Maintainability Definition, Scoping, and Measurement for Machine Learning Systems

Khan Mohammad Habibullah, Juan Garcia Diaz, Gregory Gay, and Jennifer Horkoff

Chalmers University of Technology and University of Gothenburg, Sweden
khan.mohammad.habibullah@gu.se, gusgarjub@student.gu.se,
greg@greggay.com, jennifer.horkoff@gu.se

Abstract. Machine learning (ML) systems are increasingly being applied to critical tasks. Like all systems, ML systems must satisfy maintainability requirements. However, assessing and ensuring maintainability is complicated by the dynamic, heterogeneous, and interconnected components within ML systems—a mixture of structured code, scripting, data, and models. We propose that ensuring maintainability of ML systems requires definition, specification, and measurements that can be scoped across one or more of these components, in addition to the system as a whole. To that end, we propose a component-based breakdown of ML systems, a modified definition of maintainability, and examples of modified modularity measurements. We use these to characterize the modularity of real-world ML systems. Our contributions offer a starting point for future research on maintainability for ML systems.

Keywords: Machine Learning · Requirements Engineering · Software Quality · Non-Functional Requirements · Maintainability

1 Introduction

Machine Learning (ML) systems blend code-based functionality with functionality based on supervised, unsupervised, or reinforcement learning[1] to perform critical prediction tasks in domains including healthcare and automotive. The number and complexity of ML systems is increasing. However, due to issues related to, e.g., non-deterministic behavior, explainability, and bias, ML systems can be more difficult to specify, implement, and test than traditional systems.

Additionally, ML system behavior depends not only on traditional structured code[2], such systems also contain models, data, and scripting[3] used to train models. For example, in an advanced driver assistance system, an ML model may perform object detection, while traditional code interacts with sensors and actuators based on the model's predictions. These components integrate and interact

[1] We focus, in this work, on supervised learning.
[2] Code organized into dedicated classes, services, or other organizational structures that cooperates with other structured code to deliver functionality.
[3] Unstructured code in standalone files, not using a defined organizational structure.

in potentially complex ways, have complex dependencies, and evolve in different ways, increasing the complexity of development of the system as a whole.

As with any system, ML systems must satisfy quality requirements—also known as non-functional requirements (NFRs)—for successful development, implementation, and use [12]. Past research has reported challenges specific to the definition, scoping, and measurement of NFRs for ML systems, including domain dependence, interdependence among system components, lack of measurement techniques, and lack of awareness among practitioners and users [10–12, 14]. We propose that addressing these challenges requires clear definition of NFRs, scoping over different components within an ML system, and measurement of NFRs over those scopes [10, 12, 22]. Clear definition, scope, and measurements can reduce ambiguity and miscommunications among stakeholders, and enable assessment of NFR satisfaction [10, 13].

ISO/IEC 25059 specifies eight high-level quality requirements for ML systems, including maintainability [16]—the degree of effectiveness and efficiency with which a system can be modified, corrected, or adapted to changes in its environments and requirements. Maintainability comprises five sub-characteristics, including modularity, reusability, modifiability, analyzability, and testability [16]. In particular, in this study, we focus on modularity—the degree to which a change to one discrete system component affects others [16].

For traditional systems, studies have shown an exponential relationship between decreasing maintainability and increasing development cost [6]. Although maintainability as an NFR is well established for traditional systems, its definition and interpretation when specifying ML systems are less clear [12]. ML systems introduce unique maintainability challenges, impacting not just system development, but also data and model engineering [25].

In this article, we focus on maintainability—specifically, modularity—as an example of how NFR definition and measurement can be adapted to ML systems. We introduce a breakdown of ML systems into typical components that could be used to scope requirements and measurements. Comparatively, ISO/IEC 23050:2022 provides a breakdown of elements of ML systems, including tasks, models, data, and software tools/techniques [15]. We propose that an architectural breakdown for ML systems should place more emphasis on software components present in complex ML system, including structured and unstructured code, and code which is not directly related to ML. We introduce a modified definition of maintainability and modularity metrics that take into account both structured and unstructured code, as well as dependencies on models and data.

We examine how our breakdown and modified metrics can characterize the modularity of a set of realistic ML systems from GitHub. We found that our breakdown is applicable, and that the modified metrics capture dependencies missed by traditional metrics—although typical code dependencies remain more common than ML-related dependencies. We also found that training pipelines and ML interfacing components tend to have higher-than-average coupling, and that data acquisition and ML interfacing components tend to have lower-than-average cohesion. Our contributions and observations offer a starting point for

research on the maintainability of ML systems, and our proposed breakdown and general approach can be applied to other NFRs for ML systems.

2 Background and Related Work

In this section we present existing common metrics for modularity as part of maintainability, and related work on maintainability challenges for ML systems.

2.1 Modularity Measurement

Modularity metrics are often based on cohesion and coupling [27]. Coupling refers to the degree of interdependence between system components while cohesion refers to the degree to which grouped sub-components—e.g., functions collected within a single class—belong together. *High* coupling indicates low modularity, as components are more interdependent. *Low* cohesion indicates low modularity, as components lack focused responsibility.

Coupling Measurement: One of the most common coupling metrics is **"Coupling Between Object Classes" (CBO)** [9]. CBO for a class[4] is:

$$CBO(C) = \sum_{i=1}^{c} I(C, C_i) \tag{1}$$

where, C is the class-under-assessment, c is the total number of classes in the system, and $I(C, C_i) = 1$ if class C is coupled to class C_i and 0 otherwise.

The CBO value for a class is, essentially, the number of other classes it is coupled with. Two classes are coupled when one invokes methods or references instance variables defined in the other [5]. The CBO of the entire system, then, is the average CBO value. In past research, a CBO of 0–5 is considered low coupling and > 9 is considered high [24].

Cohesion Measurement: One of the most common cohesion metrics is **Loose Class Cohesion (LCC)** [7]. This metric considers direct connections between methods in a class, when two methods read or modify a common variable, and indirect connections, when two methods are both connected to a third that shares variable with these two methods[5]. LCC measures the ratio of connected method pairs to the total number of method pairs:

$$LCC(C) = \frac{M_s}{\frac{n(n-1)}{2}} \tag{2}$$

where, C is the class-under-assessment, M_s is the number of method pairs that are connected (directly or indirectly), and n is the total number of methods in the class. LCC ranges from 0–1, where 0 indicates low and 1 indicates high cohesion. When $n = 0$—i.e., a class contains no methods—LCC is undefined. Again, the average LCC indicates the overall cohesion of the system.

[4] CBO and LCC can be straightforwardly adapted to other code structures, or—for scripting or imperative languages—to code files.

[5] Consider, for example, three methods (A, B, C). A and B both modify variable x, while B and C both modify variable y. A and B and B and C are directly connected. As a consequence, A and C are indirectly connected, via B.

2.2 Maintainability Challenges for ML Systems

Existing maintainability research for traditional systems primarily considers structured code. ML systems contain code as well as additional components that strongly affect system behavior [10, 11]. Many additional challenges are introduced by the inclusion of diverse component types, as well as dependencies between these component types. Shivashankar et al. identified interdependent maintainability challenges at each stage in the ML workflow, including data engineering (e.g., ensuring quality and consistency of data over time, data dependence), model engineering (e.g., model drift, hyperparameter tuning), and deployment and operation (e.g., scaling to increasing data and user demands) [25].

Dataset maintenance differs from code maintenance and is focused on ongoing quality assurance and data inclusion and labeling [20, 25]. Models, too, differ from code in their maintenance process, involving continuous monitoring and retraining [25]. Like code, models exhibit dynamic "behavior" when performing prediction tasks. However, understanding this behavior can be difficult, as it is difficult to infer the calculations that lead to an individual prediction [19]. This increases the difficulty of maintaining both models and the overall ML system. Models are also generally task-specific, leading to poor reusability.

The relationships between structured code components are generally well-understood. In contrast, the relationships between ML system components are not yet clear. Dependencies between structured and unstructured code, data sources, external APIs, and models lead to additional maintainability challenges such as complex inter-component communication, integration issues, version control complexity, and complex resource management [18].

Data, models, training pipelines, model testing and validation components, and models-dependent functionality are highly coupled [8]. This makes system modification more complex and error-prone. Many ML projects fail in the initial phase because setting up infrastructure for deployment is more complex than for traditional systems due to the need for integration and management of training pipelines, code, and data monitoring [25]. Modularity challenges have also been identified in modular neural networks—where a neural network is constructed from multiple independent neural networks [1].

With regard to other sub-characteristics, modifiability is complicated by a lack of design specifications—specifically affecting forecasting of time required for the training process and assessments of component redundancy [17]. Testability of ML systems has also received significant attention. Identified challenges include lack of explainability, non-determinism, large input spaces, sensitivity to training data, testing for rare or special cases, and specifying test oracles [19,21].

Overall, there is a growing awareness of maintainability challenges unique to ML systems—such as heterogeneous components, implicit dependencies, and evolving data and model behavior. Although various studies reported the complexity of maintaining ML systems, they lack systematic methods to manage and measure maintenance across ML components. Current approaches are either only code-centric or address ML-specific issues separately. Towards closing this gap, we propose a component-aware definition, breakdown, and modularity metrics that can characterize maintainability of complex ML systems.

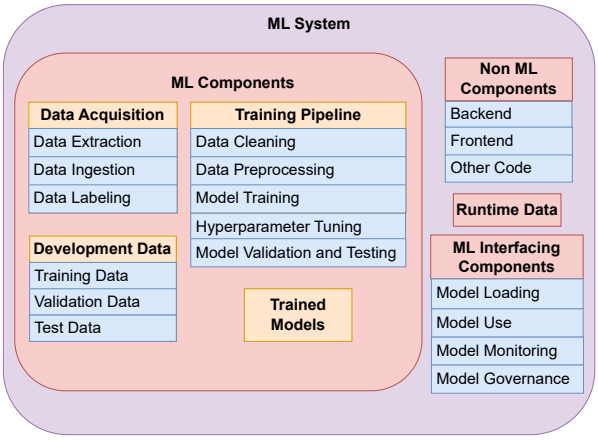


Fig. 1: ML system components, separated into groups.

3 Maintainability for ML Systems

In order to address the described challenges, we propose a breakdown of ML systems into distinct components that could affect maintainability[6]. We then propose a definition of maintainability and cohesion and coupling metrics that take into account this breakdown.

3.1 ML System Scoping

Like many complex systems, an ML system is not a monolith, but many components that work together [22]. Those components have different formats, information, and behavior. NFRs can be defined, specified, and measured for the whole system or across different scopes—individual or subsets of components [13]. How NFR types are defined, how individual NFRs are specified, or how attainment is measured, can vary between scopes. For example, the performance of a model, of the training pipeline, and of the system as a whole may be assessed differently.

In Fig. 1, we present typical components of a ML system based on supervised learning. As a basis for scoping NFRs, we propose dividing ML systems into (1) "ML components"—responsible for supporting and performing ML operations—(2) components that interface with ML, (3), components unrelated to ML, and (4) data ingested at runtime. These three groups can be broken down further. The ML components include models that perform predictions, as well as the data and training pipelines used to produce models. ML-interfacing components include code that loads, invokes, and monitors models. As compared to ISO/IEC 23050:2022 ([15]), our breakdown, covers software-related elements in more granularity, including Non-ML Components and ML-interfacing components.

When scoping NFRs, the "type" of components may influence specification and measurement. In Fig. 2, we also break down these components into four types of information represented, including data—input to either the training process or at runtime—trained models, structured code, and scripting.

[6] This breakdown builds on previous work [10,11], but has been substantially modified.

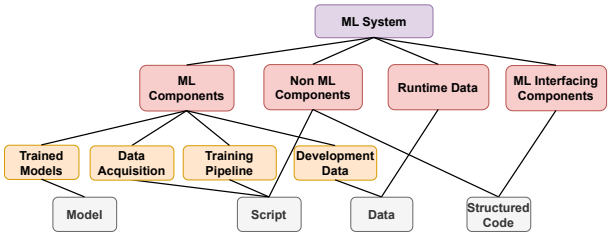


Fig. 2: Components, grouped by the type of information represented.

3.2 Maintainability Definition for ML Systems

We propose a definition of maintainability for ML systems, adapting the ISO/IEC 25059 definition of maintainability [16] and considering our breakdown in Fig. 1:

> Maintainability refers to the degree of effectiveness and efficiency with which modifications, including corrections, adaptations, or improvements, can be applied to the ML system as a whole, or to scopes (individual components or subsets of components) within the system.

Both individual components and subsets of components affect system maintainability in ways that are not fully understood. Textually, this is a simple change to the definition. However, the intention of this change is that the core concept of maintainability—as well as its five sub-characteristics—must be reconsidered in a way that incorporates the ability to scope components of a ML system, as different scopes may have unique maintenance needs and challenges.

For example, reusability must extend beyond structured code to encompass reusability of models, training pipelines, or data. How "reusable" these components are differs from how reusability of code is defined. The modifiability of a model depends on how difficult it is to retrain that model using a new or modified dataset, which in turn may depend on the modifiability of the code that performs the training process or on the sensitivity of the parameter tuning. Similarly, the analyzability or testability of a model are strongly dependent on the model's explainability. Future work on this topic should take a scoping-aware viewpoint and explore these effects.

3.3 Modified Modularity Metrics for Code

Existing modularity metrics only capture dependencies between structured code components. However, in an ML system, a code segment may exist in either a structured form or as part of a script, and that code may depend on other code, data, and models, which are not considered in traditional metrics. For example, consider one method that produces a dataset and a second that splits the dataset into training, validation, and test subsets. These methods may not invoke each other, but there is still a dependency through the shared dataset. Modifications to the dataset's format or contents would require maintenance of both methods, and changes to one method may also lead to changes in the other. Traditional metrics would not fully capture this relationship.

As an example of how metrics can be modified to better characterize ML systems, we introduce modified versions of CBO and LCC that (1) are calculated at the file level (to account for both structured code and scripting), and (2), that capture dependencies between code and data and models[7]. New metrics should be developed in future work to assess further components, for example the maintainability of data and models as individual components. However, these examples offer a starting point for scope-aware assessment of maintainability.

Modified Coupling Measurement: We introduce a modified form of the CBO metric, CBO^{ML}, that differs from the original in two ways. First, rather than calculating for each class, which assumes structured code, we calculate for each code file. Second, rather than the number of classes that the file-under-assessment is coupled to, we count the number of other files that the file-under-assessment is coupled to and the number of data files or models that the file-under-assessment is coupled to.

Coupling between code files is determined based on a reference to a variable or method in another code file. In our prototype implementation, coupling between code and data files is determined by detection of invocations of `read`, `write`, and `load` functions referencing a particular filename[8]. CBO^{ML} can be calculated using the following formula:

$$CBO^{ML}(C) = \sum_{i=1}^{c} I(C, C_i) + \sum_{j=1}^{d} I(C, D_j) \tag{3}$$

where C is the code file-under-assessment, c is the total number of code files in the system, and d is the total number of data or model files in the system. $I(C, C_i)$ and $I(C, D_j)$ are equal to 1 if C is coupled to the code or data/model file and 0 otherwise.

Modified Cohesion Measurement: LCC considers a class to be cohesive if many of its methods read or modify the same variables. We introduce a modified version of LCC, which we refer to as LCC^{ML}, with three key changes.

First, rather than per class, we calculate LCC^{ML} for each code file. Second, in addition to considering two methods to be cohesive if they have direct or incorrect connections to the same variables, we consider methods to be cohesive if they access the same data files or models. Third, we also consider two methods to be cohesive if they invoke the same functions from a common ML library[9]. Many ML systems depend on such libraries, and methods contained in the same code file that use the same library functions are likely to be related in their purpose. LCC^{ML} can be calculated using the following formula:

[7] These modified formulae could also be applied to traditional systems, considering dependencies on non-code elements such as databases.

[8] Our concrete implementation is based on Python, but could be adapted to similar functions in other languages.

[9] The specific libraries are listed at `https://anonymous.4open.science/r/maintainability-for-MLSystems-3C12/cohesion_improved.py`.

$$LCC^{ML}(C) = \frac{M_v \cup M_f \cup M_l}{\frac{n(n-1)}{2}} \tag{4}$$

where M_v is the number of method pairs that are directly or indirectly connected by sharing at least one attribute, M_f is the number of method pairs that are directly or indirectly connected by accessing at least one common file, M_l is the number of method pairs that are directly or indirectly connected by accessing the same library function, and n is the total number of methods in the class. LCC^{ML} values range between 0 and 1, with 1 indicating maximum cohesion.

Prototype Implementation: Our measurement implementations for traditional and modified CBO and LCC can be found at `https://anonymous.4open.science/r/maintainability-for-MLSystems-3C12/`. Our prototype performs static analysis to detect dependencies on code and data files. Pseudocode describing our implementations can be found at `https://anonymous.4open.science/r/maintainability-for-MLSystems-3C12/`.

4 Evaluation

We are interested in characterizing the maintainability of ML systems using our proposed breakdown and metrics:

RQ1: For the examined systems, what differences occur between traditional and modified metrics?
RQ2: What factors explain these differences?
RQ3: To what extent do the examined systems match our conceptual breakdown?
RQ4: How and why do the traditional and modified metrics differ across scopes?

ML System Selection: We selected 10 representative ML systems by searching GitHub for projects with the "ML" tag and choose the 10 systems with the most stars—that is, the most popular on GitHub. We performed this selection on July 15th, 2024. We selected 10 projects because it is a sufficient number to highlight some of the differences that can occur between systems, while still remaining a reasonable number to manually inspect. Table 1 presents information about the selected ML systems.

Table 1: Selected ML systems, description of the system, GitHub stars, the number of files per scope, and link to the project.

			Number of Files Per Scope					
System	**Description**	**Stars**	**Data Acquisition**	**Training Pipeline**	**ML Interfacing**	**Non-ML**	**Whole System**	**Link (`https://github.com/`)**
face recognition	Face Detection	53k	0	4	22	2	28	ageitgey/face_recognition
faceswap	Image Processing	52k	0	37	162	42	241	deepfakes/faceswap
Open Assistant	Chat Bot	37k	51	69	64	201	385	LAION-AI/Open-Assistant
DeepFaceLive	Image Processing	26k	1	2	45	45	93	iperov/DeepFaceLive
CLIP	Image-to-Text	25k	1	2	2	1	7	openai/CLIP
EasyOCR	Character Recognition	23k	1	46	13	0	60	JaidedAI/EasyOCR
DocsGPT	Chat Bot	14k	1	10	28	32	71	arc53/DocsGPT
Chatterbot	Chat Bot Creator	14k	0	5	17	7	29	gunthercox/ChatterBot
DeepFace	Facial Analysis	12k	1	27	41	15	84	serengil/deepface
LaTeX-OCR	Image Processing	12k	1	24	5	8	38	lukas-blecher/LaTeX-OCR

Partitioning Systems into Scopes: We manually inspected each project, partitioning code files into four scopes based on components from Figure 1:

- **Data Acquisition:** Data extraction, ingestion, and labeling components.
- **Training Pipeline:** Data cleaning, pre-processing or model training, tuning, validation, and testing components.
- **ML-Interfacing Components:** Components that load models, use models to perform functionality, monitor models, and govern the use of models.
- **Non-ML Components:** Traditional components, independent of ML.

We chose these scopes, rather than individual components, because many of the inspected projects merge individual components within the same files.

The second author performed this analysis manually by reviewing the folder structure of the GitHub projects, file names, and code. They then assigned each file to the appropriate partition based on the file's functionality and alignment with the specific scope. For all 10 projects, it took approximately four hours and thirty minutes to partition the files and map them to specific scopes (an average of 30 minutes per project). The first author checked the partitioning results for two projects, and confirmed the assignment.

Some files, which have overlapping functionality, were difficult to partition. For example, some files belonged to both the "ML-Interfacing" and "Training Pipeline" scopes, and some blended both "ML-Interfacing" and "Non-ML" within the same file. In these cases, the files were counted under both scopes. However, similarities in file, method, and variable naming conventions across projects helped to understand scope assignment.

Data Collection: We collected both traditional and modified CBO and LCC for each code file in each project. We then calculated the average for each scope and the whole system.

Data Analysis: We used descriptive statistics (averages and distribution characteristics) to compare CBO, CBO^{ML}, LCC, and LCC^{ML} values between projects and scopes within and across projects. To identify factors leading to differences in the metrics, we performed a qualitative analysis of the code of a subset of the ML systems. To help understand differences in traditional and revised metrics, we examined file and library dependencies in each system and between scopes within the systems—searching for commonalities and differences in variable access, library functions invoked, and file access.

5 Results and Discussion

5.1 Traditional and Revised Metrics, Whole System (RQ1)

Coupling Results: Table 2 presents the average CBO for each of the 10 ML systems, while Table 3 presents the average CBO^{ML}. In Table 3, colored cells

Table 2: Average CBO for each system and scope.

System	Whole System	Data Acquisition	Training Pipeline	ML Interfacing	Non-ML
face recognition	2.32	✖	2.50	2.65	1.00
faceswap	46.16	✖	43.09	50.24	41.72
Open Assistant	15.20	10.38	28.52	12.02	11.80
DeepFaceLive	22.47	17.00	35.00	27.18	27.18
CLIP	1.60	2.00	2.00	1.50	1.00
EasyOCR	11.51	13.00	11.62	12.58	✖
DocsGPT	8.73	15.00	10.11	7.92	12.19
Chatterbot	8.86	✖	13.75	11.47	12.83
DeepFace	24.94	✖	27.39	28.97	10.56
LaTeX-OCR	6.08	4.00	6.71	6.00	6.00
Average	**14.79**	**10.23**	**18.07**	**16.05**	**13.81**

Table 3: Average CBO^{ML} for each system and scope. Colored cells indicate difference from CBO.

System	Whole System	Data Acquisition	Training Pipeline	ML Interfacing	Non-ML
face recognition	2.32	✖	2.50	2.65	1.00
faceswap	46.16	✖	43.09	50.24	41.72
Open Assistant	15.25 (0.33%)	10.67 (2.79%)	28.61 (0.32%)	12.02	11.81 (0.08%)
DeepFaceLive	22.47	17.00	35.00	27.18	27.18
CLIP	1.60	2.00	2.00	1.50	1.00
EasyOCR	11.54 (0.26%)	13.00	11.62	12.58	✖
DocsGPT	8.80 (0.80%)	15.00	11.11 (9.89%)	7.96 (0.51%)	12.19
Chatterbot	8.91 (0.56%)	✖	13.75	11.47	12.83
DeepFace	24.94	✖	27.39	28.97	10.56
LaTeX-OCR	6.08	4.00	6.71	6.00	6.00
Average	**14.81 (0.14%)**	**10.28 (0.49%)**	**18.18 (0.61%)**	**16.06 (0.06%)**	**13.81**

Table 4: Average LCC for each system and scope.

System	Whole System	Data Acquisition	Training Pipeline	ML Interfacing	Non-ML
face recognition	0.70	✖	0.50	0.05	0.00
faceswap	0.66	✖	0.60	0.56	0.61
Open Assistant	0.42	0.63	0.40	0.19	0.40
DeepFaceLive	0.65	0.54	0.97	0.87	0.88
CLIP	0.54	0.00	0.38	0.00	1.00
EasyOCR	0.47	0.22	0.45	0.16	✖
DocsGPT	0.49	0.00	0.65	0.20	0.59
Chatterbot	0.87	✖	0.21	0.57	0.73
DeepFace	0.50	✖	0.37	0.20	0.64
LaTeX-OCR	0.46	0.00	0.35	0.01	0.49
Average	**0.58**	**0.23**	**0.49**	**0.28**	**0.59**

Table 5: Average LCC^{ML} for each system and scope. Colored cells indicate difference from LCC.

System	Whole System	Data Acquisition	Training Pipeline	ML Interfacing	Non-ML
face recognition	0.70	✖	0.50	0.05	0.00
faceswap	0.69 (3.79%)	✖	0.64 (5.83%)	0.59 (4.82%)	0.61
Open Assistant	0.45 (5.95%)	0.64 (0.95%)	0.44 (9.75%)	0.20 (3.16%)	0.41 (1.25%)
DeepFaceLive	0.66 (2.00%)	0.54	0.97	0.87	0.88
CLIP	0.54	0.00	0.38	0.00	1.00
EasyOCR	0.55 (17.45%)	0.33 (51.36%)	0.52 (15.11%)	0.37 (133.75%)	✖
DocsGPT	0.49	0.00	0.65	0.20	0.59
Chatterbot	0.87	✖	0.21	0.57	0.73
DeepFace	0.53 (5.40%)	✖	0.39 (5.95%)	0.23 (13.00%)	0.68 (5.78%)
LaTeX-OCR	0.49 (6.15%)	0.00	0.40 (12.86%)	0.03 (180.00%)	0.49
Average	**0.60 (3.41%)**	**0.25 (8.35%)**	**0.51 (3.98%)**	**0.31 (10.53%)**	**0.60 (0.66%)**

indicate an increase from CBO, with the magnitude of the increase in parentheses[10]. In previous research, a CBO greater than 9 indicated high coupling [24]. Five of the ten ML systems are highly coupled, under this interpretation, with the `faceswap` system being the most (46.16 on average).

We hypothesize that CBO^{ML} is more accurate than CBO for assessing coupling within ML systems, as it considers coupling between code-based components through sharing of data or models. As expected, we see some increase in the average CBO^{ML} values, as these additional dependencies are captured in the metrics. However, these changes are not substantial—there is only a 0.14% increase in the average across all systems and a 0.39% increase in the median. The average CBO^{ML} only increases in four systems. This indicates that, although there are more components considered when measuring CBO^{ML} than when measuring CBO, there were not necessarily a large number of missed dependencies. Coupling between code components remains more common.

Cohesion Results: Table 4 presents the average LCC, while Table 5 presents LCC^{ML}. In Table 5, colored cells indicate an increase from LCC, with the

[10] Because both of the revised metrics count additional elements not present in the traditional metric, values can only increase from the traditional metric.

magnitude of the increase in parentheses. Six systems have average LCC values greater than 0.5, indicating that the systems tend to be relatively highly cohesive. `Chatterbot` is the most cohesive system, with an average LCC of 0.87.

We hypothesize that LCC^{ML} is more accurate than LCC for assessing cohesion within ML systems, as it consider methods to be cohesive through sharing the same variables, data files, models, or library functions, and not just variables. We see an increase in the average LCC^{ML} over LCC for five of the ML systems, with an average increase of 3.41% (3.85% in median). The largest increase, an average of 17.45%, was seen in `EasyOCR`. Like with CBO, the magnitude of the increase indicates that cohesion through code-based mechanisms is more common than cohesion though data files or library functions. However, LCC^{ML} still highlights dependencies that are missed by the traditional metric.

5.2 Factors that Contribute to Differences (RQ2)

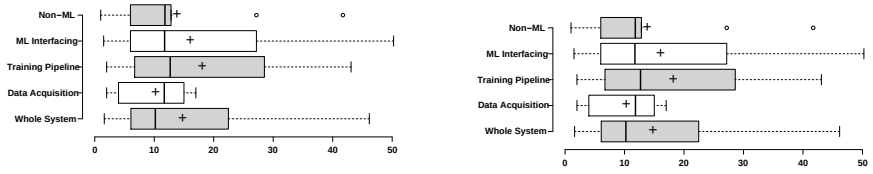
Coupling Results: We make two observations regarding why the CBO^{ML} values are not substantially higher than the CBO values. First, file access operations were not common in all examined ML systems. For example, in `DeepFace`, there are only four locations where data files are opened and the operations were not used for ML-based functionality. There are many more situations where code components interact, limiting the impact of the data components.

Second, examining `LaTeX-OCR`, we found that the CBO value did not change due to the limitations of static code analysis. We identified dependencies based on, for example, specified filenames. However, in cases where filenames were passed dynamically, we were not able to determine whether a dependency existed. In the four systems where CBO^{ML} values differed, file interactions were always based on static filenames. This is a limitation in our implementation, and in future work, we will explore dynamic analysis.

Cohesion Results: LCC^{ML} values are higher than LCC values, on average. However, the magnitude of the increase is small. On average, there is only a 3.41% increase. The reasons for the limited increase are the same as for CBO. First, traditional dependencies on variables, shared between methods within classes, are more common than shared dependencies on data or external library functions. Second, limitations in static analysis may result in missed data dependencies.

However, the magnitude of the differences is greater than it was for CBO. There are two reasons for this. First, when data dependencies exist, they are more often between two methods in the same file than across different files. For example, in `Open Assistant`, file `language_classification.py` contains methods that load and save the same model.

Second, LCC^{ML} also includes shared use of ML library calls between methods in the same code file. For example, For example, in `DeepFace`, file `VGGFace.py` has multiple methods that use the same library functions as part of loading and training models. We chose to include these calls in LCC^{ML} and not CBO^{ML} after inspecting the code of several ML systems and identifying cases where methods within individual files (i.e., cohesion) were clearly connected in their purpose through their use of library functions. Such connections were not as clear *across* files (i.e., in coupling).



(a) CBO values per scope. (b) CBO^{ML} values for per scope.

Fig. 3: Comparison of CBO and CBO^{ML} values across scopes.

5.3 Validity of Conceptual Breakdown (RQ3)

To assess the validity of our conceptual breakdown, we examined each ML system and assigned its files to four scopes—selections of components from Figure 1. Table 1 includes the number of files that we assigned to each scope. We were able to assign the files to the proposed scopes, indicating that our conceptual breakdown has validity.

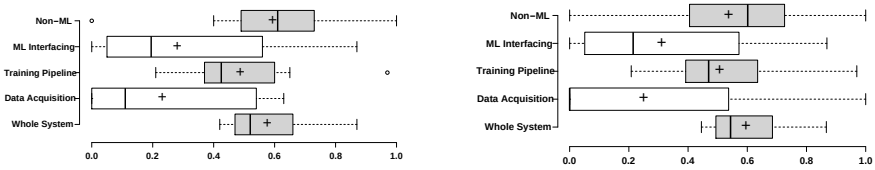
Not all projects have a clean division between individual components. For example, in the `Open Assistant` system, data extraction, ingestion, and labeling are not cleanly divided, but are blended within the same set of files. Rather than specifying NFRs for these components individually, it would be better to apply NFRs to the "Data Acquisition" scope. Within the same system, components of the "Training Pipeline" are more cleanly divided, with separate files responsible for data cleaning, pre-processing, model training, and model validation and testing. Here, NFRs could be specified over the individual components or across the whole training pipeline.

Some components or scopes are missing in some of the ML systems. For example, code related to the "Data Acquisition" scope was not found in four of the projects. This indicates that the projects do not have code for preparing datasets for use in model training. Instead, datasets may be manually created, imported from another source, or the code for preparing datasets is not included in the project. Additionally, the `EasyOCR` system does not have any "Non-ML Components" because it is intended for use as an imported library.

5.4 Differences Between Scopes (RQ4)

Coupling Results: Table 2 presents CBO values for each scope, while Table 3 presents CBO^{ML}. Figures 3a–3b also visualize the values, split by scope. For both versions of the metric, we can see differences between scopes. Although there are exceptions (e.g., `EasyOCR` and `Chatterbot`), the "Training Pipeline" and "ML Interfacing" components have higher average coupling than the average for the system as a whole or for the non-ML components. In particular, the "Training Pipeline" has the highest median and third quartile of any scope. These observations hold between both versions of CBO, and the largest difference between the two versions is in the "Training Pipeline" scope for `DocsGPT`.

In contrast, "Data Acquisition" and "Non-ML" scopes tend to have lower coupling than the average across the full system. This suggests that—even when



(a) LCC values for each scope. (b) LCC^{ML} values for each scope.

Fig. 4: Comparison of LCC and LCC^{ML} values across all scopes.

considering only code-based dependencies—the code files within the "Training Pipeline" and "ML Interfacing" scopes are more densely interdependent than code in other scopes. One reason for this, in the "ML Interfacing" components, may be because interactions with models are often through an API wrapper. Code that depends on a model, then, may have a dependency to this API.

Another factor is the number of code files in each scope. For example, in `faceswap`, the "Training Pipeline" and "ML Interfacing" scopes have 37 and 162 files, respectively. On the other hand, `face recognition`, `CLIP`, or *LaTeX-OCR* have a lower number of files in these scopes and lower coupling. The existence of more files does not automatically imply that coupling will be higher, but it does mean there are more opportunities for coupling to occur.

Cohesion Results: Table 4 present LCC values for each scope, while Table 5 presents LCC^{ML} values. Figures 4a–4b visualize values, split by scope.

For both LCC and LCC^{ML}, results differ between scopes. Figures 4a–4b show that ML components ("Data Acquisition", "Training Pipeline", and "ML Interfacing") have a lower average cohesion than non-ML components. However, looking at Tables 4–5, at least one ML component is more cohesive than the non-ML components in five projects. Thus, the average LCC^{ML} scores are biased by the other four projects—`Clip`, `Chatterbot`, `DeepFace`, and `LaTeX-OCR`—where all ML scopes have a lower average cohesion than non-ML components. Note that EasyOCR has no non-ML components. Because we do not see that ML components are consistently more cohesive than non-ML components, we cannot conclusively declare that ML components suffer from cohesion issues. However, we can make some observations: First, the "Data Acquisition" scope seems to suffer from the clearest cohesion issues, showing the lowest average cohesion scores across all scopes for five of the six projects where that scope exists. In three of those, the average cohesion is 0.00. Second, "ML Interfacing" components also have lower cohesion than non-ML components in eight of the projects.

One reason for lower cohesion might be an unclear division of functionality into distinct components. For example, data cleaning, ingestion, and labeling might be mixed in a single code file, and these three might not share a large number of variables, library functions, or files. Even if multiple functions in that code file may depend on the same ML library, the functions may not be linked through the same functions from that library.

Summary: Overall, our results support claims in the literature that ML systems are more coupled [22, 23, 26] and less cohesive [3, 23] than traditional systems, potentially hindering maintainability.

6 Limitations and Threats to Validity

Construct Validity: Maintainability has five sub-characteristics, but we have only focused on modularity—and specifically on cohesion and coupling measurements. In addition, we focus on the modularity of code, and not the modularity of data, models, or other ML system components. However, modularity is one of the most critical sub-characteristics of maintainability, and cohesion and coupling are the most common means of assessing modularity. Still, these metrics have been criticized as too narrowly focused on code-level dependencies [2, 4]. Future work should propose adapted or new measurements for other sub-characteristics and for additional ML system components or scopes.

Our measurements only account for modularity from the perspective of code and scripts, including dependencies to data and ML files, but do not account for modularity from the perspective of all ML system elements described in Fig. 1. For example, we have not developed measurements that assess the "modularity" of data, as measuring modularity of data requires a different interpretation of modularity than applies to code. Future work should consider metrics from the perspective of these other components.

External Validity: We have only examined ten systems to offer a reasonable portrait of the domain while controlling experiment costs. We used popularity (stars) to select systems. However, many of these systems are based on the use of images as a data source. Thus, this subset may not represent the full diversity of ML systems, including types of ML (e.g., we do not currently consider unsupervised or reinforcement learning) or application domains. However, we believe that this subset is sufficient to illustrate the proposed breakdown and measurements, and offers lessons for future work. Furthermore, we focused on Python. However, Python is the most popular language for ML system development, and the proposed measurements and component breakdown are not language-specific. Future studies should evaluate a wider variety of systems.

Internal Validity: Our metric implementations are based on static analysis and could miss dependencies. However, we performed manual validation on two projects to ensure accuracy. Future work should integrate dynamic analysis techniques to capture additional dependencies. The second author manually mapped files to scopes using set criteria, although this may be subjective, we mitigated this threat by having the first author validate the mapping and conclusions for two projects. There were no disagreements among the authors during manual inspection, as code naming conventions made it easier to understand which file belongs to what granular-level component, thus we have reasonable confidence in our mapping.

7 Conclusion

We have proposed a breakdown of ML systems into common components as well as a revised definition and measurements of maintainability that account

for these components. We found that our breakdown is applicable to 10 systems, and that the modified measurements capture dependencies missed by traditional measurements—although code dependencies between and within components are more common than data dependencies. We also found that training pipelines and ML interfacing components tend to have higher-than-average coupling, and that data acquisition and ML interfacing components tend to have lower-than-average cohesion. Future work should evaluate our measurements and scoping on further projects, other sub-characteristics of maintainability, other elements of ML systems, and other NFRs critical for ML systems.

Acknowledgements: This work is supported by a Swedish Research Council (VR) Project: Non-Functional Requirements for Machine Learning: Facilitating Continuous Quality Awareness (iNFoRM).

References

1. Alho, R., Raatikainen, M., Myllyaho, L., Nurminen, J.K.: On modularity of neural networks: Systematic review and open challenges. In: International Conference on Software and Software Reuse. pp. 18–36. Springer (2024)
2. Anquetil, N., Laval, J.: Legacy software restructuring: Analyzing a concrete case. In: 2011 15th European Conference on Software Maintenance and Reengineering. pp. 279–286. IEEE (2011)
3. Apel, S., Kästner, C., Kang, E.: Feature interactions on steroids: On the composition of ml models. IEEE Software **39**(3), 120–124 (2022)
4. Candela, I., Bavota, G., Russo, B., Oliveto, R.: Using cohesion and coupling for software remodularization: Is it enough? ACM Transactions on Software Engineering and Methodology (TOSEM) **25**(3), 1–28 (2016)
5. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
6. Döhmen, T., Bruntink, M., Ceolin, D., Visser, J.: Towards a benchmark for the maintainability evolution of industrial software systems. In: 2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA). pp. 11–21. IEEE (2016)
7. Etzkorn, L.H., Gholston, S.E., Fortune, J.L., Stein, C.E., Utley, D., Farrington, P.A., Cox, G.W.: A comparison of cohesion metrics for object-oriented systems. Information and Software Technology **46**(10), 677–687 (2004)
8. Fischer, L., Ehrlinger, L., Geist, V., Ramler, R., Sobieczky, F., Zellinger, W., Moser, B.: Applying ai in practice: key challenges and lessons learned. In: International Cross-Domain Conference for Machine Learning and Knowledge Extraction. pp. 451–471. Springer (2020)
9. Fregnan, E., Baum, T., Palomba, F., Bacchelli, A.: A survey on software coupling relations and tools. Information and Software Technology **107**, 159–178 (2019)
10. Habibullah, K.M., Diaz, J.G., Gay, G., Horkoff, J.: Scoping of non-functional requirements for machine learning systems. In: 2024 IEEE 32nd International Requirements Engineering Conference (RE). pp. 496–497. IEEE (2024)
11. Habibullah, K.M., Gay, G., Horkoff, J.: Non-functional requirements for machine learning: An exploration of system scope and interest. In: Proceedings of the 1st Workshop on Software Engineering for Responsible AI. pp. 29–36 (2022)

12. Habibullah, K.M., Gay, G., Horkoff, J.: Non-functional requirements for machine learning: Understanding current use and challenges among practitioners. Requirements Engineering **28**(2), 283–316 (2023)
13. Habibullah, K.M., Gay, G., Horkoff, J.: A framework for managing quality requirements for machine learning-based software systems. In: International Conference on the Quality of Information and Communications Technology. pp. 3–20. Springer (2024)
14. Habibullah, K.M., Heyn, H.M., Gay, G., Horkoff, J., Knauss, E., Borg, M., Knauss, A., Sivencrona, H., Li, P.J.: Requirements and software engineering for automotive perception systems: an interview study. Requirements Engineering pp. 1–24 (2024)
15. ISO/IEC 23053: ISO/IEC 23053:2022, information technology — framework for artificial intelligence (ai) systems using machine learning (ml) (2022)
16. ISO/IEC 25059: ISO/IEC 25059:2023, systems and software engineering — systems and software quality requirements and evaluation (square) — quality model for ai systems (2023)
17. Kuwajima, H., Yasuoka, H., Nakae, T.: Engineering problems in machine learning systems. Machine Learning **109**(5), 1103–1126 (2020)
18. Latendresse, J., Abedu, S., Abdellatif, A., Shihab, E.: An exploratory study on machine learning model management. ACM Transactions on Software Engineering and Methodology (2024)
19. Mikkonen, T., Nurminen, J.K., Raatikainen, M., Fronza, I., Mäkitalo, N., Männistö, T.: Is machine learning software just software: A maintainability view. In: Software Quality: Future Perspectives on Software Engineering Quality: 13th International Conference, SWQD 2021, Vienna, Austria, January 19–21, 2021, Proceedings 13. pp. 94–105. Springer (2021)
20. Munappy, A., Bosch, J., Olsson, H.H., Arpteg, A., Brinne, B.: Data management challenges for deep learning. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 140–147. IEEE (2019)
21. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: proceedings of the 26th Symposium on Operating Systems Principles. pp. 1–18 (2017)
22. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. Advances in neural information processing systems **28** (2015)
23. Serban, A., Visser, J.: Adapting software architectures to machine learning challenges. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 152–163. IEEE (2022)
24. Shatnawi, R.: A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. IEEE Transactions on software engineering **36**(2), 216–225 (2010)
25. Shivashankar, K., Martini, A.: Maintainability challenges in ml: A systematic literature review. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 60–67. IEEE (2022)
26. Smith, M.R., Martinez, C., Ingram, J.B., DeBonis, M., Cuellar, C.R., Jose, D.: Test and evaluation of systems with embedded machine learning components. ITEA Journal of Test and Evaluation **44**(SAND-2023-11084J) (2023)
27. Tiwari, S., Rathore, S.S.: Coupling and cohesion metrics for object-oriented software: A systematic mapping study. In: Proceedings of the 11th Innovations in Software Engineering Conference. pp. 1–11 (2018)