Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP)

Van T. K. Tran*†, Kevin Lee*†, Alan Fekete*‡, Anna Liu*† and Jacky Keung*§

*National ICT Australia Ltd., Australia

Email: {ThiKhanhVan.Tran,Kevin.Lee,Anna.Liu}@nicta.com.au

†School of Computer Science and Engineering, University of New South Wales, Australia

‡School of Information Technologies, The University of Sydney, Australia

Email: Alan.Fekete@sydney.edu.au

§The Hong Kong Polytechnic University, HKSAR

Email: Jacky.Keung@comp.polyu.edu.hk

***Abstract*—One of the major obstacles to enterprise adoption of cloud technologies has been the lack of visibility into migration effort and cost. In this paper, we present a methodology, called Cloud Migration Point (CMP), for estimating the size of cloud migration projects, by recasting a well-known software size estimation model called Function Point (FP) into the context of cloud migration. We empirically evaluate our CMP model by performing a cross-validation on six different small-scale cloud migration projects and show that our size estimation model can be used as a reliable predictor for effort estimation. Furthermore, we prove that our CMP model satisfies the fundamental properties of a software size measure.**

***Keywords*-size measures; cloud computing; migration; theoretical validation; empirical validation; effort estimation;**

I. Introduction

Since its emergence over the last decade, Cloud computing has been well recognized for its abilities to provide virtualized resources and services, such as infrastructure, platform, and software [1], [2]. Cloud computing provides computing resources on demand; therefore, Cloud users are neither required to plan far ahead for provisioning nor tied to huge up-front commitment on hardware resources and infrastructures. This elasticity of costs enables businesses to start small and acquire more resources only when needed on a short-term basis (e.g., hourly processors and daily storage), and rewards conservation by releasing computing machines and storage when they are no longer required. Cloud can also serve as additional resource (alongside existing data center) for established businesses to deal with bursts of load, perhaps seasonal, or due to intermittent activities such as stress testing. Here the cloud allows the client to delay the large commitment of funds needed to scale-up the hardware.

Many organisations write their software from scratch specifically for deployment into cloud, while others may wish to keep using their existing application software and have it run on a cloud platform. We consider the latter as "migrating" from a traditional computing platform to a cloud-based one such as Amazon EC2[1] or Microsoft Windows Azure [2]. Although the migration process is a one-off task, it is not automatic and the amount of effort required could be significant. This effort is due to discrepancies between the environment provided by a cloud platform, and that in a traditional platform. There are often differences in the version of various infrastructure, the programming models, the libraries available, even the semantics of data access are different; for example, cloud platforms typically provide eventual consistency rather than transactional guarantees.

As effort is required for migrating to cloud and the amount of effort required is diverse, early effort estimation for a migration project to a cloud platform is essential for its project management, particularly project scheduling and budget planning. Since cloud computing is new, there are not sufficient data points publicly available in regard to migration efforts to cloud. Therefore, to the best of our knowledge, no effort estimations have been proposed specifically designed for cloud application projects, while existing traditional effort estimation approaches for software development are not applicable in this context.

This strongly motivates us in our attempt to develop a Function-Point-like and cloud-specific metric, called Cloud Migration Point (CMP), to measure the size of a cloud migration project, which then serves as a basis for cloud migration effort estimation. In this paper, we introduce our approach to develop CMP and its counting method, together with theoretical and initial empirical validations.

Section II defines the scope of the paper as well as the underlying assumptions of our effort estimation model. Section III identifies cost factors of a migration project to cloud. Section IV classifies cloud migration projects into different types based on their characteristics. Section V describes our CMP metric and its counting process. Section VI shows a list of necessary properties for any size measures and proves that CMP satisfies all of them. Section VII describes our empirical validation to show the effectiveness of CMP as a basis for measuring cloud application migration effort. We also provide related work in Section VIII discussing existing

[1]http://aws.amazon.com/ec2/

[2]http://www.microsoft.com/windowsazure/

size measures and effort estimation approaches. Section IX concludes the paper.

II. Cloud Migration Project Scope

The scope of this paper is limited to a cost estimation model for cloud migration projects that satisfy the following assumptions:

- Migration between two data centers only (typically, one in-house and one in-cloud) - We assume that migration projects are directional (i.e. components are moved from local to remote).
- Cloud offerings are of Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) types - CMP considers only IaaS and PaaS, although some parts of our cost model might still be applicable to other cloud offerings. Software-as-a-Service (SaaS) is deliberately removed from the scope since users of SaaS have no control over the deployment process.
- Migrating application is object-oriented - CMP assesses application code changes at 'class' level.
- Target cloud is selected - CMP estimates the complexity of migrating to a specific cloud platform, excluding the process of determining the most suitable cloud technologies/providers, and the need to get familiar with the specific cloud technology and offering.
- Design decision is made - CMP requires inputs from the design phase and is most appropriate to apply before the implementation phase of a migration.
- Migration tasks are outlined - CMP measures the size and complexity of migration tasks, hence migration tasks must be outlined in advance.

III. Cloud Migration Cost Factors

Our approach to develop the CMP model starts with identifying important cost factors of a cloud migration project. The cost factors are defined as any aspects of the project that influence the amount of effort required. We differentiate two types of cost factors: internal and external. The former indicates what migration tasks are required and determines their complexity, the latter concerns with environmental factors that are specific to each organization, such as: development team's skills and expertise, or knowledge on cloud platforms and offerings. The internal cost factors are commonly identified first to measure the size or complexity of the project, which will then be adjusted by applying the external cost factors. Our CMP model focuses only on the internal cost factors and identify them as sole indicators of the migration tasks' complexity, regardless of who conduct those tasks and under what environment.

The reported experiments and the taxonomy of cloud migration tasks proposed in [3] enable us to understand and identify different internal cost factors of migration projects to cloud. A migration project to a cloud platform consists of a set of migration tasks involved in different categories. These categories are mutually exclusive since they cover different aspects of a cloud migration project; but on the other hand, they complement each other and altogether provide a complete picture of migration to cloud.

1) Installation and Configuration - When migrating to an IaaS cloud such as Amazon EC2, effort is required to install the necessary system software, database servers, or middlewares; environment variables and settings also need to be configured. When migrating to a PaaS cloud such as Microsoft Azure, installation and configuration effort lies in the application layer, such as libraries or plugins.
2) Database Changes - Migrating a database to cloud can result in database schema changes and query changes because of differences in versions, variants (MySQL vs. MSSQL), or database types (Relational vs. NoSQL).
3) Code Changes - In some migration cases, code modification is required to adapt to the new programming model in cloud, or database access layer needs to be changed to seamlessly work with different databases in cloud.
4) Connection Changes - Within a system S before migration, the connection between two components A and B is a LAN connection. If only B is migrated to cloud and A is kept in local data center, the LAN connection between A and B becomes a WAN connection. If both A and B are migrated to cloud, the LAN connection between A and B becomes a LAN connection in cloud (network conditions can be different in cloud).

These factors influence the amount of effort required for a cloud migration project. A metric for sizing migration projects to cloud must adequately cover these aspects in order to accommodate effort estimation for migrating to cloud.

IV. Cloud Migration Project Classification

The cost factors identified in Section III does not apply to all components of the system, but only those components that have been affected by the migration. We classify components involved in a migration into four different categories: Migrated, Removed, Unchanged and Added. These categories would help us better understand the dynamics of the migration process, as well as its impact on the effort as captured in our CMP model.

It is important to distinguish between a migrating system and a migration project. A migrating system is the system to be migrated to cloud, and is defined as a set of components required for the system to function properly, such as: third-party libraries or middlewares, system software, databases, applications' code, and network connection amongst its modules. A migration project is defined as a set of migration tasks to move a migrating system from a local data center to cloud.

We classify a migration project by, first, denoting its migrating system' states in a local data center and in cloud before and after the migration as summarized in Table I.

Table I
SYSTEM'S STATES BEFORE AND AFTER MIGRATION

	Local	Remote
Before Migration	L	R
After Migration	L'	R'

Table I depicts the components that present at each of the states, with the rows dividing the components temporally and columns dividing the components spatially. The *set* of components at each of the states are denoted by $L \neq \emptyset$, R, L' and R'. Note that, the same component may appear in different rows but they cannot appear twice in the same row (i.e. a component cannot appear both in-house and in-cloud at the same time). Hence, L and R are disjoint sets, and similarly, L' and R' are also disjoint. The allocation of components to each state can be determined using the design documents.

Definition 1: A migration project is defined as a *full migration* if $L \subseteq R'$, otherwise it is a *partial migration.*

The set of components involved in a migration project can be partitioned into three categories (or *disjoint* subsets):

- Migrated components ($\mathcal{M} = L \cap R'$) - Components moved from in-house to the cloud. These components are reused with or without modifications. For example, third-party libraries, database servers, or system software that are moved to cloud (i.e., effort involved for installation, configuration, and integration with the rest of the system); application's code (i.e., effort needed for moving and changing code); and database (i.e., efforts required for data transfer and any required modifications in schema and queries).
- Removed components ($\mathcal{R} = L \setminus (L' \cup R')$) - Components removed from in-house as a result of the migration. Removal of components is not always necessarily because some components can exist without interfering or disrupting the functionality of a system, in which case no effort is required. However, sometimes this may be necessary to ensure normal operation of the system, then effort will be required.
- Unchanged components ($\mathcal{U} = L \cap L'$) - Components that remain unchanged in-house. These components do not participate in the migration process, they simply continue to operate in-house as usual, hence no effort is required.

In addition to the above, there is also the category of Added components ($(L' \cup R') \setminus (L \cup R)$), which are components added to the system as part of the migration, such as: new libraries in cloud, newly added code for extra functionality, or integrating new middlewares. For example, when a library is not fitted for the cloud environment, a more suitable library is used if it exists in cloud or is rewritten if it is not available.

Proposition 2: If x is a component in the local data center before migration (i.e., $x \in L$), then x is one of a migrated component (i.e., $x \in \mathcal{M}$), a removed component (i.e., $x \in \mathcal{R}$) or an unchanged component (i.e., $x \in \mathcal{U}$) after migration.

Proof: It suffices to show that (1) $\mathcal{M} \cup \mathcal{R} \cup \mathcal{U} = L$, and that (2) the collection $\{\mathcal{M}, \mathcal{R}, \mathcal{U}\}$ is pairwise disjoint. For (1), $\mathcal{M} \cup \mathcal{R} \cup \mathcal{U} \equiv (L \cap R') \cup (L \setminus (L' \cup R')) \cup (L \cap L') \equiv (L \cap (L' \cup R')) \cup (L \setminus (L' \cup R')) \equiv L$. For (2), there are three cases: (i) $\mathcal{M} \cap \mathcal{R} \equiv (L \cap R') \cap (L \setminus (L' \cup R')) \equiv (L \cap R') \cap (L \cap (\neg L' \cap \neg R')) \equiv \emptyset$; (ii) $\mathcal{M} \cap \mathcal{U} \equiv (L \cap R') \cap (L \cap L') \equiv (L \cap (L' \cap R')) \equiv L \cap \emptyset \equiv \emptyset$. Note that $(L' \cap R') \equiv \emptyset$ as defined above; (iii) $\mathcal{R} \cap \mathcal{U} \equiv (L \setminus (L' \cup R')) \cap (L \cap L') \equiv (L \cap (\neg L' \cap \neg R')) \cap (L \cap L') \equiv \emptyset$. ■

The effort associated with each of the categories defined above are carefully captured in our CMP model. Roughly speaking, migrating components require the most effort, followed by adding and removing components then components with no changes.

V. CLOUD MIGRATION POINT

CMP is a metric that follows the Function Point (FP) approach [4] for measuring the size of migration projects to cloud. CMP extends FP not by adding more elements into the existing FP method, but by adopting the three-step approach of FP: (1) Classify the basic estimating units (a function in the FP context, a class in the Class Point [5] context, and a migration task in CMP context) into different pre-defined categories; then for each unit, (2) Evaluate its complexity level (Low, Average, or High); and finally (3) Compute the final sizing value.

The classification of cloud migration projects discussed in Section IV can be seen as a way to allocate components of a migration project into different categories. In this section, we delve further into the components of each category to assess the complexity of each migration task.

A. *CMP$_{conn}$*

CMP$_{\text{conn}}$ assesses all migration tasks related to network connections and evaluate their complexity. It adopts the three-step approach from FP.

First, all network connections that will be affected by the migration process and require effort to optimize performance are identified and classified into three types:

- LAN-to-LAN: A connection belongs to this type if both ends A and B of the connection are migrated from the local data center to cloud, i.e., $\{A, B\} \subseteq L \cap R'$. The LAN connection in the local site becomes a LAN connection in cloud.
- LAN-to-WAN: A connection is classified into this type if only one end A of the connection is migrated to cloud

while the other end B stays in-house (i.e., $A \in L \cap R'$ and $B \in L \cap L'$). The LAN connection in the local site becomes a WAN connection spanning from in-house to cloud over an internet connection.

- WAN-to-LAN: This type of connection happens if before migration, a part of the system is already in cloud, i.e., $R \neq \emptyset$. Before the migration, this is a WAN connection with one end A in local data center (i.e., $A \in L$) and the other end B in cloud (i.e., $B \in R$). After the migration, both ends A and B are in cloud (i.e., $A \in L \cap R'$ and $B \in R \cap R'$). The connection becomes a LAN connection in cloud environment.

Second, the complexity level (Low, Average, or High) of all migration tasks involved in *each* connection is evaluated based on its requirements for security and protocol optimization using Table II. We identify these two dimensions: Security and Protocol Optimization, as main cost factors for connection-related tasks in the cloud context, based on our cloud migration experience with cost breakdown analysis, discussion with cloud engineers, analysis of the taxonomy from [3], and close study into many cloud practitioners' blogs and discussions.

Table II
COMPLEXITY EVALUATION FOR EACH CONNECTION

Protocol Optimization	**Security**	
	Required	Not Required
Required	High	Average
Not Required	Average	Low

Lastly, a weighted value is assigned for each connection, based on its type identified from the first step and its complexity level evaluated from the second step, using Table VII. Values in Table II and VII are defined from our discussion with a group of cloud engineers involved in cloud migration projects. The value of CMP$_{\text{conn}}$ is defined as the weighted sum of all identified connections: $CMP_{conn} = \sum_{i=0}^{2} \sum_{j=0}^{2} x_{ij} \times w_{ij}$, where x_{ij} is the number of connections type i with complexity level j, and w_{ij} is the weighted value for connection type i and complexity level j.

B. *CMP$_{code}$*

CMP$_{\text{code}}$ assesses any migration tasks relating to code changes. These tasks can vary from adding new functionality, removing unnecessary code, to modifying code to use new databases or integrate with new libraries. CMP$_{\text{code}}$ is inherited from Class Point [5] but with modifications to adapt to code changes rather than just adding new functionality. Similar to CMP$_{\text{conn}}$, CMP$_{\text{code}}$ also follows FP's three-step approach.

First, all classes in application code that require modification efforts are identified and classified into four types as defined in Class Point [5]:

- Problem Domain Type (PDT): classes that represent real-world entities in the application domain of the system.
- Human Interaction Type (HIT): classes designed for information visualization and human-computer interaction.
- Data Management Type (DMT): classes that accommodate data storage and retrieval.
- Task Management Type (TMT): classes that are responsible for definition and control of tasks, communications between subsystems and with external systems.

Table III
ELEMENTS OF EACH CHANGED CLASS

Identify:	
Before changing code	After changing code
A - a set of attributes	A' - a set of attributes
M - a set of public methods	M' - a set of public methods
S - a set of services requested from other classes	S' - a set of services requested from other classes
Derive:	
$\lvert A \setminus A' \rvert$: number of attributes removed	
$\lvert A' \setminus A \rvert$: number of attributes added	
$\lvert M \setminus M' \rvert$: number of methods removed	
$\lvert M' \setminus M \rvert$: number of methods added	
$\lvert S \setminus S' \rvert$: number of requested services removed	
$\lvert S' \setminus S \rvert$: number of requested services added	
Define the changes:	
$CA = \lvert A \setminus A' \rvert \times 0.2 + \lvert A' \setminus A \rvert$: changes in attributes	
$CM = \lvert M \setminus M' \rvert \times 0.2 + \lvert M' \setminus M \rvert$: changes in methods	
$CS = \lvert S \setminus S' \rvert \times 0.2 + \lvert S' \setminus S \rvert$: changes in services requested	

Table IV
COMPLEXITY EVALUATION FOR EACH CLASS

Changes in methods (CM)	**Changes in Attributes (CA)**		
	$0-5$	$6-9$	≥ 10
$0-4$	Low	Low	Average
$5-8$	Low	Average	High
≥ 9	Average	High	High

(a) Changes in services requested (CS): $0-2$

Changes in methods (CM)	**Changes in Attributes (CA)**		
	$0-4$	$5-8$	≥ 9
$0-3$	Low	Low	Average
$4-7$	Low	Average	High
≥ 8	Average	High	High

(b) Changes in services requested (CS): $3-4$

Changes in methods (CM)	**Changes in Attributes (CA)**		
	$0-3$	$4-7$	≥ 8
$0-2$	Low	Low	Average
$3-6$	Low	Average	High
≥ 7	Average	High	High

(c) Changes in services requested (CS): ≥ 5

Second, *each* class's changes in three dimensions: attributes (CA), public methods (CM), and services requested from other classes (CS), are evaluated. These changes are made of the number of elements to be removed and added by following three steps in Table III. The sets of three elements (attributes, methods, services requested) of the system are identified both before and after code change. The number of elements to be removed and added is calculated (e.g., $|A \setminus A'|$ and $|A' \setminus A|$ are the number of attributes to be removed and added, respectively). The final values CA, CM, and CS are determined by applying a factor of 0.2 and 1 on removing and adding tasks, respectively (e.g., , $CA = |A \setminus A'| \times 0.2 + |A' \setminus A|$). These factors were suggested by Niessink and Vliet [6] since a removing task also requires effort although not as significant as an adding task.

CA, CM, and CS are defined to capture aspects of changed classes. Special circumstances happen when a class is newly added, i.e., there are no existing sets of elements before the migration, or $A = M = S = \emptyset$. In this case, $CA = |A \setminus A'| \times 0.2 + |A' \setminus A| = 0 \times 0.2 + |A'| = |A'|$, which is the number of attributes in the new class. Similarly $CM = |M'|$ and $CS = |S'|$, which are the number of methods and services requested in the new class. These three values are similar to Class Point for sizing a new class for development effort. In other words, CA, CM and CS are also valid for capturing newly added code.

These three dimensions form the basis to evaluate each changed class's complexity level as in Table IV. The complexity level indicators are inherited from Class Point.

Lastly, a weighted value is assigned for each changed class based on its type identified from the first step and its complexity level evaluated from the second step. These weights are also adopted from Class Point (shown in Table VII). The value of CMP$_{\text{code}}$ is computed as a weighted sum of all changed classes: $CMP_{code} = \sum_{i=0}^{3} \sum_{j=0}^{2} x_{ij} \times w_{ij}$, where x_{ij} is the number of classes of type i with complexity level j, and w_{ij} is the weighted value for class type i and complexity level j.

CMP$_{\text{code}}$ is analogous to Class Point in the sense that it also assesses a class' attributes, public methods, and services requested from other classes. However, it extends Class Point by evaluating the changes of elements in a class by taking into account both adding and removing tasks. Nevertheless, its validity still holds when it comes to adding an entirely new class, in which case its counting approach is exactly the same as Class Point, as shown above. As a result, all complexity levels and weighted values can be sufficiently inherited from Class Point.

C. *CMP$_{ic}$*

CMP$_{\text{ic}}$ assesses all migration tasks related to Installation and Configuration (IC), such as: installation of system software, middleware, database server, third-party library; or configuration of environment variable and basic network information. CMP$_{\text{ic}}$ is determined in a similar manner as the previous two components of CMP.

First, all required installation and configuration tasks are identified and classified into two types:

- Infrastructure level: software or servers required to set up the environment belong to this type, for example, setting up EC2 instance or image, installing operating system and middleware, or installing database server.
- Application level: this type consists of any third-party libraries that the application requires, for example, JDBC drivers for databases. When an application relies on an external library to function properly, and that library does not exist within the cloud environment, there are two options: (1) Rewrite the library from scratch for the cloud environment - This is seen by CMP as adding new code into the system and is sufficiently captured by CMP$_{\text{code}}$. Hence, the migration tasks related to this option are excluded from CMP$_{\text{ic}}$. (2) Reuse a similar library (if one exists) in the cloud environment, and change code in the system to preserve functionality and to connect with the new library seamlessly - The migration tasks involved in this option are integrating the new library into the system, which will be assessed by CMP$_{\text{ic}}$, and changing code, which is assessed by CMP$_{\text{code}}$ and excluded from CMP$_{\text{ic}}$. If the libraries are found available in the cloud environment exactly as required, the migration tasks expected are to integrate them with the system and are measured by CMP$_{\text{ic}}$.

Second, we evaluate the complexity of each IC task based on the number of configuration steps required and the installation methods (from binary files or source code) as in Table V.

Table V
COMPLEXITY EVALUATION FOR EACH IC TASK

Configuration	**Installation**		
	No installation	Package	Source Code
< 2	Low	Low	Average
$2 - 5$	Low	Average	High
≥ 6	Average	High	High

Finally, each IC task is assigned with a weighted value as in Table VII based on its type from the first step and its complexity level from the second one. The final value of CMP$_{\text{ic}}$ is determined as: $CMP_{ic} = \sum_{i=0}^{1} \sum_{j=0}^{2} x_{ij} \times w_{ij}$, where x_{ij} is the number of IC tasks of type i with complexity level j, and w_{ij} is the weighted value for IC task type i and complexity level j.

D. *CMP$_{db}$*

CMP$_{\text{db}}$ assesses all migration tasks related to modifying queries and populating data to new databases, excluding database server installation tasks and any code changes required which have been covered by CMP$_{\text{ic}}$ and CMP$_{\text{code}}$, respectively. Since the effort required for each query modification task or data population task is quite uniform, CMP$_{\text{db}}$ is easier to calculate than other CMP components.

First, all database related tasks are identified and classified into two types:

- Query modification task: when a database changes in database type (e.g., MySQL to MSSQL), or database

version, or from relational to NoSQL database, queries must be modified accordingly.
- Data population task: Data in each table must be packaged and loaded into the new database.

Table VI
COMPLEXITY EVALUATION FOR EACH DATABASE TASK

Database changes	**Complexity level**
Same relational database, same version	Low
Same relational database, different version	Average
Different relational databases	Average
Relational to NoSQL databases	High

Second, the complexity of each task is determined based on the differences between the database of the local data center and the database in cloud: same type of relational database, same type of relational database but different versions, different types of relational databases, or relational to NoSQL database. Table VI summarizes these complexity levels.

Finally, CMP_{db} is determined by the number of database tasks and for each database task its associated weight as in Table VII. The final value of CMP_{db} is calculated as: $CMP_{db} = \sum_{i=0}^{1} \sum_{j=0}^{2} x_{ij} \times w_{ij}$, where x_{ij} is the number of database tasks of type i (i.e., the number of queries to be modified or the number of tables to be populated) with complexity level j, and w_{ij} is the weighted value for database task type i and complexity level j.

E. CMP

The final value of CMP is determined as a weighted sum of its four components CMP_{i} with $i \in$ {conn, code, ic, db}:

$$CMP = \sum_{i=0}^{3} CMP_i \times w_i$$

where CMP_{i} is the value of CMP type i, and w_i is the weighted value for CMP type i (as shown in Table VII).

Table VII
EVALUATING CMP'S COMPONENTS

CMP	**Type**	**Complexity Level**			**Weight**
		Low	Average	High	
CMP_{conn}	LAN-to-LAN	1	3	4	3
	LAN-to-WAN	1	6	9	
	WAN-to-LAN	1	6	9	
CMP_{code}	PDT	3	6	10	5
	HIT	4	7	12	
	DMT	5	8	13	
	TMT	4	6	9	
CMP_{ic}	Application	1	2	7	2
	Infrastructure	1	3	9	
CMP_{db}	Query Modification	1	3	8	1
	Data Population	3	4	10	

VI. THEORETICAL VALIDATION

Validation is an essential process to justify whether a software metric meets its specification and fulfils its intended purpose [5], [7]. It is widely accepted that there are two types of validation required for software metrics, namely theoretical validation and empirical validation. The objective of the theoretical validation is to prove that a metric sufficiently satisfies the necessary conditions of a measurement metric that it claims to be (e.g., , sizing metrics, complexity metrics, cohesion metrics, coupling metrics, etc.), whereas the empirical validation is to show that the metric is practically useful within a given context.

Briand et al. [7] proposed a generic mathematical framework that defines some software measurement concepts, such as size and complexity. The framework provides different sets of convenient and intuitive properties which are used as necessary conditions for each measurement concept. In this section, we mathematically validate CMP against three properties of a size concept proposed in [7].

Based on [7], a system S can be represented as a pair $\langle E, R \rangle$, where E is the set of elements of S, and R is a binary relation of E ($R \subseteq E \times E$). In the context of this paper, a migration project is defined as a set of migration tasks as in Section IV, and each migration task examines one component to be migrated. In light of this analogy, a migration project can be represented as a system $S = \langle E, R \rangle$, where E is the set of migrating components of S, i.e., components to be migrated, removed, and added; and R is the set of connections between elements of E, i.e., method calls between classes and network connections between non-code elements.

Three properties for a size metric proposed by [7] are: Non-negativity, Null Value, and Module Additivity. These properties are formalized as:

- **Property Size 1: Non-negativity** - *The size of a system $S = \langle E, R \rangle$ is non-negative: $Size(S) \geq 0$*
  Proof: $Size(S)$ is the CMP value of the migration project S. CMP is obtained as a weighted sum of its four components, which in turn are weighted sums of non-negative numbers. Hence, $CMP = Size(S) \geq 0$, or the Non-Negativity Property is verified. ■
- **Property Size 2: Null Value** - *The size of a system $S = \langle E, R \rangle$ is null if E is empty: $E = \emptyset \Rightarrow Size(S) = 0$*
  Proof: CMP is determined by assessing each component to be migrated, evaluating its migration task's complexity, and assigning an associated weight to it. The final value of CMP is the sum of all the weights of the migration task set. If $E = \emptyset$, i.e., there exist no components to be migrated, there is no weight to be assigned. Hence, $CMP = Size(S) = 0$, or the Null Value Property holds. ■
- **Property Size 3: Module Additivity** - *The size of a system $S = \langle E, R \rangle$ is equal to the sum of the size of two of its modules $m_1 = \langle E_{m1}, R_{m1} \rangle$ and $m_2 = \langle E_{m2}, R_{m2} \rangle$ such that any element of S is an element of either m_1 or m_2: $\forall m_1, m_2((m_1 \subseteq S$ and $m_2 \subseteq S$ and $E = E_{m1} \cup E_{m2}$ and $E_{m1} \cap E_{m2} = \emptyset) \Rightarrow$*

$Size(S) = Size(m_1) + Size(m_2))$

Proof: The CMP calculation examines each element e_i (i.e., migrating component i) of $E = \{e_0, e_1, ..., e_{n-1}\}$ independently. Each element e_i is assessed by a migration task, which is assigned a weight w_i. CMP is then determined as the sum of all these weights, i.e., $CMP = \sum_{i=0}^{n-1} w_i$. If E is divided into two disjoint subsets E_{m1} and E_{m2}, with no loss of generality, E_{m1} and E_{m2} can be represented as: $E_{m1} = \{e_0, e_1, ..., e_{k-1}\}$ and $E_{m2} = \{e_k, e_{k+1}, ..., e_{n-1}\}$, where $k \leq n$. Applying the same process of determining CMP, the values CMP_{m1} and CMP_{m2} of these two subsets of migration tasks E_{m1} and E_{m2} are: $CMP_{m1} = \sum_{i=0}^{k-1} w_i$ and $CMP = \sum_{i=k}^{n-1} w_i$. As a result, $CMP_{m1} + CMP_{m2} = \sum_{i=0}^{k-1} w_i + \sum_{i=k}^{n-1} w_i = \sum_{i=0}^{n-1} w_i = CMP$. Hence, the Module Additivity Property is satisfied. ■

We have shown that CMP satisfies all three necessary conditions of a size measurement proposed by [7], hence it can sufficiently measure a migration project's size.

VII. Empirical Validation

Besides the theoretical validation, the empirical validation is necessary to ensure that CMP is practically useful as an indicator of effort estimation in terms of person-hours. Because of the limited number of data points publicly available, the data we use for this empirical validation is extracted from a number of small-scale projects we have conducted. Although the validity of these data points has not been verified externally with other research projects, they are suitable for empirical validation because: (1) We have access to all necessary information required to determine CMP. (2) These projects cover different migration project types. In an actual migration project, not all aspects of CMP happens at the same time in one project. Therefore, these data points sufficiently reflect what is likely to happen in reality. (3) The uniformity of these projects are ensured, because they were carried out by the same team. Therefore, the external cost factors as discussed in Section III have minimal impact on these data points. This is suitable for validating the CMP model since we focus on internal cost factors only.

Table VIII shows the data points extracted from our six projects. For FSO, the majority of the effort was spent on securing and optimizing WAN connection. While TPC-C, TPC-W, and Cloudstone required most effort on installation and population of data. The migration process of PetShop. Net and PetStore Java involved installation, data population, and code changes, especially migrating PetStore Java relational database to SimpleDB.

As part of our empirical validation, we followed a leave-one-out cross-validation approach on the samples of our dataset. This approach is the same as a k-fold cross-validation, in which k is equal to the number of data points. The k-fold cross-validation has been successfully used to validate cost estimation models in the literature, and is especially recommended for small data sets [5], [8]. In the leave-one-out cross-validation, each single data point is used as the validation data, whereas the remaining data are used as training sets. This is repeated until each data point is used as the validation data once.

Table VIII
Empirical validation data points

No	Project	Effort(hours)	CMP
1	FSO	45	504
2	TPC-C	4	60
3	TPC-W	6	95
4	Cloudstone	9	149
5	PetShop .Net	32	337
6	PetStore Java	51	645

In our context, we performed six rounds of validation. Each round uses five projects as the training set, and one project is left out as the validation set. Descriptive statistics were computed for each training set, based on which the boxplot and outliers of each set were analysed. Figure 1 shows that there are no outliers in the training sets of the six validation rounds which may biasedly influence the derived models from regression analysis.

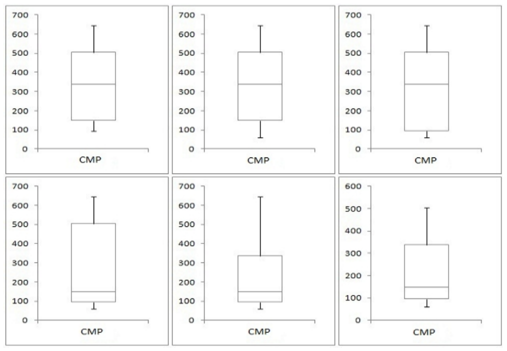

Figure 1. The boxplots for the six training datasets of variable CMP

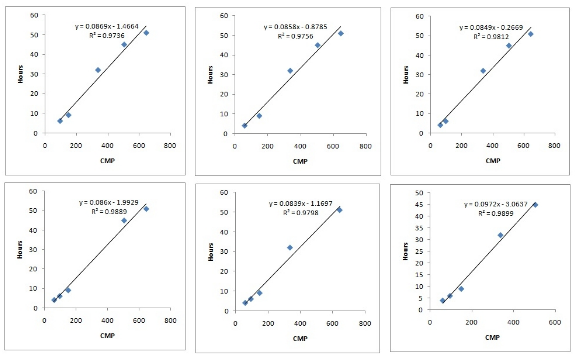

Figure 2. The scatter plots for OLS regression

The scatter plots in Figure 2 show a positive linear relationship between CMP and Effort (in hours) of each training set. As a result, an Ordinary Least-Squares (OLS) regression analysis is then applied on each training set to

Table IX
OLS Regression Analysis Results

Training Set	Coefficient			Intercept			R^2	Models: Effort = m $\times$ CMP + c
	Value	t-value	p-value	Value	t-value	p-value		
1	0.0869	10.504	0.002	-1.4664	-0.4399	0.6898	0.9736	Effort $= 0.0869 \times$ CMP $- 1.4664$
2	0.0858	10.958	0.002	-0.8785	-0.2789	0.7984	0.9756	Effort $= 0.0858 \times$ CMP $- 0.8785$
3	0.0849	12.507	0.001	-0.2669	-0.0985	0.9277	0.9812	Effort $= 0.0849 \times$ CMP $- 0.2669$
4	0.086	16.339	0.000	-1.9929	-1.0084	0.3876	0.9889	Effort $= 0.086 \times$ CMP $- 1.9929$
5	0.0839	12.069	0.001	-1.1697	-0.501	0.6508	0.9798	Effort $= 0.0839 \times$ CMP $- 1.1697$
6	0.0972	17.112	0.000	-3.0637	-1.9008	0.1535	0.9899	Effort $= 0.0972 \times$ CMP $- 3.0637$

derive the equation of the trend line, which can be used as a prediction model for effort required in hours.

The proficiency of each regression model is determined by the Coefficient of Determination R^2, representing the proportion of the dependent variable effort (in hours) explained by the independent variable CMP. Moreover, the statistical significance of CMP as a predictor of effort is evaluated with *t-test* and is determined by *t-value* and *p-value* of the coefficient of the prediction model. If *p-value* < 0.05, the null hypothesis can be rejected; in other words, it shows that CMP is a significant predictor of effort. The *t-value* is then applied to indicate the reliability of the predictor. If *t-value* > 1.5, it shows that CMP is a reliable predictor of effort. The results of R^2, *t-value*, and *p-value* of the coefficients and the intercepts of all six validation rounds are summarized in Table IX. The result suggests that the coefficients of the models are statistically significant and hence CMP is indicated to be a significant predictor of effort. Although the intercepts are statistically insignificant, each derived model has a high value of R^2 and all the coefficients pass the significant test. In other words, the OLS regression analysis results still shows a strong linear relationship between CMP and effort (in hours). For example, in the first training set, the derived model is: Effort $= 0.0869\times$ CMP -1.9929, with high value of $R^2 = 0.9736$ and the coefficient is significant at level 0.05.

The cross-validation result were determined by using the derived models to compute the predicted effort of the left-out project in each validation round (reported in Table X). The results were then evaluated using the following metrics:

- Magnitude Relative Error (MRE): $MRE = |AE - PE|/AE$, where AE is Actual Effort, and PE is Predicted Effort.
- Mean Magnitude Relative Error ($MMRE$): $MMRE = \sum MRE/n$, where n is the sample size, and $MMRE \leq 0.25$ is acceptable.
- Prediction at level l (or $PRED(l)$ in short): $PRED(l) = k/n$, where k is the number of observations such that $MRE \leq l$. Note that, $k \leq n$, hence $0 \leq PRED(l) \leq 1$. The closer the $PRED(l)$ value to 1 the better, and $PRED(0.25) \geq 0.75$ is acceptable.

Table X shows that the MMRE value is 0.199 and the prediction at level 0.25 is 0.833. This result suggests that the CMP can produce an accurate cost estimate which can be used as a predictor of effort estimation for cloud migration projects. The metric is useful and thus allows cost justification of a cloud migration project at early stage.

Table X
Results Evaluation

No	Project	CMP	AE	PE	MRE
1	FSO	504	45	41.116	0.086
2	TPC-C	60	4	3.221	0.195
3	TPC-W	95	6	7.272	0.212
4	Cloudstone	149	9	12.383	0.376
5	PetShop .Net	337	32	26.989	0.157
6	PetStore Java	645	51	59.63	0.169
				MMRE	0.199
				PRED(0.25)	0.833

VIII. Related Work

Software effort estimation is an important and challenging issue in empirical software engineering. A more complicated project typically requires more effort both in development and maintenance. Software size measurement is a conventional way to indicate a project's complexity. In this section, we review existing software size measures and effort estimation approaches to evaluate their applicability in the context of migration to cloud.

A. Software Size Measurement

Software size measurement is commonly derived in the form of a metric to measure the complexity and characteristics of a software project using either source lines of code or function points or it extended variants.

Source Line of Code (SLOC) is a traditional size measure that counts the number of lines in a software product's source code [9], [10]. However, counting SLOC is only possible after the implementation phase when source code is available, which makes SLOC not applicable for estimation in early phase of the development cycle [4] [11].

Function Point (FP) is a metric developed by Albrecht in 1983 [4] to measure a software system's size in terms of system functionality, independent of implementation language. FP is used to estimate the amount of functions a software provides, on the rationale of how much data it uses and generates. FP of a system can be developed as early as discussions with customers in the development cycle proceeds, because it is directly related to user requirements. Although FP is most applicable for only procedural business systems, it has formed a firm foundation for a number of extensions suitable for other types of systems and development paradigms [5], [12]–[17].

Karner [16] proposed *Use Case Point* (UCP) model inspired by FP. It measures a system functionality based on use cases, actors, and transactions. Abran et al. [12] extended the applicability of FP to real-time software by introducing *Full Function Point* (FFP). FFP redefines FP's function types to capture specific real-time software characteristics that FP fails to measure, such as: large number of single occurrence groups of data, or fluctuating number of sub-processes. At the same time, Antoniol et al. [14], on the other hand, developed *Object-Oriented Function Point* (OOFP) for sizing OO systems. OOFP relies on object models to map FP's function types into OO concepts. Reifer [17] extended FP to *Web Object* (WO) for sizing Web projects, by adding four new web-specific components: multimedia files, web building blocks, scripts, and links.

Costagliola et al. [5], in 2005, proposed *Class Point* (CP_1 and CP_2 for initial size estimation at the beginning of the development process and further detailed estimation when more data are available later in the development process, respectively). Class Point does not apply one-to-one mappings from FP's function types to OO concepts like other extensions, but rather focus on classes as the basic units.

SLOC, FP and its extensions have been widely used to measure size of different types of systems and development paradigms. However, their applicability is limited to software functionality development. The main purpose of migrating a system to cloud is not to develop new functionalities, but to reuse the existing ones, while, at the same time, benefited the best performance from cloud offerings. In light of this stance, none of the existing metrics are suitable for estimating size and effort of a migration project to cloud.

B. *Effort Estimation Approaches*

There is a diverse range of effort estimation approaches in the literature [18], [19]. They can be categorized into three general types: analogy, expert judgement, and algorithmic models [20]–[22].

Effort estimation using analogy is the process of identifying a problem as a new case, retrieving similar cases from a repository and using the knowledge derived from those previous cases to suggest a solution for the new case [20]. This approach is not applicable for the cloud context at this stage because cloud is still at its early adoption phase and hence there is a lack of data points for previously completed migration projects. Analogy-based estimation typically requires tens of data points.

Expert judgement is another well-known approach for estimation [21]. This approach captures knowledge, experiences, and expertise of practitioners who are recognized as experts within a domain of interest, and derives estimates based on historical data that they are well aware of, or past projects that they participated. Similar to the analogy-based approach, because of the newly emergence of cloud, there is a lack of experienced developers specialising in the task of cloud migration. Nevertheless, this approach shows a great potential when cloud gets more mature in the future.

The most dominating estimation approach is algorithmic models [18]. This approach estimates efforts using mathematical formulas to establish the relationship between dependent and independent variables of the models, which are the estimated effort and influential cost factors, respectively. This approach also required historical data to develop the algorithmic model; however, the model itself is more generic than the other two approaches, which makes model-based technique more suitable to apply for a broader range of migration projects to cloud at this stage.

Size of a project is one of the key factors in algorithmic models for the project's effort estimation. This again confirms our motivation to build the CMP model for sizing a migration project to cloud.

IX. Conclusion

Cloud Migration Point (CMP) approach has been developed in this paper as an important software size measure for legacy-to-cloud migration projects. Our study shows CMP is more suitable for cloud migration projects than other existing size metrics in the literature since it captures special aspects of the cloud migration context, as discussed in section III. Moreover, CMP emphasises on distinguished features of cloud migration, as distinct from migrating between two local data centres, such as cloud users (or developers) do not possess full control over the cloud environment as they do in a local data centre. This results in the limited range of actions for each migration task. Therefore, the CMP model takes into consideration cloud-specific dependencies for each migration task, for example, only security and protocol optimisation are assessed for each connection task, or database tasks are concerned with migrating from relational to NoSQL databases, and so on.

In a project development cycle, the CMP model fits well into before the implementation phase and after the design phase. One important assumption for CMP is all design decisions have been made. These design decisions have direct impact on how CMP is counted, since they define all anticipated migration tasks. The CMP counting process itself does not require much training and effort; however, its accuracy relies on the sufficiency and granularity of the migration task list. Therefore, it is important to carefully analyse the list of expected migration tasks to ensure it captures the cloud migration aspects adequately and with as much details as possible.

The CMP model measures the size of a migration project from a local data center to cloud with the condition $L \neq \emptyset$, as discussed in Section IV. However, the CMP model was developed without any constraint on L. In other words, the CMP model is also applicable to migration projects with $L = \emptyset$, which means the system can be migrated from cloud back to the local data center. This symmetric characteristic of CMP enables the measurement to expand beyond just two data centers. When there are more than two data centers (either from local to cloud, or vice versa) involved in the migration process, the CMP model can be repeatedly applied for each pair at a time.

The CMP model satisfies all necessary conditions of a software size measurement and it has been empirically validated as a predictor of effort estimation for cloud migration projects with the dataset extracted from our experiments and industrial projects. We have identified more possible data points from other industrial and public projects. Interviews and surveys have been planned as our next step to collect data from those projects for further validation of CMP.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., 2009.

[3] V. Tran, J. Keung, A. Liu, and A. Fekete, "Application migration to cloud: A taxonomy of critical factors," in *Proceedings of the ICSE Software Engineering For Cloud Computing Workshop, SECLOUD*. New York, NY, USA: ACM, 2011.

[4] A. Albrecht and J. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Transactions on Software Engineering*, vol. 9, pp. 639–648, 1983.

[5] G. Costagliola, F. Ferrucci, G. Tortora, and G. Vitiello, "Class point: An approach for the size estimation of object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 31, pp. 52–74, 2005.

[6] F. Niessink and H. v. Vliet, "Predicting maintenance effort with function points," in *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 32–39.

[7] L. Briand, S. Morasca, and V. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68 –86, Jan. 1996.

[8] L. Briand, K. El Emam, D. Surmann, I. Wieczorek, and K. Maxwell, "An assessment and comparison of common software cost estimation modeling techniques," in *Proceedings of the International Conference on Software Engineering ICSE*, May 1999, pp. 313 –323.

[9] J. Verner and G. Tate, "A software size model," *IEEE Transactions on Software Engineering*, vol. 18, pp. 265–278, April 1992.

[10] J. J. Dolado, "A validation of the component-based method for software size estimation," *IEEE Transactions on Software Engineering*, vol. 26, pp. 1006–1021, October 2000.

[11] R. Lai and S.-J. Huang, "A model for estimating the size of a formal communication protocol specification and its implementation," *IEEE Transactions on Software Engineering*, vol. 29, pp. 46–62, January 2003.

[12] A. Abran, "Functional size measurement for real time and embedded software," in *Proceedings of the 4th IEEE International Symposium and Forum on Software Engineering Standards*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 259–.

[13] T. Dekkers, F. Vogelezang, and S. N. B. V, "Cosmic full function points: Additional to or replacing fpa," in *Proceedings of the Ninth International Software Metrics Symposium*, ser. ACOSM, 2003.

[14] G. Antoniol, C. Lokan, G. Caldiera, and R. Fiutem, "A function point-like measure for object-oriented software," *Empirical Software Engineering*, vol. 4, pp. 263–287, September 1999.

[15] P. Mohagheghi, B. Anda, and R. Conradi, "Effort estimation of use cases for incremental large-scale software development," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE. New York, NY, USA: ACM, 2005, pp. 303–311.

[16] G. Karner, "Resource Estimation for Objectory Projects," *Objectory Systems*, 1993.

[17] D. Reifer, "Web development: estimating quick-to-market software," *Software, IEEE*, vol. 17, no. 6, pp. 57 –64, 2000.

[18] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 33–53, January 2007.

[19] B. Boehm, C. Abts, and S. Chulani, "Software development cost estimation approaches a survey," *Annals of Software Engineering*, vol. 10, pp. 177–205, January 2000.

[20] M. Shepperd and C. Schofield, "Estimating software project effort using analogies," *IEEE Transactions on Software Engineering*, vol. 23, no. 11, pp. 736 –743, Nov. 1997.

[21] M. Jorgensen, "A review of studies on expert estimation of software development effort," *Journal of Systems and Software*, vol. 70, no. 1-2, pp. 37 – 60, 2004.

[22] G. R. Finnie, G. E. Wittig, and J.-M. Desharnais, "A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models," *Journal of Systems and Software*, vol. 39, no. 3, pp. 281 – 289, 1997.