Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices

Anup K. Kalia
Jin Xiao
anup.kalia@ibm.com
jinoaix@us.ibm.com
IBM Research
Yorktown Heights, NY
USA

Rahul Krishna
Saurabh Sinha
rkrsn@ibm.com
sinhas@us.ibm.com
IBM Research
Yorktown Heights, NY
USA

Maja Vukovic
maja@us.ibm.com
IBM Research
Yorktown Heights, NY
USA

Debasish Banerjee
debasish@us.ibm.com
IBM Hybrid Cloud,
Customer Success
Rochester, MN, USA

ABSTRACT

In migrating production workloads to cloud, enterprises often face the daunting task of evolving monolithic applications toward a microservice architecture. At IBM, we developed a tool called to assist with this challenging task. Mono2Micro performs spatio-temporal decomposition, leveraging well-defined business use cases and runtime call relations to create functionally cohesive partitioning of application classes. Our preliminary evaluation of showed promising results.

How well does Mono2Micro perform against other decomposition techniques, and how do practitioners perceive the tool? This paper describes the technical foundations of Mono2Micro and presents results to answer these two questions. To answer the first question, we evaluated Mono2Micro against four existing techniques on a set of open-source and proprietary Java applications and using different metrics to assess the quality of decomposition and tool's efficiency. Our results show that Mono2Micro significantly outperforms state-of-the-art baselines in specific metrics well-defined for the problem domain. To answer the second question, we conducted a survey of twenty-one practitioners in various industry roles who have used Mono2Micro. This study highlights several benefits of the tool, interesting practitioner perceptions, and scope for further improvements. Overall, these results show that Mono2Micro can provide a valuable aid to practitioners in creating functionally cohesive and explainable microservice decompositions.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Software architectures**;

KEYWORDS

microservices, dynamic analysis, clustering

ACM Reference Format:
Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. 2021. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473915

1 INTRODUCTION

Enterprises are increasingly moving their production workloads to cloud to take advantage of cloud capabilities, such as streamlined provisioning of infrastructure and services, elasticity, scalability, reliability, and security. To leverage cloud-native capabilities, monolithic applications have to be decomposed to cloud-native architectures, such as microservices. A *microservice* encapsulates a small and well-defined set of business functionalities and interacts with other services using lightweight mechanisms, often implemented as RESTful APIs [16, 34]. In modernizing legacy applications, enterprises, however, often have to answer the challenging question of *how to transform their monolithic applications to microservices.*

Current strategies for decomposing monolithic applications fall under static- or dynamic-analysis techniques, *i.e.,* they typically compute module dependencies using static and/or dynamic analysis and apply clustering or evolutionary algorithms over these dependencies to create module partitions that have desired properties (*e.g.,* high cohesion and low coupling). Static approaches [10, 11, 13, 15, 28, 31, 38, 45, 51] suffer imprecision in computing dependencies that is inherent to static analysis. In Java Enterprise Edition (JEE) applications, which are the focus of our work, these techniques face challenges in dealing with dynamic language features, such as reflection, dynamic class loading, context, and dependency injections. In contrast, dynamic techniques (*e.g.,* [12, 19, 23, 35]) capture runtime dependencies and thus avoid the imprecision problems. However, a common challenge that still exists for both static and dynamic analysis is computing the alignment of classes and their dependencies with the business functionalities of the application, which is a primary concern in industrial practice.

In this contribution, we show how Mono2Micro [25] based on dynamic analysis achieves the alignment of classes and their dependencies with business functionalities of the application. Mono2Micro [25] was developed at IBM and recently in January 2021 made generally available as a product[1]).

[1]https://www.ibm.com/cloud/mono2micro

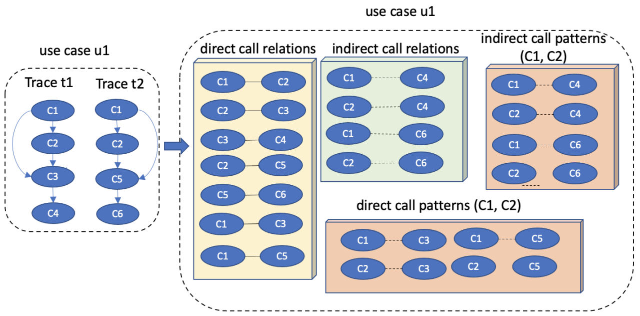


Figure 1: Illustration of execution traces and temporal relations.

Mono2Micro. We implement a *hierarchical spatio-temporal decomposition* in Mono2Micro that dynamically collects runtime traces under the execution of specific business use cases of the application and applies clustering on classes observed in the traces to recommend partitions of the application classes. In this approach, business use cases constitute the *space* dimension, whereas the control flow in the runtime traces expresses the *time* dimension.

• **Business Use Cases**. The space dimension emphasizes the importance of identifying candidate microservices as functionally cohesive groups of classes, each of which implements a small set of well-defined functionalities that can be easily explained to a business user. To implement the space dimension, Mono2Micro considers module dependencies specifically in the contexts of business use cases under which they occur. Examples of such business use cases are `Create Account`, `Browse Products`, and `Checkout Products`. In contrast, a technique that analyzes dependencies while ignoring business use cases can recommend partitions that mix different functionalities and, thus, suffer low cohesion. Moreover, the rationale for the computed groupings, agnostic to business use cases, can be hard to explain to a practitioner.

• **Runtime Call Traces**. The time dimension considers temporal relations and co-occurrence relations among classes extracted from runtime call traces (collected by executing use cases). Existing techniques in the areas of software repackaging [35, 49] and microservice extraction [12, 23, 24] analyze direct call relations only. We enhance those approaches in two ways. First, we consider indirect call relations, as shown in Figure 1, that indicate long-range temporal relations among classes. Second, we propose direct call patterns and indirect call patterns to capture the pattern of interaction among classes. The patterns capture the similarity between classes based on how they call other classes through direct or indirect relations across one or more use cases. In Figure 1, (c_1, c_2) and (c_2, c_3) are the examples of direct call relations. (c_1, c_4) and (c_1, c_6) are the examples of indirect call relations. Considering direct call patterns, c_1 and c_2 are similar based on how they call other classes such as c_3 and c_5 through direct relations and c_4 and c_6 through an indirect relations, respectively. We can derive direct and indirect call patterns for other pairs of classes in a similar manner.

Evaluation. We describe the technical details of Mono2Micro and the results of empirical studies conducted on two sets of JEE applications: four open-source web applications and three proprietary web applications. We evaluate Mono2Micro against four well-known baseline approaches from software remodularization and microservices communities i.e., Bunch [32], FoSCI [23], CoGCN [13], and MEM [31]. We perform the evaluation using five metrics: Inter-Call Percentage (ICP) [25], Business Context Purity (BCP) [25], Structural Modularity (SM) [23], Interface Number (IFN) [23] and Non-Extreme Distribution (NED) [13]. In addition, we conducted a survey among 21 industry practitioners to highlight the importance and benefits of Mono2Micro and further scope for improvement.

Our results indicate that Mono2Micro consistently performs well compared with BCP and NED and is competitive with ICP and IFN. Considering SM, Mono2Micro did not perform well when compared to Bunch and MEM. However, we observed that high SM scores in such baselines also have higher NED scores indicating extreme distributions. From the survey, we learned several benefits of Mono2Micro such as the following. 1) Mono2Micro helps implement a Strangler pattern, 2) recommendations generated using Mono2Micro capture required business functionalities and are self-explainable, 3) Mono2Micro can detect potential unreachable code. In addition, we learned the scope for further improvements of Mono2Micro such as the following. 1) minimize the number of changes a user has to make on the top of the recommendations generated, 2) add database interactions and transaction patterns to refine recommendations, and so on.

The rest of the paper is organized as follows. In the next section, we describe the technical details of Mono2Micro and illustrate it using an open-source JEE application. Section 4 provides the research questions. Section 3 presents the empirical evaluation. Section 4.3 presents the survey. Section 5 summarizes of research questions. Section 6 highlights the threats to the validity of the empirical evaluation and the survey. Section 7 discusses related work. Finally, Section 10 summarizes the paper and lists directions for future research. Section 10 provides acknowledgements to everyone who have helped build Mono2Micro.

2 MONO2MICRO: TECHNICAL APPROACH

In this section, we present the technical details of the approach implemented in Mono2Micro; Figure 2 shows the main steps of the approach. First, we introduce a sample application, `JPetStore`, to illustrate the approach and discuss analysis preliminaries, which consists of trace collection and reduction. Then, we describe the core partitioning technique in the context of the `JPetStore` application.

2.1 Analysis Preliminaries

Runtime Trace Collection: Runtime traces are defined as $T^{(u_i)} = \langle t_1, t_2, \ldots, t_T \rangle$, where each trace t_i is generated by running a use case $u_i \in U$. A user can manually create such use cases by navigating through the application's user interface (UI) and providing an appropriate label for each use case. If functional test cases are available for an application, one can use them for generating runtime traces. The tests need not be UI test cases, but a test case must correspond to a well-defined application use case (business functionality). Traces record the entries and exits to each function, including the constructors via added probes. For an open-source application `JPetStore` application, we created ten use cases, *e.g., update_item* and *click_item*. We generated runtime traces by executing the use cases navigating through the UI of the application for each use case. The use cases (traces) cover 37 of the 42 classes (88%

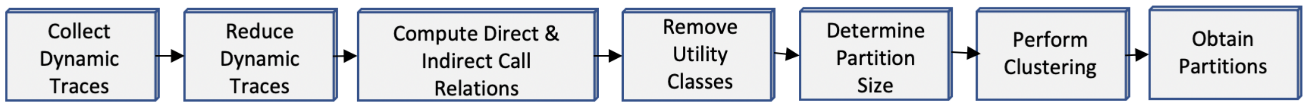


Figure 2: The main steps of the decomposition approach.

class coverage). The trace obtained via the execution of a use case is a *raw trace*. An example fragment of a raw trace is as follows:

```
t1,[32],Entering ... PetStoreImpl::getCategory
t2,[32],Entering ... SqlMapCategoryDao::getCategory
t3,[32],Entering ... Category::setCategoryId
t4,[32],Exiting ... Category::setCategoryId
```

Each trace element captures a timestamp, a thread id, and an entry/exit label with a class name and a method signature.

Trace Reduction: For each use case $u_i \in$ U, we reduce the number of traces in two ways. One, we reduce the total number of traces by considering unique traces. Two, we reduce the length of a trace by removing a redundant sequence of classes that might have invoked due to the presence of a loop. We remove the redundant sequences by converting traces to a representation similar to a calling-context tree (CCT) [3]. Specifically, each trace $t_j \in T^{(u_i)}$ is processed to build a set of CCTs, at the level of class methods, with each tree rooted at an "entry point" class that is the first one to be invoked in response to a UI event. Unlike the conventional CCT, in which nodes represent methods [3], in our CCT, nodes represent classes, thereby further reducing the length of traces.

Below we provide two class-level CCTs that are constructed from the raw traces collected by executing two use cases: *click_item* and *update_item*. In the example, `Root` corresponds to an entry-point. For *click_item*, we obtain one reduced trace (`ViewCatego- ryController` $\longrightarrow$ `PetStoreImpl` $\longrightarrow$ `SqlMapCategoryDao` $\longrightarrow$ `Category`), whereas for *update_item*, we obtain three reduced traces, each containing two classes; *e.g.,* `UpdateCartQuantitiesController` $\longrightarrow$ `Cart`.

```
click\_item, Root, ViewCategoryCont.., PetStoreImpl, SqlMapCategory..
update\_item, Root, UpdateCartQuantitiesController, Cart
update\_item, Root, UpdateCartQuantitiesController, CartIt.
update\_item, Root, UpdateCartQuantitiesController, Item
```

2.2 Computation of Partitions

The core of our technique consists of first identifying the similarities among a pair of classes c_i and c_j where $i \neq j$ and c_i and $c_j \in$ classes $C = \langle c_1, c_2, \ldots, c_C \rangle$ of an application. We identify the similarities by deriving four spatio-temporal different features (1) direct call relations (DCR), (2) indirect call relations (ICR), (3) direct call patterns (DCP), (4) indirect call patterns (ICP). Then, based on the features, we construct a similarity matrix S(c_{k_1}, c_{k_2}) where c_{k_1} and $c_{k_2} \in C$. For the purpose of similarity computation, we consider undirected edges. We apply the hierarchical clustering algorithm on the matrix to decompose the classes into a set of non-overlapping partitions that aims to execute specific business tasks based on business functionalities or use cases.

Direct Call Relations: A *direct call relation* (DCR) exists between classes c_{k_1} and c_{k_2} if and only if a directed edge (c_{k_1}, c_{k_2}) exists in an execution trace; *i.e.,* a method in c_{k_1} invokes a method in c_{k_2} in a trace. For example, in `JPetStore`, for the *click_item* use case, the `ViewCategoryController` class calls the `PetStoreImpl` class, whereas for the *update_item* use case, `UpdateCartQuantitiesContro- ller` calls `Cart`. Thus, `ViewCategoryController` and `PetStoreImpl` have a direct call relation; similarly, `UpdateCartQuantitiesController` and `Cart` also have a direct call relation.

Table 1: Example to illustrate interaction patterns of two classes (c_1 and c_2) in two use cases (u_1 and u_2).

c_1	u_1	u_2	c_2	u_1	u_2
c_1	0	0	c_1	0	0
c_2	0	0	c_2	0	0
c_3	1	1	c_3	1	0
c_4	0	0	c_4	0	1
c_5	1	0	c_5	1	1

We leverage the use-case labels u_i associated with an execution trace t_j to compute DCR as the ratio of the number of use cases where a direct call relation exists between c_{k_1} and c_{k_2}, to the union of use cases in which c_{k_1} and c_{k_2} occur: DCR$(c_{k_1}, c_{k_2}) = \frac{|\mathrm{U}_{c_{k_1} \leftrightarrow c_{k_2}}|}{|\mathrm{U}_{c_{k_1}} \cup \mathrm{U}_{c_{k_2}}|}$. For example, if c_1 participates in two use cases $\{u_1, u_2\}$, c_2 participates in three use cases $\{u_1, u_2, u_3\}$, and there is only one direct call relation between them, we compute DCR(c_1, c_2) as $\frac{1}{3}$.

Direct Call Pattern: Based on direct call relations, we derive another spatio-temporal feature *direct call pattern* (DCP) that exists between two classes c_{k_1} and c_{k_2} if and only if there exist an edge (c_{k_1}, c_l), or (c_{k_2}, c_l) in the traces; *i.e.,* both c_{k_1} and c_{k_2} have a direct call relation with c_l in some execution trace. Whereas DCR considers the interactions between two classes, DCP considers whether two classes have a similar pattern of interaction with other classes. We compute DCP(c_{k_1}, c_{k_2}) as follows: DCP(c_{k_1}, c_{k_2}) $= \frac{\sum_{c \in C, c_l \neq \{c_{k_1}, c_{k_2}\}} |\mathrm{U}_{c_{k_1} \leftrightarrow c_l} \cap \mathrm{U}_{c_{k_2} \leftrightarrow c_l}|}{(|C|-2)*(|\mathrm{U}|)}$. To illustrate, consider the call relations for classes c_1 and c_2 under different use cases shown in Table 1. As shown in Table 1, c_1 and c_2 do not have a direct call relation. However, c_1 and c_2 have two direct call relations with c_3 and c_5, respectively. We divide the total number of direct call patterns by the total number of possible call patterns $(|C| - 2) * (|\mathrm{U}|)$ for c_1 and c_2 across all use cases. Under the use case u_1, c_1 and c_2 have, in total, two direct call relations with c_3 and c_5, respectively. Therefore, we compute DCP(c_1, c_2) as $\frac{2}{3*2}$.

Indirect Call Relations: An *indirect call relation* (ICR) exists between classes c_{k_1} and c_{k_2} if and only if there exists a path (c_{k_1}, c_1, . . ., c_p, c_{k_2}), $p \geq 1$, in an execution trace. The indirect call relation (ICR) is calculated as the ratio of the number of use cases where an indirect call relation between c_{k_1} and c_{k_2} occurs to the union of use cases associated with these two classes. For example, in `JPetStore`, for the *browse* use case, the `ViewCategoryController` class has a transitive call relations with the `SqlMapCategoryDao` class and the `Category` class. We calculate ICR(c_{k_1}, c_{k_2}) as $\frac{|\mathrm{U}_{c_{k_1} \Longleftrightarrow c_{k_2}}|}{|\mathrm{U}_{c_{k_1}} \cup \mathrm{U}_{c_{k_2}}|}$.

Indirect Call Pattern: Based on indirect call relations, we define *indirect call pattern* (ICP) as a relation that exists between classes c_{k_1} and c_{k_2} if and only if there exist paths $(c_{k_1}, c_1, \ldots, c_p, c_l)$ and

$(c_{k_2}, c_1, \ldots, c_q, c_l)$, $p \geq 1$ and $q \geq 1$, in the execution traces; *i.e.,* both classes have an indirect call relation with a common class c_l. ICP is computed as: $\mathrm{ICP}(c_{k_1}, c_{k_2}) = \frac{\sum_{c \in C, c_l \neq \{c_{k_1}, c_{k_2}\}} |\mathrm{U}_{c_{k_1} \leftrightarrow c_l} \cap \mathrm{U}_{c_{k_2} \leftrightarrow c_l}|}{(|C|-2) * (|\mathrm{U}|)}$.

Computation of Similarity: Based on these call relations and patterns, the similarity score between two classes c_{k_1} and c_{k_2} is calculated as: $\mathrm{S}(c_{k_1}, c_{k_2}) = \mathrm{DCR}(c_{k_1}, c_{k_2}) + \mathrm{DCP}(c_{k_1}, c_{k_2}) + \mathrm{ICR}(c_{k_1}, c_{k_2}) + \mathrm{ICP}(c_{k_1}, c_{k_2})$. We represent $\mathrm{S}(c_{k_1}, c_{k_2})$ as a similarity matrix.

Hierarchical Clustering: We use the well-known hierarchical clustering algorithm [44] for three reasons. First, it has been investigated in prior work on software modularization [4, 35, 42] and microservice identification [23]. Second, it has less time complexity compared to the hill-climbing algorithm [29, 32] and genetic algorithms [14, 23, 32] (scalability is essential for analyzing large enterprise applications). Third, we assume that monoliths have hierarchical overlapping business processes that need to be separated into microservices and hence a non-parametric approach such as the hierarchical clustering algorithm is appropriate for the setting.

The hierarchical clustering algorithm groups similar objects into clusters (partitions) based on S. The algorithm takes the target number of clusters n as its sole input. Initially, we assign each class $c_k \in C$ to a cluster P_i. During the clustering process, the similarity score $Sim_{i,j}$ between each pair of clusters i and j as $\frac{\sum_{i=0}^{n_i} \sum_{n=0}^{n_j} S(c_{im}, c_{jn})}{|C_i||C_j|}$. We merge the pairs with the highest similarity score. We iterate the step until the stopping criterion n is achieved.

Partitions Explainability: We obtained five partitions from `JPetStore` using $n = 5$. We provide the details of the partitions and corresponding use cases in Section 9.

We observe the five partitions represent five different microservices, respectively: 1) *init*, 2) *item*, 3) *register*, 4) *order*, and 5) *browse*. Each microservice is represented as a group of classes where each class has a mapping to a tuple of use cases. For example, in case of the *init* microservice, `ListOrdersController` and `ViewOrderController` are mapped with the ⟨init⟩ tuple whereas `SearchProductsController` is mapped with the ⟨init, search⟩ tuple. The mapping of a class with a tuple indicates that a class is invoked under one or more use cases present in the tuple. Based on overlapping use cases across tuples, we find classes under the *init* microservice are aligned with the *init* specific business functionality. Similarly in case of the *register* microservice, `SignonController` are mapped with the ⟨init, login_user⟩ tuple whereas `AccountValidator` is mapped with the ⟨ register_user, submit_user ⟩ tuple. Here, both the tuples may not have overlapping use cases, however, semantically both the tuples are related to the *register* microservice. Thus, we observe classes under the *register* microservice are aligned with the *register* specific business functionality.

Accordingly, the collection of tuples of use cases for each partition provides the explainability for the partition in terms of the business functionalities for users to comprehend the partitions' correctness.

3 EMPIRICAL EVALUATION

For the evaluation, we followed this general procedure: (1) we collected execution traces based on use cases, (2) we generated reduced paths using CCTs, and (3) we ran the implementation of our partitioning approach to generate partitions.

Table 2: Subject applications and use-case-based traces used in the evaluation.

Apps	Classes	Methods	#UC	Class Coverage	Method Coverage
DayTrader [2]	109	969	83	73 (66%)	428 (44%)
AcmeAir [3]	33	163	11	28 (84%)	108 (66%)
JPetStore [4]	66	350	44	36 (54%)	236 (67%)
Plants [5]	37	463	43	25 (67%)	264 (57%)
App1	82	449	15	50 (60%)	247 (55%)
App2	245	333	7	60 (24%)	280 (8%)
App3	1286	12,066	21	241 (19%)	1517 (12%)

3.1 Subject Applications

We used seven JEE applications for the evaluation, consisting of four open-source applications and three proprietary enterprise applications. Table 2 presents the data about the applications and use-case-based execution traces collected on the applications. We chose open-source applications for the following reasons. First, all of them are JEE web applications that are available as deployable and runnable applications. Second, they have a monolithic architecture. Third, they have been used in prior evaluations in academic research [13, 23]. The open-source applications are small and have class-coverage rates ranging from 66% to 88%. The proprietary applications are larger but have lower coverage rates, in particular for `App2`, and `App3`.

3.2 Baseline Techniques

We compare Mono2Micro with four baselines: FoSCI [23], CoGCN [13], Bunch [32] and MEM [31]. We selected them based on the following criteria. 1) their source code is available to replicate their methods; 2) they are well-known techniques from microservice identification (FoSCI, CoGCN, and MEM) and software remodularization (Bunch) research areas; and 3) they require minimal manual data preparation for usage. There are other relevant baselines such as ServiceCutter [20] that requires significant manual effort in generating the inputs such as the entity-relationship model (ERM) from an application. We realized that such effort is intractable and cannot be scaled to applications with more than 1000 classes.

- **FoSCI**[6] [23], creates functional atoms using a hierarchical clustering approach and then merges the atoms using a genetic algorithm to compute partition recommendations. For FoSCI, we considered both structural and conceptual connectivity.
- **CoGCN**[7][13] proposes an approach to partition a monolith applications by minimizing the effect of outlier classes that might be present in the embeddings of other classes. For CoGCN, we construct their three matrices: EP(i, p), C(i, j), and In(i, j). EP(i, p) suggests if a class i is present in an entry point p, C(i, j) suggests if two classes i and j are present in an entry point, and In(i, j) suggests if i and j related by the inheritance relationship.
- **Bunch**[8] [32] needs an external module dependency graph (MDG) as its input to generate partitions. For Bunch, we consider a version

[2]https://github.com/WASdev/sample.daytrader7
[3]https://github.com/acmeair/acmeair
[4]https://github.com/KimJongSung/jPetStore
[5]https://github.com/WASdev/sample.mono-to-ms.pbw-monolith

[6]https://github.com/wj86/FoSCI/releases
[7]https://github.com/utkd/cogcn
[8]https://github.com/ArchitectingSoftware/Bunch

of its hill-climbing algorithm. We considered the nearest-ascend hill climbing (NAHC) as suggested by Saeidi *et al.* [39].

• **MEM** [31][9] considers the minimum spanning tree (MST) approach that uses Kruskal's algorithm [27] for computing the minimum spanning trees. We consider their logical and semantic coupling strategies to generate partitions.

Based on the input data obtained from the subject applications using Mono2Micro, we created data converters to convert the input data to the format required by each of these four baselines.

3.3 Metrics

We provide five metrics to measure the effectiveness of partitions recommended using Mono2Micro.

• **SM** [23] measures the modularity quality of partitions as the structural cohesiveness of classes within a partition (m_i) (scoh) and coupling (scop) between the partitions (M). It is computed as $\frac{1}{M}\sum_{i=1}^{M} scoh_i - \frac{1}{(M(M-1))/2}\sum_{i \neq j}^{M} scop_{i,j}$. $scoh_i$ is computed as $\frac{\mu_i}{m_i^2}$ where μ_i refers the number of edges within a partition m_i and $scop_{i,j}$ is computed as $\frac{\sigma_{i,j}}{2*(m_i * m_j)}$ where $\sigma_{i,j}$ refers the number of edges between partitions m_i and m_j. Higher the value of SM, better is the recommendation.

• **ICP** [25] measures the percentage of runtime calls occurring between two partitions $icp_{i,j} = c_{i,j} / \sum_{i=1,j=1,i \neq j}^{M} c_{i,j}$, where $c_{i,j}$ represents the number of call between partition i and partition j. Lower the value of ICP, better is the recommendation.

• **BCP** [25] measures the average entropy[10] of business use cases per partition. The use cases for a partition consists of all use-case labels associated with its member classes. A partition is considered functionally cohesive if it implements a small set of use cases. BCP is computed as $\frac{1}{M}\sum_{i=1}^{M} bcp_i$, where bcp_i is computed as -$\sum_{j=1}^{m_i} \frac{1}{|m_i|}$ $log\,(\frac{1}{|m_i|})$ where $\frac{1}{|m_i|}$ is a vector of the size m_i where m_i represents the number of use cases for a partition $i \in M$. For example, given a partition with 3 use cases, $\frac{1}{|m_i|}$ is represented as [1/3, 1/3, 1/3]. Lower values for BCP indicate better recommendations.

• **IFN** [23] measures the number of interfaces in a microservice. IFN is computed as $\frac{1}{N}\sum_{i=1}^{N} ifn_i$ where ifn_i is the number of interfaces in a microservice where N is the total number of microservices. Lower values of IFN indicates better recommendations.

• **NED** [13] measures how evenly the size of a microservice is. It is measured as 1 - $\frac{\sum_{i=1, knotextreme}^{K} n_k}{|N|}$ where k ranges in {5, 20} [41]. Lower values of NED indicates better recommendations. NED was originally proposed by Wu *et al.* [48] to evaluate the extremity of a microservice distribution.

3.4 Hyperparameter Settings

For hyperparameter settings, we first consider the number of partitions to consider for each approach. Several approaches have been used in prior work for determining a partition size. Some of these require users to choose a cut-point [35, 49]: *i.e.,* a value between 0 and the maximum height of a dendogram obtained using the hierarchical-clustering algorithm. Other approaches provide a stopping criteria; *e.g.,* Jin *et al.* [23] use Jaccard distance values greater than three to merge clusters. Such approaches require users to determine a value for each application, which in practice the user may not always know. For our experiments, we chose to adopt the approach suggested by Scanniello *et al.* [42] where we take a range of cluster sizes (partition size values) starting from $\frac{N}{2}$, N > 0 and keep going downward to a value greater than 1 where N represents the number of classes. Here, for small applications ($N \leq 50$), we use a slower rate ($N - 2^i$), whereas for larger applications ($N \geq 100$), we consider $\frac{N}{2^i}$ where $i \geq 1$. The strategy is applicable to Mono2Micro, CoGCN, FoSCI, and MEM but not Bunch. Bunch does not provide an explicit option to provide a partition size as its input, rather it provides three agglomerative output options to generate partitions: top level, median level, and the detailed level. For FoSCI, we consider the *diff* as 3 for all the applications except for App$_2$ where the number of functional atoms flattened when *diff*=1. For other hyperparameters for FoSCI, Bunch, CoGCN, and MEM, we consider the values provided by the authors for each approach.

4 RESULTS

We compare Mono2Micro against four baselines using five evaluation metrics. We also conducted a survey of Mono2Micro with industry practitioners to get their feedback. In particular, our evaluation and the survey aims to address the following research questions:

RQ1: How does Mono2Micro perform based on the quality of partitioning using a set of metrics?
RQ2: How fast is Mono2Micro's partitioning?
RQ3: How helpful do industry practitioners find Mono2Micro in refactoring their monolithic applications?

4.1 Partitioning Quality (RQ1)

Table 4 and 5 present the comparison of Mono2Micro with FoSCI, CoGCN, Bunch and MEM for seven applications across five metrics. For each application, we created a range of partitions and obtained the score for all the metrics. We removed the outliers and computed median scores for each metric. For Bunch, considering only three partition values, the IFN score for `Daytrader` and the SM score for `Jpetstore` got omitted once we remove the outliers. Table 3 indicates the overall winners across all approaches.

Considering BCP and NED, Mono2Micro significantly outperformed other approaches as shown in Table 3. Mono2Micro winning in terms of BCP indicates that use-case-based partitions are more functionally cohesive. Mono2Micro winning for NED implies that the majority of the partitions generated by Mono2Micro contain 5 to 20 classes. The result is due to the non-parametric approach based on hierarchical clustering rather than multi-objective optimization and parametric methods like k-means that other baselines use. We observed that for App3, Mono2Micro lost to FoSCI in NED, indicating the possible adjustments for the NED constraints for larger applications.

Considering ICP and IFN, Mono2Micro performed better than other approaches. However, the performance does not hold across the majority of applications. In terms of ICP, Mono2Micro outperformed other approaches for `Daytrader`, `Jpetstore`, and `App2`. Followed by Mono2Micro, FoSCI performed better than other approaches for `App2` and `App3` and CoGCN for `Acmeair`. For `App2`, using FoSCI, we obtained a significantly lower ICP score, whereas

[9]https://github.com/gmazlami/microserviceExtraction-backend
[10]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.entropy.html

Table 3: Heatmap showing the overall winners among all the approaches (0 indicates the lowest score whereas 6 indicates the highest score). Here M2M is Mono2Micro.

	M2M	FoSCI	CoGCN	Bunch	MEM
BCP	5	0	1	1	0
ICP	3	2	1	0	1
SM	0	0	0	5	2
IFN	2	1	1	1	1
NED	5	2	1	0	0

the NED score obtained is significantly higher compared to the approaches. «ANUP: THIS antecedent »This suggests that high non-extreme distribution values might have led to monolithic partitions, thereby lowering the ICP scores. However, this did not hold for `Acmeair` and `App3` where CoGCN and FoSCI performed well for both ICP and NED, respectively.

For SM, Bunch outperformed all other approaches for `Daytrader`, `Acmeair`, `App1`, `App2`, and `App3` followed by MEM that outperformed other approaches for `Jpetstore`, `Plants`. Bunch internally uses a function that optimizes for cohesion and coupling based on internal and external edges, respectively. We assume that this might be the reason for high SM values for Bunch. Although, we observed that in the majority of the applications, the NED scores for Bunch are higher than other approaches that suggest non-extreme distribution. The result is due to Bunch's technique that might lead to obtaining large monolithic partitions at the cost of high SM.

Table 4: Evaluation results for all the open source applications in terms of median BCP, ICP, SM, IFN and NED scores obtained for a range of partitions. Here M2M is Mono2Micro.

Daytrader	M2M	FoSCI	CoGCN	Bunch	MEM
BCP	0.907	1.641	1.073	1.858	1.965
ICP	0.346	0.748	0.455	0.572	0.355
SM	0.078	0.092	0.086	0.269	0.089
IFN	1.922	3.489	2.880	–	4.200
NED	0.338	0.697	0.663	0.582	1.000
AcmeAir	**M2M**	**FoSCI**	**CoGCN**	**Bunch**	**MEM**
BCP	0.953	1.539	1.221	1.545	1.827
ICP	0.527	0.706	0.444	0.55	0.589
SM	0.072	0.095	0.038	0.177	0.097
IFN	3.375	4.375	2.846	3.875	4.333
NED	0.429	0.407	0.250	0.692	0.464
Jpetstore	**M2M**	**FoSCI**	**CoGCN**	**Bunch**	**MEM**
BCP	1.625	2.181	1.905	2.433	2.496
ICP	0.333	0.478	0.582	0.477	0.434
SM	0.054	0.044	0.091	–	0.124
IFN	1.857	3.750	2.533	7.948	3.429
NED	0.257	0.516	0.392	0.667	1.000
Plants	**M2M**	**FoSCI**	**CoGCN**	**Bunch**	**MEM**
BCP	1.690	2.593	2.338	2.902	1.902
ICP	0.381	0.682	0.571	0.501	0.320
SM	0.078	0.135	0.133	0.155	0.210
IFN	6.000	4.875	4.875	6.357	4.750
NED	0.038	0.538	0.500	0.346	0.231

Table 5: Evaluation results for all the web enterprise applications in terms of median BCP, ICP, SM, IFN and NED scores obtained for a range of partitions. Here M2M is Mono2Micro.

App1	M2M	FoSCI	CoGCN	Bunch	MEM
BCP	0.888	1.433	1.347	1.209	1.429
ICP	0.214	0.58	0.456	0.426	0.489
SM	0.184	0.143	0.061	0.281	0.216
IFN	2.750	5.100	3.923	2.933	5.400
NED	0.438	0.438	0.471	0.565	1.000
App2	**M2M**	**FoSCI**	**CoGCN**	**Bunch**	**MEM**
BCP	0.404	0.828	0.424	0.297	0.543
ICP	0.329	0.021	0.759	0.267	0.561
SM	0.137	0.119	0.060	0.238	0.137
IFN	1.625	0.333	3.050	1.786	3.607
NED	0.121	0.690	0.262	0.500	1.000
App3	**M2M**	**FoSCI**	**CoGCN**	**Bunch**	**MEM**
BCP	1.511	1.495	1.275	1.432	1.542
ICP	0.626	0.233	0.899	0.647	0.758
SM	0.133	0.045	0.029	0.270	0.09
IFN	11.548	39.000	17.500	6.921	7.879
NED	0.921	0.890	1.000	0.934	1.000

Table 6: The median time in seconds required to generate partitions for all the applications using different approaches. Here M2M is Mono2Micro.

Apps	M2M	FoSCI	CoGCN	Bunch	MEM
Daytrader	0.604	2601	23.06	0.045	1.680
AcmeAir	0.049	181.4	10.92	0.044	1.620
Jpetstore	0.103	210.9	14.41	0.030	1.600
Plants	0.049	87.24	11.01	0.042	3.500
App1	0.133	718.3	17.04	0.040	1.634
App2	0.185	6.292	19.83	0.043	1.660
App3	5.005	6886	38.45	0.062	2.570

4.2 Runtime (RQ2)

Table 6 shows the median time in seconds taken by each approach to generate partitions. We compared the approaches to find that Mono2Micro takes significantly less time than FoSCI, CoGCN, and MEM to generate partitions. Bunch with the hill-climbing approach takes the least amount of time. In addition, we find that FoSCI with a genetic algorithm takes the most amount of time followed by CoGCN that takes a neural network approach, and MEM, which takes the minimum spanning tree approach.

4.3 User Survey (RQ3)

We surveyed industry practitioners to understand how they perceive Mono2Micro. For the survey, we created a questionnaire with 20 questions adopted from existing surveys [17, 26, 46, 50]. First, we conducted a pilot study with 4 participants to refine the questionnaire and estimate the total time required to complete the survey. Next, we sent out the survey questionnaire to 32 participants who have tried Mono2Micro. Among the 32 participants, 21 participants returned the survey results. The participants belonged to the following job roles: 1) technical sales (21.1%), 2) software architect (21.1%), 3) software developer (15.8%), and 4) others. Considering software industry experience, 1) 84.2% participants have 10+ years of experience, 2) 10.5% participants with 5-10 years of experience, and 3) the rest with 1-3 years. For microservices development, 1) 36.8% participants have more than five years of experience, 2) 26.3% participants with three years of experience, 3) 15.8% participants with one year of experience, 4) 10.5% participants with two years of experience, and 5) 10.5% participants with four years of experience. We asked the participants questions as shown in Table 7. We provide their response below.

4.3.1 Preliminary Information. In terms of preliminary information, we observed the following. 1) Based on Q1, most participants (42.1%) have used Mono2Micro for 1-2 months. 2) Based on Q2, we observed most participants used it on a few applications to date, e.g., 36.8% on 2-3 applications and 31.6% of the participants on one application. 3) Based on Q3, we find 35.8% of the participants have used Mono2Micro for sample applications. The responses to Q1, Q2, and Q3, respectively, are expected since Mono2Micro was recently made generally available (GA) in early 2021. 4) Based on Q4, most participants (33.3%) relied on Mono2Micro's user guide. 5) Based on Q5, the majority of the participants mentioned they did not use any tools for refactoring before Mono2Micro. One participant mentioned about CAST [11] and ADDI [12] whereas another participant mentioned about the Transformation Advisor tool [13]. We find the response interesting, considering there are plenty of refactoring tools available from academia. The tool availability aspect deserves further study. Our current hypothesis is that it is important to have active product support to gain popularity. Users are looking for tools that can support their modernization methodologies, as we will discuss shortly.

Lesson 1: Enterprise users are inclined toward using supported industry tools.

4.3.2 Architectural Decisions. In terms of architectural decisions, we observed the following responses. Based on Q6, most participants (57.1%) agreed that Mono2Micro helps to implement the Strangler pattern. The response is expected considering Mono2Micro allows users to refactor their applications incrementally. The response also corroborates Fritzsch *et al.*'s [17] study where 7 of 9 cases they analyzed were re-written applying the Strangler pattern. Based on Q7, we find that most participants (41.7%) are undecided if Mono2Micro helps them to implement DDD patterns. This partially correlates with Fritzsch *et al.* [17] findings. Though DDD has been cited frequently in the literature, only 3 of the 16 participants in Fritzsch *et al.*'s [17] study reported using it. Based on Q8, most participants (53.8%) agree that partitions created by Mono2Micro are aligned with the business functionality of their applications.

Based on Q9, we find the following. 1) In the case of structural relations [26], most participants (64.3%) considered it extremely important. 2) In the case of semantic relations [26], most participants considered it neutral (35.7%). 3) In the case of evolutionary relations [26], most participants (50.0%) considered it extremely important.

Aside from run traces and use cases, Q10 lists other reported factors for consideration [26]: structural (static call graphs), semantic (class name similarity), and evolutionary relationships (change history, commit similarity, and contributor similarity). In addition, we added database interaction patterns and database transactions that are also considered important in terms of decomposing applications. Based on Q10, we obtained the following responses from participants for each factor. 1) For static call graphs, most participants (50%) considered it extremely important. 2) For class name similarity, most participants (35.7%) considered it neutral. 3) For change history, most participants (38.5%) considered it neutral. 4) For commit similarity, most participants (42.9%) considered it neutral. 5) For contributor similarity, most participants (42.9%) considered it neutral. 6) For database interaction patterns, most participants (64.3%) considered it extremely important. 7) For database transactions, most participants (64.3%) considered it extremely important. Overall we find that participants consider database interaction patterns and database transactions as the most important factors for refactoring followed by static call graphs.

Lesson 2: Mono2Micro is helpful in implementing the Strangler pattern
Lesson 3: Database interaction patterns and database transactions should be added to enhance Mono2Micro's partitioning strategy.

4.3.3 Running Mono2Micro. Considering running Mono2Micro, we observed the following responses. Based on Q11, most participants (50%) agreed that Mono2Micro supports independent or mutually exclusive business functionalities. Based on Q12, we obtained the following reasons for dependencies in business functionalities: 1) strong coupling, 2) inheritance and database interactions, 3) interdependent operations, and 4) shared classes underlying technical components. Based on Q13, most participants affirmed that Mono2Micro provides a new perspective to their applications. For example, one participant responded that interactions among application classes got clearer due to the Mono2Micro's recommendations. Based on Q14, most participants (78.6%) manually executed the use cases. The manual effort is required since several legacy monoliths may not have sufficient coverage of automated tests aligned with business functionalities. Based on Q15, we observed that most participants (54.5%) responded that the use cases and unobserved classes align with their expectations. Most participants (54.5%) participants responded that a gap in existing test use cases coverage was found. Most participants (63.6%) mentioned that they found potentially dead or unreachable code using Mono2Micro.

Lesson 4: Mono2Micro provides a new perspective to clients' applications in terms of understanding their applications business functionalities.
Lesson 5: Legacy monoliths may not have sufficient automated tests aligned with business functionalities
Lesson 6: Mono2Micro can reveal potentially dead or unreachable code

4.3.4 Explainability, Configurations & Performance. Based on Q16, most participants responded by saying the "explainability" of partitions provided by Mono2Micro is valuable. Based on Q17, most

[11]https://www.castsoftware.com/products/code-analysis-tools
[12]https://www.ibm.com/products/app-discovery-and-delivery-intelligence
[13]https://www.ibm.com/garage/method/practices/learn/ibm-transformation-advisor

Table 7: Survey questions categorized into four groups.

Category	Questions	Response Format	Free Text?	Responses
Preliminary Information	Q1. How long have you used Mono2Micro?	MCQ	×	Fig. 3a
	Q2. How many applications have you analyzed with Mono2Micro?	MCQ	×	Fig. 3b
	Q3. What kinds of applications have you analyzed with Mono2Micro	MCQ	×	Fig. 3c
	Q4. What kind of supporting resources for Mono2Micro have you used?	MCQ	×	Fig. 3d
	Q5. Have you used applications other than Mono2Micro? If so, which one?	open	✓	Fig. 3e
Architectural Decisions	Q6. Is Mono2Micro helpful in implementing the Strangler pattern in decomposing apps?	1 to 5 scale	×	Fig. 3f
	Q7. Is Mono2Micro helpful in implementing a Domain-Driven Design (DDD) Pattern in decomposing apps?	1 to 5 scale	×	Fig. 3g
	Q8. Do the partitions created by Mono2Micro are aligned with business functionality of the apps?	1 to 5 scale	×	Fig. 3h
	Q9. Which of the following relationship types are most important to decomposing your app? a. Structural b. Semantic c. Evolutionary	1 to 5 scale	×	·
	Q10. Which of the following factors Mono2Micro should consider apart from runtime traces and use cases? a. static call graphs, b. class name similarity, c. change history, d. commit similarity, e. contributor similarity, f. database interaction patterns, g. database transactions	1 to 5 scale	×	·
Running Mono2Micro	Q11. The partitions generated by Mono2Micro support independent or mutually exclusive business functionalities?	1 to 5 scale	×	Fig. 3i
	Q12. If dependencies exist in business functionalities across partitions, what are the main reasons?	open	✓	·
	Q13. Did Mono2Micro's recommendation provides new perspective to your application that you find useful?	open	✓	·
	Q14. When collecting traces did you execute the application use cases manually or did you have an automated suite of functional test cases?	MCQ	×	Fig. 3j
	Q15. Which experiences with use cases and Mono2Micro matches your experience?	MCQ	×	Fig. 3k
Explanability, Configuration, Performance	Q16. Did you find the "explainability" of partitions, as indicated by use-case labels, to be valuable?	1 to 5 scale	×	Fig. 3l
	Q17. How many and what kind of changes did you make to the original partition suggestions?	MCQ	×	Fig. 3m
	Q18. What kinds of changes did you make?	MCQ	×	Fig. 3n
	Q19. When you chose the number of partitions what was more valuable to you? Why?	MCQ	×	Fig. 3o
	Q20. Is Mono2Micro is fast enough to generate recommendations that it does not slow down my workflow?	1 to 5 scale	×	Fig. 3p

participants (38.5%) responded that they did some minor changes and some major changes. Q18 is a follow-up question based on Q17. Based on Q18, we found that most of the participants (31.8%) suggested that they moved classes between the recommended partitions. Based on Q19, most participants (50.0%) used the default partition value provided by Mono2Micro. Q20 is a follow-up question for Q19. Based on Q20, we found that a participant went with the default value of 5 to avoid too many microservices. Another participant responded that he/she went with a value of more than 5 since the customer was expecting more than 5 partitions. One participant responded that he/she chose a value larger than 5 since their application is relatively large with multiple domain services. The responses indicate that the partition size is dependent on the domain knowledge of applications. Based on Q21, most participants (41.7%) agree that Mono2Micro is fast enough to generate recommendations.

Lesson 7: Domain knowledge of an application is needed to chose the appropriate partition size.
Lesson 8: Mono2Micro's explainability of partitions in terms of use cases is valuable to users.

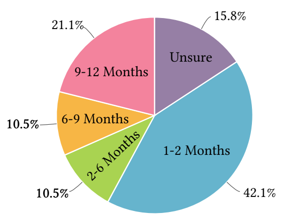
5 DISCUSSION

Summary of RQ1 and RQ2. In terms of empirical evaluation (Section 3), we observed that Mono2Micro performs well across most of the metrics and applications. The BCP and NED Mono2Micro outperformed other baselines, whereas, for ICP and IFN, the performance was competitive with a slight edge over other approaches. For SM, it lost to both Bunch and MEM; however, we also observed higher SM values lead to higher NED scores. The result needs further investigation to understand the relationship between SM and other metrics. In terms of time required Mono2Micro again lost to Bunch; however, it significantly outperformed other approaches.

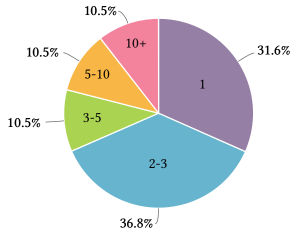
Summary of RQ3. In terms of survey (Section 4.3), we observed that Mono2Micro was beneficial in several cases. 1) It helps implement the Strangler pattern; the partitions generated by Mono2Micro align with the applications' business functionality. 2) It made the interaction among classes more evident. 3) It helped users to find potentially unreachable or dead code. 4) It discovered the gap between test cases coverage. 5) It produces explainable partitions. The survey also provided further scope for improvement, such as to 1) consider static call graphs in addition to runtime traces, 2) consider database interactions and transaction patterns to improve partitioning, 3) minimize the changes required post-recommendations.

6 THREATS TO VALIDITY

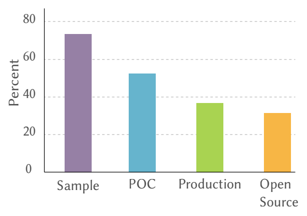
Although the empirical evaluation and the survey show the effectiveness of Mono2Micro, there are threats to the validity of our results. The most significant of these are threats to external validity, which arise when the observed results cannot be generalized to other experimental setups. Our evaluation included seven applications with varying use cases and code coverage. Therefore, we can draw limited conclusions on how our results might generalize to other applications, use cases, and coverage. Although our subjects have considerable variations in number and granularity of use cases and coverage achieved by the use cases, the effect of application



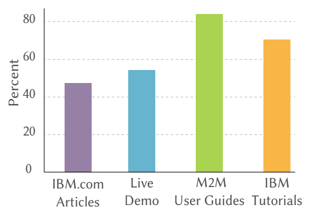
(a) Mono2Micro usage time (Q1)



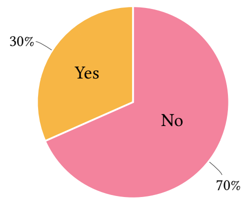
(b) No. applications assessed (Q2)



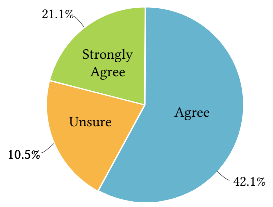
(c) Assessed application types (Q3)



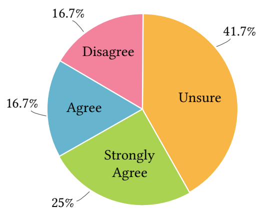
(d) Supporting resource used (Q4)



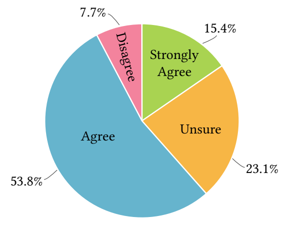
(e) Mono2Micro alternatives used (Q5)



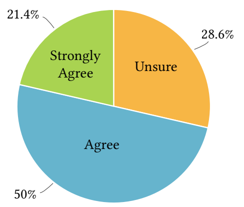
(f) Strangler pattern observed (Q6)



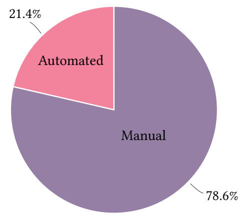
(g) DDD pattern observed (Q7)



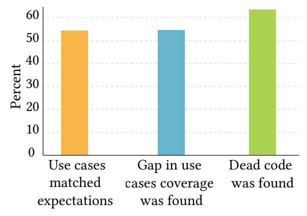
(h) Mono2Micro partitions align with business functions (Q8)



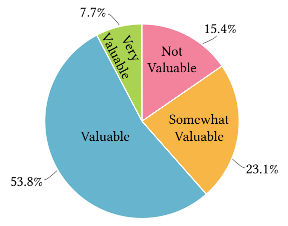
(i) Partitions were independent and/or mutually exclusive of one another (Q11)



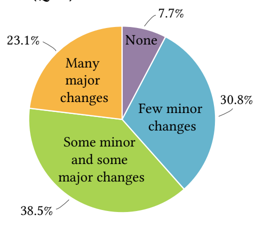
(j) Trace collection strategy (Q14)



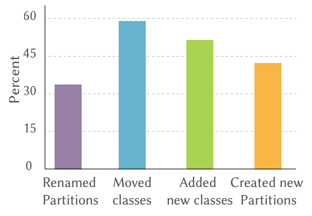
(k) Experiences with use cases and Mono2Micro (Q15)



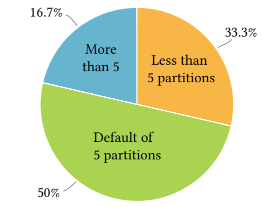
(l) Value of use-case labels (Q16)



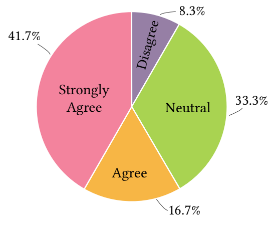
(m) Changes to partitions from Mono2Micro (Q17)



(n) Types of changes (Q18)



(o) Number of partitions (Q19)



(p) Mono2Micro does not slow down workflow (Q20)

Figure 3: Survey responses from participants for Q1 to Q21 given in Table 7.

decomposition is an aspect that requires further experimentation and investigation.

Threats to internal validity may be caused by bugs in Mono2Micro, our experimental infrastructure, and data-collection scripts. We mitigated the threat by adding validation scripts and providing appropriate error messages. For the survey, we have limited the number of participants (21) with varying degrees of job roles and experiences who completed the survey. We can address the lack of participants by creating an extensive study group to find more general results. Additionally, since Mono2Micro was generally available in January 2021, many participants did not get a chance to use it for many production applications. We think the use of Mono2Micro on a large number of production applications by survey participants could have possibly shown results favoring Mono2Micro in generating partitions for production applications.

7 RELATED WORK

In this section, first, we discuss the techniques that are most related to ours. Then, we discuss selected contributions from the software decomposition and service extraction.

Software Remodularization. Microservice decomposition is a newer instance of the long-standing problem of software (re)modularization and clustering, which has seen a long line of work (*e.g.*, [4, 7–9, 14, 22, 29, 30, 32, 33, 35, 37, 39, 40, 43, 47, 49]). We discuss select techniques from this body of work, observing that our approach, unlike the existing techniques, applies clustering on execution traces generated using functional use cases. In addition, we leverage the temporal relations as indirect call relations to generate partitions.

Commonly investigated approaches in modularization build a module dependence graph (MDG) using various types of dependence relations and then apply clustering or evolutionary algorithms to compute partitions based on different similarity metrics and objective functions. For example, Doval et al. [14] and Mitchell and Mancoridis [32] apply genetic algorithms to the MDG to optimize a metric based on cohesion and coupling. Mahdavi et al. [29] investigate multiple hill climbing for software clustering. Xiao and Tzerpos [49] consider runtime calls and associate weights with edges in the MDG. Bavota et al. [7] analyze information flowing into and out of a class via parameters of method calls; they also infer semantic information from comments and identifiers. Much of this work combines multiple goals into a single objective function, but several multi-objective formulations of modularization have been presented as well (*e.g.,* [1, 6, 33, 37]).

Decomposition via Dynamic Traces. Patel et al. [35] present a decomposition technique that applies hierarchical clustering over execution traces. Their approach performs clustering over a matrix in which rows represent classes, columns represent features, and each cell has a boolean value indicating whether a class occurs in a trace. Jin et al. [23, 24] present a technique for identifying candidate microservices that uses execution traces collected from functional test cases. Their approach first performs function atom generation, applying hierarchical clustering based on occurrences of classes in execution traces [24] followed by the application of a genetic algorithm to merge such atoms. De Alwis et al. [12] propose an approach that recommends microservices at the level of class methods. For recommendations, they rely on execution traces generated from use cases and database tables. For generating partitions, they use an approach that computes subgraphs from a given graph.

Other Decomposition Techniques. Several other techniques have been presented on software decomposition for microservice extraction (*e.g.,* [2, 5, 10, 11, 15, 18, 28, 31, 38, 45]). A couple of survey papers [18, 36] provide an overview of recent work on this topic.

Escobar *et al.* [15] present a rule-based approach for clustering highly-coupled classes in JEE applications; their approach considers entity beans (representing data) and their relationships to session beans in the business tier of the application. Levcovitz *et al.* [28] propose an approach that analyzes control flow through application tiers—from the presentation tier to the database tables to generate candidate microservices. Mazlami et al. [31] present a graph-based clustering approach for identifying microservices. They use four different extraction strategies based on change history and developer contribution to the codebase.

Other approaches for microservice decomposition match terms in OpenAPI specifications against a reference vocabulary [5], leverage domain-driven design and entity-relationship models [21], use manually constructed data-flow diagrams [11], and include security and scalability requirements [2]. Ren et al. [38] apply k-means clustering over combined static and runtime call information to generate microservices. Taibi and Systä [45] apply process mining on runtime logs files to construct call graphs for partitioning.

8 POTENTIAL ETHICAL IMPACT

We consider the current contribution does not pose any societal or ethical impact.

9 DATASETS

We have released the datasets[14] for Mono2Micro and baselines. Additionally, we provide the Python-based data converters to convert Mono2Micro's dataset to the formats required by other baselines.

10 CONCLUSION

The paper provided an approach that recommends microservices from legacy applications. The approach captures and preserves the temporal relationships; it uses the relationships to group classes into disjoint partitions. Our experimental studies show the efficacy of our approach when compared with the baselines.

In the future, we plan to continue our investigation, expand the quality metrics and provide further guidance to create efficient use cases for the practitioners. We are conducting extensive verification and validation of our approach by trying it against large enterprise real-life applications in production for several years in various industry sectors. Based on the lesson learned from the survey, we plan to take the following directions: 1) Add database interaction and transaction patterns to refine Mono2Micro's recommendation. 2) Automate test case generation for legacy monoliths to generate runtime traces. 3) Automate the generation of a partition size for a legacy application. 4) Redefine NED constraints for larger applications. 5) Finally, how we can improve the explainability of partitions further.

ACKNOWLEDGMENTS

We are grateful to Troy Bjerke for helping us with the survey. We would like to thank our partners at IBM Hybrid Cloud (Melissa Modjeski, Laura Scott, Dana Price, Erin Heximer) and the entire Mono2Micro development team. We are grateful to our colleagues at IBM Research Chen Lin, John Rofrano, Shivali Agarwal, Amith Singhee, Srikanth Tamilselvan, Yasuharu Katsuno, Fumiko Satoh, Nicholas Fuller, and Ruchir Puri for their valuable suggestions and feedback. Finally, we thank Julia Rubin, Evelien Boerstra, and Lisa Kirby at the University of British Columbia and Tim Menzies, Rahul Yedida, Munindar P. Singh, and Arvind Kumar at North Carolina State University for their valuable feedback on this work.

REFERENCES

[1] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse. 2013. Towards automatically improving package structure while respecting original design decisions. In *Proceedings of the 20th Working Conference on Reverse Engineering*. IEEE, Koblenz, Germany, 212–221.
[2] Mohsen Ahmadvand and Amjad Ibrahim. 2016. Requirements Reconciliation for Scalable and Secure Microservice Decomposition. In *Proceedings of the International Requirements Engineering Conference Workshops*. IEEE, Beijing, 68–73.
[3] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 85–96.
[4] N. Anquetil and T. C. Lethbridge. 1999. Experiments with clustering as a software remodularization method. In *Sixth Working Conference on Reverse Engineering*. IEEE, Atlanta, Georgia, 235–255.

[14]**https://github.com/kaliaanup/mono2micro-fse-industry-track-2021**

[5] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices Identification Through Interface Analysis. In *Proceedings of European Conference on Service-Oriented and Cloud Computing*. Springer, Norway, 19–33.

[6] Marcio de Oliveira Barros. 2012. An Analysis of the Effects of Composite Objectives in Multiobjective Software Module Clustering. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. ACM, Philadelphia, USA, 1205–1212.

[7] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. 2010. Software Re-Modularization Based on Structural and Semantic Metrics. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE, Beverly, MA, 195–204.

[8] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. 2014. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *ACM Trans. Softw. Eng. Methodol.* 23, 1 (Feb. 2014), 33 pages. https://doi.org/10.1145/2559935

[9] G. Caldiera and V. R. Basili. 1991. Identifying and qualifying reusable software components. *Computer* 24, 2 (1991), 61–70.

[10] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rodrigo Bonifácio, Leonardo P. Tizzei, and Thelma Elita Colanzi. 2019. Extraction of Configurable and Reusable Microservices from Legacy Systems: An Exploratory Study. In *Proceedings of the 23rd International Systems and Software Product Line Conference*. ACM, Paris, France, 26––31.

[11] Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *Proceedings of 24th Asia-Pacific Software Engineering Conference*. IEEE, Nanjing, Jiangsu, 466–475.

[12] Adambarage Anuruddha Chathuranga De Alwis, Alistair Barros, Artem Polyvyanyy, and Colin Fidge. 2018. Function-Splitting Heuristics for Discovery of Microservices in Enterprise Systems. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, Hangzhou, 37–53.

[13] Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. 2021. Graph Neural Network to Dilute Outliers for Refactoring Monolith Application. In *Proceedings of Association for the Advancement of Artificial Intelligence*. AAAI, virtual, 72–80.

[14] D. Doval, S. Mancoridis, and B. S. Mitchell. 1999. Automatic Clustering of Software Systems Using a Genetic Algorithm. In *Proceedings of Software Technology and Engineering Practice*. IEEE, Pittsburgh, PA, 73.

[15] Daniel Escobar, Diana Cardenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra, and Rubby Casallas. 2016. Towards the understanding and evolution of monolithic applications as microservices. In *Proceedings of Latin American Computing*. IEEE, Chile, 1–11.

[16] Martin Fowler. 2019. Microservices Guide. https://martinfowler.com/microservices/

[17] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In *Proceedings of International Conference on Software Maintenance and Evolution*. IEEE, Cleveland, US, 481–490.

[18] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. 2018. From Monolith to Microservices: A Classification of Refactoring Approaches. http://arxiv.org/abs/1807.10059

[19] Andreas Fuhr, Tassilo Horn, and Volker Riediger. 2011. Using Dynamic Analysis and Clustering for Implementing Services by Reusing Legacy Code. In *Proceedings of 18th Working Conference on Reverse Engineering*. IEEE, NW Washington, DC, 275–279.

[20] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *Service-Oriented and Cloud Computing*. Springer, Cham, 185–200.

[21] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *Proceedings of European Conference on Service-Oriented and Cloud Computing*. Springer, Vienna, 185–200.

[22] D. H. Hutchens and V. R. Basili. 1985. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering* 11, 8 (1985), 749–757.

[23] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. 2019. Service Candidate Identification from Monolithic Systems based on Execution Traces. *IEEE Transactions on Software Engineering* 47, 5 (Apr 2019), 1–21.

[24] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. 2018. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *Proceedings of IEEE International Conference on Web Services*. IEEE, San Francisco, 211–218.

[25] Anup K. Kalia, Chen Lin, Jin Xiao, Saurabh Sinha, John Rofrano, Maja Vukovic, and Debasish Banerjee. 2020. Mono2Micro: An AI-based Toolchain for Evolving Monolithic Enterprise Applications to a Microservice Architecture. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Sacramento, 1606–1610.

[26] Lisa J. Kirby, E. Boerstra, Z. J.C. Anderson, and J. Rubin. 2021. Weighing the Evidence: On Relationship Types in Microservice Extraction. In *IEEE/ACM International Conference on Program Comprehension*. IEEE, virtual, 1–11.

[27] Joseph B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *American Mathematical Society* 7, 1 (1956), 45–80.

[28] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. 2016. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. http://arxiv.org/abs/1605.03175

[29] K. Mahdavi, M. Harman, and R. M. Hierons. 2003. A multiple hill climbing approach to software module clustering. In *Proceedings of International Conference on Software Maintenance*. IEEE, Amsterdam, Netherlands, 315–324.

[30] O. Maqbool and H. Babri. 2007. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering* 33, 11 (2007), 759–780.

[31] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *Proceedings of International Conference on Web Services*. IEEE, Honolulu, 524–531.

[32] B. S. Mitchell and S. Mancoridis. 2006. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering* 32, 3 (2006), 193–208.

[33] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and Ali Ouni. 2015. Many-Objective Software Remodularization Using NSGA-III. *ACM TOSEM* 24, 3 (May 2015), 1–45.

[34] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc., Boston, MA.

[35] Chiragkumar Patel, Abdelwahab Hamou-Lhadj, and Juergen Rilling. 2009. Software Clustering Using Dynamic Analysis and Static Dependencies. In *Proceedings of 13th European Conference on Software Maintenance and Reengineering*. IEEE, Kaiserslautern, 27–36.

[36] F. Ponce, G. Márquez, and H. Astudillo. 2019. Migrating from monolithic architecture to microservices: A Rapid Review. In *Proceedings of the ICCCSS*. IEEE, Concepcio'n, Chile, 1–7.

[37] K. Praditwong, M. Harman, and X. Yao. 2011. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering* 37, 2 (2011), 264–282.

[38] Zhongshan Ren, Wei Wang, Guoquan Wu, Chushu Gao, Wei Chen, Jun Wei, and Tao Huang. 2018. Migrating Web Applications from Monolithic Structure to Microservices Architecture. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. ACM, Beijing, 1–10.

[39] A. M. Saeidi, J. Hage, R. Khadka, and S. Jansen. 2015. A search-based approach to multi-view clustering of software systems. In *Proceedings of IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, Montreal, Canada, 429–438.

[40] G. Santos, M. T. Valente, and N. Anquetil. 2014. Remodularization analysis using semantic clustering. In *Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE, Antwerp, Belgium, 224–233.

[41] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. 2010. An Approach for Architectural Layer Recovery. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, Sierre, Switzerland, 2198—-2202.

[42] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. 2010. Using the Kleinberg Algorithm and Vector Space Model for Software System Clustering. In *Proceedings of IEEE 18th International Conference on Program Comprehension*. IEEE, Braga, Portugal, 180–189.

[43] R. W. Schwanke. 1991. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 83–92.

[44] Robin Sibson. 1973. SLINK: An optimally efficient algorithm for the single-link cluster method. *Comput. J.* 16, 1 (jan 1973), 30–34.

[45] Davide Taibi and Kari Systä. 2019. From monolithic systems to microservices: a decomposition framework based on process mining. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER)*. Springer, Heraklion, Greece, 1–12.

[46] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. 2021. Promises and Challenges of Microservices: an Exploratory Study. *Empirical Software Engineering* 1, 1 (2021), 1–45.

[47] T. A. Wiggerts. 1997. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE, Amsterdam, Netherlands, 33–43.

[48] Jingwei Wu, Ahmed E. Hassan, and Richard C. Holt. 2005. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of IEEE International Conference on Software Maintenance*. IEEE, Budapest, 525–535.

[49] Chenchen Xiao and Vassilios Tzerpos. 2005. Software Clustering based on Dynamic Dependencies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering*. IEEE, Manchester, 124–133.

[50] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. 2019. Microservice Architecture in Reality: An Industrial Inquiry. In *IEEE International Conference on Software Architecture*. IEEE, Hamburgh, 51–60.

[51] Junfeng Zhao, Jiantao Zhou, Hongji Yang, and Guoping Liu. 2015. An Orthogonal Approach to Reusable Component Discovery in Cloud Migration. *China Communications* 12, 5 (2015), 134–151.