DistMeasure: A Framework for Run-Time Characterization and Quality Assessment of Distributed Software via Interprocess Communications

XIAOQIN FU, Washington State University, USA
ASIF ZAMAN, Washington State University, USA
HAIPENG CAI*, University at Buffalo, SUNY, USA

A defining, unique aspect of distributed systems lies in interprocess communication (IPC) through which distributed components interact and collaborate toward the holistic system behaviors. This highly decoupled construction intuitively contributes to the scalability, performance, and resiliency advantages of distributed software, but also adds largely to their greater complexity, compared to centralized software. Yet despite the importance of IPC in distributed systems, little is known about *how to quantify IPC-induced behaviors in these systems through IPC measurement and how such behaviors may be related to the quality of distributed software.* To answer these questions, in this paper, we present DistMeasure, a framework for measuring distributed software systems via the lens of IPC hence enabling the study of its correlation with distributed system quality. Underlying DistMeasure is a novel set of IPC metrics that focus on gauging the coupling and cohesion of distributed processes. Through these metrics, DistMeasure quantifies relevant run-time characteristics of distributed systems and their quality relevance, covering a range of quality aspects each via respective direct quality metrics. Further, DistMeasure enables predictive assessment of distributed system quality in those aspects via learning-based anomaly detection with respect to the corresponding quality metrics based on their significant correlations with related IPC metrics. Using DistMeasure, we demonstrated the practicality and usefulness of IPC measurement against 11 real-world distributed systems and their diverse execution scenarios. Among other findings, our results revealed that IPC has a strong correlation with distributed system complexity, performance efficiency, and security. Higher IPC coupling between distributed processes tended to be negatively indicative of distributed software quality, while more cohesive processes have positive quality implications. Yet overall IPC-induced behaviors are largely independent of the system scale, and higher (lower) process coupling does not necessarily come with lower (higher) process cohesion. We also show promising merits (with 98% precision/recall/F1) of IPC measurement (e.g., class-level coupling and process-level cohesion) for predictive anomaly assessment of various aspects (e.g., attack surface and performance efficiency) of distributed system quality.

CCS Concepts: • **Software and its engineering → Maintaining software**.

Additional Key Words and Phrases: distributed system, interprocess communication, dynamic metrics, software quality

ACM Reference Format:
Xiaoqin Fu, Asif Zaman, and Haipeng Cai. 2024. DistMeasure: A Framework for Run-Time Characterization and Quality Assessment of Distributed Software via Interprocess Communications . *ACM Trans. Softw. Eng. Methodol.* 1, 1 (November 2024), 52 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Haipeng Cai is the corresponding author.

Authors' addresses: Xiaoqin Fu, Washington State University, Washington, 99163, Pullman, USA; Asif Zaman, Washington State University, Washington, 99163, Pullman, USA; Haipeng Cai, University at Buffalo, SUNY, New York, 14228, Buffalo, USA, haipengc@buffalo.edu.

1 INTRODUCTION

In response to growing societal and scientific demands on various data storage and computation tasks, modern software systems are increasingly distributed by design. These systems (e.g., financial management software, supply chain coordination services, and web search engines) are meritorious in exploiting decentralized and scalable computing infrastructures and resources [34] for high performance-efficiency, resiliency, and scalability. Due to their critical roles, assuring the quality of these systems is of paramount importance. And one common pathway to that end is through software measurement. Indeed, measuring a software system, especially in relation to its quality, plays an essential role in quality assurance of the system because it enables quality assessment and hence guides quality improvement [54, 74].

In particular, in the most common type of distributed systems [34], the constituent components[1] are located at separate machines hence communicate only through network-based interprocess communication (IPC). From an architectural perspective, IPC is a primary trait that differentiates distributed systems from centralized software, hence a defining, unique aspect of the behaviors of distributed software. Intuitively, while this trait contributes to the scalability, performance efficiency, and resiliency advantages of distributed systems, IPC also largely adds to the complexity hence potential quality assurance challenges for such systems. Thus, given the recognized role that software measurement plays in quality assurance [86], it is natural to assess distributed system quality with respect to IPC-induced behaviors—via IPC measurement. Yet despite the rich literature on software measurement in general, most existing works only address centralized software [9, 31, 41, 60, 61, 72]. A few addressed distributed systems, yet with a scope other than the software itself (e.g., monitoring system environments [142] or user dynamics [140]). As a result, very little has been studied [71] or known about *"what are the characteristics of IPC-induced behaviors in distributed systems and how are these characteristics related to the quality of those systems?"*

To answer these overarching questions, we present DistMeasure[2], a framework for characterizing IPC-induced behaviors of distributed systems via IPC measurement hence enabling our exploratory study on the relevance of such characteristics to distributed system quality. Underlying the framework is a set of six IPC metrics aiming to measure the run-time communication structure, complexity, and reusability of common distributed systems [34]. Specifically, DistMeasure measures the *coupling* between concurrent, distributed processes, as well as the *cohesion* of individual processes, at method, class, component/process, and whole-system levels, in terms of message passing and method-level dynamic dependencies across process boundaries. Using these measurement/characterization capabilities, DistMeasure then allows for examining the quality relevance of IPC via statistical analyses, revealing correlations between the (six) IPC metrics and (eight) existing quality metrics that directly measure nine quality sub-characteristics/sub-factors in four quality characteristics/factors in the ISO/IEC 25010 software quality model. Finally, based on significant correlations, DistMeasure further enables *quality anomaly* detection with respect to those direct quality metrics via both supervised and unsupervised learning to meet diverse use-scenario needs.

We have implemented DistMeasure for Java and applied it to 11 real-world distributed software applications, mostly enterprise-scale systems, in varied operation scenarios (with respect to a large number of run-time inputs that drive distinct executions). Our results revealed, among other findings, that (1) the six IPC metrics are generally independent of each other—no metric subsumes or consistently tells about another; (2) IPC characteristics of a distributed system, in terms of any of the IPC metrics, are not tied to the scale (code size) of the system; (3) higher (lower) process coupling does not necessarily imply lower (higher) process cohesion; (4) IPC-induced behaviors have strong relevance to the security, complexity, and performance efficiency of distributed systems; and (5) high IPC coupling generally has negative

[1] We define a distributed system component as *the code entities that run in a process that is separate (decoupled) from others.*
[2] Short for *Distributed software system Measurement via IPC.*

correlations with quality while high IPC cohesion has positive correlations. In addition, the statistical quality anomaly detection models in DistMeasure achieved 88.2% precision, 66.7% recall, and 74.5% F1 accuracy on average when using IPC metrics to detect anomalous quality status via unsupervised learning. With supervised learning, those models achieved on average 98% precision, recall, and F1. Some of the IPC metrics contributed more to the anomaly detection for certain quality metrics than others, as generally justified by the varying correlation strengths between the two metric classes. Further association studies revealed that (1) low class-level interprocess coupling is strongly associated with *normal* performance efficiency and (2) low cross-process communication loads is strongly associated with *abnormal* attack surface, so is low process cohesion.

Notably, since each process is the run-time instance of its corresponding component and components in distributed software are decoupled, by measuring the IPC coupling/cohesion we essentially reveal the hidden/implicit coupling/cohesion of corresponding distributed components. While coupling and cohesion have been extensively studied among various types of software metrics [135], static or dynamic, they have not yet been addressed as regards IPC during the execution of distributed systems. Our technical approaches enable practical measurement of IPC characteristics in real-world, industry-scale distributed systems, as well as predictive assessment of their quality with respect to a wide range of concerns (quality metrics)—which may not be always readily practical or even feasible to measure directly. Our empirical results, and the insights distilled from those results as presented in our extension discussions, provide new knowledge and understanding about the characteristics of IPC-induced behaviors in distributed systems and the relationships between IPC characteristics and distributed software quality. Since DistMeasure is based on the statistical analyses and learning-based predictive assessment, it only addresses *associations and correlations, not causation*. Thus, we never claim that distributed system quality, with respect to any of the considered direct quality metrics, is caused by any of the proposed IPC metrics taking certain values.

Contributions. In summary, we make the following contributions:

- We proposed a novel set of six metrics for measuring IPC at varied (from method through system) levels in distributed system executions, based on message-passing semantics and method-level dynamic dependencies.
- We developed DistMeasure, a framework that computes the proposed IPC metrics and their correlation with various quality metrics, so as to enable in-depth understandings of IPC characteristics and IPC-induced run-time system behaviors in terms of the predictive relationships between the quality metrics and IPC measurements.
- We implemented DistMeasure to work with industry-scale distributed systems and extensively characterized the IPC-induced behaviors and their quality relevance via predictive assessment of various quality aspects via IPC metrics against 11 diverse real-world systems against 16,880 execution scenarios, which demonstrated the merits of our framework and generated new knowledge about the implications of IPC characteristics to distributed system quality.
- We made our framework implementation open source and all datasets publicly accessible [48] to facilitate future research on distributed systems measurement and quality assessment.

This paper is an extended version of our preliminary work presented in [47], which to the best of our knowledge is the first study of dynamic *coupling* measurement in distributed systems in relation to their quality. Technically, the extension includes (1) one more *coupling* metric and a new *cohesion* metric and (2) the development of learning-based classification models for understanding IPCs, using the coupling/cohesion metrics significantly related to quality metrics as features. Experimentally, we expanded the study on the association between these IPC metrics and distributed system quality by (1) considering the two new IPC metrics introduced and two more quality metrics and (2) using

two additional real-world distributed systems as study subjects,in addition to (3) evaluating the newly developed unsupervised and supervised quality classification models. Notably, we manually augmented the run-time inputs based on original test cases for each subject at an extensive scale, increasing the executions considered per subject from 1~3 to 26~2,000 and total subject executions from 11 to 16,880. The large-scale input augmentation provides a much more solid data basis for the statistical analyses; it also effectively enables valid learning-based classifications and associated performance evaluations. Moreover, we substantially enhanced both the technical and empirical presentations, with expanded and more in-depth discussions on relevant background, related work, metrics definitions, experimental results, and insights based on the empirical findings. Overall, we reformulated the work as a framework that tightly connects three parts, where the first two parts subsume the previous work as a whole and the third part demonstrates the practical usefulness of IPC measurement in distributed software, hence significantly more extensively answering the newly clarified overarching questions.

Paper organization. In the rest of this paper, we first provide the necessary technical background of our approach along with an illustrating/motivating example (§2). Then, we give an overview of DistMeasure (§3), including its design rationale and implementation. Next, we present the first part of the framework (§4), which defines and justifies the proposed IPC metrics, including how each metric is computed with illustrating examples. In the second part (§5), we characterize the IPC in a set of distributed system executions hence examine the quality relevance of IPC, discovering correlations between IPC and various direct quality metrics. These correlations establish the underpinnings of the third part (§6), where we further investigate the quality relevance of IPC-induced behaviors of distributed systems in terms of the predictive relationships of IPC to significantly correlated direct quality metrics. Finally, we discuss the threats to validity of our results (§7) and related work (§8) before making concluding remarks and outlining future work (§9).

2 BACKGROUND AND MOTIVATION

This section gives a brief background on IPC (§2.1) and information flow (§2.2) in distributed systems, and general/software measurement fundamentals (§2.3). It then introduces a code example (§2.4) used for motivating our work, which is also used as a working example for illustrating our approach.

2.1 IPC in Distributed Systems

Interprocess communication (IPC) is a fundamental mechanism commonly used by distributed systems whose communications often depend on IPC primitives. In this paper, we focus on the common type of distributed systems in which communications among processes are realized via message passing [34]. With message passing, two or more processes exchange messages using at least two types of basic IPC primitives: sending messages and receiving messages.

Based on the concept of IPC and following the general notions of cohesion and coupling [41], a unique aspect of such notions in distributed systems is the cohesion/coupling in relation to IPC (i.e., at process level). Intuitively, process-level coupling is the degree to which different processes in a distributed system execution are connected to each other, while process-level cohesion is the degree to which different parts of a distributed system that execute in a single process are connected to each other at runtime.

2.2 Dynamic Information Flow

A dynamic information flow path, or called a dynamic taint flow path, is a program path from a *source* to a *sink* that is exercised during a concrete execution of the program. A (taint) source is a program point where sensitive data is

retrieved, and a sink is a program point where sensitive data leaves the program [79]. The program point as a source or a sink may be a method or statement, depending on the granularity level of the underlying dynamic information flow analysis algorithm with respect to the user interest. Particularly in a distributed system execution, a dynamic information flow path may be across different processes [46]. For those taint flow paths, IPC means through which the taint flow within individual processes propagates across process boundaries.

2.3 Measurement Fundamentals

For software engineering, measurement is a key step in quality assurance [54], for which appropriate metrics must be defined and applied. Before software implementation, the metrics provide a means for specifying quality requirements. After the implementation, the metrics serve as crucial guidance for evaluating the software product with respect to the quality requirements.

There are two categories of software metrics: static and dynamic. Static metrics are generally easier to compute relative to dynamic counterparts. Additionally, static metrics are not subject to limited code coverage or generalizability as are dynamic metrics. On the other hand, static metrics are not sufficient for measuring and interpreting actual (i.e., dynamic) behaviors of software, for which dynamic metrics offer more precise indicators. In fact, concerning quality metrics that are ultimately attested at runtime (e.g., performance and reliability), dynamic metrics are much more preferable. Meanwhile, understanding software behaviors does not always need complete code coverage [116]. Therefore, the fact that dynamic metrics only address particular executions does not necessarily compromise the usefulness of dynamic measurement.

Importantly, a metric should be validated with respect to its representation condition before applying it in empirical assessments. The representation condition for a metric holds if and only if a measurement mapping maps the entity being measured into a number in terms of the metric, and its empirical relation into a numerical relation in the measurement, in such a way that the empirical relation is preserved by the numerical relation [41]. This means that (1) if we have an intuitive understanding that an object A is physically higher than an object B, then also the measurement mapping M must give that M(A) > M(B); and (2) if M(A) > M(B) then it must be that the object A is intuitively understood to be higher than B. (1) and (2) here are the representation condition. For example, consider the metric for measuring human height. Suppose the measurement mapping here is *height*, which maps a person being measured into a number (e.g., the person's height in centimeters). If we intuitively understand (e.g., via visual observations) that a person A (of height 175cm) is taller than another person B (of height 172cm), this mapping will give us *height*(A) > *height*(B) (i.e., 175cm > 172cm) according to the measurement results. Meanwhile, if we have *height*(A) > *height*(B) based on our measurements (e.g., *height*(A)=175cm and *height*(B)=172cm), we will intuitively understand that A is taller than B [40] via visual observations. Then, we can conclude that the height metric is validated.

2.4 Motivating/Working Example

By design, a distributed system consists of multiple collaborating components each running in a process typically located at a separate computing node. Since these components interact primarily through IPC [123], measuring/characterizing hence examining quality relevance of IPC is essential for studying the quality implications of the behaviors of distributed systems. To illustrate, consider the simple example of Figure 1, where the system includes two components. The *Server* component, implemented in class `SC`, provides the service for querying a data store `ds` (line 7) according to the `dataid` sent by the client (lines 10–11).The *Client* component, implemented in classes `CC1` (for data querying) and `CC2` (for managing connections), prepares a query from user inputs (line 25), and then looks up the database server (line 26)

```
1   // The Server component
2   public class SC {
3     private Datastore ds;
4     private ServerSocket ssock;
5     void setup (int port) {
6         ssock = new ServerSocket(port);
7         ds = new Datastore();
8     }
9     int serve() {
10        long dataid = Long.valueOf ( ssock.readLine() );
11        return ssock.write ( ds.retrieve(dataid) );
12    }
13    public static void main(String[] args) {
14        SC s = new SC();
15        s.setup( Long.valueOf(args[1]) );
16        s.serve();
17    }
18  }
19  ...
20  // The Client component
21  public class CC1 {
22    public static void main(String[] args) {
23        CC2 conn = new CC2();
24        CC1 c = new CC1();
25        long dataid = Long.valueOf(args[1]);
26        System.out.println( c.lookup(conn,dataid) );
27    }
28    String lookup(CC2 conn,long id) {
29        return conn.getD(id);
30    }
31  }
32  public class CC2 {
33    private Socket csock = connectServer(authenticateInfo);
34    String getD(long id) {
35        csock.write(id);
36        return csock.read();
37    }
38  }
```

Fig. 1. A distributed system as an illustrating/working example.

through IPC (lines 35–36). Apparently, IPC dominates the high-level functionalities of this system, excluding which each component alone would be largely trivial. Thus, the complexity of this system's execution essentially lies in that of the IPC. Notably, the client runs in a low-security area as it is interfaced with public access (e.g., to low-security data such as `dataid`), while the data records managed by the server (via `Datastore`) are high-security information.

Suppose the developer now wants to understand the communication security issues of this system reported by users who also provided the inputs that can reproduce the issues. A rewarding first step would be to understand how the communication currently works in terms of the IPC with respect to the user inputs. Measuring IPC would help in this scenario. For example, based on the security policy given above, the data record associated with the `dataid` originated in the client process (at line 25) is considered private or sensitive. Then, there is a potential data leak—at line 11, the record is retrieved in the server process due to the call to `ds.retrieve` and then sent to the network due to the call to `ssock.write`, which can send the sensitive data record to a untrustworthy party if the party exists to intercept the network communication here. Understanding attack surface anomaly due to this possible leak can be facilitated by measuring the class-level coupling between the server and client processes and process-level cohesion of each of these two processes, as we demonstrate later (Section 6), because the data leak is induced by the IPC between the two processes. Of course, an accurate determination of the data leak here would need a further, more precise analysis such as information flow tracking/control. Yet, knowing the presence of the anomaly as informed by the IPC measurement constitutes an alert as a necessary first step and helps narrow down the scope (i.e., to the two particular communicating processes here) of the precise (typically more expensive) analysis.

Moreover, the IPC measurements may also help the developer assess (in a predictive manner) some quality aspects other than security. For instance, if the message coupling is relatively low while class-level coupling is relatively high between the communicating processes, intuitively the system may have abnormal complexity (in terms of cyclomatic complexity measures) as we demonstrate later in Section 6; as a result, the system might be difficult to update and test. Thus, we can anticipate that its testing and maintenance cost may be abnormally high also. Consider the example of

Figure 1 again, the system-level (class) coupling between the server and client processes is relatively low (0.45 as we later computed in Section 4.3), suggesting updating one of the two components is potentially easier (because of not having to update much of the other component)—an important aspect of the system maintainability.

Furthermore, it may be difficult to immediately understand or assess distributed system quality through direct quality metrics. For instance, as a performance efficiency metric, *execution time* can be hard to measure directly because it is hard to simulate the real-world use scenarios of the system in order to compute the metric. In addition, the execution environments of end users can be quite diverse and complex, and there could be many factors affecting the execution time, making a reliable execution time measurement difficult in practice. Another example is *attack surface*, a direct metric of security, which may be difficult to measure directly also. To compute the attack surface with an existing approach [87] as we followed, we would need to obtain the number of network ports possibly used and the number of files possibly read/written by the program, for which we would need to perform static analysis of the whole distributed system. Yet that may not be quite feasible because there is no accurate and scalable (to large-scale software) static analysis of whole distributed systems available yet. Indeed, for study purposes in this paper, we had to compute these direct quality metrics largely manually, which is laborious and clearly undesirable and not scalable. Again, characterizing IPC-induced behaviors of distributed systems could help address these difficulties.

Unfortunately, there is a lack of knowledge, and support for gaining such knowledge, regarding the characteristics of distributed systems executions hence their run-time behaviors with respect to IPC, and it is unknown what quality relevance and implications certain kinds of IPC characteristics may have. In this paper, we address these questions by developing new IPC metrics and applying them to study the correlation of IPC with the quality, concerning various quality aspects, of distributed systems. We also apply the correlations found to distributed software quality assessment.

Application scope. While our focus in this paper is on the common distributed systems [18] defined in the classical textbook on this subject [34], the proposed metrics apply to other systems that are similar to distributed systems in terms of the decoupled structure [31]. For example, systems adopting microservice architectures or IPC-based inter-component communication [69, 72] can also be measured using our IPC metrics. In particular, for those systems, looking into IPC coupling and cohesion against microservice systems with our respective metrics would help better understand or even validate some of the architectural merits and advantages of the common design practices for those systems. For instance, our measurement framework can be adapted to quantify location and temporal coupling in microservices through IPC coupling measurement since those coupling measures are closely related to interactions among distributed processes (running at physically separate locations). Moreover, integrating open specifications such as OpenTelemetry [59] in DistMeasure would facilitate distributed tracing to support the underlying IPC measurements.

3 DISTMEASURE OVERVIEW

We first justify the overall design of our framework and give an overview of its architecture and high-level workflow. We also discuss key implementation issues with DistMeasure, stressing the challenges encountered and summarizing our solutions to those implementation challenges. with real-world distributed systems.

3.1 Design Rationale

Figure 2 depicts the overarching composition of DistMeasure, highlighting its three integral, closely connected parts: IPC Metrics Definition and Computation (Part 1), Characterizing IPC and its Quality Relevance (Part 2), and Predictive Quality Assessment (Part 3). The rationale behind the framework design is two-fold.

First, IPC is a key aspect of the behaviors of distributed software systems. It is also a defining aspect that differentiates distributed software from single-process and centralized programs. Thus, with respect to our overarching questions, it is essential to be able to measure IPC, for which the metrics must be clearly defined including how each should be computed. In fact, Parts 2 and 3 both immediately rely on the definition/computation of IPC metrics because studying the quality relevance and assessing quality status predictively both require quantifying IPC characteristics, for which having the IPC metrics defined is key. Thus, in **Part 1**, we propose a set of intuitive and justifiable IPC metrics, offering the basis capabilities underlying the rest of DistMeasure.

Second, an essential part of our overarching questions (and the main goal) of this work is to investigate how IPC-induced distributed system behaviors are related to the quality of these software systems. By statistically characterizing the correlations between IPC metrics and direct metrics of various quality factors/sub-factors, **Part 2** aims to reveal how IPC measurements, in terms of *individual* metrics, may inform about diverse facets of distributed software quality, discovering significant relationships between the two that become the basis (features of learning-based prediction models) of **Part 3**.

Third, it might be difficult or even impossible to directly measure some quality factors/sub-factors with respect to certain quality metrics. For example, it is common to use the quality metric *code churn size* to quantify the modifiability sub-factor (in the maintainability factor). Yet computing this quality metric typically needs data on historical software releases; thus, it is not directly measurable for the first release of a software project. Other examples include those mentioned earlier (§2.4). Therefore, **Part 3** explicitly models the predictive relationships between IPC metrics and direct quality metrics, hence illuminating how the IPC metrics *holistically* relate to distributed system quality. This predictive assessment also demonstrates a way in which the studied quality correlations and statistical relationships can be practically useful in potentially assisting with distributed system quality assurance.

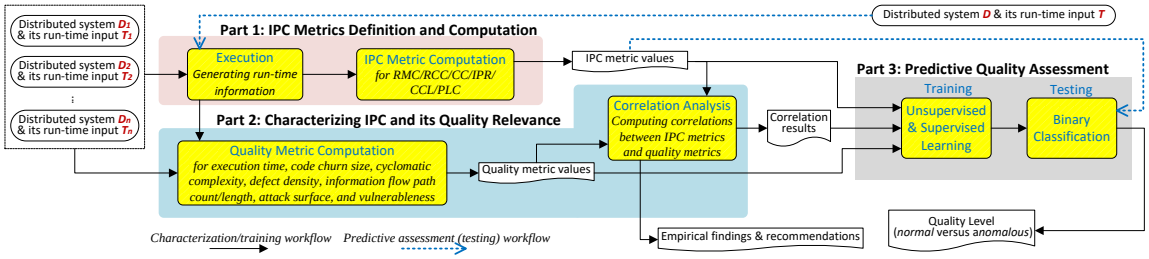

Fig. 2. An overview of the DistMeasure workflow, including its inputs, outputs, and 3 integral parts.

3.2 Overall Workflow

As shown in Figure 2, DistMeasure consists of three parts that are connected in a way we just described and justified (§3.1). Overall, the framework works in two modes hence has two different workflows: *characterizing/training* and *predictive assessment/testing*, as summarized as follows.

In the characterizing/training mode, DistMeasure leverages its capabilities of measuring IPC in distributed systems in **Part 1** to characterize IPC-induced behaviors in a sample set of distributed software and their executions, hence identifying the statistical relationships (i.e., correlations) between IPC metrics and a range of direct quality metrics in **Part 2**. These correlations are then utilized to train unsupervised and supervised anomaly detectors for strongly correlated direct quality metrics in **Part 3**.

In particular, **Part 1: IPC Metrics Definition and Computation** serves as the foundation of DistMeasure, which provides definitions of underlying IPC metrics and specifies how each metric is computed. These capabilities then

enable DistMeasure to measure IPC for one distributed system D_i against its one run-time input set T_i at a time: first, it executes D_i against T_i to generate the run-time system information needed for the metrics computations—all of our IPC metrics are *dynamic*, thus they require run-time system information; then, DistMeasure computes six IPC metrics (i.e., RMC (Runtime Message Coupling), RCC (Runtime Class Coupling), CCC (Class Central Coupling), IPR (InterProcess Reuse), CCL (Class Communication Load), and PLC (Process-Level Cohesion)) from that information.

Next, in **Part 2: Characterizing IPC and its Quality Relevance**, DistMeasure takes two kinds of inputs: multiple (N) distributed systems D_1, D_2, ..., D_n (in an executable format such as Java bytecode) and run-time input sets T_1, T_2, ..., T_n, where T_i is the input set (e.g., text messages, system commands, or SQL statements) driving the executions of D_i. For system D_i and its corresponding input set T_i, DistMeasure invokes **Part 1** to obtain the IPC measurement results in terms of the six IPC metrics. Meanwhile, eight direct quality metrics (i.e., *execution time, code churn size, cyclomatic complexity, defect density, information flow path count, information flow path length, attack surface*, and *vulnerableness*) of these systems are measured directly. After obtaining all values of these IPC metrics and those of the direct quality metrics, DistMeasure analyzes the correlations between them. These statistical analysis results lead to *empirical findings and recommendations* about distributed software quality.

In **Part 3: Predictive Quality Assessment**, DistMeasure takes the IPC and direct quality metric values, from **Part 1** and **Part 2**, respectively, to train unsupervised and supervised learning based quality anomaly detection models. Both kinds of models are built in order to accommodate diverse use scenarios and to enable our study on the trade-offs between effectiveness and usability, as discussed later. Also, one unique model is trained for each of the quality metrics with which at least one of the IPC metrics has a significant (negative or positive) correlation.

As such, the three parts are related and differentiated as follows. **Part 1** offers the capabilities of IPC measurement needed by **Part 2** for actual characterization of real-world distributed systems—the general computation methods/formulae are given in **Part 1**, while **Part 2** applies those capabilities (methods/formulae) to specific systems. Meanwhile, **Part 2** offers the correlation results as the statistical basis of the machine learning-based quality assessment classifications designed/evaluated in **Part 3**.

In the predictive assessment/testing mode, DistMeasure takes as inputs a distributed systems D and its run-time input set T, for which the same eight direct quality metrics are checked against anomalies. Feeding the trained supervised/unsupervised models with the IPC measurement results obtained by invoking **Part 1** on D against T, (**Part 3** of) DistMeasure predicts each of the direct quality metrics as `anomalous`, indicating a warning of low (lower-than-average-case) quality, or `normal` indicating no such a warning. These *quality level (normal versus anomalous)* results are the output of DistMeasure.

3.3 Framework Implementation

We have implemented the entire DistMeasure framework as a toolkit that works with real-world distributed systems of diverse domains and architectures, offering immediate tooling support for the measurement and understanding of IPCs in their executions. Measuring interprocess coupling and cohesion in distributed systems faces a practical challenge [26] as follows. Such metrics are often defined on the basis of certain relationships (e.g., dependency and inheritance) [135]. However, deriving the interprocess dependencies, from which our metrics are computed, is not trivial in the context of distributed system executions [37, 52]. The main reason lies in the lack of global timing across the system together with the lack of explicit references/invocations across distributed components [49]. To overcome this challenge, we leverage our dynamic dependence [51] and taint analyzers [46] for distributed programs. Using these analyzers, we reason about interprocess dependencies through the happens-before relations between executing

methods across processes, derived from a global partial ordering of method execution events. We further exploit the semantics of message passing to improve the precision of such derived dependencies, so as to enhance the validity of our IPC metrics.

Part of the DistMeasure implementation can be found in our tool demo, DistFax [53]. On top of the artifact [48] of DistMeasure, the tool demo and its documentation provide an additional reference for replicating our studies presented in this paper. Yet we note that the tool demo is not a research paper and is solely focused on the *demonstration* (e.g., installation/configuration and usage) of the tool. More importantly, our DistMeasure implementation has shifted considerably from what is in the tool demo even just from a tool perspective: a key component of the tool is how to train the learning-based prediction models, which involves constructing the training data and labeling them. In DistFax [53], we simply used mean feature values to differentiate positive and negative samples—samples with feature values below the means are considered normal and otherwise anomalous. This simple approach may suffer from poor robustness as the feature value distributions may be skewed. To address this issue, for DistMeasure (Part 3), we adopt a rigorous statistical approach, identifying feature-vector outliers among the training samples and labeling a sample as anomalous if it is an outlier and normal otherwise. As a result, the performance evaluation of DistMeasure (Part 3) led to different findings/conclusions from that of DistFax.

4 PART 1: IPC METRICS DEFINITION AND COMPUTATION

This section presents the basis of our framework—the metrics in which IPC-induced behaviors are measured, concerning their definitions and computation formulations. We first give the necessary preliminaries of our work by summarizing the main ideas of our method-level dynamic dependence approximation, which commonly underlies the definitions of the IPC metrics proposed. Then, we elaborate on the definition and computation of each of the metrics. As an overview, Table 1 lists for each (**IPC metric**) the definition (**Definition**), underlying rationale (**Rationale**), the most similar existing metric (**Reference metric**) from which the proposed IPC metric is drawn, and the corresponding previous evaluations of that reference metric—including the evaluation tool (**Tool**) and the granularity level (**Level**) of the evaluation results.

In what follows the preliminaries, for each metric, we elaborate in a respective subsection the rationale/justification behind the definition, motivating the metric by discussing its potential use for evaluating relevant quality metrics. Then, we illustrate each metric using our example system of Figure 1. Following the guideline in [41], we also provide the theoretical and empirical validations (explanations) with respect to the representation condition (scientific foundation) and one example of the condition, for our IPC metrics. Lastly, we summarize the metric definitions and computations, explicitly mapping these individual IPC metrics to various groups (e.g., cohesion and coupling).

4.1 Preliminaries

All of our proposed metrics are (*explicitly* for all but RMC) based on dynamic dependencies at the method level [21] across processes. Understanding how we compute such dependencies is hence essential for understanding our IPC metrics and measures. Thus, this section provides preliminaries of dependence computation for distributed programs.

Given a method m_{P_i} in one process P_i, all methods in any another process P_j, $j{\neq}i$ in the system's execution that *depend* on m_{P_i} form a set, referred to as the *dependence set* of m_{P_i}. Below, we outline the two core steps of our approach for determining the dynamic dependence between two methods across processes. Further details can be found in [27].

Table 1. An overview of the proposed IPC metrics in DistMeasure

IPC Metric	Definition	Rationale	Reference metric	Previous evaluation	
				Tool	Level
RMC	Interprocess message coupling at both process and system levels	The Extent of run-time interactivity among system components	MPC (Message Passing Coupling)[85]	CCMETRICS [67]	Class
RCC	Class coupling between two processes hence further the system-level class coupling	How methods from a class in one process access methods from a class in another process	DCC (Distinct Class Coupling)[119]	GMN tool [57]	Class
CCC	Aggregate coupling of a class that is executed in a local process with classes in all remote processes	The importance of a class in terms of its influence on all classes in remote processes	CBO (Coupling Between Objects)[32]	QScope [39]	Class
IPR	Interprocess reuse (coupling) at method/system levels	The size of functionalities shared between system components	MPC (Method Pair Coupling)[43]	[108]	Method
CCL	Communication loads of a class communicating with others in all remote processes	How much a class contributes to communication loads between its process and other processes	RFC (Response For a Class)[32]	QScope [39]	Class
PLC	Internal connections within an individual process	The degree to which the methods of a process belong together	ISCI (Inter-Service Coupling Index)[126]	SSP tool [126]	Service

(1) partial ordering of method execution events. The basis of our dynamic method-level dependence approximation is the happens-before relations between method entry (i.e., program control enters the method that is called) and returned-into (i.e., program control returns back to the calling method) events [24]. These events are partially ordered using the Lamport timestamps (LTS) algorithm [18, 82], realized in dedicated runtime monitors for those method execution events. In addition, two types of communication events, *message sending* and *message receiving*, are monitored to update the per-process logic clocks in the LTS algorithm. This is done in order to synchronize the logic clocks throughout distributed processes in the system execution so that all the method entry and returned-into events are timestamped consistently (i.e., with the synchronized logic clock for each process). Then, a method $m2$ is considered dependent on $m1$ if the first execution event of $m1$ happens before the last execution event of $m2$, according to the global partial ordering. Essentially, method-level dependencies between $m1$ and $m2$ are derived from their execute-after relation [25], which is further deduced from the happens-before relation between their associated method-execution events. Apparently, this is a rough approximation of dependency (only capturing temporal relationships). Thus, it will be refined via pruning next.

(2) pruning based on message-passing semantics. For a more precise dynamic dependence approximation, we further leverage the semantics of message-passing events across processes. Suppose methods $m1$ and $m2$ are executed in two processes P_i and P_j respectively, and $m2$ is considered dependent on $m1$ according to the purely control-flow-based approximation described above (i.e., *Step 1*). From a data-flow perspective, $m1$ would not affect $m2$ if during the system's execution (1) P_i never sends any messages to P_j, or (2) the event that P_j receives the first message from P_i never happens before the last execution event of $m2$. The rationale is intuitive: for a method in one process to influence (i.e., causing dependence to) a method in another process, the two processes must have actually communicated (by passing messages from one to the other), and the possible dependence between the two methods again relies on their happens-before relation. In other words, two methods have dependencies between them only if both they have execute-after relations and message passing has occurred between their respective processes, since message passing is the only communication channel between processes in the common type of distributed systems we address [22]. Spurious dependencies as a result of *Step 1* are pruned using these two intuitive rules. Then, with this more precise [27] dependence computation, we compute the dependence set underlying our IPC coupling metrics, as detailed next.

Against six distributed systems and ten system executions, the average precision of this dynamic dependence analysis, with both Steps (1) and (2) above applied, was 71% and the recall was 100% (i.e., 83% F1 accuracy) [27].

4.2 RMC (Runtime Message Coupling)

At a high level, processes in distributed systems executions interact through passing messages. Thus, this metric may capture the coupling between processes at a high (i.e., messaging) level.

Definition/computation. We define interprocess message coupling at both process and system levels. First, given two different processes P_i and P_j, their message coupling $RMC(P_i, P_j)$ is intuitively defined as the number of messages sent from P_i to P_j, counting all instance of each unique message. This process-level metric can be computed according to the communication events monitored in our framework (Section 2.1). Then, the system-level RMC is defined as the average of such process-level measures overall all communicating pairs of processes as (hereafter, we suppose the system runs in N processes in total)

$$RMC = \frac{\sum_{j=1}^{N} \sum_{i=1}^{N} RMC(P_i, P_j)}{N(N-1)}, i \neq j, i, j \in [1, N] \tag{1}$$

Rationale/justification. The value of the RMC metric indicates the extent of run-time interactivity among system components [12]. A higher RMC implies greater reliance of a component on (i.e., sending more messages to) others in the execution considered. Since these components are distributed over separate locations, larger RMC values also indicate higher communication costs (e.g., for greater network bandwidth use) at the granularity of message passing. Finally, a higher RMC suggests greater effort for systems understanding (even at a very high level) [69] if the messages counted in the RMC computation are all unique. Thus, RMC can be used as an *understandability/cost-of-quality* [54] metric. In essence, RMC measures the implicit coupling between (statically) decoupled components because the two processes involved in the definition correspond to two components of the system.

Illustration. Consider the example system of Figure 1. Suppose during the system execution under analysis the client process (P_{client}) sends the server process (P_{server}) two messages—the first for authentication (line 23, eventually realized in `connectServer()` at line 33) and the second for querying (line 35),and the server then sends back to the client the querying result in three messages. Thus,

$RMC(P_{server}, P_{client})$ = 3 and

$RMC(P_{client}, P_{server})$ = 2.

With two processes (N = 2), the system-level message coupling is

(2 + 3)/ (2 *(2 - 1)) = 2.5.

For another example, consider Apache ZooKeeper [5], an enterprise-scale distributed system that consists of three major components each running in one or multiple processes. Further, consider its built-in integration test for an example execution, during which only one server and one client process are involved and they exchanged 13 messages during the test. Thus, both process-level RMC measures are 13, and the system-level

RMC = (13 + 13) / (2 * (2 - 1)) = 13.

Note that, as for other IPC metrics as follows, the metric definition/computation does not rely on the functionality role (e.g., server versus client) of a process to be identified. We mention these roles with respect to the illustrating example of Figure 1 only for ease of presentation.

4.3 RCC (Runtime Class Coupling)

The coupling between two processes is ultimately due to their coupling at a lower, code level. When considering distributed systems developed in an object-oriented language, a lower-level unit of code is class, hence this class-level coupling metric.

Definition/computation. We measure finer-grained interprocess coupling at the class level, from which we define the class coupling between two processes hence further the system-level class coupling. Specifically, the coupling metric for two classes, A_{P_i} in process P_i and B_{P_j} in process P_j, is defined as the ratio of the total number of methods in B_{P_j} that are dependent on any method in A_{P_i}, to the total number of methods in any process other than P_j that are dependent on any method in A_{P_i}. Let $DS(m)$ denotes the dependence set of method m, the class-level RCC metric is defined as

$$RCC(A_{P_i}, B_{P_j}) = \frac{|\bigcup_{m \in A_{P_i}} \{f | f \in DS(m) \wedge \ f \in B_{P_j}\}|}{|\bigcup_{m \in A_{P_i}} DS(m)|} \tag{2}$$

Given a query m (i.e., an exercised method), the entire dependence set $DS(m)$ includes all methods that dynamically depend on m for the execution being analyzed [19], regardless of the dependant methods being executed in the same process as the query m (i.e., *local* process) or in other processes (i.e., *remote* processes). Accordingly, the dependant methods in the local and remote processes form the *local dependence set* and *remote dependence set*, respectively. The denominator in Equation 2 is the size of the union set of entire dependence sets of all methods in A_{P_i}, while the numerator is the size of the union set of remote dependence sets of those methods.

Next, the process-level RCC metric is defined as

$$\begin{aligned} RCC(P_i, P_j) &= \sum_{A \in P_i} \sum_{B \in P_j} RCC(A_{P_i}, B_{P_j}), \\ where \quad & RCC(A_{P_i}, B_{P_j}) \neq 0 \end{aligned} \tag{3}$$

While the constraint $RCC(A_{P_i}, B_{P_j}) \neq 0$ is trivial for computation, it is not for the definition which only considers the pairs of classes that are actually coupled—a zero RCC would indicate that they are not coupled.

Finally, the system-level RCC metric is defined as

$$RCC = \frac{\sum_{j=1}^{N} \sum_{i=1}^{N} RCC(P_i, P_j)}{N(N-1)}, i \neq \ j, i, j \in [1, N] \tag{4}$$

Rationale/justification. The rationale for RCC is that its value indicates how methods from a class in one process access methods from a class in another process. In contrast to RMC which is a *message coupling* metric, RCC measures functionality coupling. A higher RCC implies greater functional interdependency among system components in the execution considered. From a change management perspective, this higher RCC suggests that making changes for the associated system use case would be more difficult and costlier. Also, the denser interprocess dependence associated with a greater RCC makes it harder to test and debug the system with respect to the use case. Thus, RCC may inform about the *maintainability* of the system being measured [54].

Table 2. Illustrating dependence sets of the system of Figure 1

Method m	Dependence set $DS(m)$
`SC::main`	{`SC::main`, `SC::setup`, `SC::serve`, **CC1::main**, **CC1::lookup**, **CC2::getD**}
`SC::setup`	{`SC::setup`,`SC:serve`, **CC2::getD**}
`SC::serve`	{`SC::serve`, **CC2::getD**}
`CC1::main`	{`CC1::main`,`CC1::lookup`,`CC2::getD`, **SC::setup**,**SC::serve**}
`CC1::lookup`	{`CC1::lookup`,`CC2::getD`, **SC::serve**}
`CC2::getD`	{`CC2::getD`}

Illustration. For example, consider an operation profile of the system of Figure 1, in which the method-level dependencies[3] are listed in Table 2. For each query (first column), the entire dependence set is given in the second column, including methods executed in remote processes as marked in boldface. Based on these local/remote dependencies, for class $SC_{P_{server}}$, the union set of all its methods' entire dependence sets is

{`SC::main`, `SC::setup`, `SC::serve`, `CC1::main`, `CC1::lookup`, `CC2::getD`}

while the union set of all the methods' remote dependence sets that belong to $CC1_{P_{client}}$ is

{`CC1::main`, `CC1::lookup`}.

Thus,

$RCC(SC_{P_{server}}, CC1_{P_{client}})$ = 2 / 6 = 0.33.

Similarly,

$RCC(SC_{P_{server}}, CC2_{P_{client}})$ = 1 / 6 = 0.17,

$RCC(CC1_{P_{client}}, SC_{P_{server}})$ = 2 / 5 = 0.4, and

$RCC(CC2_{P_{client}}, SC_{P_{server}})$ = 0 / 1 = 0.

Then, the process-level RCC measures are

$RCC(P_{server}, P_{client})$ = 0.33 + 0.17 = 0.5,

$RCC(P_{client}, P_{server})$ = 0.4.

Finally, the system-level RCC is computed as

(0.5 + 0.4) / (2 *(2 - 1)) = 0.45.

4.4 CCC (Class Central Coupling)

Based on the previous metric of class-level coupling, it is intuitively useful to assess how an individual class in one process would influence (through that class-level coupling) the classes in other processes. This assessment helps capture classes that are more influential than others.

Definition/computation. On the basis of the RCC metric, we further measure the aggregate coupling as regards an individual class executed in a local process with respect to classes in all remote processes. Specifically, given a class A_{P_i} in process P_i, the CCC metric is defined as

$$CCC(A_{P_i}) = \sum_{j=1}^{N} \sum_{B \in P_j} RCC(A_{P_i}, B_{P_j}),$$
$$where \quad RCC(A_{P_i}, B_{P_j}) \neq 0, i \neq j, i, j \in [1, N] \tag{5}$$

The constraint $RCC(A_{P_i}, B_{P_j}) \neq 0$ is given for the rigor of the definition despite its triviality in the metric computation. The system-level CCC is then defined as the mean of class-level CCC measures over all classes executed (in any process).

Rationale/justification. Intuitively, the CCC metric of a class c characterizes the importance of c in terms of its influence (coupling strength) on all classes in remote processes by being coupled with them. If we construct a coupling graph where each node is a class and each edge represents RCC between two classes, the CCC metric is akin to the *centrality* metric in network measurement [88]. Our definition of CCC was originally motivated by the centrality metric indeed, which has been widely used in network analysis to identify the most important vertices in a graph (network). Thus, the CCC metric can be used to understand key classes in IPC, which could potentially inform about localizing quality issues (e.g., identifying faulty classes). Further, this metric may be utilized to assess functional *correctness* and other quality metrics of the distributed system under measurement at class level.

[3]For the simplicity of our illustrations, we dismissed methods `dlDB` and `connectServer`, which are both *library* routines.

Illustration. Let us consider the same example system (Figure 1) and execution scenario as used for illustrating the RCC metric. Given all those class-level RCC measures, we can readily compute the CCC measures for the three classes:

$CCC(SC_{P_{server}}) = RCC(SC_{P_{server}}, CC1_{P_{client}}) +$
$RCC(SC_{P_{server}}, CC2_{P_{client}}) = 0.33 + 0.17 = 0.5$,
$CCC(CC1_{P_{client}}) = RCC(CC1_{P_{client}}, SC_{P_{server}}) = 0.4$, and
$CCC(CC2_{P_{client}}) = 0$.

The system-level CCC measure is thus computed as
$(0.5 + 0.4 + 0) / 3 = 0.3$.

4.5 IPR (InterProcess Reuse)

Given that we have process- and class-level coupling metrics, we may measure coupling between two processes at an even finer-grained level: method level. At this level, the coupling measures functionality overlapping and code reuse across the processes in the system execution under analysis.

Definition/computation. Complementing the previous three forms of coupling metric, we further propose a metric of interprocess coupling at method level. And the system-level metric is derived from the method-level measures. Specifically, given a method m_{P_i} in any process P_i, let $LDS(m_{P_i})$ and $RDS(m_{P_i})$ denote the local dependence set (set of methods in P_i, which depend on m_{P_i}) and remote dependence set (set of methods that depend on m_{P_i} but are in any process other than P_i) of m_{P_i}, respectively. Also, we denote as M the entire set of methods covered in the system execution under analysis—M is the union set of methods executed in any process during the execution: $M = \bigcup_{i \in [1,N]} \{m_{P_i} | m_{P_i} \in P_i\}$.

First, the method-level IPR metric with respect to an individual method m_{P_i} in a process P_i is defined as

$$IPR(m_{P_i}) = \frac{|LDS(m_{P_i}) \bigcap RDS(m_{P_i})|}{|M|} \tag{6}$$

Then, the system-level IPR metric is defined as

$$IPR = \frac{\sum_{m_{P_i} \in M} IPR(m_{P_i})}{|M|}, i \in [1, N] \tag{7}$$

Rationale/justification. The key rationale for this metric is that its value indicates the size of functionalities shared between system components. Real-world distributed systems are usually found to have common code modules used by two or more distributed components. As a typical example, the process-level common functionalities exercised reflect the fact that multiple components of the distributed system use the same third-party libraries. Therefore, IPR can serve as an intuitive metric measuring component-level code reusability [26]. On the other hand, despite the name of this metric suggesting code reuse, IPR is essentially still an (albeit derivative/variant) metric of interprocess coupling. Intuitively, a higher IPR metric value of a system indicates more dynamically coupled processes of the system.

Illustration. For the same example system and execution as used for illustrating other metrics and in reference to the dependence sets of Table 2, the method-level IPR for any of the six methods covered by the execution is zero. Thus, the system-level IPR is also zero. (Note that the methods in boldface form the remote dependence set of each query.) For an alternative example, let us consider the library functions `dlDB` and `connectServer`. Now suppose these two functions both invoked another library function `libHttpConn` (for managing HTTP connections). Then,

LDS(`SC::main`) = {`SC::main`, `SC::setup`, `libHttpConn`, `SC::serve`}
and

RDS(`SC::main`) = {`CC1::main`, `CC1::lookup`, `libHttpConn`, `CC2::getD`}.

Thus,

IPR(`SC::main`) = 1/7.

Similarly,

IPR(`SC::setup`) = 1/7,

IPR(`CC1::main`) = 1/7,

and the IPR of other methods remains zero. Accordingly, the system-level IPR would be

(1/7+1/7+1/7) / 7 = 3/49.

4.6 CCL (Class Communication Load)

The intuitive core concept underlying IPC is *communication*. Thus, it is natural choice to have a metric capturing the communication load. More specifically, we may attribute such load to individual classes.

Definition/computation. We measure the communication loads of an individual class communicating with others in all remote processes. Specifically, given a method m in a given class A_{P_i} in a process P_i, let $RDS(m_{P_i})$ denotes the remote dependence set (set of methods that depends on m but are in any process other than P_i) of m_{P_i}. Also, we denote as $M_{A_{P_i}}$ the entire set of executed methods in the class A_{P_i}. First, the class-level CCL metric with respect to an individual class A_{P_i} in a process P_i is defined as

$$CCL(A_{P_i}) = \frac{\sum_{m \in A_{P_i}} |RDS(m)|}{|M_{A_{P_i}}|} \tag{8}$$

The system-level CCL is then the mean of class-level CCL measures over all classes executed (across all processes).

Rationale/justification. Intuitively, the CCL metric measures how much a class contributes to loads of communications between the process that executes the class and any other process. A higher CCL means that a class contributes more to the communications among system components. Thus, the CCL metric could be used to identify high-throughput classes which may need to be particularly attended in diagnosing communication *performance* issues of the distributed system under measurement. Accordingly, a high system-level CCL would inform about the system's overall communication performance. Also, since such high-throughput classes have the densest external dependencies across process boundaries, they may be prioritized when debugging interprocess faults; thus, a high system-level CCL would also inform about the *correctness* of the system with respect to process interfaces. It might seem that CCL is very similar to the previous four metrics (RMC/CCC/RCC/IPR), yet there are actually key differences, as discussed below.

Compared to RMC informing about the communication costs between two processes directly at a coarse (i.e., process) and high (i.e., in terms of messages passed) level, CCL measures the communication loads between two processes in terms of interprocess dependencies at a finer (class) and lower (in terms of code dependencies between methods) level. Accordingly, CCL and RMC have different measurement granularity and scopes. While the message passing directly measured by RMC are underlaid by the interprocess dependencies measured by CCL—after all, the dependencies are induced by the message passing, RMC simply considers the number of messages but not the amount of information underneath, whereas CCL captures how many dependencies are involved underlying the messages sent and received.

Different from the CCL of a class measuring how much the class contributes to loads of the communications between relevant processes, RCC measures functionality coupling among components in the execution considered. Compared to CCC, which characterizes the importance of a class by summarizing RCC measurements, CCL focuses on the communication load contributions of classes. Finally, complementary to IPR measuring component-level code

reusability among system components in terms of *common* dependencies between processes, the CCL measurement is based on how heavily one process communicates with others in terms of dependencies *across* processes.

Illustration. Consider the same example system of Figure 1 with the dependence sets shown in Table 2 for the same execution scenario used to demonstrate the RCC metric. We have

RDS(`SC::main`)={`CC1::main`, `CC1::lookup`, `CC2::getD`},

RDS(`SC::setup`) = {`CC2::getD`}, and

RDS(`SC::serve`) = {`CC2::getD`}.

Thus,

$CCL(SC_{P_{server}})$ = (3+1+1) / 3 = 1.67.

Similarly, we can compute

$CCL(CC1_{P_{client}})$ = (2 + 1) / 2 = 1.5.

Therefore, the system-level CCL is calculated as

(1.67+1.5) / 2 = 1.585.

4.7 PLC (Process Level Cohesion)

All of the previous five metrics are concerned with *coupling* (albeit at various granularity levels). Traditionally, cohesion is studied as opposed to coupling. With respect to IPC, the cohesion of individual processes may also have quality implications similar to those of the cohesion of components in centralized software.

Definition/computation. Complementary to the previous metrics which are all about coupling, we define a process-level cohesion metric, to measure internal connections within an individual process. Specifically, for a given method m_{P_i} executed in a process P_i, let $LDS(m_{P_i})$ be the local dependence set of the method (i.e., the set of methods executed in P_i that depend on m). Also, we denote as M the entire set of methods covered in P_i in the system execution under analysis. First, the process-level PLC metric with respect to an individual process P_i is defined as

$$PLC(P_i) = \frac{\sum_{m \in P_i} |LDS(m)|}{|M|} \tag{9}$$

Then, the system-level PLC is the mean of process-level PLC measures over all processes.

Rationale/justification. The cohesion measures the degree to which the methods of a process belong together [38, 132]. A higher PLC implies methods in a given process rely more on each other. Accordingly, PLC can also serve as an intuitive metric measuring the component-level code cohesion in distributed systems. A component/process of a higher PLC is easier to be identified for reuse and easier to change.

Comparatively, PLC measures *cohesion* while the other five metrics (i.e., RMC, RCC, CCC, IPR, and CCL) measure the *coupling* of a distributed system. Accordingly, compared to these other metrics measuring process pairs, PLC measures an individual process.

Illustration. For the same example system and execution as used for the demonstration of other metrics, as shown in Table 2, we can calculate process-level PLC firstly.

LDS(`SC::main`) = {`SC::main`, `SC::setup`, `SC::serve`},

LDS(`SC::setup`) = {`SC::setup`, `SC::serve`}, and

LDS(`SC::serve`) = {`SC::serve`}.

Thus,

$PLC(P_{server})$ = (3 + 2 + 1) / 3 = 2.

Similarly, we can compute

$PLC(P_{client})$ = (3 + 2 + 1) / 3 = 2.

Lastly, the system-level

PLC is (2 + 2) / 2 = 2.

4.8 Theoretical and empirical validations

In software measurement, there are two ways to validate a metric: *theoretical* and *empirical* validations. The theoretical validation assures that the measurement using the metric does not violate the necessary properties of what is being measured, while an empirical validation confirms that the metric values are consistent with the values predicted by the model that involves what is being measured [130]. Moreover, the representation condition is a prerequisite for a valid measure [17]. This condition asserts that a measurement mapping maps measured entities into numbers. And the mapping also maps empirical relations into numerical relations for preserving the empirical relations.

As mathematical properties, our IPC metrics map coupling/cohesion (in distributed systems) into numbers and coupling/cohesion relations into numerical relations [41]. We suppose that a mapping function RMC(D) indicates the RMC value of a distributed system D. If a system D_1 intuitively exchanges more messages among its distributed processes than another system D_2 does during their executions, then the function RMC() ensures that RMC(D_1) > RMC(D_2). Meanwhile, if RMC(D_1) > RMC(D_2), we should be able to empirically observe that D_1 has more messages exchanged than D_2 does during the executions. Any of our other five IPC metrics was theoretically and empirically validated in a similar manner. For example, Voldemort exchanges more (241 in total) messages among its processes than Netty did (only 2 messages) during one of their executions, as we understood through observing these execution behaviors via the run-time logs of these two systems. In our IPC measurements, we obtained RMC(Voldemort)=40.17, greater than RMC(Netty)=1.00. Meanwhile, whenever we had RMC(Voldemort)>RMC(Netty) per our measurement results, we did observe more messages being passed among the system components in Voldemort during its executions than in Netty according to their logs.

5 PART 2: CHARACTERIZING IPC AND ITS QUALITY RELEVANCE

In this section, we apply DistMeasure's capabilities of measuring IPC as described in **Part 1** to characterize the IPC-induced behaviors of diverse real-world distributed systems using the proposed IPC (coupling and cohesion) metrics. Such measurements will then enable us to study the quality relevance of IPC. Accordingly, the goal of **Part 2** of our framework subsumes two aims: (1) it characterizes real-world distributed system executions to reveal the traits of their IPC-induced behaviors in terms of cohesion and coupling using the proposed metrics and (2) it examines the quality relevance of IPC through statistical (correlation) analyses between those IPC metrics and various aspects of distributed software quality in terms of respective direct quality metrics. These two aims can be fulfilled by answering two questions:

- **RQ1: How coupled and cohesive are the processes in distributed systems in terms of the proposed IPC metrics?** The goal of answering this question is to demonstrate the practicality of IPC measurement with DistMeasure for distributed systems hence its usefulness in quantifying their IPC characteristics (i.e., component/process-level coupling and cohesion) as a way of measuring their IPC-induced behaviors.
- **RQ2: What are the quality relevance of the proposed IPC metrics in terms of their correlations with the eight direct quality metrics?** The goal of answering this question is to show the merits of IPC measurement in

terms of its statistical relationship with distributed software quality and demonstrate the capabilities of DistMeasure in offering the merits by enabling in-depth characterization of IPC-induced behaviors of distributed software systems. We first describe our methodology for the IPC measurement and statistical analyses (§5.1), including the direct quality metrics considered, underlying datasets used, and our procedure for answering each of the two research questions. We then present the results for each question (§5.2) in accordance with the respective aim above. Lastly, we discuss the lessons learned from the empirical results and recommendations made based on the results and lessons (§5.3).

5.1 Methodology

This section clarifies our measurement design, including the measured subject distributed systems and run-time test inputs used for exercising these subjects to generate systems executions needed for computing the IPC metrics. We also describe our reference-quality model and quality metrics in our framework, followed by measurement procedures.

Table 3. Statistics of measured subject systems

Subject (version)	Logical SLOC	Test Type	Execution Scenario	#Augmented Test Inputs
XNIO (2.0.0)	3,963	Integration	Client/Server	247
OpenChord (1.0.5)	6,391	Integration	Peer-to-Peer	1,999
xSocket (2.8.15)	11,628	Integration	Peer-to-Peer	1,698
QuickServer (1.4.6)	13,369	Integration	Client/Server	34
Thrift (0.11.0)	13,543	Integration	Client/Server	525
Grizzly (2.4.0)	22,725	Integration	Client/Server	2,000
Karaf (2.4.4)	46,810	Integration	Three-tier	26
ZooKeeper (3.4.11)	50,577	Integration	Client/Server	2,000
		Load	N-tier	1,409
		System	N-tier	2,000
Voldemort (1.9.6)	66,754	Integration	Client/Server	1,631
Netty (4.1.19)	109,450	Integration	N-tier	1,919
Derby (13.1.1)	423,662	Integration	Peer-to-Peer	1,392

5.1.1 Subjects and Test Inputs. Table 3 lists the 11 Java subjects used in our study (the first column), the logical Source Lines Of Code (noted as logical SLOC, as shown in the second column), and the original types (i.e., integration, load, system) of test inputs (the third column). For all the subjects but ZooKeeper, only one type of test inputs was available. We report logical SLOC (and use it for normalizing various other metrics in DistMeasure) as opposed to physical SLOC because the former is less sensitive to programming styles and format [99]. The fourth column shows the execution scenarios of each system execution, covering all the major types of distributed system architecture: Client/Server (C/S), Peer-to-Peer (P2P), Three-tier, and N-tier. The last column shows the size of input set we eventually used for each subject. Such test inputs were built through manual test augmentation as detailed below.

For IPC characterization against one system execution, a single test input suffices. However, for measuring IPCs in relation to various quality metrics by computing statistical correlations, we need a substantial number of executions by executing the chosen subjects against a large number of test inputs. Yet for each subject, only one test case was originally available for each test type. For ZooKeeper, the load and system tests came with the project package. For any subject, the single integration test was created by ourselves putting together the steps in the quick start guide of the respective subject found from its official project website. This led to a total of 13 subject executions, far from being

adequate for statistical analyses and learning-based classifications in DistMeasure. Thus, we manually augmented the test inputs for each subject. We now elaborate on each subject and how we expanded its original run-time inputs.

XNIO is a non-blocking I/O layer and library used to build efficient networking applications [143]. In its original integration test, we started one server and one client and then sent arbitrary text messages from the client to the server. To augment the test suite, we created 2,000 files each of which includes different text contents randomly generated. During each execution, the XNIO client read one file and sent the full content to the server.

OpenChord is a peer-to-peer network service using a distributed hash table [103]. In its original integration test, after we started three nodes (e.g., A, B, and C), the following operations were performed: create an overlay network on a machine (node) A; join the network on other nodes B and C; insert a new data entry to the network on node C; search and then remove the data entry on node A; finally, list all data entries on node B. To augment the test suite, we created 2,000 files each of which includes various a different set of OpenChord commands (e.g., `retrieveN -key test`). In some of these files, we purposely included invalid commands or command combinations to construct malformed inputs. During each execution, the nodes read the commands from one file and performed them in order.

xSocket is an NIO-based library for the development of high-performance network applications [144]. In its original integration test, after one server and one client were started, the client sent a sequence of manually composed text messages to the server. We expanded the test suite using the same way that we used for XNIO.

QuickServer is an open-source library for users to quickly develop multi-client TCP applications [111]. In its original integration test, after its server was started, the client connected to it and then sent a set of text messages to the server. We expanded the original test suite as for XNIO.

Thrift is a scalable framework for developing cross-language services [136]. In its original integration test, we used its libraries to develop a calculator consisting of a server and a client component. (The Thrift file was transferred to Java programs firstly). We ran the calculator (from its client) against basic arithmetic operations (i.e., addition, subtraction, multiplication, and division). To expand the test suite, we created 2,000 files each of which includes a distinct arithmetic expression (e.g., `43 minus 27 multi 39 add 11`), with some invalid ones to represent malformed inputs. During each execution, the Thrift client read the expression from one file, sent it to the server, and then took the computation result back from the server.

Grizzly is an NIO-based server framework from the GlassFish community [62]. In its original integration test, we started a server and a client, and then sent random text messages from the client to the server, and finally waited for the echo of each message. We expanded the original test inputs almost like for XNIO, except for the addition of a command for awaiting each message's echo sent by the client.

Karaf is a modular container as an open-source runtime environment supporting the standard OSGi [75]. In its original integration test, we created a container hosted by the server and then executed two commands: list all packages (`la`) and list OSGi bundles (`list`). To expand the original input set, we created 2,000 files each of which includes various Karaf commands (e.g., `config:proplist`). We purposely included invalid ones in some cases to construct abnormal inputs. During each execution, the Karaf client read the commands from one file and performed them in order.

ZooKeeper is a widely used distributed coordination service for consistency and synchronization [155]. In its original integration test, after we started two instances of the server and one instance of the client, the sequence of our operations was: create two nodes; search the two nodes just created; look for their attributes; update the data association between the two nodes; and remove the two nodes. In the original load test, after the server was started,

we started a container instance and then generated workloads. In the original system test, we started a server instance and a system test container, and then launched the system test.

To expand the integration test inputs, we created 2,000 files each of which includes various ZooKeeper commands (e.g., `ls /zk-temp`), including invalid ones. During each execution, the Zookeeper client read the commands from one file and then performed them in order. To expand the load test suite, we generated the workloads with 2,000 different sets of configuration parameters (e.g., `the number of clients involved in the load`, `the request size`). During each execution, we ran the load test with one of the parameter sets. To expand the system test suite, we created 2,000 configuration files each with different configuration parameters (e.g., `tickTime`, `initLimit`, `syncLimit`). During each execution, we ran the original system test code but with one of the configuration files.

Voldemort is a distributed key-value storage system used at LinkedIn [134]. In its original integration test, we first started a server and a client, and then our operations were: add a key-value pair, find the key for its value, remove the key, and retrieve the pair. To expand the original test suite, we created 2,000 files each of which includes various Voldemort commands (e.g., `getmetadata`). Invalid commands or command sequences were included for invalid inputs. During each execution, the Voldemort client read the commands from one file and then performed each of the commands.

Netty is a non-blocking I/O event-driven framework used for the rapid development of Java network protocol servers and clients [98]. In its original integration test, after starting a server and a client, we sent a series of text messages from the client to the server. We expanded the original input set using the same way as for XINO.

Derby is an open-source relational database [35]. In its original integration test, we searched all the data records (`SELECT *`) from a relational database (including one table) created beforehand. To expand the original test, we created 2,000 files each of which includes a distinct set of SQL statements that are compatible with Derby (e.g., `show settable_roles`). Invalid SQL statements or invalid statement sequences were included to construct invalid inputs. For each execution, the Derby client read all of the SQL statements from one file and then executed them in sequence.

We aimed at 2,000 tests per subject as it will give us a substantial number of sample subject executions and IPC measurements to enable meaningful statistical analysis. In augmenting each type of test inputs for a subject, for each candidate new test, we ran the subject against the test twice each to record a sequence of method entry and returned-into events. If the two sequences differ, which indicates the execution was non-deterministic against the test, we discarded it. Otherwise, if the sequence is *unique* relative to all previously collected sequences, the candidate test was added to the suite. As a result, we had 26 to 2,000 test inputs for each subject and test type, for a total of 16,880 test inputs for the 11 subjects. These 16,880 subject executions formed the basis of all the experiments presented in the rest of this paper. Depending on the nature of different systems, creating additional inputs while enforcing each to cover different program paths turned out to be quite difficult for some subjects. For instance, for QuickServer and Karaf, although we followed the same strategy (e.g., creating 2000 different inputs) as for other subjects, we ended up with much less inputs with unique coverage because these two systems execute highly similar program paths against the different inputs we created. Importantly, the generation of each of these additional tests was achieved via mutation as in mutation-based fuzz testing, using different mutation operators we manually carefully chose for each project according to its input format and functionality domain. While such added tests may not always represent real-world operational profiles of respective subjects, these tests do help exercise interesting run-time behaviors of the systems just like what fuzzing and other test generation/augmentation methods typically aim to. Thus, the exercised behaviors do fit DistMeasure's run-time characterization purposes. Also, the randomness in our test-input augmentation is

justified by the exploratory/discovering nature of our study that aims to reveal statistical relationships between our IPC metrics and relevant direct quality metrics. Ideally, input generation strategies tailored for different quality metrics would be more desirable. Furthermore, for some of the metrics (e.g., security focused ones), random generation may not guarantee accurate analysis that is preferred. However, prior to our study, we did not know about those relationships. And the random part in our generation strategies may suffice for statistical (albeit not for accurate) analysis.

Note that our subject selection is justified by two principles we follow. First, with DistMeasure we aim to address common distributed systems for general-purpose distributed computations, as defined in the classical textbook on distributed systems [34], rather than specialized/domain-specific ones such as distributed event-based systems which are built on specialized/customized communication frameworks/protocols (e.g., CORBA). While analyzing the later would be less challenging as those special/customized framework/protocols make it easier to reason about inter-process/component dependencies, we focus on common types of distributed systems because they are more widely deployed in real-world environments. Second, we chose common distributed systems and executions in different application domains and of varying scales (code sizes) and architectures, including `Client/Server` (e.g., Thrift and ZooKeeper against their integration tests), `Peer-to-Peer` (e.g., xSocket and OpenChord against their integration tests), `Three-tier` (i.e., Karaf against its integration tests), and `N-tier` (e.g., Netty against its integration test and ZooKeeper against their system and load tests). The selection was done before we developed DistMeasure—we have used them for prior works [22, 26, 50], rather than cherry-picking them after developing DistMeasure.

5.1.2 Reference-Quality Model. In order to systematically measure the relationships between IPC and distributed software quality—to fulfill the second aim of **Part 2** (§5), we need a reference-quality model for distributed software to ensure the direct quality metrics we considered in DistMeasure are well-grounded on relevant software quality standards. To that end, we adopted the standard quality model ISO/IEC 25010 [68] with necessary customization applied, as depicted in Figure 3. The original model defines three layers: quality characteristics (i.e., factors), sub-characteristics (i.e., sub-factors) under each characteristic, and quality metrics for each sub-characteristic. The first two layers are standardized in terms of the specific quality characteristic/sub-characteristic names, yet the bottom layer is not specified by the standard (in terms of the concrete metrics for each sub-characteristic).

For DistMeasure, we only considered the characteristics and sub-characteristics that are intuitively related to distributed system behaviors that are likely induced by IPCs. Similarly, for each sub-characteristic, we considered some quality metrics that are potentially related to our IPC metrics. As shown, we defined our reference-quality model with four quality characteristics (i.e., *performance efficiency*, *maintainability*, *functional suitability*, and *security*), nine sub-characteristics, and eight quality metrics. Some quality metrics are used to measure multiple sub-characteristics.

Next, we elaborate on each of the quality metrics within the corresponding sub-characteristic/characteristic in reference to the model, including how it is defined, why we included it, and how the associated metrics were computed.

Performance Efficiency. According to the ISO/IEC 25010 standard, performance efficiency concerns the system performance relative to the number of resources used under stated conditions. As its sub-characteristic, *time behavior* is the degree to which the response/processing times and throughput rates of a system meet requirements [68]. The IPC characteristics of a distributed system account, at least in part, for the system's performance efficiency in terms of time behavior, since IPCs immediately incur time costs due to the networking activities underlying the IPCs.

We measure the time behavior of a system through a straightforward metric *execution time*, normalized by the subject size in terms of logical SLOC. Different systems can be quite different in terms of their size/scale. To make the value of a metric comparable between two systems of different sizes, it is common to normalize the value of such

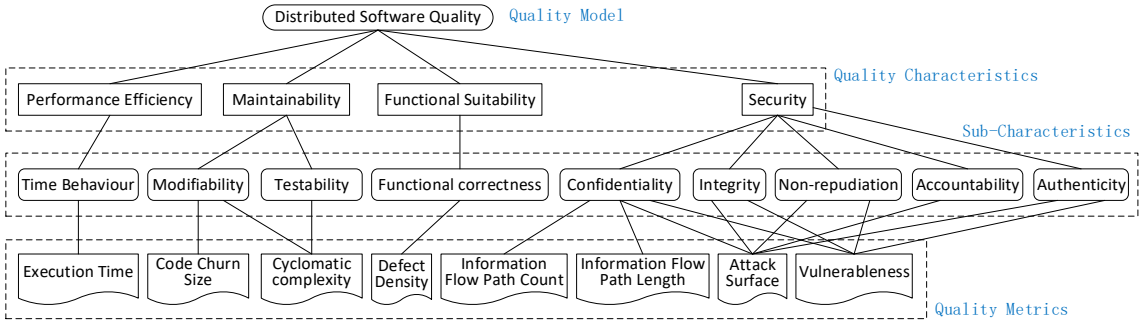

Fig. 3. The reference-quality model underlying DistMeasure, adopted from and compliant with the ISO/IEC 25010 [68] standard.

metrics by a size measure [74]. A common size measure is logical SLOC [56]. Thus, as for a few other quality metrics, we normalize the metric values of *execution time* by logical SLOC.

Maintainability. In ISO/IEC 25010, maintainability is defined as the degree of effectiveness and efficiency with which a system can be modified by the intended maintainers. As a maintainability sub-characteristic, *modifiability* is the degree to which a system can be effectively and efficiently modified without introducing defects or degrading existing product quality; *testability* is another sub-characteristic, defined as the degree of effectiveness and efficiency with which test criteria can be established for a system or component and tests can be performed to determine whether those criteria have been met [68]. As IPCs represent run-time interactions among components in a distributed system, IPC characteristics of the system are clearly relevant to its modifiability and testability. Due to the (implicit) interdependencies of one component on another underlying an instance of IPC, changing one component will impact the other and even imply the other having to be changed too. Similarly, such interdependencies also necessitate isolation in testing, thus testability is also clearly related to IPCs.

We measure the modifiability of a system using a direct metric *code churn size*—the number of source lines of code changed (i.e., added, deleted, or updated), between consecutive versions of the system [149]. It is then normalized by the code size (logical SLOC) of the later version. The system-level code churn size is computed as the mean code churn size across all pairs of consecutive versions in the system's release history. We collected the entire release history accessible to us through respective online resources, including the release date and source code of each version. Specifically, the history of XNIO [143], xSocket [144], Grizzly [62], Karaf [75], and Netty [98] was all from respective Maven repository; the history of OpenChord was obtained from SourceForge [103]; the history of QuickServer [111] was from its project site; the history of Thrift [136] and Derby [35] was from Apache Projects, and the history of ZooKeeper [155] and Voldemort [139] from their GitHub repositories.

To illustrate our direct measure of code churn size, Table 4 shows the change history of one of our subject systems Thrift, including the version (the first row), size (logical SLOC) of each version (the second row), number of changed lines of code in each version relative to the previous version (i.e., code churn size, as shown in the third row), and the normalized code churn size (the last row). For example, from version 0.3.0 to 0.4.0, 909 lines of code were changed; thus, the changed code size for this pair of versions was 909, the logical SLOC of the later version (0.4.0) was 6,302, and thus the normalized code churn size is 909 / 6,302 = 0.1442. The system code churn is the average of the per-pair code churn sizes, which is 0.0657 in this example.

Table 4. Direct measures of the sub-characteristic *modifiability* using the quality metric *code churn sizes* of Thrift

Version	0.3.0	0.4.0	0.5.0	0.6.0	0.6.1	0.7.0	0.8.0	0.9.0	0.9.1	0.9.2	0.9.3	0.10.0	0.11.0
Logical SLOC	5,499	6,302	8,064	9,302	9,302	9,427	9,983	10,384	11,092	12,554	12,844	13,264	13,543
Code churn size	984	909	296	1828	126	179	650	189	543	596	179	749	209
Normalized code churn size	0.1789	0.1442	0.0367	0.1965	0.0135	0.0190	0.0651	0.0182	0.0490	0.0475	0.0139	0.0565	0.0154

In addition, we quantify both maintainability sub-characteristics (modifiability and testability) of a distributed system using the *run-time cyclomatic complexity* as a direct quality metric [94], normalized by the size (logical SLOC) of the system. We computed this direct metric, from each of the executions used for computing the IPC metrics for the respective subject, as the number of independent paths exercised during the execution. For each subject, these executions were all obtained by executing its single version as listed in Table 3 (first column).

Functional Suitability. As per ISO/IEC 25010, functionality suitability is the degree to which a system provides functions that meet stated and implied needs when used under specified conditions. As its sub-characteristic, *functional correctness* is the degree to which a system provides the correct results with the needed degree of precision [68]. Functional correctness is obviously related to the IPC characteristics of a system: after all, IPCs are integral building blocks of the system's functionalities; the system as a whole would not provide correct results without having the IPC-induced behaviors conforming to the system's holistic functional requirements.

We quantify the functional correctness of a system via the *defect density* of the system—the number of defects normalized by the system's size in terms of logical SLOC. This is an intuitive, reverse measure of functional correctness: the higher the defect density, the lower the functional correctness of the system considered. To compute the defect density for each of the 11 subjects, we targeted the versions shown in Table 3 and collected the data on defects available to us from relevant resources (e.g., project repositories, bug databases). Specifically, we gathered the defect data for XNIO [115] from its Maven repository. For OpenChord [128] and xSocket [91], the defects were collected from SourceForge [129]. For Thrift [6], ZooKeeper [109], Derby [7], and Karaf [92], we found their defects in the Jira database. The defects of Volemort [110], Netty [93], and Grizzly [58] were obtained from respective GitHub issue collections. For QuickServer [8], we searched issues from the Google Code Archive storage [120].

For each subject, the total number of defects was then divided by the logical SLOC to obtain the defense density. For example, we found 33 defects for Voldemort (1.9.6) whose logical SLOC is 66,754; thus, its defect density was 33/66,754 = 0.000493, meaning that there were 0.000493 bugs per line of source code in this system.

Security. In ISO/IEC 25010, security is defined as the degree to which a system protects information and data so that persons or other systems have the degree of data access appropriate to their types and levels of authorization. It is composed of five sub-characteristics: *confidentiality*, the degree to which a system ensures that data are accessible only to those authorized to have access; *integrity*, the degree to which a system prevents unauthorized access to, or modification of, computer programs or data; *non-repudiation*, the degree to which actions or events can be proven to have taken place so that the events or actions cannot be repudiated later; *accountability*, the degree to which the actions of an entity can be traced uniquely to the entity; and *authenticity*, the degree to which the identity of a subject or resource can be proved to be the one claimed [68]. IPCs are the links connecting distributed components hence constitute a major attack source and source of vulnerabilities. For instance, it essentially enables a remote attacker to intrude a local component of the entire distributed system. For another example, these links are the channels for data leaks. Thus, IPC-induced behaviors are highly relevant to all the five aspects (sub-characteristics) of security.

We measure the sub-characteristic confidentiality with two direct quality metrics in terms of dynamic information flow in the analyzed execution: *Information flow path count*, computed as the total number of dynamic information flow paths, and *information flow path length*, computed as the average length of the paths. The length of an information flow path is the number of statements on the path. As for quantifying other dynamic quality metrics, we computed these two security metrics for a single version of each of the 11 subjects (as listed in Table 3). We also normalized both metrics by the subject size in terms of logic SLOC. Table 5 gives our confidentiality measures of all subjects with respect to the executions generated using the chosen test inputs (Table 3). For some subjects (i.e., Derby, Karaf, Grizzly, OpenChord, QuickServer, and XNIO) and test types (i.e., integration test and system test of ZooKeeper), we did not find any information flow path. Thus, they are absent in Table 5.

Table 5. Measurement results for the sub-characteristic *confidentiality* using two direct quality metrics *information flow path count* and *information flow path length*

Subject Execution	Thrift	xSocket	Voldemort	ZooKeeper_Load	Netty
Raw information flow path count	3	2	42	64	2
Logical SLOC	13,543	11,628	66,754	50,577	109,450
Normalized information flow path count	**2.22E-04**	**1.72E-04**	**6.28E-04**	**1.27E-03**	**1.83E-05**
Raw information flow path length	171	55	362	466	2,477
Normalized information flow path length	**1.26E-02**	**4.69E-03**	**5.43E-03**	**9.22E-03**	**2.26E-05**

For all the five security sub-characteristics, we consider *attack surface* as a common, direct quality metric, which characterizes the system security in three dimensions: methods, channels, and data. In prior works, such measurements of attack surface have been used to determine whether one system is more secure than others [87]. We quantify the attack surface of a subject system via a triple <**M**, **C**, **D**>, where **M** is the number of executed methods including entry/exit points (sources/sinks), **C** is the number of channels (network ports) used across all the distributed processes of the subject, and **D** measures untrusted data (the number of files which are read or written by the subject during the execution considered). Then, we treated the triple as a coordinate and calculated the Euclidean distance between it and the origin (0,0,0) as a direct measure of the whole-system attack surface:

$$\sqrt{M^2 + C^2 + D^2} \tag{10}$$

We then normalized the distance by the subject's source code size (logical SLOC).

In addition, we consider another direct quality metric, *vulnerableness*, as a common measure of the five sub-characteristics of security. We qualify the vulnerableness of a subject via its Common Vulnerability Scoring System—CVSS (Version 2.0) [102] score, a vulnerability metric based on public security databases (e.g, the National Vulnerability Database (NVD) [100]) as used in prior works [4, 147]. We also consider the timeliness of a vulnerability's CVSS score. If some vulnerabilities were found in recent years, they should be more concerning than those discovered long ago. Some subjects (i.e., OpenChord, xSocket, Voldemort, XNIO, and Grizzly) do not have Common Vulnerabilities and Exposures (CVE) [101] vulnerabilities documented in the NVD. For those subjects, we considered the numbers of security vulnerabilities (as explicitly documented so) in the same online defect sources as used for directly measuring *functional correctness*. Also, a vulnerability documented as a CVE in the NVD generally carries a higher weight than a vulnerability in a defect database in the wild, because the former has been publicly disclosed and rated as generally critical (otherwise they would not have been accepted into the NVD). Thus, we assigned CVSS scores (which are always >= 1) as the weights for vulnerabilities with CVE records; other security vulnerabilities were assigned a weight of 1. We then computed the vulnerableness of a subject as

$$N_{misc} + \sum_{CVE\ e} (CVSS_e * (100 - age_e/100)) \tag{11}$$

where $CVSS_e$ is the CVSS score of a vulnerability e reported as a CVE, age_e is the age of e calculated as the difference between the current year and the year the vulnerability e was first entered into the NVD, and N_{misc} is the number of vulnerabilities found in miscellaneous defect sources. Given the nature of this computation, we normalized the vulnerability as computed above by the subject's logical SLOC.

In summary, we use a total of eight direct quality metrics (shown at the bottom layer of Figure 3) to measure nine sub-characteristics for four characteristics that form our reference-quality model. Among these quality metrics, *code churn size*, *defect density*, and *vulnerableness* are static—we computed one metric value for each subject system; the others five metrics are dynamic, computed per subject execution—the system-level metric value is simply computed as the mean of per-execution measures for all the executions considered for each subject (see §5.1.1). As a high-level overview, Table 5 lists the system-level measurement results of the two information flow based quality metrics. The results of the other metrics are summarized in Table 6.

Table 6. Measurement results of other (six) direct quality metrics for all subjects—all normalized by code size in logical SLOC

Subject	Execution Time (seconds)	Code Churn Size	Cyclomatic Complexity	Defect Density	Attack Surface	Vulnerableness
XNIO	1.51E-03	9.28E-02	2.03E-01	9.34E-03	4.55E-03	1.01E+00
OpenChord	8.45E-03	8.87E-02	1.72E-01	2.35E-03	1.13E-02	6.26E-01
xSocket	9.46E-04	4.83E-02	2.38E-01	5.16E-04	1.98E-03	2.58E-01
QuickServer	8.23E-04	4.73E-02	2.45E-01	2.99E-04	7.10E-04	4.38E-01
Thrift	5.91E-04	6.57E-02	1.45E-01	4.28E-03	2.51E-03	8.14E+00
Grizzly	3.08E-04	7.81E-02	2.21E-01	1.98E-03	6.30E-04	8.80E-01
Karaf	5.13E-04	1.17E-02	1.66E-01	9.08E-03	3.01E-04	1.96E+00
ZooKeeper	4.60E-04	1.37E-02	5.17E-02	7.41E-03	6.16E-03	9.00E-01
Voldemort	4.19E-04	3.97E-03	2.42E-01	4.93E-04	4.61E-03	1.80E-01
Netty	1.10E-04	1.11E-02	2.33E-01	5.03E-03	7.13E-04	1.80E-01
Derby	5.90E-05	2.70E-02	1.55E-01	9.70E-03	4.91E-04	5.12E-01

5.1.3 Experimental Procedure. Now that we have the IPC and direct quality measurement facilities and subject datasets, we are ready to answer the two research questions laid out earlier as follows.

For RQ1, we characterize the IPC coupling/cohesion and relevant other traits of distributed systems in different execution scenarios, by measuring IPC in those executions with respect to the proposed IPC metrics. We computed all the IPC metrics (§4) against all the subjects and run-time inputs (Table 3), using our dynamic dependence abstraction framework [22]. The goal of answering this question is to demonstrate the practicality of IPC measurement with DistMeasure for distributed systems hence its usefulness in quantifying their IPC characteristics (i.e., component/process-level coupling and cohesion) as a way of measuring their IPC-induced behaviors.

For RQ2, with the IPC measures computed for RQ1 and the direct measures of these quality metrics computed as detailed above (§5.1.2), we performed extensive statistical analyses to examine the correlation between each of the six IPC metrics and each of the eight quality metrics. We used Spearman's rank correlation analysis [122], a non-parametric method making no assumptions about the relationship between the two variables involved (which is the reason why we chose Spearman's method over alternatives such as Kendall's [1] and Pearson's [13] methods). Further, we adopted the interpretations of correlation strength according to varied value ranges of Spearman's rank coefficient r in [121]: the correlation is *very weak* if $|r|$ is below 0.20, *weak* if $|r|$ is between 0.20 and 0.39, *moderate* if $|r|$ is between 0.40 and 0.59, *strong* if $|r|$ is between 0.60 and 0.79, *very strong* if $|r|$ is 0.8 or above. For each coefficient, we also computed the p value associated with it at a typical 0.95 confidence level, which indicates the statistical significance of the correlation.

Table 7. Measurement results of system-level IPC metrics

Subject Executions	RMC	RCC	CCC	IPR	CCL	PLC
XNIO	16.82	58.96	2.76	0.51	74.43	41.10
OpenChord	3.14	67.15	5.29	0.78	267.36	255.78
xSocket	20.74	76.77	3.63	0.36	257.98	208.19
QuickServer	2.12	36.21	2.42	0.40	113.27	70.85
Thrift	11.59	25.27	3.17	0.56	23.40	41.59
Grizzly	39.68	152.09	2.16	0.67	665.13	673.34
Karaf	2.85	22.31	1.06	0.45	70.16	78.32
ZooKeeper	6.26	191.83	3.04	0.42	506.32	461.36
ZooKeeper Load	4.01	90.02	1.19	0.37	391.41	369.29
ZooKeeper System	4.84	131.32	2.62	0.39	382.56	332.70
Voldemort	40.17	301.54	5.02	0.54	528.32	569.79
Netty	1.00	129.00	2.38	0.54	863.39	765.36
Derby	3.31	29.45	2.22	0.72	717.70	734.12

We used our dynamic analyzers [22, 50] for distributed programs to compute information flow paths hence the two direct quality metrics based on such paths. The lists of information sources and sinks involved were curated based on the documentation of Java security/cryptography APIs. We used a Java source code measurement tool LocMetrics [10, 76] to compute the code size of a system (i.e., logical SLOC). To compute the run-time cyclomatic complexity of each subject execution, we instrumented the subject and counted the number of simple decisions (predicates) exercised in the instrumented execution. The `diff` tool [11] was used to compute the code churn sizes between consecutive versions of each subject. All the machines used in our study were Ubuntu Linux 18.04.1 LTS workstations, each with an Intel E7-4860 2.27GHz CPU and 256GB DRAM.

5.2 Results

This section presents our major results for the measurements of our subject systems using the proposed IPC metrics and their relations to the quality metrics, in response to the two research questions (RQ1 and RQ2).

5.2.1 RQ1: IPC Characterization. Our results on IPC characterization are summarized in Table 7. Each number represents one of the six IPC metrics computed for one subject and one type of test (i.e., at system level). For all subjects but ZooKeeper, the test type is omitted as only one type of (i.e., integration) test inputs was available. Recall that for each subject and test type, we have extensively augmented the test inputs as described earlier (§5.1.1). The results shown are the mean IPC metric values over all the executions per subject and test type. We also note that since our per-execution (system-level) IPC metrics were computed as averages (over all processes/classes in the execution), they do not need to be further normalized (as for the quality metrics using logical SLOCs).

The RMC results show that two enterprise systems, Grizzly and Voldemort, had the highest degree of message coupling among their distributed processes. The main reason is that during their integration-test executions these two systems exhibited an extraordinary level of inter-component dependency leading to a larger number of message exchanges among the processes. The integration test needed all the processes to collaborate closely, hence a high RMC as a result. Intuitively, the RMC measures immediately indicate the complexity of IPC in terms of message passing among processes. The higher RMC metric values also implied heavier network communications. We observed that the RMC values did not seem to be consistently related to the system size (in terms of logical SLOC)—the processes in the largest system (Derby) were very lightly coupled, less than those in the smallest system (XNIO), despite both being with respect to the same type of test inputs (i.e., integration test).

The RCC measures appeared to be independent of RMC, evidenced by the absence of association between the two metrics: In short, higher or lower RCC did not consistently co-occur with higher or lower RMC. Similar to RMC, RCC saw no correlation with system size either. The numbers revealed that during the integration test, Voldemort had the most highly coupled processes at class level, followed by Zookeeper, while in Karaf, a class in one process did not influence much the classes in other processes.

Like RMC and RCC, the other four metrics (i.e., CCC/IPR/CCL/PLC) saw no significant correlation with subject size nor with any other IPC metrics—per our computation, the correlations between any two of these IPC metrics were weak or very weak. An implication of the latter is that each of them is uniquely expressive/significant—none of them subsumed any others. What was clearly different between RMC/RCC/CCL/PLC and CCC/IPR was that the 13 measures saw substantial variations in RMC/RCC/CCL/PLC, yet very small ones in CCC/IPR. This contrast suggests that RMC/RCC/CCL/PLC can vary widely across systems, while CCC/IPR tends to fall in relatively narrow ranges.

In terms of the numbers, the mean CCC was mostly between one and five, meaning that every class collaborated with about one to five other classes in remote processes. Complementary to the four metrics (i.e., RMC/RCC/CCC/CCL) that concern the dependence/reliance of one process on others (albeit at different levels of granularity), IPR informs the *common* dependence of processes. That said, OpenChord's processes had a much higher degree of common functionalities shared among them than other systems; in contrast, xSocket had very little functional sharing among its processes during the integration test.

Regarding CCL, our measurement results show that by average, every single class contributed noticeably, if not substantially, to the interprocess communication loads in any execution of any subject system. The observation that these subject executions had CCL values between 23 to 863 suggests that every class depended on 23~863 times as many methods in remote processes as those in itself—in other words, on overall average every method in an executed class depended on 23~863 methods throughout all remote processes in the studied system executions. We also found that the remote dependence sets of methods within a class largely overlapped. Nevertheless, the CCL numbers still indicate generally substantial interdependence (hence functional coupling) among the distributed processes in these systems. The largest CCL was seen by Netty, indicating its greatest (among the 13 cases) per-class contribution to the communication loads among its distributed processes—this highest level of interprocess communication complexity may suggest the greatest difficulty in debugging the IPC performance/correctness with this system.

Our results point to some similarities between PLC and CCL—the PLC values ranged from 41 to 765, meaning every method within a process depended on 41~765 other methods in the same process. Thus, in terms of PLC for cohesion and CCL for coupling, our subject executions have seen overall notable closeness between process-level cohesion and process-level coupling. Despite a lack of strictly consistent correlation between PLC and CCL, higher/lower PLC generally came along with higher/lower CCL. For instance, Thrift had the lowest CCL and its PLC was almost the lowest too. On the other hand, Netty had the highest PLC while having the highest CCL as well, suggesting busy interactions both within and across processes in this subject's executions. In all, at process level, having generally more (less) cohesive processes did not make the processes less (more) coupled.

Answer to RQ1: The six IPC metrics each measured/characterized the IPC coupling or cohesion in a uniquely significant perspective—no one metric subsumes/implies another. Meanwhile, RCC/RMC/CCL/PLC can be sharply different among different systems while other metrics (i.e., CCC/IPR) seem to be relatively stable across systems. No IPC metric seems to be correlated with system sizes, even in the same type of execution scenarios; and no consistent correlation was seen between any two IPC metrics—the only exception was between CCL and PLC, suggesting at process level high/low coupling did not make cohesion low/high.

5.2.2 RQ2: Quality Relevance of IPC . The data groups underlying our correlation analysis include (i) the dynamic measures with respect to each of the six IPC metrics overall subject executions (as summarized in Table 7) and (ii) the direct measures of each of the eight quality metrics overall subjects or subject executions (as summarized in Table 5 and Table 6). Since we examine the correlation (p-value) between each IPC metric and each quality metric, we performed the correlation analysis for 6x8=48 pairs of (i) and (ii) data groups.

Table 8. Spearman's correlation coefficients (p-values) between our IPC metrics and quality metrics

Quality metrics			IPC metrics					
Level	**Type**	**Name**	**RMC**	**RCC**	**CCC**	**IPR**	**CCL**	**PLC**
System	Static	Code Churn Size	0.2320 (4.46E-01)	-0.3260 (2.77E-01)	0.2541 (4.02E-01)	0.2707 (3.71E-01)	-0.3978 (1.78E-01)	**-0.4972** (8.38E-02)
		Defect Density	-0.2210 (4.68E-01)	-0.3149 (2.95E-01)	**-0.4475** (1.25E-01)	0.0497 (8.72E-01)	0.0166 (9.57E-01)	0.0331 (9.14E-01)
		Vulnerableness	0.0553 (8.58E-01)	**-0.4205** (1.53E-01)	-0.2407 (4.28E-01)	-0.0470 (8.79E-01)	**-0.6224** (2.31E-02)	**-0.5616** (4.58E-02)
Execution	Dynamic	Execution Time	-0.2845 (0.00E+00)	-0.0864 (2.39E-29)	**0.4735** (0.00E+00)	0.0461 (2.11E-09)	-0.1554 (1.10E-91)	-0.2471 (3.32E-233)
		Cyclomatic Complexity	**0.4239** (0.00E+00)	0.0573 (9.16E-14)	**0.4831** (0.00E+00)	0.1988 (5.77E-150)	-0.3644 (0.00E+00)	-0.3941 (0.00E+00)
		Information Flow Path Count	0.1295 (4.99E-64)	0.0777 (4.97E-24)	-0.0091 (2.38E-01)	-0.3771 (0.00E+00)	-0.1327 (3.31E-67)	-0.1570 (1.15E-93)
		Information Flow Path Length	-0.1616 (3.45E-99)	0.0349 (5.77E-06)	-0.1159 (1.50E-51)	-0.2494 (1.24E-237)	0.1134 (2.07E-49)	0.0757 (6.59E-23)
		Attack Surface	0.1556 (5.35E-92)	-0.2132 (9.49E-173)	**0.4534** (0.00E+00)	-0.0868 (1.42E-29)	**-0.7424** (0.00E+00)	**-0.7968** (0.00E+00)

Note that the two groups in each pair need to be of equal sizes. For each of the three quality metrics that are *static* (one measure per subject)—code churn size, defect density, and vulnerableness, the group had 11 data points (as seen from the first column of Table 3); accordingly, we had to use one aggregate IPC measure for each subject as well. Thus, the data points underlying the statistical analysis were from different systems, not from different versions of the same systems. Because these systems are quite different in terms of run-time behaviors and application logic, we did not consider that they were dependent on each other with respect to their measures in our IPC and quality metrics. To that end, we took the mean of the measures per IPC metric over all the executions of each subject (further across the three test types for ZooKeeper). The other five quality metrics are *dynamic* (one measure per subject execution), the group had a total of 16,880 data points (as seen from the last column of Table 3); accordingly, we just used the IPC metric value computed for each subject execution without any aggregation.

Table 8 lists the 48 resulting Spearman's rank coefficients and their associated statistical significance (p values at 0.95 confidence level, shown in the parentheses). The results show widely varying relations of our IPC metrics to the eight quality metrics. We regard a moderate or strong correlation as *noteworthy* and marked such cases in boldface. Yet we also note that these coefficient numbers themselves are not supposed to convey significance or signify a conclusion; rather, their *contrasts* tell the *comparative* quality levels among systems/executions.

Regarding **code churn size**, our results revealed that PLC tended to have a noteworthy correlation with this quality (modifiability) metric. Recall that PLC is a process-level cohesion metric and that high cohesion is commonly known to benefit maintainability in general [54, 55]. Thus, the negativity of this correlation is intuitive: the systems with more cohesive processes/components are easier to modify—that is, changes made to any process/component more readily propagate throughout the process/component. As a result, fewer changes are required, leading to lower code churn sizes in an average case.

With **defect density**, CCC was the only IPC metric noteworthily correlated. By definition, a higher CCC is a strong indicator of higher coupling among all the processes throughout the entire system execution (as opposed to RCC indicating the average coupling between any two processes), hence that of overall higher interprocess complexity. Meanwhile, recall that in our framework defect density is computed based on defects reported publicly. Intuitively, finding defects can be more difficult in a system with more complex inter-component interactions (i.e., interdependencies), resulting in less defects having been reported. This justifies the correlation between CCC and defect density being negative. In fact, our original rationale for introducing the CCC metric was mainly due to its potential connection to the *correctness* quality sub-characteristic, for which defect density is the key quality metric considered in DistMeasure.

Our results indicate that CCL and PLC were notably informative about system **vulnerableness** with respect to our direct measure of vulnerableness in terms of the number of vulnerabilities reported publicly in the past. The negativity of the correlation further indicates that higher degrees of coupling and communication loads among classes and a higher level of cohesion between processes were associated with lower vulnerableness. One reason is that it is hard for users to find vulnerabilities in complex systems with higher class coupling, communication loads, or process cohesion. In particular, for distributed systems, there are few available tools for discovering security vulnerabilities induced by implicit dependencies across processes [46], which are used to compute RCC/CCL. Accordingly, a higher RCC or CCL means more such implicit dependencies hence greater difficulties in discovering those vulnerabilities. In fact, it was found that in distributed systems more vulnerabilities were found on those implicit (interprocess) dependence chains than those found within individual processes [46]. As a result, systems with higher RCC/CCL may have fewer vulnerabilities reported, hence lower vulnerableness measured directly.

The positive correlation of CCC with **execution time** revealed that a higher degree of class central coupling was consistently related to longer execution time. Justifiably, given that a system needs collaborations among its classes to finish a task when one class has to communicate with more others across processes (as implied by the higher interactivity), it takes longer to carry out the task. As mentioned above, CCC measures the *overall* coupling among all processes in the underlying distributed system execution; a higher CCC strongly indicates a higher level of the overall complexity of the execution, which is intuitively associated with greater time cost of that execution.

The above observation that the higher overall system execution complexity was associated with higher CCC intuitively explains the positive correlation between CCC and run-time cyclomatic complexity—a direct quality metric dedicated to measuring the execution complexity. We also observed that higher message coupling, measured by RMC, was indicative of higher execution complexity as well. It turned out that a process typically sent messages to others

under certain conditions. Thus, that more messages were exchanged among processes, as captured by a higher RMC, implies that more such predicates (in those conditions) were exercised, hence a greater run-time cyclomatic complexity.

Although more functionality reuse, indicated by higher IPR, tended to be associated with fewer and shorter information flow paths, the correlation was relatively weak.

Among our six IPC metrics, CCC was found to be noteworthily and positively correlated to our direct **attack surface** measure. Thus, larger class central coupling during a system execution implies a broader attack surface as measured. Justifiably, a higher degree of overall process coupling, indicated by a greater CCC, points to a higher level of system execution complexity as discussed earlier, while it is known that more complex software tends to have more vulnerabilities hence broader attack surface [3]. Meanwhile, CCL and PLC were found to be noteworthily but negatively correlated to the quality metric *attack surface*. This is because a higher *internal* class-level communication or process-level cohesion means a lower attack surface exposed to the *external* of the system (e.g., adversaries).

In addition to the correlation coefficients, we also reported the p-value of each coefficient, which is the probability that the coefficient is not statistically significant. Since we computed these p-values at the 0.95 confidence level, $p<0.05$ here indicates a statistically significant correlation. As shown in Table 8, between any dynamic quality metric and any of our IPC metrics that had a noteworthy correlation (shown in boldface), the p-values were all 0.00E+00 (zero). That is, all of the noteworthy correlations were also (strongly) statistically significant.

On the other hand, regarding the static quality metrics, only the noteworthy correlations between CCL or PLC and **vulnerableness** were statistically significant; the other three noteworthy ones were not. The main reason was that the data points underlying the statistical analysis were only few for the static quality metrics. For the same reason, we need to be cautious about the noteworthy correlations here even when they were associated with a $p<0.5$: due to the small sample size, such obtained statistical significance may have been biased. Also, more broadly, despite our consideration of diverse subjects and their various executions, the strength or even significance of some of the correlations may not always hold as we observed, due to factors such as software development process, application domain, codebase complexity, and technical debt, among others. For instance, embedded systems typically aim for minimal code churn due to high stability requirements, meaning PLC correlations could become weak or irrelevant in such domains. Thus, such additional factors need to be considered by practitioners when deciding which IPC metrics to use or be expected to be effective for understanding a given quality concern.

Answer to RQ2: Five of our six IPC metrics were significantly correlated with one or more of six (out of eight total) quality metrics. Higher IPC coupling in terms of dependencies of one process/component on others (via higher RMC/RCC/CCC) was significantly correlated to lower defect density, fewer reported vulnerabilities, longer execution time, higher complexity, and greater attack surface. Also, more class communication loads (via higher CCL) implied lower vulnerableness and smaller attack surface(s). In addition, higher process cohesion (via higher PLC) informed smaller code churn size, fewer reported vulnerabilities, and smaller attack surface(s). The small (zero) p-values for the coefficients further indicate strongly statistically significant correlations between the five IPC metrics and corresponding dynamic quality metrics considered.

5.3 Discussion

Our exploration of IPC metrics not only demonstrated the practicality of measuring IPC in large, real-world distributed systems but also revealed the substantial presence (albeit with varying degrees) of *implicit* coupling among distributed

components which are generally decoupled in the architectural design of common distributed software. And we showed that one way to reveal such implicit coupling is through measuring interprocess coupling.

Our results on IPC measurements revealed that higher coupling in terms of inter-process/component dependencies is generally bad for quality in three of the four IPC-relevant quality characteristics considered. Specially, in distributed system executions, higher coupling tended to be associated with undesirable time behavior hence lower performance efficiency, lower modifiability and testability hence compromised maintainability, and lower overall security in terms of greater vulnerableness and attack surface. This largely confirmed the drawbacks of high coupling in general [74] and that they are correlated with (integration) failure proneness in component-based software in particular [64]. Higher process-level cohesion can benefit quality via enhanced security in multiple sub-characteristics (e.g., reduced lower attack surface and vulnerableness). This consolidates the previous finding that high cohesion is a positive property of software processes/components that tend to have high quality [55].

We observed that CCC was significantly correlated with three quality metrics (i.e., execution time, run-time cyclomatic complexity, and attack surface) while RCC was not significantly correlated with any quality metric, although these two IPC metrics are very closely related to each other—in fact, in terms of metric computation, CCC is derived from RCC (Equation 5). Yet as noted earlier, CCC captures the overall coupling among *all* of the distributed processes in a system execution, while RCC is the average coupling between *two* processes in the execution. This contrast suggests that the more holistic coupling measure may be more indicative of distributed system quality.

Among the four quality characteristics in our reference-quality model (Figure 3), security was significantly correlated with half (CCC, CCL, PLC) of the six IPC metrics via the direct quality metric of attack surface. In comparison, any of the other three quality characteristics was significantly correlated with at most two IPC metrics. This result indicates that IPC metrics were most informative about security, with respect to metrics of attack surface and (information flow) vulnerabilities, in distributed systems for their confidentiality, integrity, non-repudiation, accountability, and authenticity according to the ISO/IEC 25010 standard, as shown in Figure 3. For example, in a distributed system execution, a low PLC (i.e., cohesion among the distributed processes) suggests high attack surface hence low security (confidentiality/integrity/non-repudiation/accountability/authenticity) of the execution, according to the strong, significant (negative) correlation between PLC and the direct attack surface metric (Table 8). In the contrary, a comparatively higher process-level cohesion would signify a lower security risk in terms of attack surface.

Based on our empirical findings, distributed system developers are recommended to attain and maintain a low degree of *overall* (implicit) coupling among systems components in order to achieve and sustain high quality, especially when concerning performance efficiency, functional suitability, and security. To that end, the content of messages can be optimized to reduce the number of messages sent between processes hence reducing RMC, while interprocess dependencies at class and method levels should be reduced (e.g., by implementing more functionalities in local components rather than heavily relying on remote ones) hence to reduce RCC, CCC, and CCL. On the other hand, developers are encouraged to promote cohesive components in light of the benefits of higher process-level cohesion to better maintainability and security. For that purpose, conventional design practices for higher cohesion (e.g., increasing modularity via refactoring the code of local components) may be adopted. Yet despite these benefits of low coupling and high cohesion in IPCs, a caveat to developers is that they should not assume that achieving one would automatically help achieve the other; instead, low coupling and high cohesion are better-targeted each with dedicated efforts.

Meanwhile, given all these correlations between IPC and quality, developers can leverage IPC measurements in terms of our proposed metrics to understand/analyze hence improve the quality of distributed systems with respect to a

variety of quality characteristics/sub-characteristics and metrics. Accordingly, IPC metrics may be leveraged to assess direct quality measures according to their correlations, as we investigate next.

6 PART 3: PREDICTIVE QUALITY ASSESSMENT

In this section, we present how DistMeasure enables us to further understand the characteristics of IPC-induced behaviors and the usefulness of IPC measurements from the perspective of their predictive relationships with different quality characteristics/sub-characteristics through relevant direct quality metrics, in reference to the quality model defined earlier. To that end, DistMeasure builds learning-based anomaly detection models for classifying the high-level status (`normal` versus `anomalous`) of distributed software quality in terms of those quality metrics. These classification models are based on (1) different subsets of the IPC coupling/cohesion metrics, (2) the direct metrics of various quality characteristics/sub-characteristics, and (3) the statistical correlations between (1) and (2) as found earlier. The main goal of DistMeasure, through its **Part 3**, is *not* to refute/disprove the justification and needs for those direct quality metrics, but rather to demonstrate in further depth the correlation and implications of IPC-induced behaviors on the quality, in many ways (i.e., via the various quality aspects), of distributed software. This is consistent with the primary goal of this paper, which is to answer the overarching questions (§1). Meanwhile, through this part of DistMeasure, we also aim to demonstrate the potential of IPC measurement in terms of supporting a practical and useful means for assessing various aspects of distributed system quality. While the high-level quality anomaly detection capability offered by this **Part 3** of DistMeasure alone may not immediately suffice for diagnosing/locating quality issues or providing detailed guidance for fixing those issues, the predictive capability can be useful in at least two ways. First, it provides useful information towards ultimate quality issues diagnosis and quality improvement—for instance, developers may use our measurement results to first identify anomalous quality conditions as a first step, before they proceed with further diagnostic actions. Second, developers/researchers can use our measurement capabilities and results to build tools that help predict/localize issues at a fine-grained level, given that we do have fine-grained metrics such as those at method (IPR), class (RCC, CCC, and CCL), and component/process (RMC, RCC, and PLC) levels. The IPC measurement enabled results are also explainable per the definition of the IPC metrics and their relationships with direct quality metrics.

In the following, we first describe our approach to these classifications (i.e., the quality anomaly detection models) (§6.1), followed by evaluating the classifiers (§6.2) via two research questions as listed below.

- **RQ3: How effective are the IPC metrics based quality classifications (anomaly detection)?** The goal of answering this question is to further quantitatively assess the quality relevance of IPC-induced behaviors in distributed systems as characterized using our IPC metrics. The effectiveness quantifies the strength of that relevance.
- **RQ4: Which factors affect the performance of our quality classifiers (anomaly detectors)?** The goal of answering this question is to better understand which IPC metrics have particularly significant correlation with the quality of distributed systems, and potentially how the measurement capabilities of DistMeasure may enable designing future effective quality (anomaly) classifiers.

In addition to answering these questions, we then also discuss the implications of our evaluation results (§6.3).

6.1 Approach

To accommodate different use scenarios and meet diverse user needs, DistMeasure uses both unsupervised and supervised learning strategies in constructing the quality classifiers, while experimenting with different learning algorithms to identify the most performant one for each strategy. The supervised classifiers apply to situations in which

users can afford to compute all the dynamic quality metrics hence have labeled training datasets to enable potentially more accurate classifications. In contrast, the unsupervised classifiers may offer lower accuracy but accommodate scenarios where the users do not have enough labeled data for supervised training. Computing some (if not all) direct quality metrics (e.g., those based on information flow paths) can be very expensive [22, 46]. Even when a direct quality metric can be affordably computed, a single metric value may not be sufficient to tell the quality status with respect to the quality characteristic/sub-characteristic measured by the metric. These are also the reasons why it may be better to use IPC metrics to indirectly classify quality status rather than immediately using the direct quality metrics in the training of and inference with such classifiers.

6.1.1 Learning Features. We consider each of the six proposed IPC metrics (i.e., RMC, RCC, CCC, IPR, CCL, and PLC) as a *potential* feature in building a supervised and unsupervised classifier for each of the direct quality metrics with which at least one of the IPC metrics was found to be significantly correlated. We refer to such quality metrics as *predictable quality metrics*. As per Table 8, DistMeasure currently supports quality classifications with respect to three static and three dynamic quality metrics that are predictable. While most of these IPC metrics are defined at different granularity (i.e., method, class, process, system) levels, here for each subject system execution we only use the values of the system-level metrics as feature values in **Part 2** of our framework.

Specifically, for each predictable quality metric Y, DistMeasure splits the set X of IPC metrics that have been found significantly correlated with Y into two subsets: $X1$ consisting of IPC metrics with which the correlations were positive, and $X2$ of those with which the correlations were negative. Then, DistMeasure trains a classifier using each non-empty subset, as elaborated as follows. For each classifier $X \rightarrow Y$, we only included in X the IPC metrics with which Y's correlation coefficient was found statistically significant, because those that are not significantly correlated, if included in the model, would reduce the classification precision [84]. Accordingly (as per Table 8), for dynamic quality metrics, we formed the following classifiers: (CCC)→execution time, (RMC, CCC)→run-time cyclomatic complexity, (CCC)→attack surface, and (CCL, PLC)→attack surface. Similarly, for static quality metrics, we formed one classifier only: (CCL, PLC)→vulnerableness, which we will discuss in §6.3.

6.1.2 Unsupervised Classification. For unsupervised learning based quality classifications, we chose the k-means clustering algorithm from all relevant candidates available in the Scikit-learn library [104] after exhaustive considerations. Data clustering is a data exploration technique, grouping together objects with similar characteristics for facilitating their further processing. The k-means algorithm is a popular data-clustering algorithm, which partitions given data points into k clusters by assigning every data point to the nearest cluster centers [107]. And this algorithm has been widely used for anomaly detection in many domains (e.g.,[30, 83]).

For each classifier $X \rightarrow Y$, Y values were not available in the training set where samples were not labeled, nor were the samples used for testing. To obtain the ground-truth label of each sample (i.e., between `normal` and `anomalous`), we performed outlier detection using k-means clustering [73] such that we label a sample as `anomalous` if it is detected as an outlier, and `normal` otherwise. Given this rationale, intuitively we set k=1, and we first applied the clustering algorithm to all the samples in our dataset, including those in the training and testing sets. Then, we computed for each cluster entry its distance from the cluster center, and regarded the entries as *outliers* whose distance is greater than a threshold τ of percentile over all such distances. In reference to prior similar approaches [73, 89, 138], we chose τ=90, which worked reasonably well for the *attack surface* classifier; for the *execution time* and *cyclomatic complexity* classifiers, however, this threshold resulted in too many outliers, for which we thus set τ=95. Also, while in principle k

could take other values greater than 1, we found those k values tend to produce unstable clustering results. Thus, we stayed with k=1 eventually. With only one cluster, all *normal* data points are covered within the threshold percentile distance (from the cluster centroid). Such obtained ground-truth labels were used for our evaluation purposes.

The unsupervised classifier was trained to cluster the training samples only based on their X values. At inference time, for a given subject execution E under classification, DistMeasure first computes the values E_X of IPC metrics in X. Then, the classifier $X \rightarrow Y$ will classify E according to E_X. The resulting status of E_X as an outlier or not indicates the subject's quality as `anomalous` or `normal` with respect to the quality metric Y. Essentially, the classifier treats normal points as part of the cluster and outliers as outside the cluster.

6.1.3 Supervised Classification. For supervised learning-based quality classifications, we chose the bagging algorithm [15] after comparatively exploring several alternatives. Bagging is a method for generating multiple versions of a model and then using these versions to get an aggregated model. These versions are generated via bootstrap replication of the training data set and using these as new training data sets. When classifying (predicting a class), the aggregation performs a vote to decide the class. The bagging model can provide higher accuracy than many other supervised learning models, especially for unstable data—the prediction accuracy can be retained when there are large variations across the training samples [15]. Due to the diverse scales, domains, and execution scenarios of our subjects, the executions represent very different training samples. The expected large variations in these samples suggest that the bagging algorithm would be a good fit here. Similar to the unsupervised classifications, the supervised classifications in DistMeasure are also formulated as a binary, anomaly detection problem.

For each classifier $X \rightarrow Y$, each training sample was the X vector value computed from a system execution (D_i against T_i of Figure 2). To obtain the ground-truth label of each sample, we used the classical z-score based outlier detection approach [118], such that we label a sample as `anomalous` if it is detected as an outlier, and `normal` otherwise. In particular, we determined a sample as an outlier if its z score is beyond a threshold. To determine this threshold, for each feature set X, we obtained the value distribution of corresponding quality metric (that we intend to predict with X values) via boxplots. We found that in our datasets these distributions were generally right skewed. Thus, we only considered the upper fence of boxplot as the threshold. Specifically, the boxplot upper fence z-score threshold values for the *execution time*, *cyclomatic complexity*, and *attack surface* classifiers are -0.32, 0.90, and 1.46, respectively.

Each supervised classifier was trained on such labeled samples in a given training set. At inference time, for a given subject execution E under classification, DistMeasure first computes the values E_X of IPC metrics in X. Then, the classifier $X \rightarrow Y$ will classify E as `normal` or `anomalous` according to E_X, indicating the subject's quality as normal or not with respect to the quality metric Y.

6.1.4 Experimental Procedure. Our empirical validation of the IPC metrics based quality classifiers is guided by the two research questions outlined earlier (Section 6), which are answered by following the experimental procedure below.

For RQ3, taking each unique subject execution as a sample, we had a total of 16,880 samples. Using this large-scale dataset, for each classifier, we first performed a hold-out validation as in prior works [20, 23, 133]: we randomly selected 30% of all the samples of each class and held them as unseen/novel samples for testing, and used the rest for training. To mitigate the potential bias in the single hold-out split, we further performed a 10-fold cross-validation (CV): the sample set was randomly partitioned into 10 disjoint subsets of equal size; the model was trained on 9 of these subsets and then tested against the remaining subset—this procedure was repeated 10 times.

For testing a supervised classifier $X \rightarrow Y$ against a sample E, the ground-truth classification result was the class label originally assigned to E—we took the mean of the 16,880 Y values as the threshold to label each of the 16,880 samples with one of the two class labels (i.e., `normal` and `anomalous`), as described earlier (Section 6.1.3).

When testing an unsupervised classifier $X \rightarrow Y$ against a sample E, the ground-truth classification result was determined as follows. Suppose the classification model has two clusters C_1 and C_2, for which we computed the mean of Y values of the subject executions each corresponding to a sample (i.e., feature vector) in each cluster, as $M1_Y$ and $M2_Y$, respectively. For E, we also computed the Y value, noted as E_Y. Then, the ground-truth classification result was C_1 if E_Y was closer to $M1_Y$ than to $M2_Y$; otherwise, the ground-truth result was C_2.

Based on the ground-truth results determined as above, we computed the precision, recall, and F1 accuracy for each supervised and unsupervised classifier with both hold-out and 10-fold validations. To explain the classification results (i.e., why the classifiers work), for each classifier $X \rightarrow Y$, we further conducted association analysis between the characteristics (i.e., high/low value ranges) of the IPC metrics in X and the normal/anomalous status of Y.

For RQ4, we aimed to study two common factors that affect the classification performance of the quality anomaly detectors: the choice of learning algorithm and the selection of features. Regarding the algorithmic choice, for unsupervised learning, we only considered the k-means clustering algorithm because it was the only one available in the Scikit-learn library that was suitable for our clustering situation; for supervised learning, we compared the performance of each classifier using the default (bagging) algorithm to that of the same classifier using eight alternative algorithms (with the same training/testing data and setting): Naive Bayes [117], Multinomial Naive Bayes [95], (KBF Kernel) SVM [29], kNN (short for k-nearest neighbor) [131], AdaBoost [44], Voting [81], C4.5 decision tree [112], and Random Forest [16]. We used default parameters for these ML models (e.g., #estimators=50 for AdaBoost and #estimators=10 for Bagging) as in the scikit-learn library. Regarding feature selection, for each bagging classifier that uses more than one feature, we compared feature importance scores and ranked the features according to the scores to identify the most important feature(s). The scores here are what a technique (on feature importance) assigns to each input feature of a predictive model that indicates the relative importance of the feature to the model's decision making (i.e., prediction) [66].

6.2 Results

In this section, we present our experimental results and conclusions for the two research questions regarding **Part 3** of (i.e., quality classifications in) our framework. We mainly focus on the results for the classifiers of *dynamic* quality metrics because we had a reasonably large number of (16,880) subject executions as samples. For the classifiers of *static* quality metrics, the (11) samples available were too few to suffice for reasonably reliable training and testing that could lead to solid conclusions—we discuss these classifiers later (§6.3).

Table 9. The effectiveness of unsupervised (k-means) classification for dynamic predictable quality metrics

Model		**Hold-out Validation**			**10-fold Cross-validation**		
IPC Metric	**Quality Metric**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
CCC	Execution Time	81.02%	90.01%	85.28%	81.52%	90.28%	85.68%
RMC, CCC	Cyclomatic Complexity	81.66%	90.37%	85.79%	81.52%	90.28%	85.68%
CCC	Attack Surface	90.37%	95.06%	92.66%	90.4%	95.08%	92.68%
CCL, PLC	Attack Surface	91.16%	95.48%	93.27%	90.4%	95.08%	92.68%
Average:		**86.05%**	**92.73%**	**89.25%**	**85.96%**	**92.68%**	**89.18%**

6.2.1 RQ3: Effectiveness. Table 9 shows the results for three effectiveness metrics (precision, recall, and F1 accuracy) of the unsupervised classifiers for all of the dynamic predictable quality metrics. For each classifier $X \rightarrow Y$, the first and

second column shows the X and Y respectively. Both of the hold-out validation (3–5th columns) and 10-fold validation (6–8th columns) results revealed that DistMeasure achieved a generally useful level of accuracy (89% F1 on overall average) for classifying the distributed system quality with respect to any of the three quality metrics (*execution time*, *run-time cyclomatic complexity*, and *attack surface*). And four of the six IPC metrics contributed to these classification capabilities: CCC, RMC, CCL, and PLC, which were found in **Part 2** to be significantly correlated with those quality metrics (see Table 8). Notably, the two validation schemes showed very similar precision (86.05% versus 85.96%), recall (92.73% versus 92.68%), and F1 accuracy (89.25% versus 89.18%), not only on overall average but also for each individual classifier This consistency helped consolidate the validity and reliability of these evaluation results.

Overall, the results show that measuring the IPC-induced behaviors in distributed systems using the proposed metrics such as CCC, RMC, CCL, and PLC can reasonably accurately tell about the anomaly status of the three out of the four quality characteristics we considered: performance efficiency (in terms of execution time), maintainability (in terms of cyclomatic complexity), and security (in terms of attack surface). In contrast, the anomaly prediction performance was even better for security than for the other two (F1 accuracy of 92.68% versus 85.68%). An intuitive explanation for this difference is that *attack surface* was computed in a much coarse-grained manner (i.e., based on the number of network ports and files accessed and number of methods executed) than the other two quality metrics (e.g., cyclomatic complexity is computed based on the number of branch predicates exercised in the program). Recall that the (k-means) clustering algorithm works on the basis of similarity. Capturing similarities at finer-grained levels is intuitively more challenging than capturing coarser-grained similarities.

Table 10. The effectiveness of supervised (bagging) classification for dynamic predictable quality metrics

Model		**Hold-out Validation**			**10-fold Cross-validation**		
IPC Metric	**Quality Metric**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
CCC	Execution Time	99.61%	99.61%	99.61%	98.73%	97.55%	97.86%
RMC, CCC	Cyclomatic Complexity	99.84%	99.84%	99.84%	99.76%	99.75%	99.76%
CCC	Attack Surface	99.02%	98.99%	99.0%	98.9%	97.44%	97.9%
CCL, PLC	Attack Surface	98.97%	98.78%	98.83%	98.55%	95.36%	96.23%
Average:		**99.36%**	**99.3%**	**99.32%**	**98.98%**	**97.52%**	**97.94%**

Instead, learning the association patterns between IPC metrics and the predictable direct quality metrics would be more effective for classifying the related quality characteristics/sub-characteristics with respect to those quality metrics. This was corroborated by the effectiveness results of our supervised quality classification in DistMeasure, as shown in Table 10. For the same classification tasks using the same IPC metrics, both the hold-out validation and 10-fold CV revealed that any of the supervised classifiers performed quite effectively—precision, recall, and F1 accuracy were all above 95%; the overall average F1 accuracy was 98%. As in the evaluation results for our unsupervised classifications, these supervised classifiers also had highly consistent numbers in any of the three effectiveness metrics, both for individual classifiers and overall. Thus, when labeled training samples are available, supervised classification was clearly more desirable in terms of classification accuracy in our framework.

To explain why/how these classifiers worked, we also computed the frequent if-then associations consisting of an antecedent (if, IPC metric value being high/low) and a consequent (then, quality metric being anomalous/normal), using the Apriori algorithm [105] that was implemented in the Python Mlxtend library [97], in two steps:

(1) *Data formatting*. From the quality and IPC metric values computed from the subject executions against the corresponding test inputs (cf. Table 3), we set dynamic quality metrics as anomalous or normal according to our ground-truth labeling process as described earlier, and IPC metrics as high or low when their values are > or <= the mean values, respectively.

Table 11. Status frequencies of dynamic quality metrics

Quality metric	Frequency
Anomalous *Execution Time*	11.3%
Normal *Execution Time*	88.7%
Anomalous *(run-time) Cyclomatic Complexity*	11.3%
Normal *(run-time) Cyclomatic Complexity*	88.7%
Anomalous *Attack Surface*	7.1%
Normal *Attack Surface*	92.9%

(2) *Association computation.* With the data obtained above, we computed the association rules (i.e., the if-then association matrix) including relevant support, confidence, and lift values.

Table 11 lists the frequency values of anomalous and normal instances of each dynamic quality metric in our formed classifiers. And Table 12 summarizes the results of our association analysis on all the classifiers. For each pair of the anomalous/normal quality metrics and high/low IPC metrics, the support indicates how frequently the pair occurs in the dataset; the confidence indicates the conditional probability of the appearance of the high or low IPC metric given the anomalous or normal quality metric; and the lift factor indicates the association strength: lift<1 means that the two variables in the pair are mutually exclusive; lift==1 means no association; and lift>1 means that there is an association of the pair with a greater value indicating a stronger association[65].

The results in Table 12 revealed that there were noticeable associations between anomalous/normal quality metrics (*Execution Time*, *(run-time) Cyclomatic Complexity*, and *Attack Surface*) and high/low IPC metrics (RMC, CCC, CCL, and PLC), respectively. These associations were consistent with the correlation coefficients between IPC metrics and dynamic quality metrics discussed earlier (cf. Table 8). For example, for the classifier (RMC, CCC)→run-time cyclomatic complexity, the association rules mined indicated that normal run-time cyclomatic complexity was associated with RMC and CCC being either *both* low or *both* high, while the earlier correlation analysis results revealed that RMC and CCC were *both* positively correlated with this quality metric—that is, the correlation direction/sign is consistent with these two IPC metrics. For another example, according to the association rules for attack surface, an anomalous status of this quality metric was associated with both CCL and PLC being low, which is consistent with the negative correlation between CCL/PLC and attack surface. These association rules help explain why our classifiers were able to predict the normal/anomalous status of respective quality metrics based on the (normal/outlier) value ranges of the IPC metrics that were used as classification features.

Moreover, to further understand how/why the unsupervised classification based on k-means clustering works in particular, we looked into the clustering tendency of our datasets. We were able to validate these tendencies, which justifies that the clustering-based classification worked reasonably well (with an average F1 accuracy of 89%) as shown in Table 9. For instance, Figure 4 shows the clustering tendencies for the classifiers that have more than one feature—the clustering tendency of a one-dimensional variable is difficult to visualize properly. As shown, for both classifiers, the underlying clustering tendency is reasonably clear: the outliers are generally on the outskirts, fairly discernible from the rest of the clusters. Thus, the classification models were able to learn the decision boundary between the anomalous and normal classes with a reasonable level of accuracy.

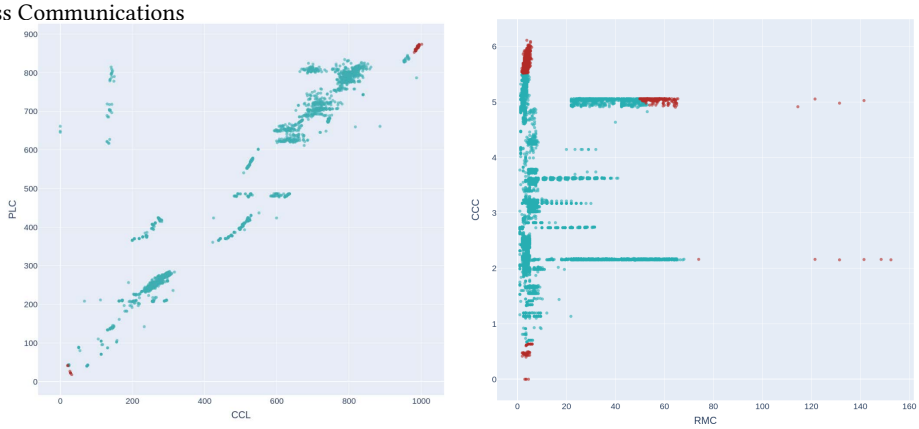

Fig. 4. Clustering tendencies of the datasets underlying the (CCL, PLC)→attack surface (top plot) and (RMC, CCC)→run-time cyclomatic complexity (bottom plot) classifiers: among the single clusters, the dark-red dots represent outlier data points.

Table 12. Overall associations between anomalous/normal quality metrics and high/low IPC metrics

Quality metric	IPC metrics	Support	Confidence	Lift
Anomalous *Execution Time*	Low CCC	11.26%	100.00%	1.95
Normal *Execution Time*	High CCC	48.73%	54.92%	1.13
Anomalous *(run-time) Cyclomatic Complexity*	Low RMC and High CCC	11.24%	99.73%	3.32
Normal *(run-time) Cyclomatic Complexity*	Low RMC and Low CCC	38.77%	43.69%	1.13
	High RMC and High CCC	18.71%	21.10%	1.13
Anomalous *Attack Surface*	High CCC	7.13%	99.92%	2.05
	Low CCL and Low PLC	7.13%	100.00%	2.08
Normal *Attack Surface*	Low CCC	51.2%	55.20%	1.08
	High CCL and High PLC	44.10%	47.45%	1.08

Answer to RQ3: On overall average, by measuring IPC coupling and cohesion, our unsupervised quality classifiers achieved 85.96% precision, 92.68% recall, and 89.18% F1, while the supervised classifiers gained about 97% precision/recall/F1, in recognizing the anomaly level of distributed system quality concerning three (dynamic direct) quality metrics (i.e., execution time, run-time cyclomatic complexity, and attack surface). The association analysis further explained the classification capabilities and revealed which IPC metric value ranges were associated with the anomalous/normal status of corresponding dynamic predictable quality metrics.

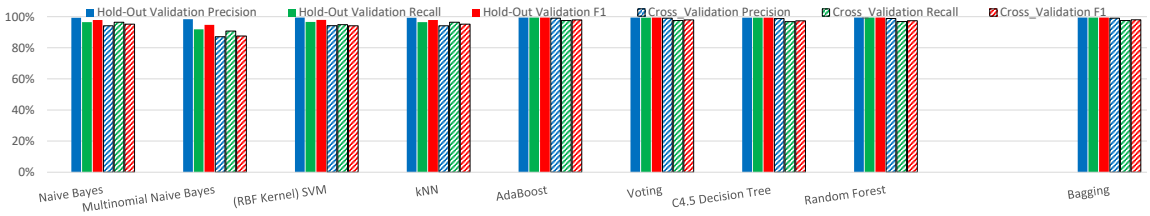

Fig. 5. The average accuracy of alternative learning algorithms compared to the default one (bagging)

6.2.2 RQ4: Influence Factors. As explained earlier, we only studied such effects for supervised classifiers in DistMeasure. Figure 5 compares the default classification (bagging) algorithm to eight alternatives when applied to quality

classification based on our IPC metrics. The length of each bar represents the overall average precision, recall, or F1 accuracy across the classifiers for all the predictable quality metrics. Our results show that the *bagging* algorithm performed almost equally well as or better than any of the alternative supervised learning algorithms considered. *Random Forest* achieved the second-highest F1 accuracy overall, followed by *Voting* and *C4.5 Decision Tree*, while *Multinomial Naive Bayes* performed the worst. Other algorithms had close classification performance in terms of any of the three effectiveness metrics in both the hold-out validation and 10-fold validation. Overall, the differences in precision, recall, and F1 between any two classifiers did not appear to be quite substantial.

Table 13. P-values (effect sizes) of the F1 accuracy for other classifiers versus bagging with 10-fold cross-validations

Model	P-value (effect size)
K-Means	2.09E-02 (-1)
Naive Bayes	7.73E-01 (0.125)
Multinomial Naive Bayes	1.48E-01 (-0.625)
SVC (RBF Kernel)	7.73E-01 (-0.125)
kNN	5.64E-01 (-0.25)
AdaBoost	1 (0)
Voting	1 (0)
C4.5 Decision Tree	3.86E-01(-0.375)
Random Forest	3.86E-01(-0.375)

In addition, we computed Wilcoxon test p-values [127] and Cliff's delta effect sizes [33] on the F1 accuracy of 10-fold cross-validations between other unsupervised/supervised classifiers and our default (bagging) classifier. We used 10-fold cross-validations over hold-out validations for the computations [146]. As listed in Table 13, almost all of the p-values are large (>0.05), except the one for the difference between the bagging (supervised) classifier and the k-means (unsupervised) classifier—that difference is significant as we have already expected from the result comparison between Tables 9 and 10. The statistics here show the difference was not only significant (p=0.02) but quite large too (effect size=1). In particular, statistically there was no difference in classification performance between the default bagging classifier and any of the two compared classifiers: *AdaBoost* and *Voting*—p value of 1 and effect size of 0, and the difference from any other compared supervised classifier was also insignificant. The reason behind large effect sizes but still with large p values (e.g., when compared to *Multinomial Naive Bayes*) is because of the relatively small number of samples involved in the statistical analysis. Overall, these statistical results show that the classification algorithm choice did not matter that much for our supervised quality anomaly detection problem, suggesting that using the proposed IPC metrics can readily and stably help predict the anomaly status of respective (correlated) quality metrics.

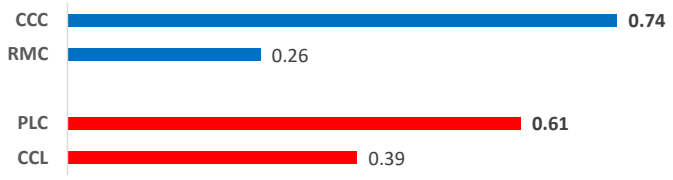

Fig. 6. Ranking of features (IPC metrics) by importance score (shown on the x axis) for supervised (bagging) classifiers for (dynamic) quality metrics. The top chart shows the importance of RMC versus CCC in classifying *run-time cyclomatic complexity* while the bottom chart shows the importance of CCL versus PLC in classifying *attack surface*.

We further studied the effects of feature selection by quantifying the contribution (importance) of each feature in the dynamic quality metric classifiers in DistMeasure. Figure 6 depicts the feature ranking by importance scores for our supervised (bagging) classifiers for two of the (dynamic) predictable quality metrics: *run-time cyclomatic complexity* and

attack surface. These feature importance scores were computed as the standard deviation and mean of the impurity decrease accumulation [28] (i.e., feature_importances_ in Python models) using the Scikit-learn library [104] of Python.

The classifier for *execution time* used one feature (CCC) only, for which feature selection was not relevant—the same applied to the other *attack surface* classifier (also based solely on CCC). Our results showed that CCC, the strongest IPC coupling metric as discussed earlier, was clearly more important than RMC for classifying the quality metric *run-time cyclomatic complexity* (with the importance score of 0.74 versus 0.26). And PLC, the only IPC cohesion metric in our framework, was much more important than CCL for classifying the quality metric *attack surface* (with the importance score of 0.61 versus 0.39). Note that feature importance and ranking are not relevant in *k*-means clustering, thus this study was only conducted on our supervised classifiers.

Answer to RQ4: Bagging was the most effective supervised learning algorithm for our binary quality classifiers with respect to the quality metrics considered for distributed systems, but overall the supervised-classification algorithm choice did not have a significant impact. For the only two dynamic quality metric classifiers (for *run-time cyclomatic complexity* and *attack surface*) that used more than one feature, the (strongest) IPC coupling and the only cohesion metric (i.e., CCC and PLC, respectively) played the most important role.

6.3 Discussion

Per the statistical results for its **Part 2**, DistMeasure has only one supervised/unsupervised classifier for one *static* predictable quality metric: (CCL, PLC)→vulnerableness, as discussed in §6.1.1. For each subject system (as listed in the first column of Table 3), we computed the static quality metric *vulnerableness* (as shown in Tables 6); we also computed the mean (system-level) IPC metrics CCL/PLC over all of each subject's executions as the IPC metric values for that subject (as shown in Table 7). Thus, we had 11 data points in total. Then, we followed the same procedure for RQ3 to evaluate the classifier, except for the inability to perform the 10-fold cross-validation because we did not have enough samples for that validation scheme. With the hold-out validation, we obtained almost perfect precision, recall, and F1 accuracy of the classifier. While our dataset was not sufficient (due to the small size) for drawing solid, generalizable conclusions, these results along with the correlations found for RQ2 suggest potentially promising predictive capabilities of our IPC metrics in assessing distributed software quality with respect to the *static* predictable quality metric (i.e., *vulnerableness*) as well. Nevertheless, we would need much more data points to train a robust classifier for the quality metric *vulnerableness* and more samples for testing the classifier.

While prior works [45, 148] found that unsupervised classifiers may perform better than supervised ones for (albeit static) defect prediction, our results showed clear advantages of supervised classification, at least for dynamic quality metrics of distributed systems. This contrast implied that assessing those quality metrics purely based on the similarity of (significantly correlated) IPC metrics across different system executions may not be effective; instead, a better way would be to base the classification on the patterns of association between IPC metrics and the quality metrics. The key reason is that the various system executions may have largely different measurements of the quality metrics even if their IPC measurements are close enough. On the other hand, the merits of supervised over unsupervised classification in DistMeasure came at a cost—computing the quality metrics in order to label the training samples can be costly in practice, especially when a large number of such samples are used as we did in the evaluation of DistMeasure.

Among the six IPC metrics in DistMeasure, IPR did not appear to statistically significantly inform any of the direct quality metrics considered. Yet the IPR measurement results can still be useful for understanding run-time code reuse

in distributed systems. Any of the other five IPC metrics has shown to be indicative of distributed software quality with respect to at least one of the quality metrics, static and/or dynamic. In particular, the strongest indicator of interprocess coupling, CCC, which captures the coupling *among all* of the distributed processes (as opposed to measuring that *between two* processes as RCC does) at a fine-grained granularity level (as opposed to measuring that at a coarse level as RMC does) tended to be the most indicative of distributed software quality. Indeed, CCC was found to be significantly correlated with more (four) quality metrics than any other IPC metrics in our experiments. As a learning feature, CCC also contributed to the quality classifications more than other features wherever more than one feature was used in the classifiers. Meanwhile, PLC, the only metric in our framework that captures process cohesion, exhibited similar merits to CCC—it was found to be significantly correlated with three quality metrics, more than other IPC metrics except for CCC, while having contributed to relevant classifiers much more than other features. These results demonstrated the great usefulness of measuring both coupling and cohesion at process level for assessing distributed software quality.

The above observations hinted at the relationship between correlation coefficients and feature importance scores. Indeed, our Spearman's rank correlation coefficients and the feature importance scores for our supervised (bagging) classifiers were consistent to a certain extent, as seen from Table 8 versus Figure 6. For the quality metric *run-time cyclomatic complexity*, the IPC metric CCC was more strongly correlated to it than RMC (correlation coefficient of 0.448 versus 0.42); in the classifier for this quality metric, CCC was seen to be more important than RMC as well (importance scores of 0.56 versus 0.44). The same consistency was observed between PLC and CCL for quality classification with respect to *attack surface*—the former was more strongly correlated with this quality metric, and also more important for the classification, than the latter.

However, the contrasts of effectiveness across our classifiers were not always consistent with those of the correlation coefficients between corresponding IPC metrics and quality metrics, according to Table 8 versus Table 9 and Table 10. For instance, with the quality metric *attack surface*, CCL and PLC had the strongest correlation among all IPC metrics, yet the classifier (CCL,PLC)→attack surface was not the most accurate among all the dynamic metric classifiers, in either the hold-out validation or 10-fold CV when the classification is supervised. For another example, the other three dynamic metric classifiers had noticeably different F1 accuracy (ranging from 96.23% to 99.86% for supervised classification), especially for unsupervised classifications (ranging from 85.28% to 93.27%), although the underlying IPC metrics and quality metrics had very close absolute correlation coefficients (ranging from 0.42 to 0.48); and the classifier with more strongly correlated underlying IPC and quality metrics was *not* always more accurate.

This inconsistency can be explained by the fact that the underlying learning algorithms of these classifiers do not work based on statistical correlations. Moreover, the correlation analyses and learning-based classifications focus on different aspects of the relationships between the two kinds of variables (i.e., the IPC metrics and the quality metrics)—the former quantifies the strength of linear association between these two kinds of variables [122] as regards how the second kind of variable changes with the first, while the latter reflects the influences of the first kind of variable on how the two ranges of values (for `normal` versus `anomalous`) of the second can be differentiated. The statistical correlations provided a general direction regarding how IPC measurements may help assess distributed software quality with respect to which direct quality metrics. These correlations also essentially enabled the design of the quality classifiers in DistMeasure. Yet the correlation coefficients themselves are not sufficient for understanding the quality connections/implications of the IPC measurements, which is why we built the quality classifiers in our framework.

7 THREATS TO VALIDITY

The main threat to internal validity lies in possible implementation errors in our computation for IPC metrics, direct measures of quality metrics, correlation analysis, and quality classification. To reduce this threat, we performed careful code reviews for our tools and used the two smallest subjects to manually validate their functionalities and analysis/prediction results. In particular, we confirmed the correctness of the dynamic dependencies underlying most of our IPC measurements using these subjects. While we did not find such issues in our subject executions, missing dependencies induced by dynamic code constructs (e.g., dynamically loaded code) may generally exist in distributed software, which would lead to false dependencies hence incorrect IPC measurements.

In addition, our results might have been affected by various factors that we have not considered but have influenced the process for collecting the data needed for directly quantifying the studied quality metrics. For instance, in order to quantify code churn size as a direct quality metric, we needed to collect the data on system releases from the respective version histories. While we considered various sources for collecting such data, our collection might have been incomplete because some system releases were not available online. To mitigate this threat, we carefully searched and reviewed all possible release data of our subjects, and reached out to the developers with data requests when we noticed likely missing releases from the sources we initially found.

Likewise, the validity of our results is subject to the particular direct metrics we used for quantifying related quality characteristics/sub-characteristics—the results might not be valid with respect to possible other relevant direct measures. For instance, we currently used (run-time) cyclomatic complexity to quantify testability, without considering other metrics that may affect this quality sub-characteristic. Yet systems with similar (run-time) cyclomatic complexity may not always be similarly testable. For example, suppose one program deals with complex data structures while frequently reading/writing external files while another program only has simple data operations. Intuitively, the testability of these two programs should be quite different. However, their control flow structures might be very similar during their executions, leading to close (run-time) cyclomatic complexity between them. We currently do not consider other quality metrics (e.g., throughput, response time, availability, scalability) that might be related to IPC metrics as well. The reason is that we would like to be more focused—on quality metrics that we can justifiably connect to IPC-related system behaviors. Yet we cannot claim that no other quality metric can be informed by IPC measurements.

The main threat to external validity is that our study results may not generalize to other distributed systems and executions. To reduce this threat, we have chosen subjects of various scales and application domains, focusing on real-world/industry-scale systems of varied architectures. For instance, among the subjects chosen, ZooKeeper is widely used by a number of Apache projects (e.g., Hadoop, Spark, and HBase) and large companies (e.g., Facebook, Twitter, and Yahoo!) while Voldemort is a system underlying LinkedIn's products. We further note that our major findings (i.e., the correlations between IPC metrics and quality metrics) for **Part 2** of our framework were consistent with those from the preliminary version of our study [47]—for the common IPC metrics and quality metrics considered in both studies. This consistency corroborates both the validity of our earlier results and the credibility of the findings presented in this paper. Nevertheless, our results are best interpreted with respect to the systems and executions actually studied.

While limited run-time input coverage is a common limitation of dynamic analysis, it is irrelevant for **Part 1** of our framework because it is supposed to focus on measuring the IPC characteristics (which is *dynamic* by nature) just for the particular executions of interest. However, a relevant validity threat does arise for **Part 3** where we used these dynamic metrics to classify the quality of the distributed software with respect to *static* quality metrics. With different run-time inputs, the correlations between our IPC metrics and static quality metrics, and the performance of

our classifiers built on the IPC metrics for quality classification with respect to those static quality metrics, may deviate from what we reported. Regarding the run-time inputs, another external validity threat is that, while we desired to consider different types of inputs (e.g., integration, system, and load tests) for each subject system and tried hard to collect such inputs, we ended up being able to do so only for ZooKeeper. Fortunately, the integration tests, which were available and used for each subject, are indeed intended to exercise the *whole-system* behaviors. Further, to reduce the threat, we have manually constructed a large number of run-time inputs for our subjects while ensuring each corresponding execution was unique, which did largely increase the total run-time coverage for each subject.

The main threat to construct validity lies in our use of statistical analysis for drawing our conclusions. In computing the system-level IPC metrics, we took the means of lower-level metrics without accounting for the variations (e.g., standard deviations), which would need to be considered in more thorough measurement schemes. For a group of data with large variations, there may not be a perfect summary statistic using a single number, for which mean is still a widely used aggregation function—other choices like min and max may not be better for us. Also, in our experiments for this paper, we calculated such variations and found that at process level they are generally negligible in our subject executions. This is not surprising because, per our configurations of these executions as described earlier (§5.1.1), most involved only 2 to 3 machines/processes. The variations at method and class levels are larger, but they are still not large enough to change comparisons in respective metrics between any pair of system executions hence changing conclusions regarding our research questions, especially for class-level metrics. Note that we only have one metric (i.e., IPR) that can be computed at method level, and it is not involved in our major findings in any research question, nor used in our quality anomaly detection classifiers. Yet, in a larger-scale system deployment, these variations could be much larger. To reduce the threat as regards our correlation analysis, we purposely chose Spearman's method over alternatives as it is a non-parametric method that does not assume normality of underlying data distribution or relationships between the data groups. On the other hand, although we endeavored to largely augment the run-time inputs for our subjects, this augmentation did not help get more samples for training and testing the quality classifiers with respect to static quality metrics. As a result, our evaluation for these classifiers was quite premature. We would need thousands of different Java distributed systems as subjects for a much stronger evaluation in this regard, which is not readily feasible at this point. Another construct validity threat lies in our process of labeling the ground-truth quality status between `normal` and `anomalous` in evaluating our quality anomaly detection classifiers. Since these ground-truth labels were determined based on the outlier values of target quality metrics among the given set of samples, the labels might change when that set changes (e.g., when we consider more systems or augment their test inputs further). Thus, the ground-truth quality status is defined relatively to the given dataset; hence, the evaluation results of the classifiers should be interpreted with respect to the set of subject systems and executions we considered in the evaluation.

The lack of enough distributed system subjects also led to a threat to conclusion validity regarding the static quality metric classifiers. In building these classifiers, we measured the IPC metrics during each subject execution and aggregated the measurement results over all executions of the subject into one data point. This aggregation ignored the varying characteristics of dynamic behavioral profiles across different executions of each subject, and essentially treated all of the executions of each subject as being collectively representative of that subject's holistic behaviors. Such a treatment may cause biases even in our preliminary conclusions about the performance of these classifiers. Finally, while we have shown the usefulness of IPC measurements of distributed systems using our IPC metrics for assessing their various quality characteristics/sub-characteristics, we must also acknowledge that our quality classification is only a high-level assessment of the quality being anomalous (hence worth more inspection/attention) or not. Currently, DistMeasure is not sufficient for localizing/diagnosing such anomalies when signaled.

8 RELATED WORK

We discuss previous works that are most related to ours in three categories: dynamic coupling metrics (§8.1), run-time cohesion (§8.2), and predictive software quality assessment based on machine learning (§8.3).

8.1 Dynamic Coupling Metrics

Dynamic coupling metrics have been well studied for single-process systems [31, 74, 135]. They also have been indirectly utilized to assist with refactoring monolithic systems to microservices [80, 90]. Arisholm et al. [9] defined a set of dynamic coupling metrics for object-oriented software and studied the relationship between dynamic coupling measures and software change-proneness. Their dynamic class export/import coupling (*IC_CD* and *EC_CD*) metrics initially motivated the definition of our IPC metric *RMC (Runtime Message Coupling)*. On the other hand, compared to their coupling measurement between *classes*, RMC measures the coupling between *processes*.

Dynamic coupling metrics also have been used to estimate architectural risks [145] and complexity [61] in relation to quality metrics such as maintainability [60, 106]. Most of these metrics were defined under the assumption that there exists an explicit reference/invocation between the entities (e.g., object, method, and class) involved in the coupling measure, thus they are not suitable for measuring interprocess communication in distributed systems. Meanwhile, our IPC metrics can also be used as complexity metrics and indicators of various quality metrics.

Jin et al. [71] defined a dynamic component coupling metric (*CPC*) directly based on inter-component dependencies derived from method executions with timing information. Conceptually, the *CPC* metric is closely related to our *IPR* metric, in that both are based on approximated dynamic dependencies across components. However, the interprocess dependencies on which our *IPR* computation is based are significantly more precise than the purely control-flow-based dependencies approximated in [71], according to our previous study [27]. In addition, *CPC* was defined for measuring structural complexity, while *IPR* is proposed primarily as a reusability metric. Previous reuse metrics mainly concern reusing library code and connectivity between server and client nodes as a whole [42]. Instead, we measure interprocess reusability at code level in terms of metrics defined based on code dependencies.

8.2 Run-Time Cohesion

The cohesion of a software component refers to the extent to which the elements of the component are related [14]. A highly cohesive component performs a set of closely relevant actions and is difficult to be split into separate components [145]. Static cohesion has been widely explored in software measurements. More recent relevant works increasingly focused on run-time (i.e., dynamic) cohesion [36, 63, 96, 153].

In particular, Jin et al. [70] proposed a dynamic cohesion measurement approach for distributed software, which includes two component-level cohesion metrics (i.e., *CC* and *CCW*) by extending the metric *lack of cohesion in methods (LCOM)*, a classical cohesion metric for single-process programs. A structural quality attribute *cohesion factor of component (CHC)* was later introduced also for distributed software [71]. These cohesion metrics were evaluated against *specialized* distributed programs (e.g., Netflix RSS Reader, RSS Reader Recipes, and/or the distributed version of iBATIS JPetStore) [70, 71]. The underlying measurement tool used was also designed for these specialized systems. In comparison, our cohesion metric (PLC) is defined based on method-level dynamic dependencies and evaluated against real-world *common* distributed systems.

8.3 Software Quality Assessment based on Machine Learning

Software quality prediction helps developers with better utilization of resources, effort estimates, and making testing plans for components that may have defects, hence reducing development costs and mitigating risks at initial stages [113, 124]. Various machine learning techniques, using unsupervised or supervised learning algorithms (e.g., logistic regression, support vector machine, neural networks, and k-means clustering) have been used for software quality assessment.

For instance, Khoshgoftaar and Allen proposed a software quality assessment model using logistic regression [77]. Xing et al. employed a support vector machine technique for the classification of software modules based on a complexity metric to predict software quality in early development stages [114]. Abe et al. used a Bayesian classifier to estimate the success or failure of a software project [2]. In addition, neural network techniques were also applied to the prediction of software quality [78, 124, 137, 141]. Furthermore, SMPLearner [152] trains a software maintainability prediction model by gathering the real maintenance efforts computed from code change histories; it experimented with 24 common machine learning techniques, including support vector machine regression, random forest, K-star, and k-nearest neighbors. In comparison, while we also explored a number of alternative learning algorithms, our quality classifiers are built on dynamic coupling and cohesion metrics and target distributed system executions while focusing on IPC characteristics, a defining aspect of distributed system behaviors.

As a type of software quality assessment model, numerous software defect prediction models have been built from single or multiple software projects for within- or cross-project prediction [150, 151, 154], respectively. For example, Zhang et al. built a universal defect prediction model with a large number of software projects from various contexts by clustering projects based on the similarity of distribution across multiple predictors, deriving the rank transformations, and fitting the model on the data transformed beforehand [150]. An unsupervised learning model does not require any training data and thus avoids any homogeneity requirement (e.g., a similar distribution of metrics) among projects [151]. Moreover, some simple unsupervised models outperformed supervised models for a special type of software defect prediction (i.e., effort-aware just-in-time defect prediction) for open-source software systems [45, 148]. In particular, unlike distance-based unsupervised learning (e.g., k-means clustering) models, connectivity-based unsupervised defect classifiers are based on the assumption of a similar intuition that defective entities likely cluster around the same neighborhood (i.e., area) [151].

In contrast to these prior defect prediction models that are commonly *static*—they are based on project features rather than system execution traits, the quality classifiers in DistMeasure are dynamic as they are based on IPC measurements in specific executions. We also differ from prior defect prediction works in that we leverage IPC characteristics as a particular aspect of system behaviors, which have not been exploited before. DistMeasure also addresses the quality of distributed software systems, which were not particularly addressed in earlier works. Finally, unlike prior works on defect prediction that mainly aim at predicting whether a software unit (e.g., a file) contains functional defects, our quality classifiers address a variety of (both functional and non-functional) quality characteristics. Meanwhile, we recall that the main goal of DistMeasure is to enable and explore measuring IPC-induced behaviors and understanding the measurement results from a perspective of their quality correlations, rather than developing a defect prediction model, for distributed systems.

9 CONCLUSION AND FUTURE WORK

This paper contributed to dynamic software measurement with a novel set of six metrics for distributed systems that characterize their IPC, a vital aspect of distributed systems' run-time behaviors, at various levels (whole-system,

process/component, class, and method). In addition, with these metrics, we measured the IPC in eleven real-world distributed systems with respect to varied execution scenarios and demonstrated that each of the metrics was uniquely expressive of IPC traits in those systems and their executions. We extensively analyzed the statistical correlations among these IPC metrics and various direct metrics of different quality characteristics/sub-characteristics in reference to a standard software quality model. Then, based on the discovered correlations, we built learning-based classifiers classifying the quality metrics of distributed systems for further understanding the characteristics and usefulness of IPC measurements from the perspective of their predictive relationships with different aspects of the quality of the systems. Our experiments revealed that IPC measures can be significantly indicative of the various quality aspects of distributed systems hence potentially help developers assess, and improve the quality of these systems.

One future step is to examine the evolution history of distributed systems by considering multiple versions of each system. Then, we will characterize how IPC-induced behaviors change over time, also in terms of the proposed IPC metrics and using our measurement framework, with respect to how the system quality varies in terms of the correlated direct quality metrics. For instance, we may observe how the PLC and CCL measures change in relation to the changes in the subject system's attack surface (e.g., whether these metrics' values decrease from one version of the system to another version when more vulnerabilities are reported between the two versions). It would also be rewarding to consider different performance/quality models (e.g., [125]) when studying our research questions. Also valuable to look into in the future are other types of distributed systems such as high-performance computing frameworks like Spark and Hadoop.

ACKNOWLEDGMENT

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the U.S. Office of Naval Research (ONR) under Grant N000142212111.

REFERENCES

[1] Hervé Abdi. 2007. The Kendall Rank Correlation Coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA* 1 (2007), 508–510.

[2] Seiya Abe, Osamu Mizuno, Tohru Kikuno, Nahomi Kikuchi, and Masayuki Hirayama. 2006. Estimation of Project Success Using Bayesian Classifier. In *Proceedings of the 28th international conference on Software engineering*. 600–603.

[3] Mamdouh Alenezi and Mohammad Zarour. 2020. On the relationship between software complexity and security. *International Journal of Software Engineering & Applications (IJSEA)* 11 (2020), 1–10.

[4] Luca Allodi and Fabio Massacci. 2012. A Preliminary Analysis of Vulnerability Scores for Attacks in Wild: The EKITS and SYM Datasets. In *Proceedings of the ACM Workshop on Building analysis datasets and gathering experience returns for security*. 17–24.

[5] Apache. 2015. ZooKeeper. https://zookeeper.apache.org/.

[6] Apache. 2018. Thrift bugs. http://archive.apache.org/dist/thrift/.

[7] APACHE - Open issues. 2019. Derby bugs. https://issues.apache.org/jira/projects/DERBY/issues.

[8] Google Code Archive. 2022. quickserver. https://code.google.com/archive/p/quickserver/.

[9] Erik Arisholm, Lionel C Briand, and Audun Foyen. 2004. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering* 30, 8 (2004), 491–506.

[10] S Aswini and M Yazhini. 2017. An Assessment Framework of Routing Complexities using LOC Metrics. In *Proceedings of the Innovations in Power and Advanced Computing Technologies (i-PACT)*. IEEE, 1–6.

[11] Nikolaus Baer and Robert Zeidman. 2009. Measuring Software Evolution with Changing Lines of Code. In *Proceedings of CATA*. 264–170.

[12] Najwa Abu Bakar and Ali Selamat. 2013. Runtime verification of multi-agent systems interaction quality. In *Intelligent Information and Database Systems: 5th Asian Conference, ACIIDS 2013, Kuala Lumpur, Malaysia, March 18-20, 2013, Proceedings, Part I 5*. Springer, 435–444.

[13] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson Correlation Coefficient. In *Proceedings of Noise reduction in speech processing*. Springer, 1–4.

[14] James M Bieman and Linda M Ott. 1994. Measuring Functional Cohesion. *IEEE transactions on Software Engineering* 20, 8 (1994), 644–657.

[15] Leo Breiman. 1996. Bagging Predictors. *Machine learning* 24, 2 (1996), 123–140.

[16] Leo Breiman. 2001. Random Forests. *Machine learning* 45, 1 (2001), 5–32.
[17] Lionel C Briand, Sandro Morasca, and Victor R Basili. 1996. Property-Based Software Engineering Measurement. *IEEE transactions on software Engineering* 22, 1 (1996), 68–86.
[18] Haipeng Cai. 2015. *Cost-effective dependency analysis for reliable software evolution.* Ph. D. Dissertation. University of Notre Dame.
[19] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *IEEE Transactions on Software Engineering* 44, 4 (2018), 334–364.
[20] Haipeng Cai. 2020. Assessing and Improving Malware Detection Sustainability through App Evolution Studies. *ACM Transactions on Software Engineering and Methodology* 29, 2 (2020), 1–28.
[21] Haipeng Cai. 2020. A Reflection on the Predictive Accuracy of Dynamic Impact Analysis. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*. 562–566.
[22] Haipeng Cai and Xiaoqin Fu. 2021. D^2Abs: A Framework for Dynamic Dependence Analysis of Distributed Programs. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4733–4761.
[23] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2018. DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling. *IEEE Transactions on Information Forensics and Security* 14, 6 (2018), 1455–1470.
[24] Haipeng Cai and Raul Santelices. 2014. Diver: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proceedings of International Conference on Automated Software Engineering*. 343–348.
[25] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying dynamic impact analyses for improved and variable cost-effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 1–50.
[26] Haipeng Cai and Douglas Thain. 2016. DistEA: Efficient Dynamic Impact Analysis for Distributed Systems. *arXiv preprint arXiv:1604.04638* 1604 (2016), 1–10.
[27] Haipeng Cai and Douglas Thain. 2016. DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. 344–355.
[28] Andrew Phelps Cassidy and Frank A Deviney. 2014. Calculating Feature Importance in Data Streams with Concept Drift using Online Random Forest. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 23–28.
[29] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM transactions on intelligent systems and technology* 2, 3 (2011), 1–27.
[30] Sanjay Chawla and Aristides Gionis. 2013. k-means−: A unified approach to clustering and outlier detection. In *Proceedings of the 2013 SIAM international conference on data mining*. SIAM, 189–197.
[31] Jitender Kumar Chhabra and Varun Gupta. 2010. A Survey of Dynamic Software Metrics. *Journal of Computer Science and Technology* 25, 5 (2010), 1016–1029.
[32] Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.
[33] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin* 114, 3 (1993), 494.
[34] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company.
[35] Derby Repository. 2019. Apache Derby. https://db.apache.org/derby/derby_downloads.html.
[36] Amr F Desouky and Letha H Etzkorn. 2014. Object Oriented Cohesion Metrics: A Qualitative Empirical Analysis of Runtime Behavior. In *Proceedings of ACM Southeast Regional Conference*. 1–6.
[37] Chandan Dhal, Xiaoqin Fu, and Haipeng Cai. 2023. A Control-Theoretic Approach to Auto-Tuning Dynamic Analysis for Distributed Services. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings*. 330–331.
[38] Johann Eder, Gerti Kappel, and Michael Schrefl. 1994. *Coupling and cohesion in object-oriented systems.* Technical Report. Citeseer.
[39] Michael Eichberg, Daniel Germanus, Mira Mezini, Lukas Mrokon, and Thorsten Schafer. 2006. QScope: an Open, Extensible Framework for Measuring Software Project. In *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 1–10.
[40] Norman Fenton. 1994. Software measurement: A necessary scientific basis. *IEEE Transactions on software engineering* 20, 3 (1994), 199–206.
[41] Norman Fenton and James Bieman. 2014. *Software Metrics: A Rigorous and Practical Approach.* CRC Press.
[42] William Frakes and Carol Terry. 1996. Software Reuse: Metrics and Models. *Comput. Surveys* 28, 2 (1996), 415–435.
[43] Enrico Fregnan, Tobias Baum, Fabio Palomba, and Alberto Bacchelli. 2019. A Survey on Software Coupling Relations and Tools. *Information and Software Technology* 107 (2019), 159–178.
[44] Yoav Freund, Robert E Schapire, et al. 1996. Experiments with a New Boosting Algorithm. In *icml*, Vol. 96. Citeseer, 148–156.
[45] Wei Fu and Tim Menzies. 2017. Revisiting Unsupervised Learning for Defect Prediction. In *Proceedings of joint meeting on foundations of software engineering*. 72–83.
[46] Xiaoqin Fu and Haipeng Cai. 2019. A Dynamic Taint Analyzer for Distributed Systems. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1115–1119.
[47] Xiaoqin Fu and Haipeng Cai. 2019. Measuring Interprocess Communications in Distributed Systems. In *IEEE/ACM International Conference on Program Comprehension*. 323–334.
[48] Xiaoqin Fu and Haipeng Cai. 2020. DistMeasure artifact package. https://figshare.com/s/aa827075e3e1964dad98.

[49] Xiaoqin Fu and Haipeng Cai. 2020. Scaling application-level dynamic taint analysis to enterprise-scale distributed systems. In *Proceedings of ACM/IEEE International Conference on Software Engineering: Companion Proceedings*. 270–271.
[50] Xiaoqin Fu and Haipeng Cai. 2021. FlowDist: Multi-Staged Refinement-Based Dynamic Information Flow Analysis for Distributed Software Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 2093–2110.
[51] Xiaoqin Fu, Haipeng Cai, and Li Li. 2020. Dads: Dynamic Slicing Continuously-Running Distributed Programs with Budget Constraints. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1566–1570.
[52] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. 2020. Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–45.
[53] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: a toolkit for measuring interprocess communications and quality of distributed systems. In *Proceedings of the ACM/IEEE International Conference on Software Engineering: Companion Proceedings*. 51–55.
[54] Daniel Galin. 2004. *Software Quality Assurance: From Theory to Implementation*. Pearson Education India.
[55] Jean-François Gélinas, Mourad Badri, and Linda Badri. 2006. A Cohesion Measure for Aspects. *J. Object Technol.* 5, 7 (2006), 75–95.
[56] Cigdem Gencel, Rogardt Heldal, and Kenneth Lind. 2009. On the relationship between different size measures in the software life cycle. In *2009 16th Asia-Pacific Software Engineering Conference*. IEEE, 19–26.
[57] Moheb R Girgis, Tarek M Mahmoud, and Rehab R Nour. 2009. UML Class Diagram Metrics Tool. In *2009 International Conference on Computer Engineering & Systems*. IEEE, 423–428.
[58] Github. 2020. eclipse-ee4j / grizzly. https://github.com/eclipse-ee4j/grizzly/issues.
[59] Daniel Gomez Blanco. 2023. OpenTelemetry Fundamentals. In *Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization*. Springer, 27–49.
[60] Anjana Gosain and Ganga Sharma. 2016. Predicting Software Maintainability Using Object Oriented Dynamic Complexity Measures. In *International Conference on Smart Trends for Information Technology and Computer Communications*. Springer, 218–230.
[61] Anjana Gosain and Ganga Sharma. 2017. Object-oriented dynamic complexity measures for software understandability. *Innovations in Systems and Software Engineering* 13, 2-3 (2017), 177–190.
[62] Grizzly Repository. 2020. grizzly-framework. https://repo1.maven.org/maven2/org/glassfish/grizzly/grizzly-framework/.
[63] Varun Gupta and Jitender Kumar Chhabra. 2011. Dynamic cohesion measures for object-oriented software. *Journal of Systems Architecture* 57, 4 (2011), 452–462.
[64] Dominik Hellhake, Justus Bogner, Tobias Schmid, and Stefan Wagner. 2022. Towards using coupling measures to guide black-box integration testing in component-based systems. *Software Testing, Verification and Reliability* 32, 4 (2022), e1811.
[65] Fauna Herawati, Muhamad Satria Mandala Pua Upa, Rika Yulia, and Retnosari Andrajati. 2019. Antibiotic Consumption At A Pediatric Ward At A Public Hospital In Indonesia. *Asian Journal of Pharmaceutical and Clinical Research* 12, 8 (2019), 64–67.
[66] Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. 2018. Evaluating Feature Importance Estimates. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*. 1–17.
[67] Sukainah Husein and Alan Oxley. 2009. A Coupling and Cohesion Metrics Suite for Object-Oriented Software. In *International Conference on Computer Technology and Development*, Vol. 1. 421–425.
[68] ISO/IEC. 2011. ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability.
[69] John Jenkins and Haipeng Cai. 2017. Dissecting Android Inter-Component Communications via Interactive Visual Explorations. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 519–523.
[70] Wuxia Jin, Ting Liu, Yu Qu, Jianlei Chi, Di Cui, and Qinghua Zheng. 2016. Dynamic Cohesion Measurement for Distributed System. In *Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs*. ACM, 20–26.
[71] Wuxia Jin, Ting Liu, Yu Qu, Qinghua Zheng, Di Cui, and Jianlei Chi. 2018. Dynamic structure measurement for distributed software. *Software Quality Journal* 26, 3 (2018), 119–1145.
[72] John Jenkins and Haipeng Cai. 2018. ICC-Inspect: Supporting Runtime Inspection of Android Inter-Component Communications. In *Proceedings of International Conference on Mobile Software Engineering and Systems*. ACM, 80–83.
[73] Petra J Jones, Matthew K James, Melanie J Davies, Kamlesh Khunti, Mike Catt, Tom Yates, Alex V Rowlands, and Evgeny M Mirkes. 2020. FilterK: A new outlier detection method for k-means clustering of physical activity. *Journal of biomedical informatics* 104 (2020), 103397.
[74] Stephen H Kan. 2002. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc.
[75] Karaf Repository. 2018. Apache Karaf. https://mvnrepository.com/artifact/org.apache.karaf/karaf.
[76] Anureet Kaur. 2016. Comparative Analysis of Line of Code Metric Tools. *Int J Sci Res Sci Eng Technol (ijsrset. com)* 2 (2016), 1–4.
[77] Taghi M Khoshgoftaar and Edward B Allen. 1999. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering* 6, 04 (1999), 303–317.
[78] Taghi M Khoshgoftaar and Robert M Szabo. 1996. Using Neural Networks to Predict Software Faults During Testing. *IEEE Transactions on Reliability* 45, 3 (1996), 456–462.
[79] Hyung Chan Kim, Angelos D Keromytis, Michael Covington, and Ravi Sahita. 2009. Capturing Information Flow with Concatenated Dynamic Taint Analysis. In *Proceedings of International Conference on Availability, Reliability and Security*. IEEE, 355–362.

[80] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. 2020. Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 9–16.
[81] Ludmila I Kuncheva. 2014. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley & Sons.
[82] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.
[83] Md Tahmid Rahman Laskar, Jimmy Xiangji Huang, Vladan Smetana, Chris Stewart, Kees Pouw, Aijun An, Stephen Chan, and Lei Liu. 2021. Extending isolation forest for anomaly detection in big data via K-means. *ACM Transactions on Cyber-Physical Systems (TCPS)* 5, 4 (2021), 1–26.
[84] Miao-Xin Li, Juilian MY Yeung, Stacey S Cherny, and Pak C Sham. 2012. Evaluating the effective numbers of independent tests and significant p-value thresholds in commercial genotyping arrays and public imputation reference datasets. *Human genetics* 131, 5 (2012), 747–756.
[85] Wei Li and Sallie Henry. 1993. Object-oriented metrics that predict maintainability. *Journal of systems and software* 23, 2 (1993), 111–122.
[86] Lidia López, Xavier Burgués, Silverio Martínez-Fernández, Anna Maria Vollmer, Woubshet Behutiye, Pertti Karhapää, Xavier Franch, Pilar Rodríguez, and Markku Oivo. 2022. Quality measurement in agile and rapid software development: A systematic mapping. *Journal of Systems and Software* 186 (2022), 111187.
[87] Pratyusa K Manadhata and Jeannette M Wing. 2010. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 3 (2010), 371–386.
[88] Peter V Marsden. 2005. Recent Developments in Network Measurement. *Models and methods in social network analysis* 8 (2005), 30.
[89] Michela Carlotta Massi, Francesca Ieva, and Emanuele Lettieri. 2020. Data mining application to healthcare fraud detection: a two-step unsupervised clustering method for outlier detection with administrative databases. *BMC medical informatics and decision making* 20 (2020), 1–11.
[90] Tiago Matias, Filipe F Correia, Jonas Fritzsch, Justus Bogner, Hugo S Ferreira, and André Restivo. 2020. Determining microservice boundaries: a case study using static and dynamic software analysis. In *Software Architecture: 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14–18, 2020, Proceedings 14*. Springer, 315–332.
[91] Maven Repository. 2008. XSocket bugs. https://mvnrepository.com/artifact/org.xsocket/xSocket.
[92] Maven Repository. 2018. Apache Karaf bugs. https://mvnrepository.com/artifact/org.apache.karaf/karaf.
[93] Maven Repository. 2019. Netty bugs. https://mvnrepository.com/artifact/io.netty/netty-all.
[94] Thomas J McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2, 4 (1976), 308–320.
[95] Andrew McCallum, Kamal Nigam, et al. 1998. A Comparison of Event Models for Naive Bayes Text Classification. In *AAAI-98 workshop on learning for text categorization*, Vol. 752. Citeseer, 41–48.
[96] Aine Mitchell and James F Power. 2004. Run-Time Cohesion Metrics: An Empirical Investigation. In *International Conference on Software Engineering Research and Practice*. 1–6.
[97] MlXTEND. 2020. Mlxtend: (machine learning extensions), a Python library of useful tools for the day-to-day data science tasks. http://rasbt.github.io/mlxtend.
[98] Netty Repository. 2019. Netty All In One. https://mvnrepository.com/artifact/io.netty/netty-all.
[99] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC Counting Standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.
[100] NIST. 2020. National Vulnerability Database. https://nvd.nist.gov/.
[101] NIST. 2023. Common Vulnerabilities and Exposures (CVE). https://www.cve.org/.
[102] NIST. 2023. Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss/.
[103] Openchord Repository. 2008. Open Chord. https://sourceforge.net/projects/open-chord/files/Open%20Chord%201.0/.
[104] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[105] Raffaele Perego, Salvatore Orlando, and P Palmerini. 2001. Enhancing the Apriori algorithm for Frequent Set Counting. In *international conference on data warehousing and knowledge discovery*. Springer, 71–82.
[106] Mikhail Perepletchikov and Caspar Ryan. 2011. A Controlled Experiment for Evaluating the Impact of Coupling on the Maintainability of Service-Oriented Software. *IEEE Transactions on Software Engineering* 37, 4 (2011), 449–465.
[107] Duc Truong Pham, Stefan S Dimov, and Chi D Nguyen. 2005. Selection of K in K-means clustering. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 219, 1 (2005), 103–119.
[108] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. 2009. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering* 14, 1 (2009), 5–32.
[109] Pull request - apache/zookeeper. 2019. Zookeeper bugs. https://github.com/apache/zookeeper/pulls.
[110] Pull request - voldemort. 2017. Voldemort bugs. http://https://github.com/voldemort/voldemort//pulls.
[111] QuickServer Repository. 2022. QuickServer. http://www.quickserver.org/download.html.
[112] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.
[113] ZA Rana, S Shamail, and MM Awais. 2007. *A survey of measurement-based software quality prediction techniques*. Technical Report. Tech. Rep. Lahore University of Management Sciences.
[114] Ekbal A Rashid, Srikanta B Patnaik, and Vandana C Bhattacherjee. 2014. Machine Learning and Software Quality Prediction: As an Expert System. *International Journal of Information Engineering and Electronic Business* 6, 2 (2014), 9.
[115] Redhat Issues. 2024. XNIO bugs. https://issues.redhat.com/projects/XNIO/issues/XNIO-330?filter=allopenissues.
[116] Tamar Richner and Stéphane Ducasse. 1999. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of IEEE International Conference on Software Maintenance*. IEEE, 13–22.

[117] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.

[118] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[119] Shallu Sarvari, Paramvir Singh, and Geeta Sikka. 2015. Efficient and Scalable Collection of Dynamic Metrics Using MapReduce. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 127–134.

[120] Nick Sayer. 2014. Google Code Archive-Long-term storage for Google Code Project Hosting.

[121] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation Coefficients: Appropriate Use and Interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.

[122] Philip Sedgwick. 2014. Spearman's rank correlation coefficient. *Bmj* 349 (2014), 7327.

[123] Mohsen Sharifi, Ehsan Mousavi Khaneghah, Morteza Kashyian, and Seyedeh Leili Mirtaheri. 2012. A platform independent distributed IPC mechanism in support of programming heterogeneous distributed systems. *The Journal of Supercomputing* 59, 1 (2012), 548–567.

[124] Kavita Sheoran, Pradeep Tomar, and Rajesh Mishra. 2016. Software Quality Prediction Model with the Aid of Advanced Neural Network with HCS. *Procedia Computer Science* 92 (2016), 418–424.

[125] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294.

[126] Renuka Sindhgatta, Bikram Sengupta, and Karthikeyan Ponnalagu. 2009. Measuring the Quality of Service Oriented Design. In *Service-Oriented Computing*. Springer, 485–499.

[127] Eric Slud and LJ Wei. 1982. Two-Sample Repeated Significance Tests Based on the Modified Wilcoxon Statistic. *J. Amer. Statist. Assoc.* 77, 380 (1982), 862–868.

[128] SourceForge. 2008. Open Chord bugs. https://sourceforge.net/p/open-chord/bugs/.

[129] SourceForge.net. 2013. SourceForge. http://sourceforge.net. Last accessed: July 2013.

[130] KP Srinivasan and T Devi. 2014. SOFTWARE METRICS VALIDATION METHODOLOGIES IN SOFTWARE ENGINEERING. *International Journal of Software Engineering & Applications* 5, 6 (2014), 87.

[131] Michael Steinbach and Pang-Ning Tan. 2009. *kNN: k-Nearest Neighbors*. Chapman and Hall/CRC. 151–162 pages.

[132] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. 1974. Structured design. *IBM systems journal* 13, 2 (1974), 115–139.

[133] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 309–320.

[134] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. 2012. Serving Large-scale Batch Computed Data with Project Voldemort. In *FAST*, Vol. 12. 18–18.

[135] Amjed Tahir and Stephen G MacDonell. 2012. A Systematic Mapping Study on Dynamic Metrics and Software Quality. In *Proceedings of IEEE International Conference on Software Maintenance*. IEEE, 326–335.

[136] Thrift Repository. 2018. Index of /dist/thrift. http://archive.apache.org/dist/thrift/.

[137] Mie Mie Thet Thwin and Tong-Seng Quah. 2002. APPLICATION OF NEURAL NETWORK FOR PREDICTING SOFTWARE DEVELOPMENT FAULTS USING OBJECT-ORIENTED DESIGN METRICS. In *Proceedings of the 9th International Conference on Neural Information Processing*, Vol. 5. IEEE, 2312–2316.

[138] Hung Tong and Cristina Tortora. 2022. Model-based clustering and outlier detection with missing data. *Advances in Data Analysis and Classification* 16, 1 (2022), 5–30.

[139] Voldemort Repository. 2017. Voldemort. http://https://github.com/voldemort/voldemort/releases.

[140] Liang Wang and Jussi Kangasharju. 2013. Measuring Large-Scale Distributed Systems: Case of BitTorrent Mainline DHT. In *Proceedings of International Conference on Peer-to-Peer Computing*. IEEE, 1–10.

[141] Qi Wang, Bo Yu, and Jie Zhu. 2004. Extract Rules from Software Quality Prediction Model Based on Neural Network. In *Proceedings of IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 191–195.

[142] Dieter Wybranietz and Dieter Haban. 1988. MONITORING AND PERFORMANCE MEASURING DISTRIBUTED SYSTEMS DURING OPERATION. *ACM SIGMETRICS Performance Evaluation Review* 16, 1 (1988), 197–206.

[143] XNIO Repository. 2010. XNIO API. https://mvnrepository.com/artifact/org.jboss.xnio/xnio-api.

[144] xSocket Repository. 2011. XSocket. https://mvnrepository.com/artifact/org.xsocket/xSocket.

[145] Sherif M Yacoub and Hany H Ammar. 2002. A Methodology for Architecture-Level Reliability Risk Analysis. *IEEE Transactions on Software Engineering* 28, 6 (2002), 529–547.

[146] Sanjay Yadav and Sanyam Shukla. 2016. Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In *2016 IEEE 6th International conference on advanced computing (IACC)*. IEEE, 78–83.

[147] Yasuhiro Yamamoto, Daisuke Miyamoto, and Masaya Nakayama. 2015. Text-mining Approach for Estimating Vulnerability Score. In *Proceedings of International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. IEEE, 67–73.

[148] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better Than Supervised Models. In *Proceedings of ACM SIGSOFT international symposium*

on foundations of software engineering. 157–168.

[149] Benjamin Zeiss, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. 2007. Applying the ISO 9126 quality model to test specifications–exemplified for TTCN-3 test specifications. *Software Engineering 2007–Fachtagung des GI-Fachbereichs Softwaretechnik* 1 (2007), 231–242.

[150] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards Building a Universal Defect Prediction Model. In *Proceedings of Working Conference on Mining Software Repositories*. 182–191.

[151] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project Defect Prediction Using a Connectivity-based Unsupervised Classifier. In *IEEE/ACM International Conference on Software Engineering*. IEEE, 309–320.

[152] Wei Zhang, Liguo Huang, Vincent Ng, and Jidong Ge. 2015. SMPLearner: learning to predict software maintainability. *Automated Software Engineering* 22, 1 (2015), 111–141.

[153] QingHua Zheng, ZhiJiang Ou, Ting Liu, ZiJiang Yang, YuQiao Hou, and Chao Zheng. 2012. Software structure evaluation based on the interaction and encapsulation of methods. *Science China Information Sciences* 55, 12 (2012), 2816–2825.

[154] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. 2018. How Far We Have Progressed in the Journey? An Examination of Cross-Project Defect Prediction. *ACM Transactions on Software Engineering and Methodology* 27, 1 (2018), 1–51.

[155] Zookeeper Repository. 2019. Apache Zookeeper. https://github.com/apache/zookeeper/releases.