Finder) but, quite surprisingly, this is not the case for Python, a widely used programming language.**

Inspired by Refactoring Miner, we present PyRef, a tool that automatically detects method-level refactoring operations in Python projects. We evaluated PyRef against a manually built oracle and compared it with a Python-adapted RefactoringMiner, which converts Python program to Java and detects refactoring operations with Refactoring Miner. Our results indicate that PyRef can achieve satisfactory precision and detect more refactorings than the current state-of-the-art.

***Index Terms*—refactoring detection, Python, software maintenance**

I. Introduction

Refactoring, the process of improving the internal structure of a software system without changing its external behavior [1], has received significant attention by the software engineering research community. Understanding how refactoring is applied in software systems can help to gain insights into software maintenance and evolution, learning good software design practices and improve code comprehension. However, detecting refactoring is not a trivial task due to the fact that developers rarely document the refactoring operations they perform [2]. Besides, refactoring operations are often performed together with –or as a consequence of– other Therefore, detecting refactoring in Python can allow to gain specific insights in these domains.

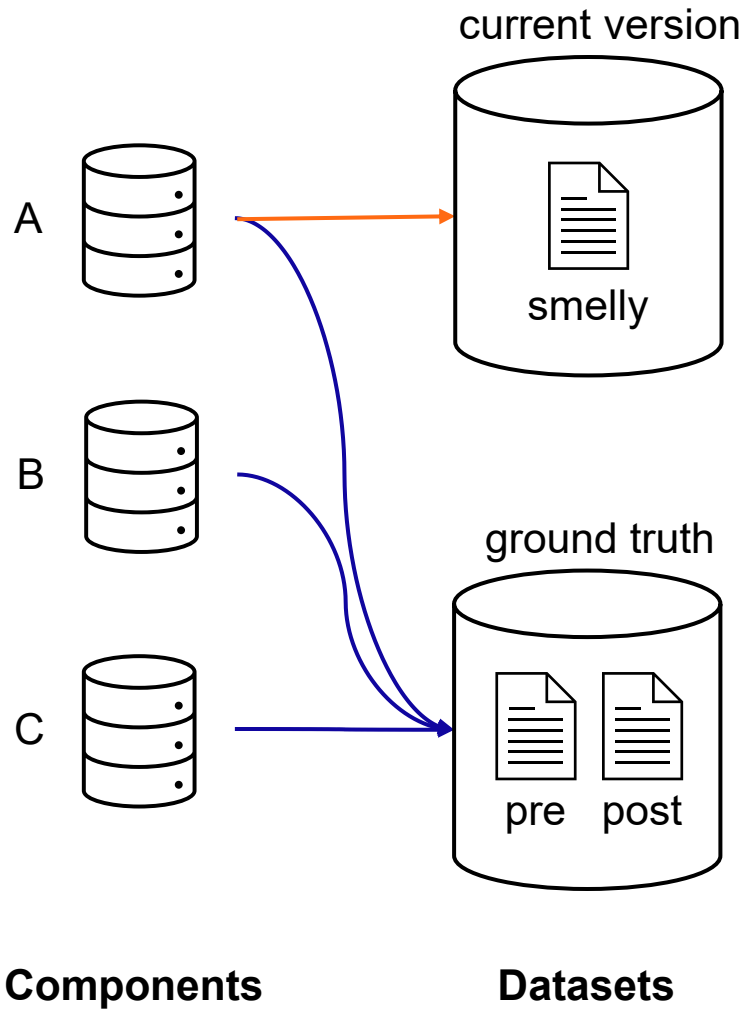
Dilhara and Dig [9] have taken the first step to address this issue and developed Python-adapted RefactoringMiner[2], which converts Python programs into Java and uses RefactoringMiner [4] to detect refactorings. However, there are considerable differences between these two languages, let alone the language grammar. For example, Python checks types at runtime while Java is a statically typed language. Moreover, Java is class-based and object-oriented, while Python projects can also follow other programming styles such as functional and imperative programming.

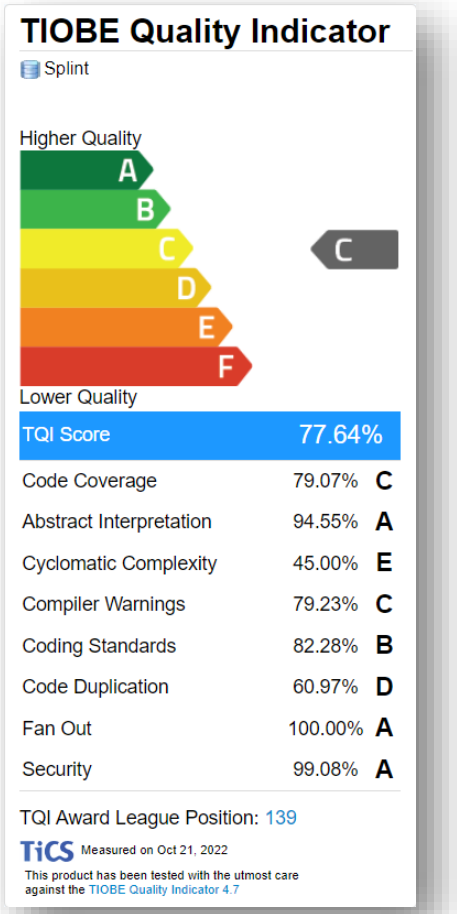
Inspired by RefactoringMiner [4], we present PyRef, a tool that automatically detects mainly method-level refactoring operations from Python projects. To evaluate the performance of PyRef, we ran it on three real-world Python projects, and manually validated the refactoring detection. We also compared PyRef with the only publicly available refactoring detection tool for Python, namely Python-adapted Refactoring Miner. On average, PyRef achieves a precision of 89.6% and a recall of 76.1%, which are both higher than the current state-of-the-art. This results show the potential of PyRef for refactoring detection in Python projects.

The remainder of the paper is structured as follows. Section II introduces current refactoring detection tools. The detailed techniques behind PyRef are described in Section III. Section IV reports the design and results of the study we performed to assess the performance of PyRef. The limitations of our tool are discussed in Section V. Finally, Section VI concludes the paper.

Study Design

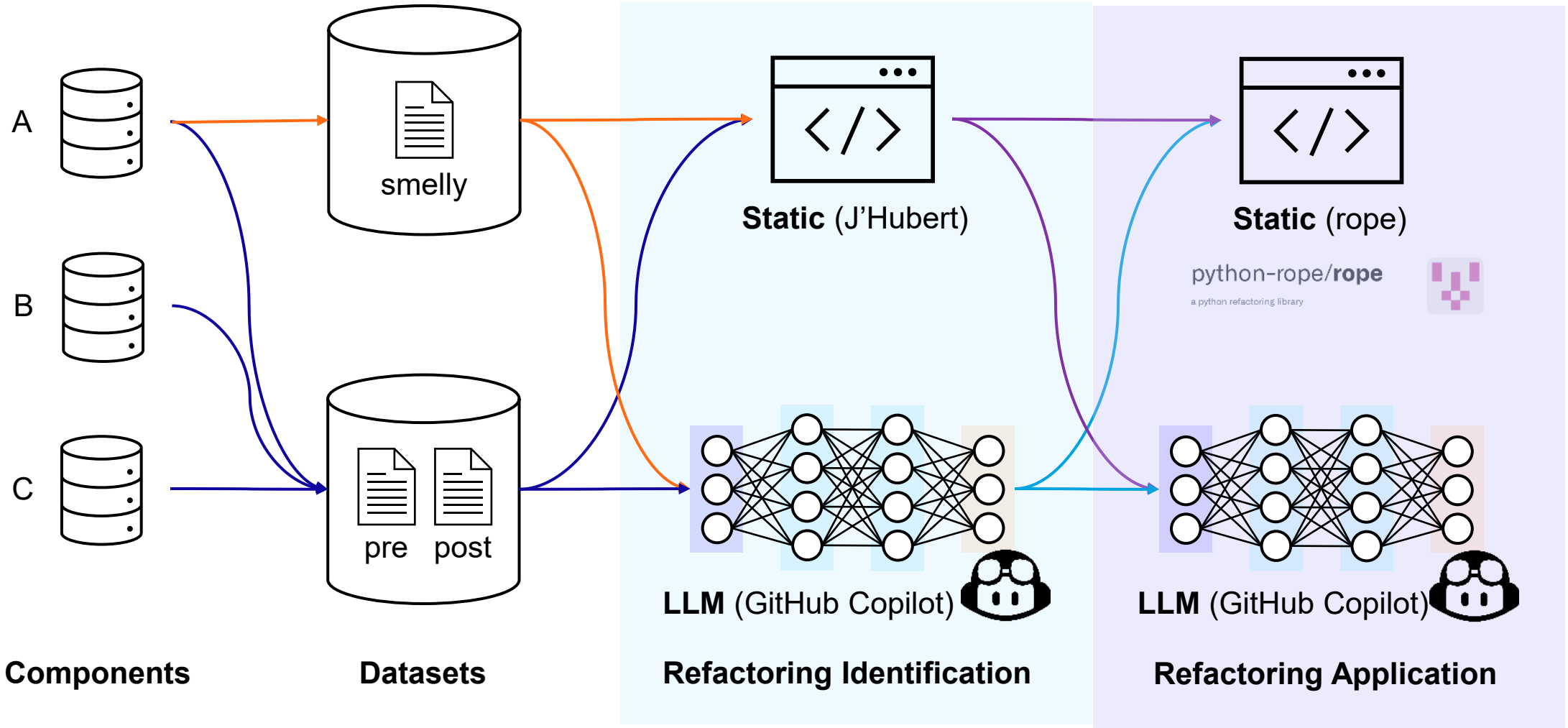
Datasets





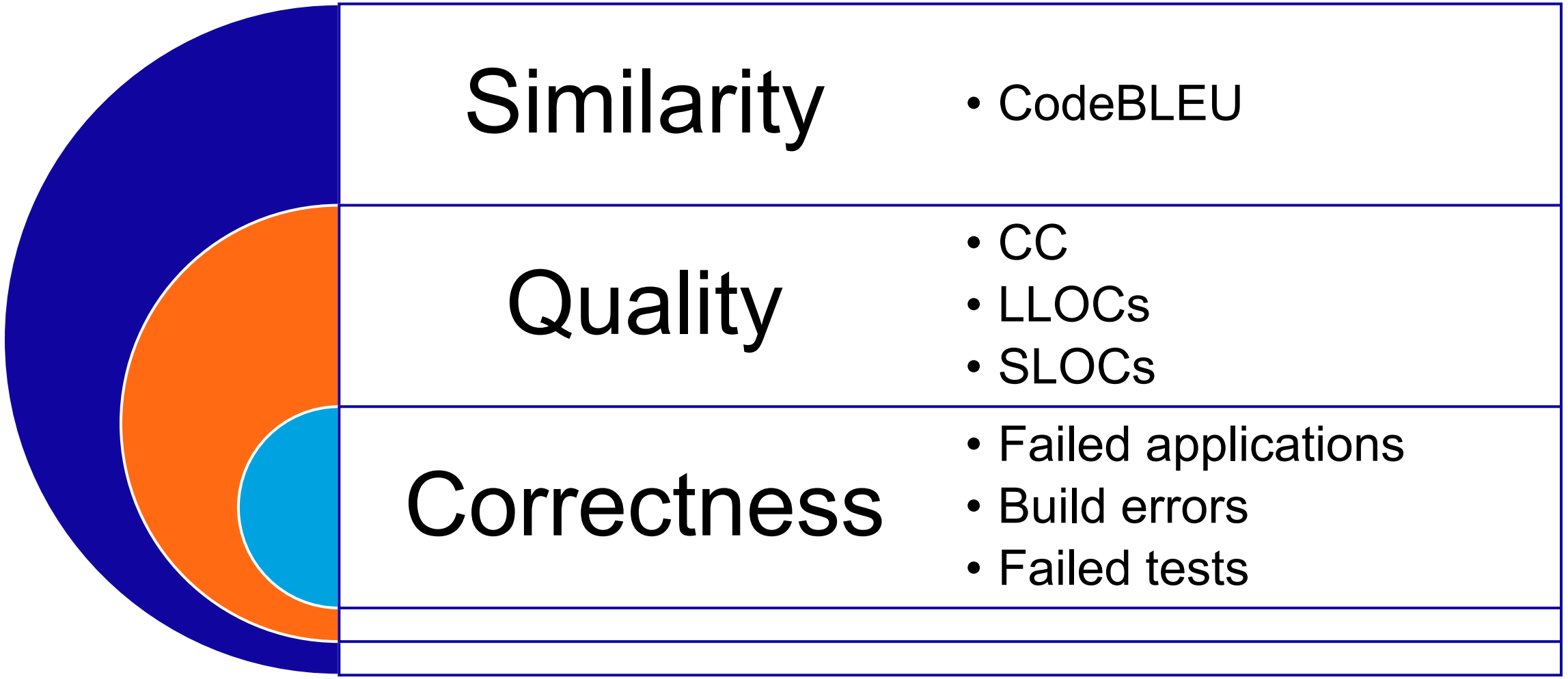
Study Design

Approaches



Study Design

Evaluation



What results we got?

Correctness

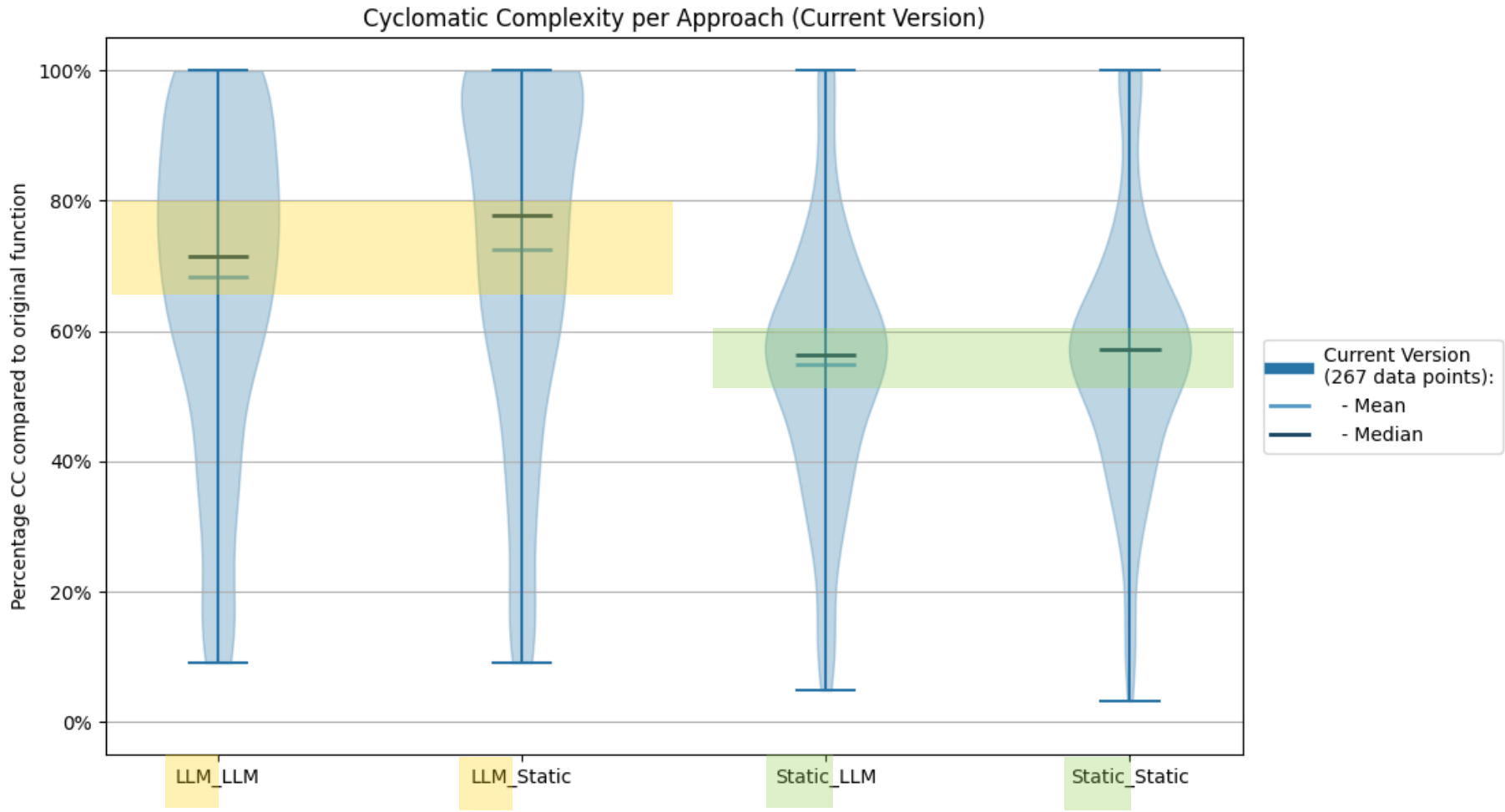
Current Version Component A

Identification	Application	Failed applications	Build errors	Executed tests	Failed test
LLM	LLM	1	**7**	1,061	**113 (10.7%)**
LLM	Static	**40**	0	955	51 (5.3%)
Static	LLM	1	**3**	1,061	**157 (14.8%)**
Static	Static	**8**	0	1,010	65 (6.4%)
Worst case		**-**	**-**	**1,061**	**420 (39.6%)**

Takeaway 1: LLMs are more flexible during application, while static approaches account for syntactic and semantic correctness.

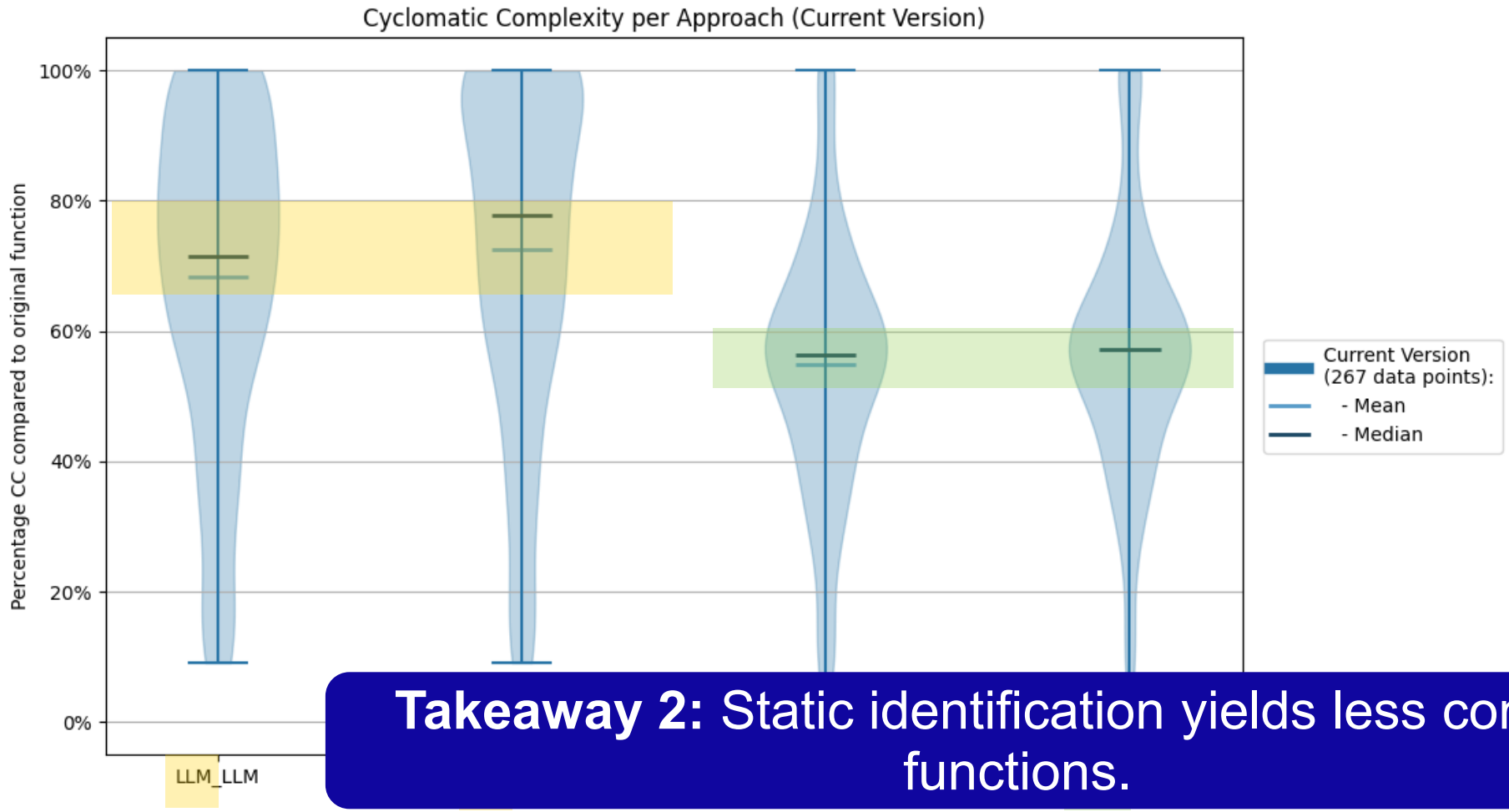
Quality

Cyclomatic Complexity (CC) in Current Version Component A



Quality

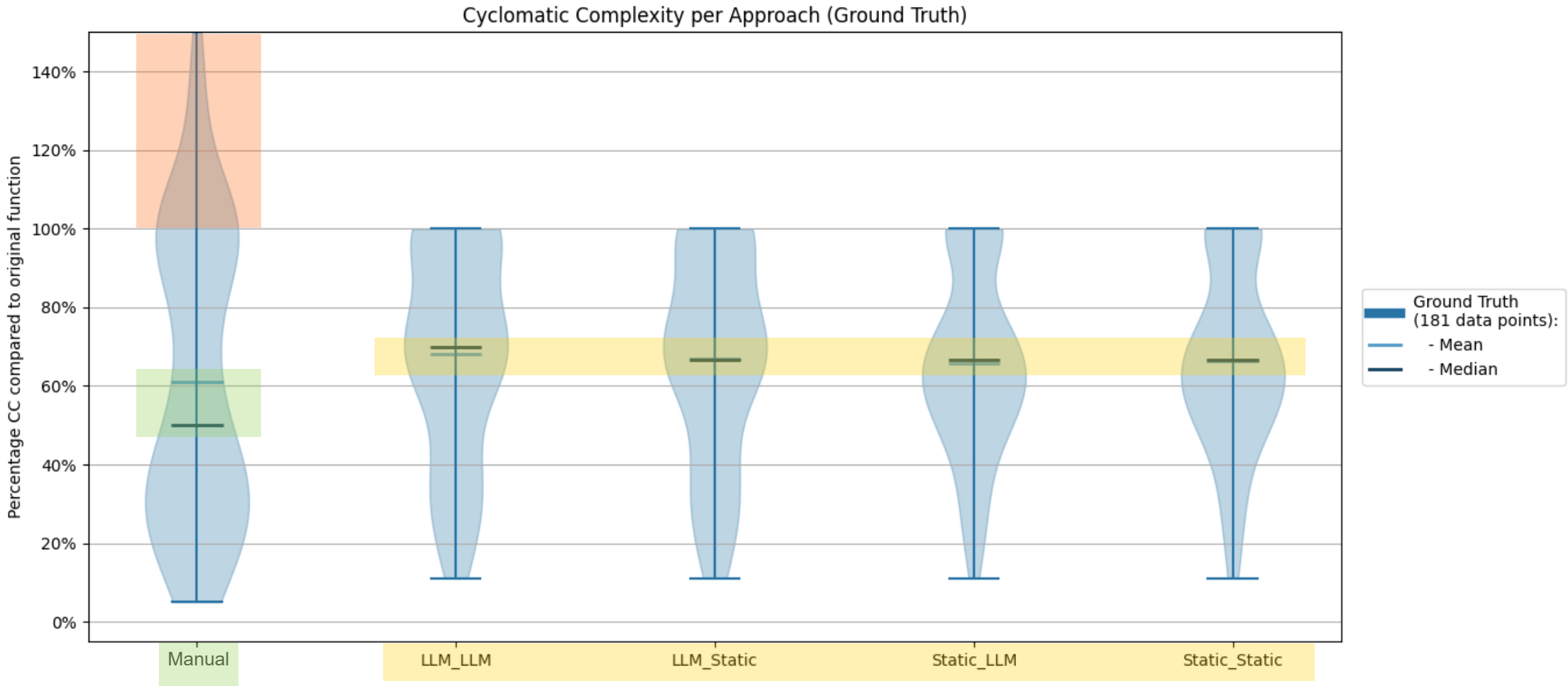
Cyclomatic Complexity (CC) in Current Version Component A



Takeaway 2: Static identification yields less complex functions.

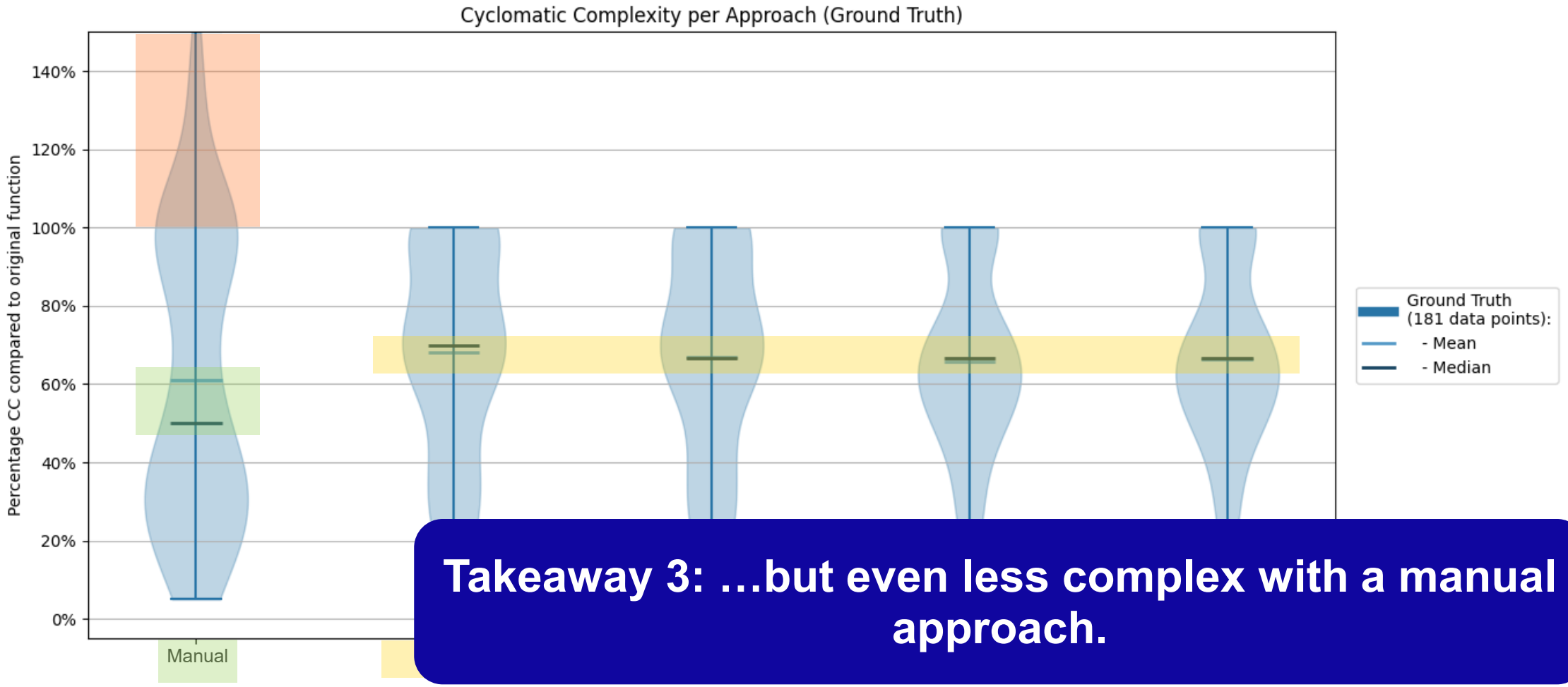
Quality

Cyclomatic Complexity (CC) in Ground Truth



Quality

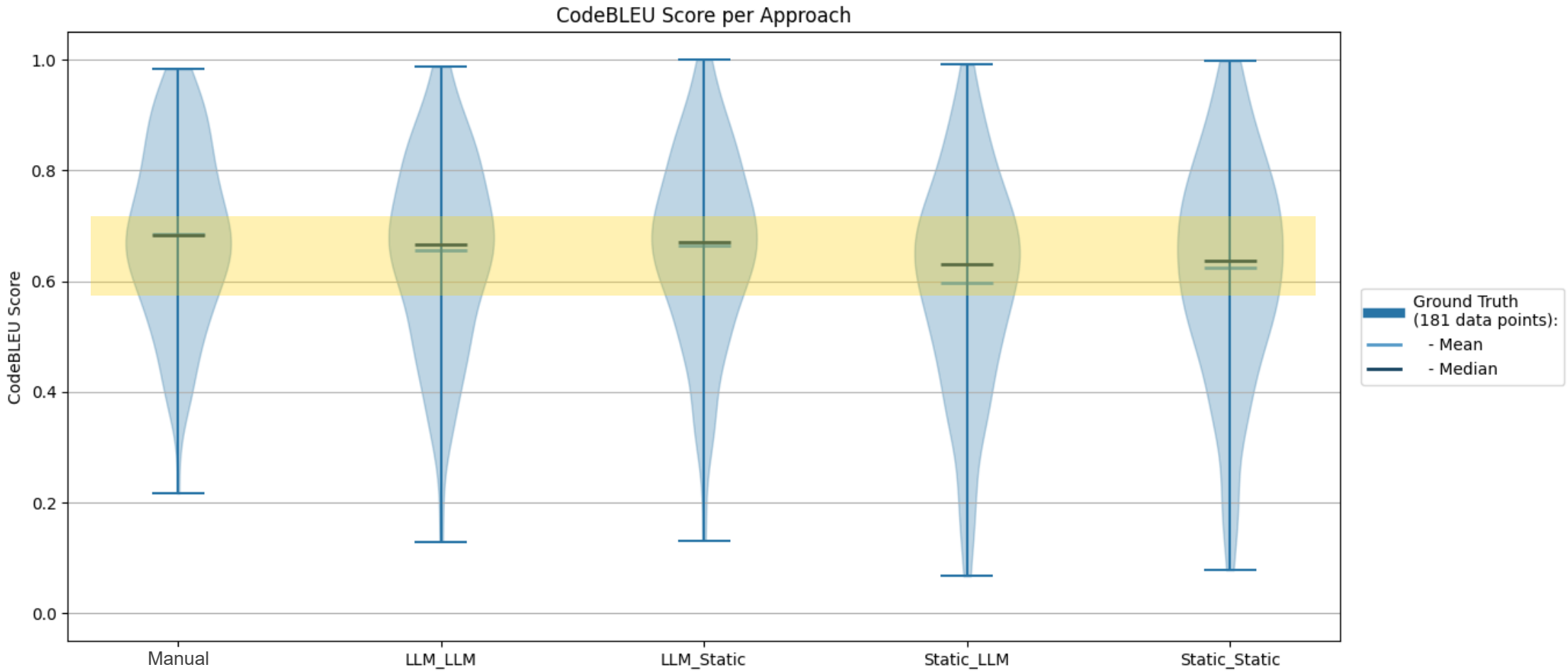
Cyclomatic Complexity (CC) in Ground Truth



Takeaway 3: …but even less complex with a manual approach.

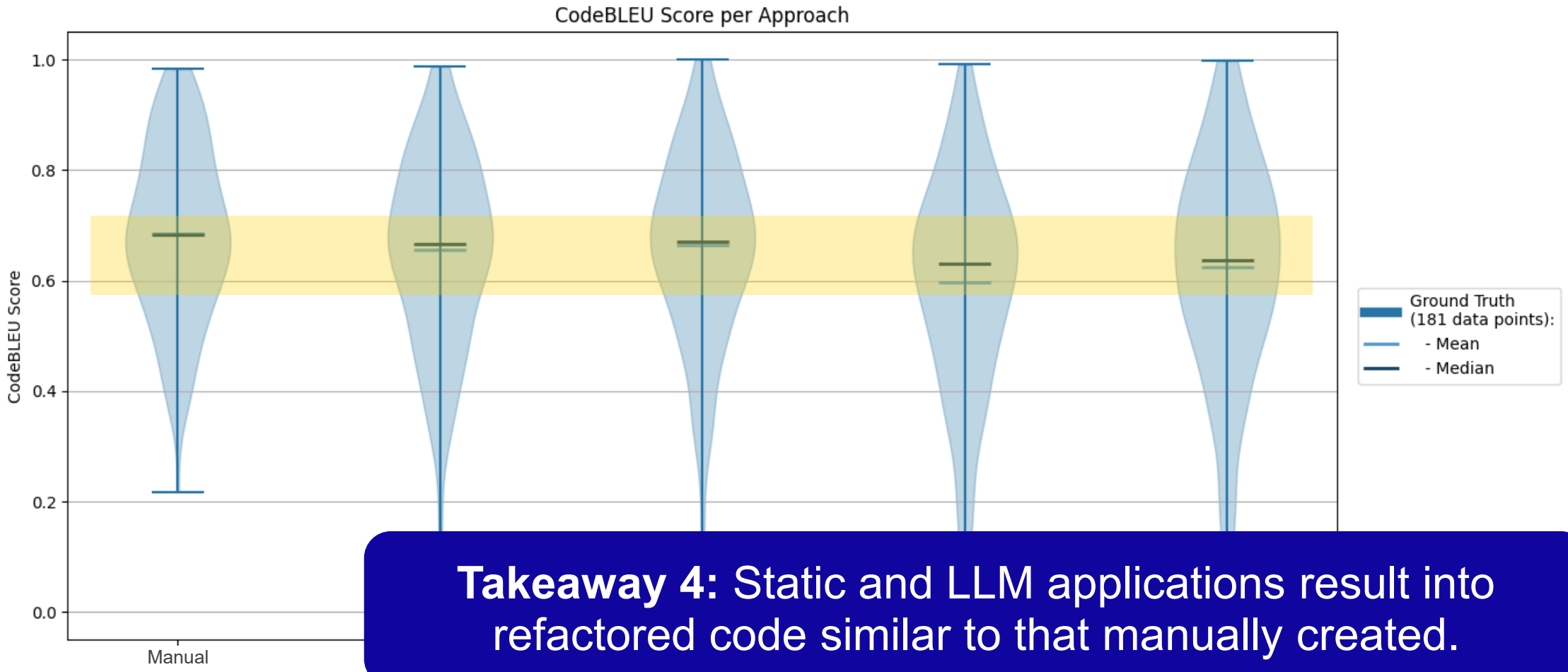
Similarity

CodeBLEU in Ground Truth



Similarity

CodeBLEU in Ground Truth



Takeaway 4: Static and LLM applications result into refactored code similar to that manually created.

What did we learn?

The AI Reality

Generative AI demonstrates notorious capability in understanding **context**, being **flexible** at applying complex **tasks**, and generating **comprehensible outputs** at scale.

Observation: Unconstrained AI generation produces inconsistent outputs, introduces bugs, and lacks the rigorous guarantees required for production systems.

Summary

- **Maintaining** large scale legacy software systems is **increasingly complex**.
- The key appeal of LLMs is their ability to produce **human comprehensible output**.
- Establishing **trustworthiness** of probabilistic solution requires further **innovation** of the technology but also in its application.

ASML

Program Analysis and AI for Legacy Code Modernisation

Thanks!

Dr. Alok Lele
alok.lele@asml.com

Dr. Lina Ochoa
l.m.ochoa.venegas@tue.nl

Eindhoven, The Netherlands

Open to Work!

Jeffrey Lint
j.a.h.lint@student.tue.nl