STRUCTURAL COMPLEXITY METRICS FOR LARAVEL SOFTWARE

Kevin Agina Onyango[1], Geoffrey Muchiri Muketha[2] and John Gichuki Ndia[1]

[1] Department of Information Technology, School of Computing and Information Technology, Murang'a University of Technology, Kenya
[2] Department of Computer Science, School of Computing and Information Technology, Murang'a University of Technology, Kenya

ABSTRACT

Existing software complexity metrics do not adequately address the unique architectural patterns of Laravel. This research, therefore, solves this problem by proposing a suite of novel complexity metrics for Laravel software. The metric definition employs the Entity-Attribute-Metric-Tooling (EAMT) model. These proposed metrics are designed to assess the complexity of Laravel software at the class level within Laravel's Model-View-Controller (MVC) architecture as guided by an Architecture-based Complexity Classification Framework for Laravel Software ($ACCF_{LS}$). The metrics offer a better approach to understanding and managing software complexity in Laravel projects. The study defined three composite metrics namely Controller Complexity Metrics for Laravel (CCM_{LV}), Model Complexity Metrics for Laravel (MCM_{LV}), and View Complexity Metrics for Laravel (VCM_{LV}). They were theoretically validated with Weyuker's properties framework and satisfied seven out of the nine properties, which is an acceptable compliance level. Moreover, the validation of the metrics against the Kaner framework further emphasizes their practicability and relevance to real-world software development scenarios. This research not only contributes to a deeper understanding of software complexity in Laravel applications but also lays the groundwork for future empirical validation and the development of automated tools for complexity measurement.

KEYWORDS

Software Metrics, Laravel Software, Theoretical Validation, Software Quality, MVC Design Pattern & EAMT Model.

1. INTRODUCTION

In the evolving field of web development, Laravel has emerged as a preferred PHP- development framework for many developers, known for its elegant syntax and robust features that facilitate rapid application development [1-2]. However, as with any software development process, understanding and managing the complexity of code is paramount to maintaining high quality, scalability, and ease of modification [3]. Traditional software complexity metrics, while providing a generalized understanding, often fall short in addressing the nuances and architectural specifics of frameworks like Laravel [1, 4-5]. This gap underscores the need for a set of novel metrics that are specifically designed to evaluate the complexity of Laravel applications. Moreover, complexity metrics tailored to Laravel can provide insights into the cognitive load required to understand and modify the code, thereby facilitating better project planning and risk management. This study aims to address this gap by proposing a suite of theoretically sound complexity novel metrics designed for Laravel applications guided by the Model-View-Controller (MVC) design pattern.

Software complexity, also known as program complexity describes the attributes of software that affect its internal interactions i.e. how the attributes are intertwined with one another [6]. Focusing on how the code interacts with other pieces or entities of code [6-7]. Previous studies indicate that, in software development, software complexity cannot be eliminated in totality but instead, the concept can only be controlled [5, 8]. Software metrics over time have been appreciated by researchers as one of the measures of software characteristics that are quantifiable [7]. Metrics play a crucial role in understanding and managing software complexity as they can be used to evaluate and predict software complexity [4].

Previously, an Architecture-based Complexity Classification Framework for Laravel Software (ACCF$_{LS}$) was developed to identify the unique Laravel attributes that course inherent complexity in software developed using Laravel and classify those attributes under the three main classes of Laravel architecture guided by the MVC design pattern theory. Based on the identified attributes by the ACCF$_{LS}$ classification framework, this study, therefore, proposes a novel set of metrics that accurately reflect the complexity of Laravel applications at the class level. Kaner framework and Weyuker's properties were used to validate the proposed metrics to establish their practicality and theoretical soundness.

The remainder of this paper is organized as follows: Section 2 reviews related works. Section 3 details the identification of attributes, followed by Section 4, where the proposed metrics are defined. Section 5 presents the theoretical validation of these metrics. Discussion, Conclusion,and suggestions for future research are presented in sections 6 and 7 respectively.

2. Related Works

This section describes various structural metrics that that been mostly adopted to measure software complexity in web-based domains and other paradigms for instance:

i) *Control Flow Complexity Metrics* ("McCabe's' Cyclomatic complexity metrics"). McCabe's metrics are based on a control flow representation of the program. In this measure, a program graph is used to depict control flow whereas nodes represent processing tasks, and the edge of the program graph represents the control flow between nodes [9]. According to this metric, the complexity M is then defined as shown in Eq. 1:

$$M=E-N+2P \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Eq. (1)}$$

where: "E = the number of edges of the graph. N = the number of nodes of the graph. P = the number of connected components."

ii) *Language Complexity Metrics* ("Halstead Metrics"). The Halstead complexity metrics use distinct operators and operands to compute the volume, difficulty, and effort among other parameters of a piece of code. This metric is a way of determining a quantitative measure of complexity directly from the operators and operands of a module. It measures the complexity of a given programming language by summarizing the number of operators and operands contained in a program [10].

iii) *Interface Complexity* ("Henry fan-in/fan-out metrics"). Interface complexity measures the complexity as a function of fan in and fan out. Fan in is defined as the number of local flows into a given procedure plus the number of data structures from which that procedure retrieves information. Fun out on the other hand is the number of local flows outs out of a given procedure plus the number of data structures that the procedure updates [11]. The metric is given as shown in Eq. 2:

$$\text{Complexity} = (\text{Procedure Length}) * (\text{fan-in} * \text{fan-out})^2 \text{ ………………… Eq. (2)}$$

iv) *Throughput and Load Time.* These are performance metrics that can indirectly indicate the complexity of the Laravel software. Throughput measures the number of requests the Laravel system can handle without going down, and load time measures the amount of time it takes for an HTTP request to respond [12].

v) *Average Complexity per LLOC, Class, and Method.* These are specific metrics that measure the average complexity of lines of code, classes, and methods in a given Laravel application [12].

vi) *Query complexity and database size measures for performance, and coupling and cohesion measures for maintainability.* These measures were proposed to evaluate the maintainability and performance of object-relational mapping of Laravel [12].

Although these studies are promising, they have overlooked metrics that focus on the unique structural features of Laravel like restful controller functions, their function calls, the special array variables, entity relationships, view bade template inheritance, and view blade template nesting as guided by the MVC design pattern of a Laravel Software. Therefore, in Laravel, the existing traditional software complexity metrics need to be adopted directly to analyze the complexity of Laravel software due to the structural differences.

3. ATTRIBUTES IDENTIFICATION

This study adopted the Entity-Attribute-Metric-Tool (EAMT) metrics definition model [2-3] to define the proposed metrics. The Laravel software was identified as the entity of concern which is made up of three main classes that is Model class, View class, and Controller class, following the MVC design pattern, which is a conventional design pattern for Laravel software development.

In a Laravel Controller class, the main attributes to measure are functions and function calls. In a Laravel Model class, entity relationships and array variables stood out as the main attributes to measure while in a View class, blade template directives are the main attributes of focus. $ACCF_{LS}$ framework also highlighted granularized measurable sub-attributes of each of these attributes of concern. Therefore, to define metrics for each attribute of concern, the measurable attributes and corresponding sub-attributes were considered as presented in the subsequent section.

4. METRICS DEFINITION

The proposed metrics are defined at the class level, according to the attributes identified and classified by the $ACCF_{LS}$, the attributes are classified, the classes concerned are Laravel controller class, Laravel model class, and Laravel view class.

Therefore, the three proposed composite complexity metrics measure the complexity of Laravel software at the class level and are formally defined as a 3-tuple $< M, V, C >$. They include; Controller Complexity Metrics for Laravel (CCM_{LV}), Model Complexity Metrics for Laravel (MCM_{LV}), and View Complexity Metrics for Laravel (VCM_{LV}).

4.1. Controller Complexity Metrics for Laravel (CCM_{LV})

The Laravel controller class is majorly composed of Laravel Functions and Function calls [13]. Therefore, as shown in Eq. 3, the proposed CCM_{LV} is a composite metric containing two independent metrics informed by the two main controller-based attributes. These independent metrics are Laravel Function Complexity Metric *(LF)* and Laravel Function Call Complexity Metric *(LFC)*.

$$CCM_{LV} = LF + LFC \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{Eq. (3)}$$

Where

LF is the Laravel Function Complexity Metric
LFC is the Laravel Function Call Complexity Metric

4.1.1. Laravel Function Complexity Metric (LF)

Laravel function is determined by considering the two measurable sub-attributes that is Parameterized functions (P_f) and Non-Parameterized Laravel functions (*NP_f*). Therefore, the definition of a metric to assess the complexity of the Laravel function here is done in two stages, the first stage is the definition of base metrics, i.e. where the study collects and computes the metrics directly from a Laravel controller class. This will measure the number of P_f and the number of NP_f [13]. Therefore, in the definition of the derived *LF* metric, weights are assigned to the count of individual measurable sub-attributes.

In Laravel, when a function is parameterized, its complexity is increased since there are other elements (parameters) that must be executed within the braces before proceeding to the next line of execution e.g. ***public function update (company $company)***. This implies that before execution of the next line of code, the content inside the brace ( ) must first be executed. On the other hand, non-parameterized functions are less complex since it does not have instructions (parameters) inside the braces to be executed before going to the next line of code for instance ***public function index ( ).*** This implies that in Laravel, P_f is weightier and more complex compared to NP_f.

In Laravel, non-parameterized Laravel functions are logical program statements inside the outermost level of control structures with empty parenthesis therefore are assigned a weight of 1 while parametrized Laravel functions are logical statement structures inside the outermost level of control structures with parameters inside the parenthesis hence a weight of 1.5 is assigned to it as shown in Table 1, based on existing weighting theories [14-16].

Table 1: Weights assigned to Laravel Functions

Types of Laravel Functions (F_i)	**Weight Description**	**Corresponding Weights (W_i)**
Non-Parameterized Function *(NP_f)*	Weight of Non-Parameterized Function (W_{NPf})	W_{NPf} =1.0
Parameterized Function *(P_f)*	Weight of Parameterized Function (W_{Pf})	W_{Pf} = 1.5

Therefore, to compute the complexity of LF, the corresponding weight is multiplied by the number of each sub-attribute of Parameterized Function and Non-Parameterized function as shown in Eq. 4:

$$\mathrm{LF} = \sum_{\mathrm{i}=1}^{\mathrm{n}} (\mathrm{F_i W_i}) \text{ .... Eq. (4)}$$

Where,
F is the various types of Laravel functions, i is the start of the first Laravel function, W_i is the weight assigned to the corresponding various Laravel functions, and n is the last Laravel function
.

4.1.2. Laravel Function Call Complexity Metric (LFC)

Function calls are the other attribute that makes up a Laravel controller class. Laravel function calls can be categorized as Regular Function calls (R_{FC}), Nesting Function call (N_{FC}), Chaining Function call (C_{FC}) and Hybrid Function call (H_{FC}). These four classifications form the measurable sub-attributes in the definition of the proposed LFC metric [13]. Therefore, the metric to measure the complexity of the Laravel function call is done in two stages. The first stage is the definition of the base metrics, where these measurable sub-attributes are directly collected and computed by counting the number of R_{FC}, N_{FC}, C_{FC} and H_{FC}. The second stage of the metrics definition process is viewed as derived metrics, this will give the overall LFC. In the definition of the derived metrics, weights are assigned to measurable sub-attributes.

When structural compositions of these Laravel function calls implementation are considered, the regular function call is the least complex since there is only a single function being called e.g. ***return view('home').*** This makes the regular function call less complex compared to the nesting, chaining, and hybrid function calls. In a nesting function call, there is a function call inside another function call e.g. ***return view ('company. index', compact(companies));*** so in this scenario, the function call ***compact('companies')*** is a function called inside another function ***return view('company.index').*** In a chaining function call, one function call is chained/points/directs or leads to another function call in the same execution line of code after another function call e.g. ***$comapnies=Company::with('customers') -> paginate(5)***, in this scenario the ***with('customers')*** function call is chaining to ***-> paginate(5)*** function call and are executed together with the nesting function call in the same execution block. Hybrid function calls on the other hand are those function calls that comprise more than one function call in one execution line of code e.g. ***return redirect()->route('customer.show', compact('customer')),*** in this scenario the ***redirect()*** function call which is a regular function call is chained and nested in the same execution line with ***->route('customer.show', compact('customer'))*** function call. Such function calls will be more complex compared to regular, nesting, and chaining function calls.

This implies that Laravel function call complexity increases from regular function call to nesting function call then to chaining function call finally the most complex is the hybrid function call. Therefore, when assigning weights based on the previous weight assignment criteria, the regular function calls will be assigned a weight of 1.0, nesting function calls will be assigned a weight of 2.0, chaining function calls will be assigned a weight of 2.5 and the hybrid function calls will be assigned a weight of 3 as shown in Table 2, based on existing weighting theories [14-16].

Table 2: Weights assigned to Laravel Function Calls

Types of Laravel Function Calls (FC_j)	**Weight Description**	**Corresponding Weights (W_j)**
Regular Function Call (R_{FC})	Weight of the Regular Function call (W_{RFC})	W_{RFC} = 1.0
Nesting Function Call (N_{FC})	Weight of the Nesting Function call (W_{NFC})	W_{NFC} = 2.0
Chaining Function Call (C_{FC})	Weight of the Chaining Function call (W_{CFC})	W_{CFC} = 2.5
Hybrid Function Call (H_{FC})	Weight of the Hybrid Function call (W_{HFC})	W_{HFC} = 3.0

Thus, when computing the derived metrics for the function call complexity, the individual function calls are computed and multiplied by the corresponding weights to give the LFC, as shown in Eq. 5.

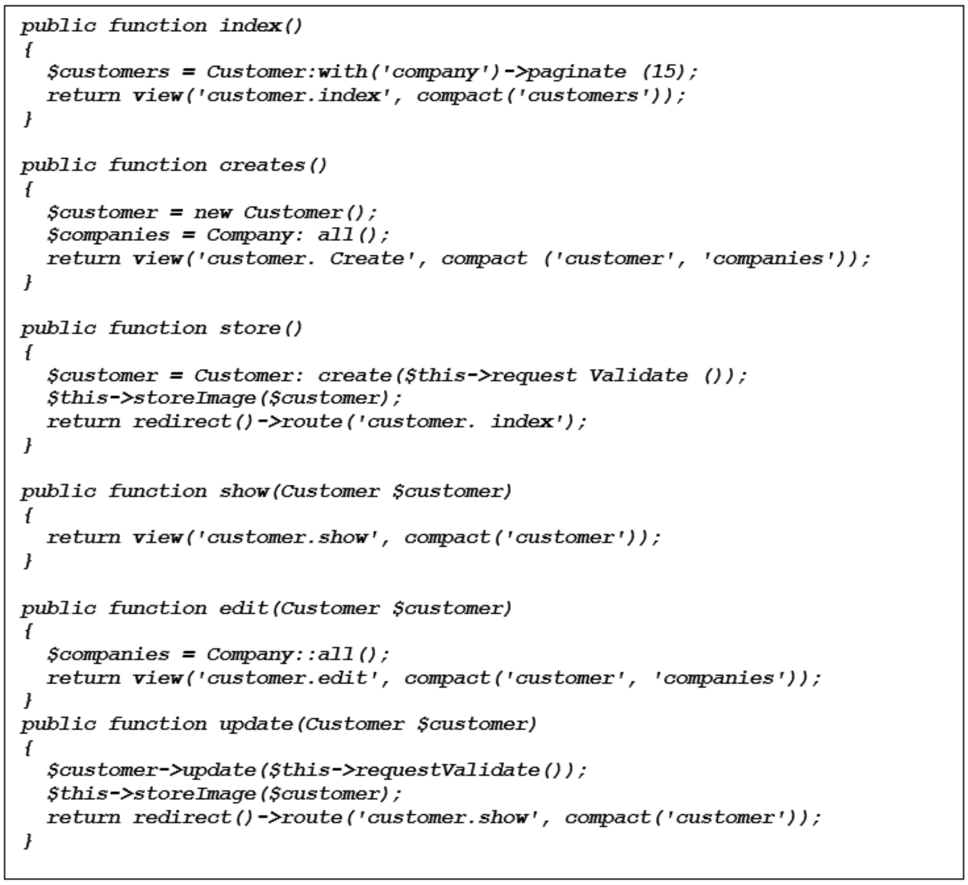
$$LFC = \sum_{j=1}^{n}(FC_j W_j) \ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{ Eq. (5)}$$

Where,

FC is the various types of Laravel function calls, *j* is the start of the first Laravel function call, W_j is the weight assigned to the corresponding various Laravel function calls and *n* is the last Laravel function call.

Operationalization of CCM$_{LV}$ Metric

The code snippet represented in Figure 1, helps to demonstrate how to operationalize the computation process of the metrics values for the CCM$_{LV}$ composite metric. This computation happens in two steps, with each step showing the computation of the metric values for the two derived metrics *LF* and *LFC* as shown*:*



```
public function index()
{
  $customers = Customer:with('company')->paginate (15);
  return view('customer.index', compact('customers'));
}

public function creates()
{
  $customer = new Customer();
  $companies = Company: all();
  return view('customer. Create', compact ('customer', 'companies'));
}

public function store()
{
  $customer = Customer: create($this->request Validate ());
  $this->storeImage($customer);
  return redirect()->route('customer. index');
}

public function show(Customer $customer)
{
  return view('customer.show', compact('customer'));
}

public function edit(Customer $customer)
{
  $companies = Company::all();
  return view('customer.edit', compact('customer', 'companies'));
}
public function update(Customer $customer)
{
  $customer->update($this->requestValidate());
  $this->storeImage($customer);
  return redirect()->route('customer.show', compact('customer'));
}
```

Figure 1: A Code Snippet Scenario to Compute CCM$_{LV}$

Calculating the Metrics values for CCM$_{LV}$

Step 1: Calculating the Metrics values for *LF*

From the code snippet in Figure 1;

The number of Parameterized Functions (P_f) = 3
The Weight of the Parameterized Function (W_{Pf}) = 1.5
The number of Non-Parameterized Functions (NP_f) = 3
The Weight of the Non-Parameterized Function (W_{NPf}) =1

Therefore, guided by Eq. (4);

$LF = \sum_{i=1}^{n}(F_i W_i)$
$= \sum_{i=1}^{n}\left(P_{f(i)} * W_{Pf(i)}\right) + \sum_{i=1}^{n}(NP_{f(i)} * W_{NPf(i)})$
$= \sum_{i=1}^{3}\left(P_{f(i)} * 1.5\right) + \sum_{i=1}^{3}(NP_{f(i)} * 1)$
$= (3 * 1.5) + (3 * 1)$
$= 4.5 + 3$
$= 7.5$

Step 2: Calculating the Metrics values for *LFC*

From the code snippet in Figure 1;

The number of Regular Function calls (R_{FC}) =3
The Weight of the Regular Function call (W_{RFC}) =1
The number of Nesting Function calls (N_{FC}) = 4
The Weight of the Nesting Function call (W_{NFC}) = 2
The number of Chaining Function calls (C_{FC}) = 4
The Weight of the Chaining Function call (W_{CFC}) = 2.5
The number of Hybrid Function calls (H_{FC}) = 3
The Weight of the Hybrid Function call (W_{HFC}) = 3

Therefore, following Eq. (5);

$LFC = \sum_{j=1}^{n}(FC_j W_j)$
$LFC = \sum_{j=1}^{n}\left(R_{FC(j)} * W_{RFC(j)}\right) + \sum_{j=1}^{n}\left(N_{FC(j)} * W_{NFC(j)}\right) + \sum_{j=1}^{n}\left(C_{FC(j)} * W_{CFC(j)}\right) + \sum_{j=1}^{n}\left(H_{FC(j)} * W_{HFC(j)}\right)$
$= \sum_{j=1}^{3}\left(R_{FC(j)} * 1\right) + \sum_{j=1}^{4}\left(N_{FC(j)} * 2\right) + \sum_{j=1}^{4}\left(C_{FC(j)} * 2.5\right) + \sum_{j=1}^{3}\left(H_{FC(j)} * 3\right)$
$= (3 * 1) + (4 * 2) + (4 * 2.5) + (3 * 3)$
$= 3 + 8 + 10 + 9$
$= 30$

Finally, as per Eq. (3), to get the metric values for the composite metric CCM$_{LV}$, the summation of both *LF* and *LFC* is done as shown;

$$\begin{aligned} \text{CCM}_{LV} &= LF + LFC \\ &= \sum_{i=1}^{n}(F_i W_i) + \sum_{j=1}^{n}(FC_j W_j) \\ &= 7.5 + 30 \\ &= 37.5 \end{aligned}$$

4.2. Model Complexity Metrics for Laravel (MCM$_{LV}$)

In Laravel, a model class majorly handles the operations in the Laravel database. It is part of Laravel's Eloquent Object-Relational Mapper (ORM) which provides an enjoyable and efficient way to interact with a database [17-19]. Models in Laravel are responsible for retrieving, storing, and processing data as they contain the logic related to the data being manipulated [13]. Model-based attributes are composed of entity relationships for manipulating database cardinalities and the array variables that execute the various functions in the development process [17, 18]. Therefore, as shown in Eq. 6, the composite metric MCM$_{LV}$ is calculated in two levels. The first level is the Laravel Array Variable Complexity Metrics (LAV) which is the complexity brought about by the Laravel array variables and the second one is the Laravel Entity Relationship Complexity Metrics (LER) which is the complexity contributed by the Laravel entity relationship Variables.

$$\begin{aligned} MCM_{LV} &= LAV + LER \text{ .... Eq. (6)} \\ &= \textstyle\sum_{i=1}^{n}(AV_i) + \sum_{i=1}^{n}(ER_i W_i) \end{aligned}$$

Where

LAV is Laravel Array Variable Complexity Metric
LER is Laravel Entity Relationship Complexity Metric

4.2.1. Laravel Array Variable Complexity Metrics (LAV)

Laravel adopts special array variables to perfume mass assignment of database fields [17-19]. Mass assignment in Laravel is a feature that allows one to assign multiple attributes to a model at once [20]. This is useful when saving data to the database, as it saves the programmer time and effort, for instance, *$user = new User(request()->all());* using this code-snippet, with a single push the programmer can mass assign multiple fields on a model. However, sometimes mass assignment can pose certain security risks, let's say the programmer has a field in the user table that can have values of "user" or "admin" [20-21]. To prevent this, Laravel provides three special array variables $attributes, $fillable, and $guarded. The $attributes [array items] array enables the programmer to set default values for the user logging into the system e.g. the programmer can set the user access right to "user" or "admin" during the login phase to avoid users accessing admin rights in the system. $fillable [array items] array variable contains all the attributes that should be mass assignable while $guarded [array items] array variable contains attributes that should not be mass assignable [20-21].

To compute the complexity metric for LAV, each array variable is counted as shown in Eq. 7:

$$LAV = \textstyle\sum_{i=1}^{n}(AV_i) \text{..................... Eq. (7)}$$

Where,

AV is the various types of Laravel Array variable, *i* is the start of the first Laravel Array variable and *n* is the last Laravel Array variable.

4.2.2. Laravel Entity Relationship Complexity Metrics (LER)

In Laravel, Database Entity Relationship is managed by ORM [21]. ORM supports a variety of unique entity relationships, such as; BelongsTo (*BT*), HasMany (*HM*), HasOneThrough (H_OT), and HasManyThrough (H_MT) [17-19].

Uncontrolled usage of these entity relationships increases the model complexity due to their interrelation. For instance, let us take a scenario of three models; Country, Team, and Athlete. The BT relationship shows the direct relation of a single model to another single model instance, hence can be assigned the simplest weight of 1.1 while the "*HM*" relationship shows that one model is related to many model instances, so this is a little bit complex presuming a weight of 1.3. In the scenario of the three models, the entity relationship can illustrate that a team can only belong to one country, which is represented using the "*BT*" relationship while a team has many athletes which is represented using the "*HM*" relationship. An athlete has one country through a team and this can be represented using a "H_OT" relationship making it more complex and hence can presume a weight of 1.5. Finally, you can also have a scenario where a country can have many athletes through a team, this is represented using "H_MT" relationships, hence the most complex of the four entity relationships with a weight of 2.0 as summarized in Table 3, based on existing weighting theories [14-16].

Table 3: Weights assigned to Laravel Entity Relationships

Types of Entity Relationships (ER$_i$)	**Weight Description**	**Corresponding Weights (W$_i$)**
BelongsTo relationship (BT)	Weight of the BelongsTo entity relationship (W_{BT})	W_{BT} = *1.1*
HasMany relationship (HM)	Weight of the HasMany entity relationship (W_{HM})	W_{HM} = *1.3*
HasOneThrough relationship (H_OT)	Weight of the HasOneThrough entity relationship (W_{HOT})	W_{HoT} = *1.5*
HasManyThrough relationship (H_MT)	Weight of the HasManyThrough entity relationship (W_{HMT})	W_{HmT} = *2.0*

Therefore, to calculate the complexity of LER, each entity relationship is counted and multiplied by individual corresponding weights as shown in Eq. 8:

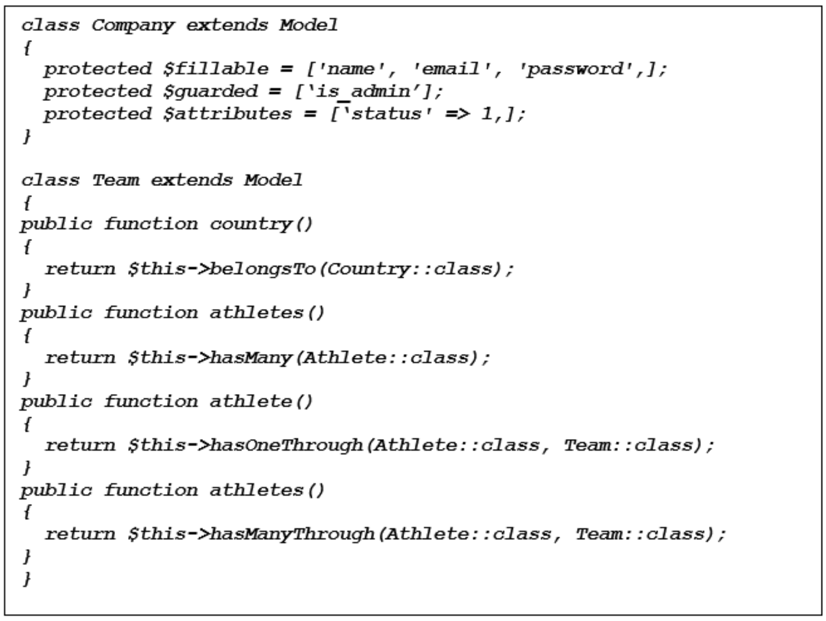
$$LER = \sum_{i=1}^{n}(ER_i W_i) \text{.....................} \text{ Eq. (8)}$$

Where,

ER is the various types of Laravel Entity Relationships, i is the start of the first Laravel Entity Relationship, W_e are the complexity weight assigned to the corresponding various types of Laravel Entity Relationships, and n is the last Laravel Entity Relationships.

Operationalization of MCM$_{LV}$ Metric

The code snippet represented in Figure 2, helps to demonstrate how to operationalize the computation process of the metrics values for the MCM$_{LV}$ composite metric. This computation happens in two steps, with each step showing the computation of the metric values for the two derived metrics *LAV* and *LER* as shown*:*



```
class Company extends Model
{
  protected $fillable = ['name', 'email', 'password',];
  protected $guarded = ['is_admin'];
  protected $attributes = ['status' => 1,];
}

class Team extends Model
{
public function country()
{
  return $this->belongsTo(Country::class);
}
public function athletes()
{
  return $this->hasMany(Athlete::class);
}
public function athlete()
{
  return $this->hasOneThrough(Athlete::class, Team::class);
}
public function athletes()
{
  return $this->hasManyThrough(Athlete::class, Team::class);
}
}
```

Figure 2: A Code Snippet Scenario to Compute Model Complexity Metrics for Laravel (MCM_{LV})

Calculating the Metrics values for MCM_{LV}

Step 1: Calculating the Metrics values for *LAV*

From the code snippet in Figure 2;

The number of $default array variable = 1
The number $fillable array variable = 1
The number $guarded array variable = 1

Therefore, following Eq. (7);

$LAV = \sum_{i=1}^{n}(AV_i)$

$= \sum_{i=1}^{n}(defaul\ array\ variable) + \sum_{i=1}^{n}(fillbale\ array\ variable) + \sum_{i=1}^{n}(guarded\ array\ varibale)$

$= \sum_{i=1}^{1}(1) + \sum_{i=1}^{1}(1) + \sum_{i=1}^{1}(1)$

$= (1) + (1) + (1)$

$= 3$

Step 2: Calculating the Metrics values for *LER*

From the code snippet in Figure 2;

The number of BelongsTo entity relationships = 1
The weight of the BelongsTo entity relationship = 1.1
The number of HasMany entity relationships = 1
The weight of the HasMany entity relationship = 1.3
The number of HasOneThrough entity relationships= 1
The weight of the HasOneThrough entity relationship = 1.5
The number of HasManyThrough entity relationships = 1
The weight of the HasManyThrough entity relationship = 2.0

Therefore, as per Eq. (8);

$$
\begin{aligned}
LER &= \sum_{i=1}^{n}(ER_i W_i) \\
&= \sum_{i=1}^{n}\left(BT_{(i)} * W_{BT(i)}\right) + \sum_{i=1}^{n}\left(HM_{(i)} * W_{HM(i)}\right) + \sum_{i=1}^{n}\left(H_O T_{(i)} * W_{HOT(i)}\right) + \sum_{i=1}^{n}\left(H_M T_{(i)} * W_{HMT(i)}\right) \\
&= \sum_{i=1}^{1}\left(BT_{(i)} * 1.1\right) + \sum_{i=1}^{1}\left(HM_{(i)} * 1.3\right) + \sum_{i=1}^{1}\left(H_O T_{(i)} * 1.5\right) + \sum_{i=1}^{1}\left(H_M T_{(i)} * 2.0\right) \\
&= (1 * 1.1) + (1 * 1.3) + (1 * 1.5) + (1 * 2.0) \\
&= 5.9
\end{aligned}
$$

Finally, as per Eq. (6), to get the metric values for the composite metric $\mathrm{MCM_{LV}}$, the summation of both *LAV* and *LER* is done as shown;

$$
\begin{aligned}
\mathrm{MCM_{LV}} &= \mathrm{LAV} + \mathrm{LER} \\
&= \sum_{i=1}^{n}(AV_i) + \sum_{i=1}^{n}(ER_i W_i) \\
&= 3 + 5.9 \\
&= 8.9
\end{aligned}
$$

4.3. View Complexity Metrics for Laravel ($\mathrm{VCM_{LV}}$)

In Laravel, a View is a class that handles the presentation logic of an application. It separates the controller application logic from the presentation logic to render output to the users [13]. The view makes use of blade directives to echo values and data to be displayed to the user. The directives in the Laravel View class are inherited from either Controller Class, Model class, or any other sections of the code, the inheritance happens hierarchically. For instance, @extends is a View blade directive that inherits and specifies a parent blade template from which the current template will inherit its layout at the outer level of the hierarchy, they can therefore be named as Level 1 Inheriting View Directives (*L1IVD*). It is like creating a base structure for a page and then filling in the specific parts of the contents later. @section on the other hand is a directive that is used to define a section of content being inherited. The contents of these @sections directives are injected into the layout defined by the @extends template inherited at *L1IVD*.

Finally, @include and @csrf are directives that are implemented at the inner level of the hierarchy within @section to include a blade view from another view and to generate security tokens to manage each active user session in the application respectively [22], these directives can be named as Level 2 Inheriting View Directives (*L2IVD*).

It implies that the @extends directive acts as a parent blade template layout, that inherits an external layout file [13, 22] at the *L1IVD*. The @section directive also defined at *L1IVD* specifies the content of the sections to be injected into the extended layouts, it defines the section of the layout that has been inherited by the @extends directive. These two view directives presume a weight of 1.3 since they work on externally inherited files. Then the other two directives i.e. @csrf and @include are executed at the *L2IVD* within the @section directive, hence, presume a weight of 1.5 as shown in Table 4, based on existing weighting theories [14-16].

Table 4: Weights assigned to Laravel View Directives

Types of Inheriting View Directives (IVD)	**Weight Description**	**Corresponding Weights (W)**
Level 1 Inheriting View Directives (*L1IVD*)	Weight of *L1IVD*	*1.3*
Level 2 Inheriting View Directives (*L2IVD*)	Weight of *L2IVD*	*1.5*

The VCM$_{LV}$ metric measures the complexity brought about by inheriting view template directives in Laravel software. The definition of this inheritance metric is borrowed from object-oriented software since it does not directly exist in Laravel [12]. To define VCM$_{LV}$, the complexity of individual inheriting view directives must be defined. This is done by considering the count of the *L1IVD* and *L2IVD* directives multiplied by their respective weights then a summation of all is done to obtain the overall composite metrics to yield the complexity of the Laravel View class as shown in Eq. 9.

Therefore,

$$VCM_{LV} = \sum_{i=1}^{n}(L1IVD_i * 1.3) + \sum_{j=1}^{n}\left(L2IVD_j * 1.5\right) \text{ ……………… Eq. (9)}$$

Where

$L1IVD$ = Level 1 Inheriting View Directives
$L2IVD$ = Level 2 Inheriting View Directives

Operationalization of VCM$_{LV}$ Metric

The code snippet represented in Figure 3, helps to demonstrate how to operationalize the computation process of the metrics values for the VCM$_{LV}$ composite metric. This computation happens in two steps, with each step showing the computation of the metric values for the two derived metrics *L1IVD* and *L2IVD* as shown below*:*

```
@extends('layouts.app')
@section ('title','Company List')
@section('content')
  <div class="row">
   <h1>Create Company</h1>
    <p class="text-muted">Fill in the details below to create a new company record</p>
     @csrf
     @include('company.form')
  <div class="d-grid">
  <button type="submit" class="btn btn-block btn-primary">Submit
Info</button>
  </div>
  </div>
@endsection
```

Figure 3: A Code Snippet Scenario to Compute View Complexity Metrics for Laravel (VCM$_{LV}$)

Calculating the Metrics values for VCM$_{LV}$

From the code snippet in Figure 3;

The number of Level 1 Inheriting View Directives = 3
The weight of the Level 1 Inheriting View Directives = 1.3
The number of Level 2 Inheriting View Directives = 2
The weight of the Level 2 Inheriting View Directives = 1.5

Therefore, as following Eq. (9);

$$
\begin{aligned}
\mathrm{VCM}_{LV} &= \textstyle\sum_{i=1}^{n}(\mathrm{L1IVD}_i * 1.3) + \sum_{j=1}^{n}(\mathrm{L2IVD}_j * 1.5) \\
&= \textstyle\sum_{i=1}^{3}(@extends * 1.3 + @section * 1.3 + @section * 1.3) + \sum_{j=1}^{2}(@\mathrm{csrf} * 1.5 + @\mathrm{include} * 1.5) \\
&= (3 * 1.3) + (2 * 1.5) \\
&= (3.9) + (3.0) \\
&= 6.9
\end{aligned}
$$

5. Validation of the Proposed Metrics

In software measurement, each newly defined metric must be validated either internally or externally [3, 5, 23-24]. Internal validation is the theoretical validation to assess the mathematical soundness of the defined metrics. External validation, on the other hand, involves the empirical study of the software metrics, it shows how the metrics can be illustrated using a real-world scenario for intuition, and it's always a complement to theoretical validation [25-26]. However, both validations are necessary for a newly defined metric, these approaches are illustrated in Figure 4.

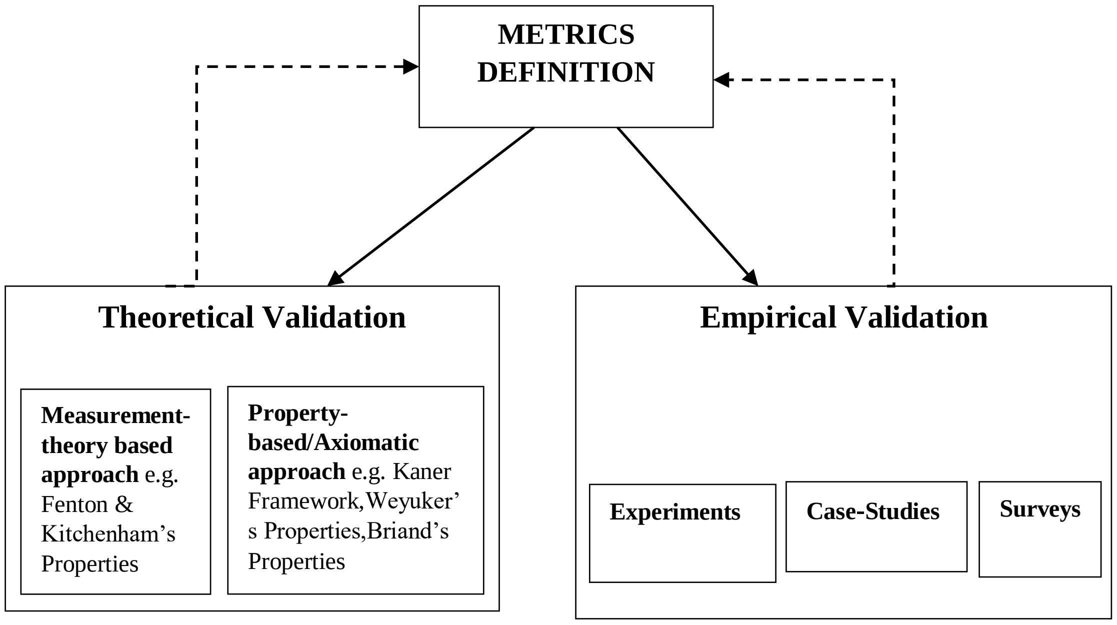

METRICS DEFINITION
Theoretical Validation
Measurement-theory based approach e.g. Fenton & Kitchenham's Properties
Property-based/Axiomatic approach e.g. Kaner Framework,Weyuker's Properties,Briand's Properties
Empirical Validation
Experiments
Case-Studies
Surveys

Figure 4: Software Metrics Validation Approaches

This process is recommended for any newly defined metrics as they ascertain their mathematical soundness and practicability. Theoretical validation also is seen as a crucial process because it provides a scientific basis for the discipline of software measurement, without it, there would be no confidence in the consumption of the newly defined metrics from the software engineering realm [27-29].

In this study, the three composite metrics newly defined are purposed to measure the inherent attributes that cause the complexity of software products developed using Laravel that might negatively affect the modifiability of such products. Therefore, the three newly proposed metrics $\mathrm{CCM}_{\mathrm{LV}}$, $\mathrm{MCM}_{\mathrm{LV}}$, and $\mathrm{VCM}_{\mathrm{LV}}$ are validated to ascertain their practicality using the Kaner framework, their mathematical soundness was also attested using Weyuker's nine properties as summarized in Table 5.

5.1. Theoretical Validation using Weyuker's Nine Properties

"Property 1: (∃P) (∃Q) (|P| ≠ |Q|) Non-coarseness:" There exist Laravel projects Q and P in a way that |Q| is not equivalent to |P|. The property implies that a metric is not supposed to rank each Laravel project with equivalent complexity if they are not identical. These defined metrics return non-identical complexity values for any two different Laravel projects. Therefore, all three proposed metrics satisfy this property.

"Property 2: Granularity:" If C is a positive number. Then there are a finite number of Laravel software of complexity C. This property states that a change in a Laravel project must also lead to a change in its complexity. Therefore, the three proposed metrics CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} satisfy this property.

"Property 3: Non-uniqueness (Notion of Equivalence):" There can exist two distinct Laravel projects Q and P where |P| = |Q|. The property states that two different Laravel projects can have the same metric value if they have the same attributes. Thus, this property holds for all the defined metrics.

"Property 4: (∃P) (∃Q) (P ≡ Q & |P| ≠ |Q|) Design Details are Important:" There exist two Laravel projects P and Q in a way that the external effects of P and Q are similar, but |P| is unequal to |Q|. This property indicates that two Laravel projects P and Q could look identical in terms of the fact that they contain the same number of class attributes, but could have different complexities if the types of these attributes are different, this is because the attributes are assigned different weights. Therefore, the three proposed metrics namely CCM_{LV}, $MCM_{LV,}$ and VCM_{LV}satisfy this property.

"Property 5: (∃P) (∃Q) (|P| ≤ |P; Q| & (|Q| ≤ |P; Q|) Monotonicity:" This property states that if two Laravel projects P and Q are concatenated, then the resulting metric value shall be greater than or equal to the individual Laravel project. Therefore, all three proposed metrics satisfy this property.

"Property 6: (∃P) (∃Q) (∃R) (|P| =|Q| and |P; R| ≠ |Q; R|) Nonequivalence of interaction:" There are similar Laravel projects P, R, and Q in a way that |P| is equivalent to |Q| however |P; R| is unequal to |Q; R|. This indicates that two similar Laravel projects can exist, but if presented to a third code in the same program, their proceeding complexities are different. This indicates that the action of combining two projects has the capability of initiating complexity additional to that inherent in real projects. Also, this newly incorporated complexity is not fully discovered by any of the interacting projects. All the defined metrics consider the physical measurable attributes of Laravel projects by assigning them fixed values and weights. Due to the existence of these constant values, anytime two Laravel projects are sequenced, it is impossible to introduce external complexity. Therefore, all the proposed metrics fail to satisfy this property.

Property 7: Permutation. There exist two Laravel projects P and Q which have the same number and type of attributes in a permuted order, then |P| is not equal to |Q|. This property implies that the order of similar attributes affects their complexity. For instance, if two Laravel projects have the same number and types of attributes but differ in ordering, then they don't need to have the same complexity level. Therefore, property 7 does not apply to all the metrics defined.

"Property 8: Renaming:" If P is a renaming of Q, then |P| = |Q|. If a Laravel project P is a renaming of a Laravel project Q, then the complexity of Laravel project P should be equal to the complexity of project Q (|P| is equal to |Q|). Therefore, all proposed metrics satisfied this property.

"Property 9: (∃P) (∃Q) (|P| +|Q| < (|P; Q|) Interaction Increases Complexity:" There exist Laravel projects Q and P where |P|+|Q| is less than |P; Q|. This attribute argues that the interrelation between sections of a project causes additional positive complexity. When a Laravel project is modified by introducing new class attributes, the complexity values of the new Laravel project will be higher than the original project. All three proposed metrics satisfy property 9.

Table 5: Metrics Validation using Weyuker's nine properties

Property	CCM_{LV}	MCM_{LV}	VCM_{LV}
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	✓
4	✓	✓	✓
5	✓	✓	✓
6	✕	✕	✕
7	✕	✕	✕
8	✓	✓	✓
9	✓	✓	✓

Key

✓ = Satisfying Property ✕ = not satisfying property

5.2. Validation using Kaner Framework

Kaner proposed an eleven-question framework to validate the practicality of any newly proposed metrics [27]. Therefore, the newly proposed Laravel metrics are subjected to the Kaner framework to ascertain their practicality.

i. "What is the purpose of this measure?"

The purpose of the three newly proposed metrics CCM_{LV}, MCM_{LV}, and VCM_{LV} is to measure and help in the evaluation of the complexity of software developed using Laravel.

ii. "What is the scope of this measure?"

The three newly proposed metrics CCM_{LV}, MCM_{LV}, and VCM_{LV} focus on measuring the complexity of software developed using Laravel at the class level.

iii. "What attribute are we trying to measure?"

The defined metrics are classified under controller-based attributes, view-based attributes, and model-based attributes. These attributes measure the complexity of Laravel software which has a direct contribution to the modifiability of such software products.

iv. "What is the natural scale of the attribute we are trying to measure?"

The ordinal scale is used as the natural scale of measure for the defined attributes.

v. "What is the natural variability of the attribute?"

Different Laravel software will have attributes that naturally vary from one software to another, hence the quality attributes are subjective.

vi. "Metrics definition"

All the three newly proposed metrics CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} have been defined clearly following the known measurement theory.

vii. "What is the metric and what measuring instrument do we use to perform the measurement?"

Three novel metrics CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} are defined to manually compute the complexity of Laravel software. Besides, a static metrics tool will be developed as a measurement instrument in the future to automate the metrics computation process.

viii. "What is the natural scale for this metric?"

All the three proposed metrics give metric values on a numerical natural scale.

ix. "What is the natural variability of readings from this instrument?"

The instrument that will be used to automate the metrics computation process will be validated to ensure that there is no natural variability of the readings.

x. "What is the relationship of the attribute to the metric value?"

All the three newly proposed metrics CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} are defined based on the identified attributes. Meaning that the metric values have a direct relationship and are directly proportional to the metric values.

xi. ***"What are the natural and foreseeable side effects of using this instrument?"***

The instrument is a static analyzer tool to be used in the automation of the metric computation process. It does not have any natural and foreseeable side effects.

6. DISCUSSION

Three novel structural complexity metrics namely CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} were defined and validated theoretically. These metrics are designed to assess the complexity of Laravel software at the class level within Laravel's MVC architecture. The metrics offer a better approach to understanding and managing software complexity in Laravel projects. The definition and validation of the metrics tailored for Laravel applications present a substantive contribution to the software complexity metrics field.

The theoretical validation of these metrics, employed Weyuker's nine properties and the Kaner framework, indicated a substantial alignment with established properties of mathematical soundness and practicability of the complexity metrics respectively. Notably, the metrics showed a high degree of compliance satisfying seven out of nine of Weyuker's properties. This implies that the proposed metrics are mathematically sound, robust, and reliable in measuring the complexity of Laravel software. The failure to comply with properties six and seven can be attributed to the specialized nature of Laravel's architectural elements, which require a more tailored approach to complexity assessment.

Moreover, the validation of the metrics against the Kaner framework further emphasizes their practical applicability and relevance to real-world software development scenarios. The comprehensive nature of the Kaner framework's questions implied that the metrics are not only theoretically sound but also practically applicable to the Laravel domain.

These results, therefore, imply that all the three newly proposed metrics CCM_{LV}, $MCM_{LV,}$ and VCM_{LV} present a significant contribution to the domain of software complexity metrics and can be adopted by Laravel developers to measure and therefore, control the complexity of Laravel software.

7. CONCLUSIONS AND FUTURE WORK

In conclusion, this study highlights the importance of developing framework-specific complexity metrics that consider the unique features and architectural patterns of modern web development frameworks. By focusing on Laravel, a widely used PHP development framework, the study addresses a critical gap in the literature and practice of software engineering metrics. The proposed metrics provide a valuable measure for developers and project managers to identify potential complexities, inform refactoring decisions, and ultimately improve the modifiability and quality of Laravel applications.

The study lays a robust foundation for measuring complexity in Laravel applications, several avenues for future research emerge, such as tool development; the development of automated tools based on these metrics for static code analysis would significantly enhance their applicability in the industry, allowing for real-time complexity assessment during development. Empirical validation; in the future, the study recommends conducting empirical studies to validate the proposed metrics against real-world Laravel projects to further establish their usefulness and effectiveness in practical scenarios.

REFERENCES

[1] N. Yadav, D. S. Rajpoot & S. K. Dhakad, "LARAVEL: A PHP Framework for E-Commerce Website," 2019 Fifth International Conference on Image Information Processing (ICIIP), Nov. 2019, doi: https://doi.org/10.1109/iciip47207.2019.8985771.

[2] S. Tenzin, "PHP Framework for Web Application Development," IARJSET, vol. 9, no. 2, Feb. 2022, doi: https://doi.org/10.17148/iarjset.2022.9218.

[3] J. G. Ndia, "Structural Complexity Framework and Metrics for Analyzing the Maintainability of Sassy Cascading Style Sheets" (Doctoral dissertation, MMUST), 2019.

[4] J. G. Ndia, G. M. Muketha & K. K. Omieno, "Structural Complexity Attribute Classification Framework (SCACF) for Sassy Cascading Style Sheets", International Journal of Software Engineering & Applications (IJSEA), Vol.11, No.1, DOI: 10.5121/ijsea.2020.11105, January 2020.

[5] G. M. Muketha, A. A. A. Ghani, M. H., Selamat & R. Atan, "Complexity Metrics for Executable Business Processes", Information Technology Journal. 9(7), 2010a, 1317-1326.

[6] J. Estdale & E. Georgiadou, "Applying the ISO/IEC 25010 quality models to software products," In Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25 (pp. 492-503). Springer International Publishing.

[7] H. Panduwiyasa, M. Saputra, Z. F. Azzahra & A. R. Aniko, “Accounting and smart system: functional evaluation of iso/iec 25010: 2011 quality model (a case study)”, In IOP Conference Series: Materials Science and Engineering (Vol. 1092, No. 1, p. 012065). IOP Publishing, March, 2021.

[8] H. Zhang & M. A. Babar, "On the complexity of Laravel application models: A framework-specific metrics analysis," Proceedings of the 5th International Workshop on Software Quality and Maintainability, 2011.

[9] T. J. McCabe, "A complexity measures." IEEE Transactions on Software Engineering, SE-2(4), 308-320. This seminal paper introduced Cyclomatic complexity, a foundational metric for understanding control flow complexity in software systems, 1976.

[10] M. H. Halstead, “Elements of Software Science”. Elsevier North-Holland, Inc., 1977.

[11] S. Henry & D. Kafura, "Software structure metrics based on information flow." IEEE Transactions on Software Engineering, SE-7(5), 1981, 510-518.

[12] S. R. Chidamber & C. F. Kemerer, "A metrics suite for object-oriented design." IEEE Transactions on Software Engineering, 20(6), 1994, 476-493.

[13] “Laravel - The PHP Framework for Web Artisans,” laravel.com. https://laravel.com/docs/10.x/eloquent

[14] Y. Wang & J. Shao, “A new measure of software complexity based on cognitive Weights.” *IEEE Canadian Journal of Electrical and Computer Engineering*, 2003, pp. 69-74

[15] C. J. Selvaraj, A. Aloysius & L. Arockiam, “A Comparision of Proposed Cognitive weights for control structures and Object-Oriented programming languages,” In Proceedings of International Conference on Advanced Computing ICAC09, 2009 (pp. 380-385).

[16] U. Chhillar & S. Bhasin, “A new weighted composite complexity measure for object-oriented systems, ”International journal of information and communication technology research, 1(3), 2011.

[17] “Laravel Models | Defining Eloquent Models with CRUD Operations, ”EDUCBA, Jan. 09, 2020. https://www.educba.com/laravel-models/, accessed Jul. 30, 2024.

[18] “Illuminate\Database\Eloquent\Model | Laravel API, ”laravel.com. https://laravel.com/api/7.x/Illuminate/Database/Eloquent/Model.html, accessed Jul. 30, 2024.

[19] “Understanding Laravel Models: A Comprehensive Guide,” www.gyata.ai. https://www.gyata.ai/laravel/laravel-model/, accessed Jul. 30, 2024.

[20] Admin, “What Does ‘Mass Assignment’ Mean in Laravel? - LaravelTuts,” Laravel Tuts, Jun. 12, 2023. https://laraveltuts.com/what-does-mass-assignment-mean-in-laravel/, accessed Jul. 30, 2024.

[21] “What does ‘Mass Assignment’ mean in Laravel?, ”Stack Overflow. https://stackoverflow.com/questions/22279435/what-does-mass-assignment-mean-in-laravel, accessed Mar. 11, 2024.

[22] “Laravel @extends and @include,” Stack Overflow. https://stackoverflow.com/questions/39749683/laravel-extends-and-include , accessed Jul. 30, 2024.

[23] D. Soni, R. Shrivastava & M. Kumar, “A Framework for Validation of Object-Oriented Design Metrics,” International Journal of Computer Science and Information Security (IJCSIS), vol. 6, no. 3, 2009, Accessed: Jul. 30, 2024. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1001/1001.1970.pdf

[24] N. Fenton, "Software Metrics: Theory, Tools and Validation," Software Engineering Journal, pp. 65-78, January 1990.

[25] K. P. Srinivasan & T. Devi, “Software Metrics Validation Methodologies in Software Engineering, ”International Journal of Software Engineering & Applications, vol. 5, no. 6, pp. 87–102, Nov. 2014, doi: https://doi.org/10.5121/ijsea.2014.5606.

[26] B. Smith, “Software metrics validation criteria: a systematic literature review,” North Carolina State University Department of Computer Science, Raleigh, NC, Jan. 2010, Accessed: Jul. 30, 2024. [Online]. Available: https://www.academia.edu/94649556/Software_metrics_validation_criteria_a_systematic_literature_review.

[27] W. Bond, “Software Engineering Metrics: What Do They Measure and How Do We Know?” Accessed: Jul. 30, 2024. [Online]. Available: https://kaner.com/pdfs/metrics2004.pdf

[28] E. J. Weyuker, “Evaluating software complexity measures,: IEEE Transactions on Software on Software Engineering, 1988, 14: 1357-1365.

[29] A. W. King'ori, G. M. Muketha & J.G. Ndia, "A SUITE OF METRICS FOR UML BEHAVIORAL DIAGRAMS BASED ON COMPLEXITY PERSPECTIVES," International Journal of Software Engineering & Applications (IJSEA), vol. 15, no. 2, 2024, doi: https://doi.org/10.5121/ijsea.2024.15201.

AUTHORS

Kevin Agina Onyango is a Tutorial Fellow in the Department of Information Technology, School of Computing and Information Technology, Murang'a University of Technology. He received his BSc. and MSc. in Information Technology from Murang'a University of Technology, Kenya. He is currently pursuing his PhD Information Technology at Murang'a University of Technology. His research interests mainly include software engineering, software measurement, program analysis, and soft computing. He is a Student Professional Member of the Institute of Electrical and Electronics Engineers (IEEE).

Geoffrey Muchiri Muketha is a Professor of Computer Science and Director of Postgraduate Studies at Murang'a University of Technology, Kenya. He received his BSc. in Information Sciences from Moi University, Kenya in 1995, his MSc. in Computer Science from Periyar University, India in 2004, and his PhD in Software Engineering from Universiti Putra Malaysia in 2011. He has wide experience in teaching and supervision of postgraduate students. His research interests include software and business process metrics, software quality, verification and validation, empirical methods in software engineering, and computer security. He is a member of the International Association of Engineers (IAENG).

John Gichuki Ndia is a Senior Lecturer at the Department of Information Technology and the Dean School of Computing and Information Technology, Murang'a University of Technology, Kenya. He earned his Bachelor of Information Technology from Busoga University in 2009, and his MSc. in Data Communications from KCA-University in 2013. He pursued his PhD in Information Technology at Masinde Muliro University of Science and Technology in 2020. His research interests include Software quality, artificial intelligence applications in software engineering, and computer network security. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM)