Best Practices and Tradeoffs in Microservice Design

Andy Cheong, Christopher Teubert

Team 1

COEN 241: Introduction to Cloud Computing

Term Project, Spring 2018

Santa Clara University

Preface

This document serves as the official project deliverable for COEN 241 Team 1's term-project. This document described the scope, intentions, work, and results for Team 1's term project for the intended audience, Professor Ming-Hwa Wang. The authors hope that, upon reviewing this document, Professor Wang will have a detailed understanding of the conducted work.

Acknowledgements

The authors of this proposal thank Professor Ming-Hwa Wang for his guidance and support during the formation of this project. They further thank him for his flexibility in allowing one of the authors to present his portion of the proposal remotely. Without this flexibility, he would not be able to complete his studies while working full-time.

Table of Contents

Table of Tables

Table of Figures

1. Introduction

This chapter describes the introduction for the project. It starts with the project objective in Section 1.1. Section 1.2 defines the problem that microservices have. Then in Section 1.3, it describes how this topic has relevance to cloud computing. Section 1.4 describes the shortcomings of previous solutions and helps transition to Section 1.5 which defines the strengths of the approach of this paper. The problem statement states the issue that needs to be addressed and it defined in Section 1.6 and in Section 1.7 the paper will describe the area or scope of investigation that will solve the problem statement.

1.1. Objective

The objective of this paper is to design a quality model in microservices that achieves service reusability, flexibility, and maintainability. The design will provide a standard that others will be able to follow when designing microservices so that it can provide the best quality in use.

1. Service Reusability
    a. Improves productivity
2. Service Flexibility
    a. Tool independent
3. Maintainability
    a. Performance, availability, reliability

1.2. Problem

This project is about designing a quality model standard for microservices. Currently there is no universal standard for quality in microservices when it comes to designing them. This issue allows anyone to design a microservice to their own discretion without abading to any quality standards so each microservice differ in quality. This mean that their service reusability, flexibility, and maintainability could be compromised. A quality model standard will allow all microservices to have the same quality.

1.3. Relevance

The emergence of cloud computing and its surrounding technology has dictated the business models of many organization to adopt the technology to survive in the cloud era. Microservices allows organizations to quickly deploy products into the market through continuous delivery and internal development environments for employers to deliver projects. It is important for microservices to be designed correctly so it can achieve its use.

1.4. Shortcomings of Previous Solutions

The reason the team is creating a quality model standard for microservices is because currently there aren't any. Through rigorous research to find an acceptable quality model, we have found several quality models but we have determined that they do not meet the standard for quality assurance.The shortcomings of other solutions are that they only cover a certain aspect of quality or they are not targeted towards microservices.

Previous solutions in quality model

- Bogner Maintainability Quality Model
- Wen Model
- McCall Model
- Boehm Model
- Dromey Model
- ISO 25010 Model

Section 2.3 will describe in more depth the above solutions and show their advantages and disadvantages.

1.5. Strengths of Approach

This paper will target different microservices in production and experimental and determine the best practices based on their implementation. The strength of this approach is the ability to look at how other teams have used microservices and determine where they have exceeded or failed in providing a quality model. By looking at different microservices it will create trends in quality and will help create a set of best practices.

1.6. Problem Statement

In today's world almost everyone has been impacted by cloud computing either by using it as data storage, instant access of data from anywhere and the use of applications. This has been made possible with the growth in technology and internet access including microservices. This high impact that cloud computing has made on the world, it is important that all microservices provides high quality service to its users.

There are so many microservices out in production that quality has been compromised in at least one product. As people continue to design and develop more microservices quality will keep diminishing and affect the integrity of the products.

1.7. Investigation Scope

This paper will be looking into different microservices that are currently in production and determine where each microservice fails or exceeds in providing quality in service reusability, flexibility, and maintainability.

In conclusion, this chapter described the objective of the paper and the problem that microservices has. It also described the relevance it has to cloud computing. Then this chapter described the shortcomings of previous solutions, defined the strengths of the new approach and stated the issue this paper is addressing through the problem statement. Lastly, it described the area or scope of investigation that will solve the issue in the problem statement. The next chapter describes in depth the problem and background of the topic, and the proposed solution.

2. Theoretical Basis

This chapter describes the theoretical basis for the project. It starts with a definition of the problem (Section 2.1). Section 2.2 defines the theoretical background for the work. Related Research and the advantages and disadvantages of each work are described in Sections 2.3 and 2.4, respectively. The following sections describe the author's solution, what makes it unique, and its strengths.

2.1. Problem Definition

Microservices is a type of service oriented architecture in which the application is composed of loosely coupled services. Based on its definition a microservice needs to be small, independent processes that communicate with each other using language-agnostic APIs. This approach lets microservices to be modular, reusable and flexible.

The figure below, Figure 2.1.1, shows a general microservice architecture.

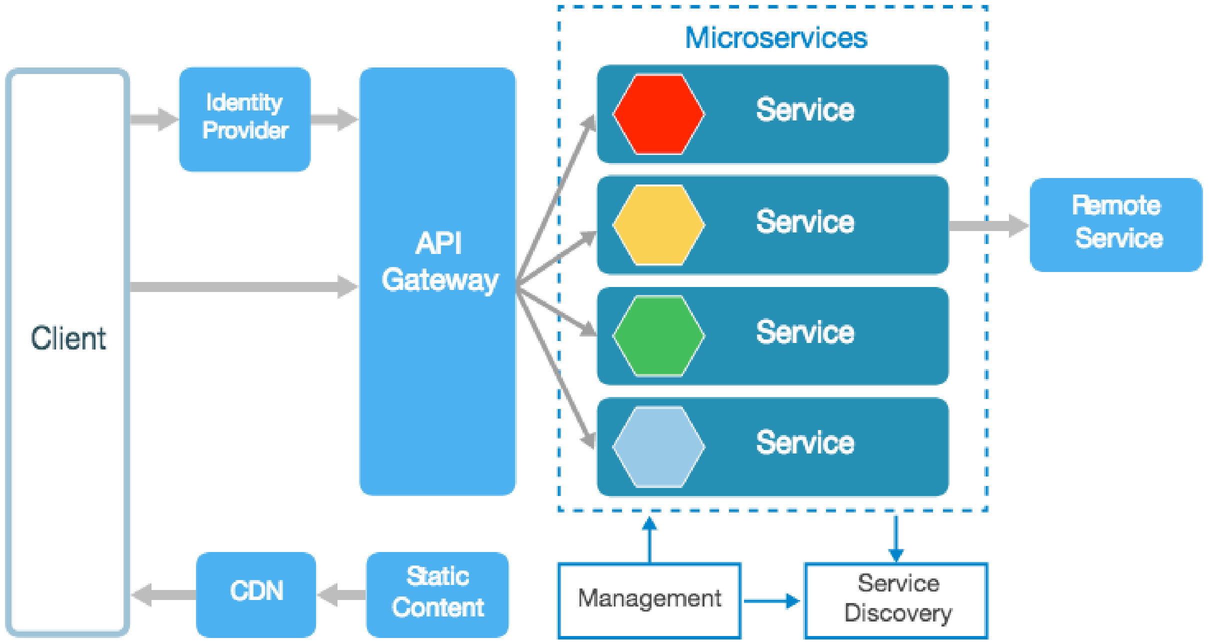


Figure 2.1.1: General Microservice Architecture (Wasson, Celarier, 2017)

The basic patterns in microservices that created modularity, reusability, and flexibility are:

Location independence pattern

- Other components can discover it within a directory and leverage it through the late binding process
- Dynamic Discovery

Communications independence pattern

- All components can talk to each other, no matter how they communicate at the interface or protocol levels

Security independence pattern

- Trust the security of each component

Instance independence patterns

- Support component-to-component communications using both a synchronous and an asynchronous model
- Not require that the other component be in any particular state before receiving the request or message

There are many use cases for microservices and this section will describe two of the main uses. The first use case of microservices is inside an organization and the second use case is in a client-server model.

In the first use case, microservices is be used to assist DevOps in organization, increase productivity for developers in product development and quality for customers.

DevOps can use microservices can be used to quickly deploy systems and development environment for teams in agile development. This use of microservices is done through container technology and it minimizes environment issues as all the development environments would be identical and contains the correct versions of tools and software. A typical container would be a virtual machine with an operating system installed, development IDE, tools and languages that can be deployed for an employer when needed for development.

The figure below, figure 2.1.2, shows a container example of what it contains for a development environment that DevOps can deploy quickly.

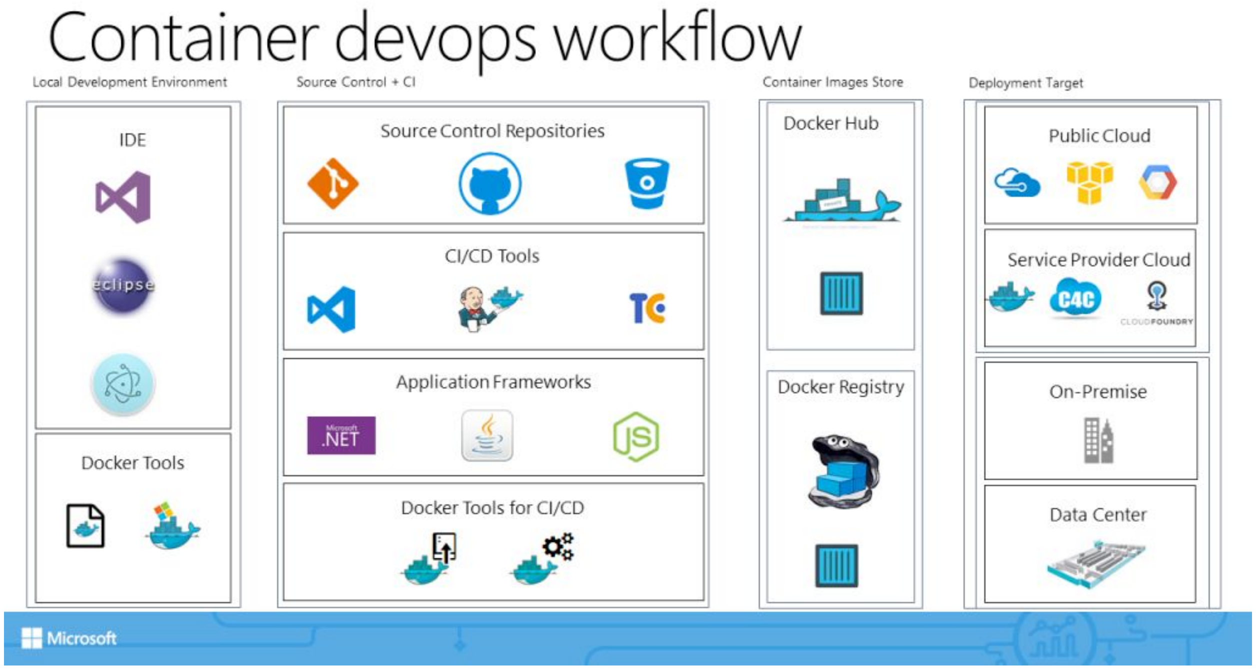


Figure 2.1.2: Container Example (Kshirsagar, 2016)

The second use of microservices is in a client-server model where the user can access information based on the application that they are using. The client would be a user interface that mainly displays information in a user friendly manner and the server would be doing all the computational work to provide the requested information from the client. This can be achieved through an RESTful API (Representational State Transfer) where the client sends requests to a server, the server does the work requested by the client and sends back the results to the client. To create these type of products, developers create application programming interface (API) which then they can develop the product with it or even provide the API to the public to allow external developers to use the API and develop their own products.

The figure below, figure 2.1.3, shows an architectural example on how a RESTful API is designed to follow the client-server model for users

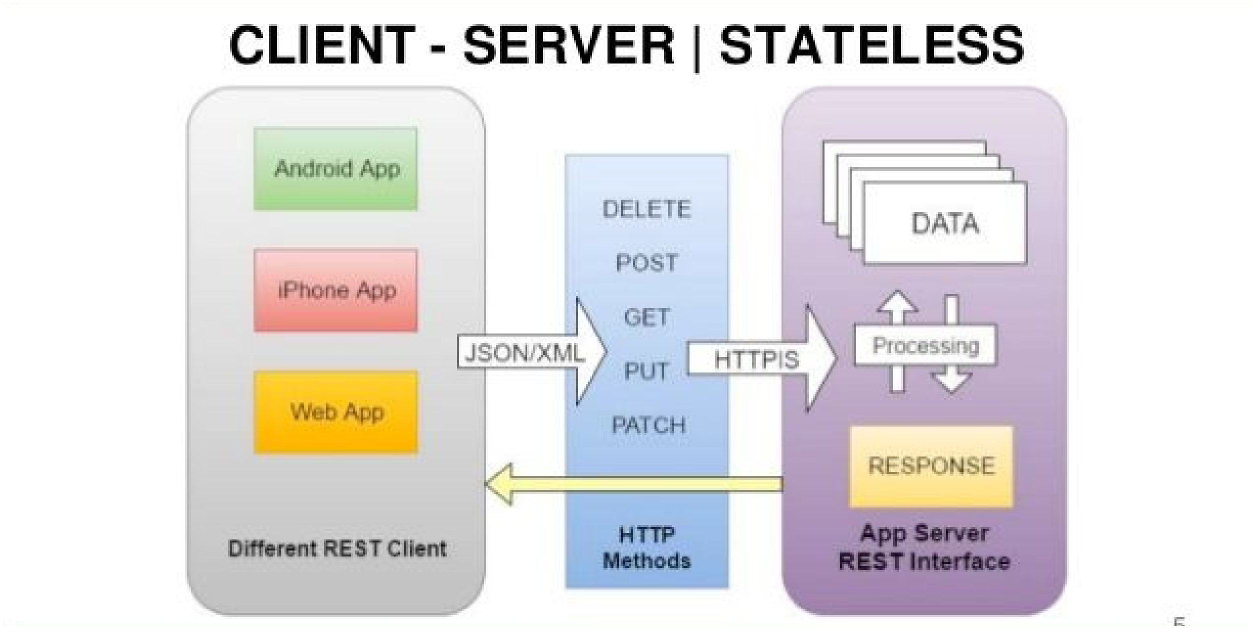


Figure 2.1.3: RESTful API Architecture (Kryvtsov, 2016)

It is important for microservices to be designed correctly the use cases affects both organizations and users. Microservices in a cloud environment allows the client to be lightweight since the server is doing all the computational work. An organization can have a broader customer base with a microservice product as this allows the customers initial investment to be minimal since all they would need is a device with a screen and has access to the internet. Customers want to be able to access data and use the product instantaneously so it is important the the microservice is implemented correctly.

For developers microservices allows a product to be modular and allows continuous delivery since microservices should be independent components.

Key factors in microservice design

- Driven by business need or capability
  - The business need will determine the functionality of the microservice.
- Size of application
  - Determines the scalability of the microservice.
- Size of development team
  - Determines the feasibility of completing the product.
- Database design
  - Determines the optimization and quality of the microservice.
- Reuse
  - Create productivity and efficiency in development.

2.2. Theoretical Background

IEEE's Draft Standard for Software Quality Assurance Processes defines software quality as "the degree to which a software product meets established requirements; however, quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations" (IEEE, 2012) and the IEEE Glossary of Terms defined quality as "the degree to which a system, component, or process meets the specified requirements" (IEEE Standards Coordinating Committee, 1999). This definition of quality is highly personal to the application for which the software is developed.

There are, however, a set of non-functional requirements (NFR's) that is common between most software. There is a larger and more personalized set of NFR's that are common between microservices. These NFR's often cannot be directly tested (unlike most functional requirements), so, in its place, a set of metrics must be established that assess the degree to which that NFR is met to a reasonable degree. These common NFR's and metrics define a quality model.

2.3. Related Research

There have been a number of other efforts to define quality models for software in general and some limited efforts to define quality models for services. Some of this related research is summarized in Table 2.3.1. Some key points of these papers are highlighted in greater detail below the table.

Table 2.3.1: Related Research

Related Research	**Summary**
Bogner Maintainability Quality Model (Bogner, 2017)	A service software quality model towards the non-functional requirement of maintainability. The authors present a layered logical-decomposition based model with quantifiable metrics.
Wen Quality Model (Wen, 2013), (Banerjee, 2014)	A general software quality model for Software As-A-Service (SaaS) applications. The author's software quality model provides multiple discrete levels of quality, and relates it back to the classification of the SaaS
McCall Model (McCall, 1977)	A general software quality model, decomposed from three perspectives, into features and metrics. Quality is assessed through answering yes/no questions.

Boehm Model (Boehm, 1978)	Highly-cited software quality model where high-level factors are decomposed from the concept of quality.
Dromey Model (Dromey, 1995)	A theoretical perspective-based general multi-layer software quality model
ISO 25011 Model (ISO, 2011)	This standard describes the product quality “characteristics” (i.e., NFRs). These characteristics are divided into “sub-characteristics”.

Bogner Model

Bogner, et al. (Bogner, 2017) present a service software quality model towards the non-functional requirement of maintainability. The authors present a layered logical-decomposition based model with quantifiable metrics. Their quality model is based on past research and professional experience. The final quality model is included below in Figure 2.3.1.

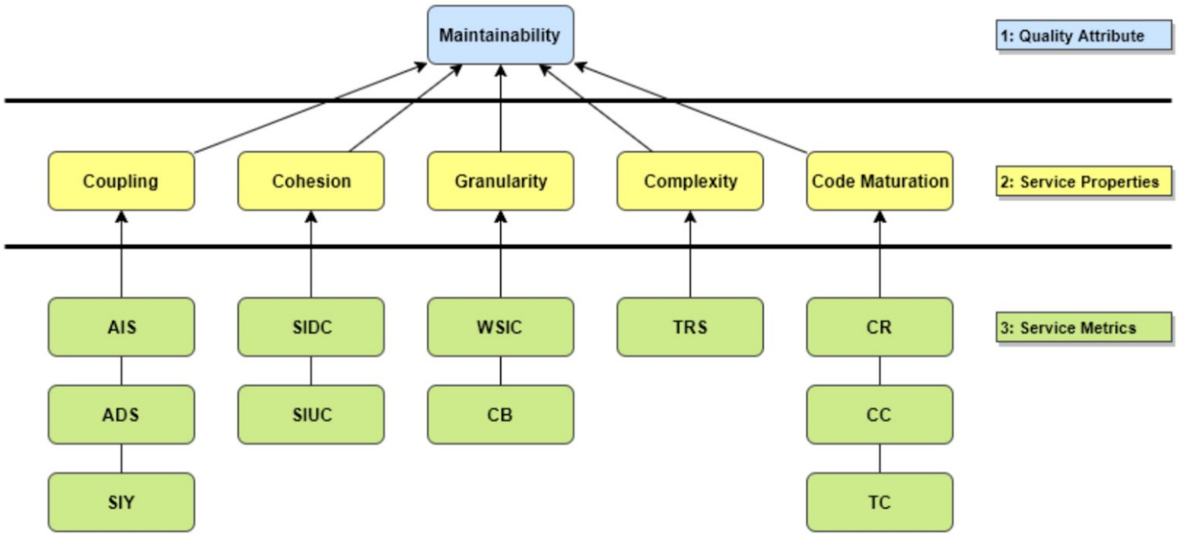


Figure 2.3.1: Maintainability Model for Services (Bogner, 2017)

Wen Model

Wen, et al. (Wen, 2013) present a general software quality model for Software As-A-Service (SaaS) applications. The author’s software quality model divides SaaS services into four discrete levels: Basic SaaS, Standard SaaS, Optimized SaaS, and Integrated SaaS. Each of these SaaS level has a set of quality metrics that it must meet. This model is shown in Figure 2.3.2.

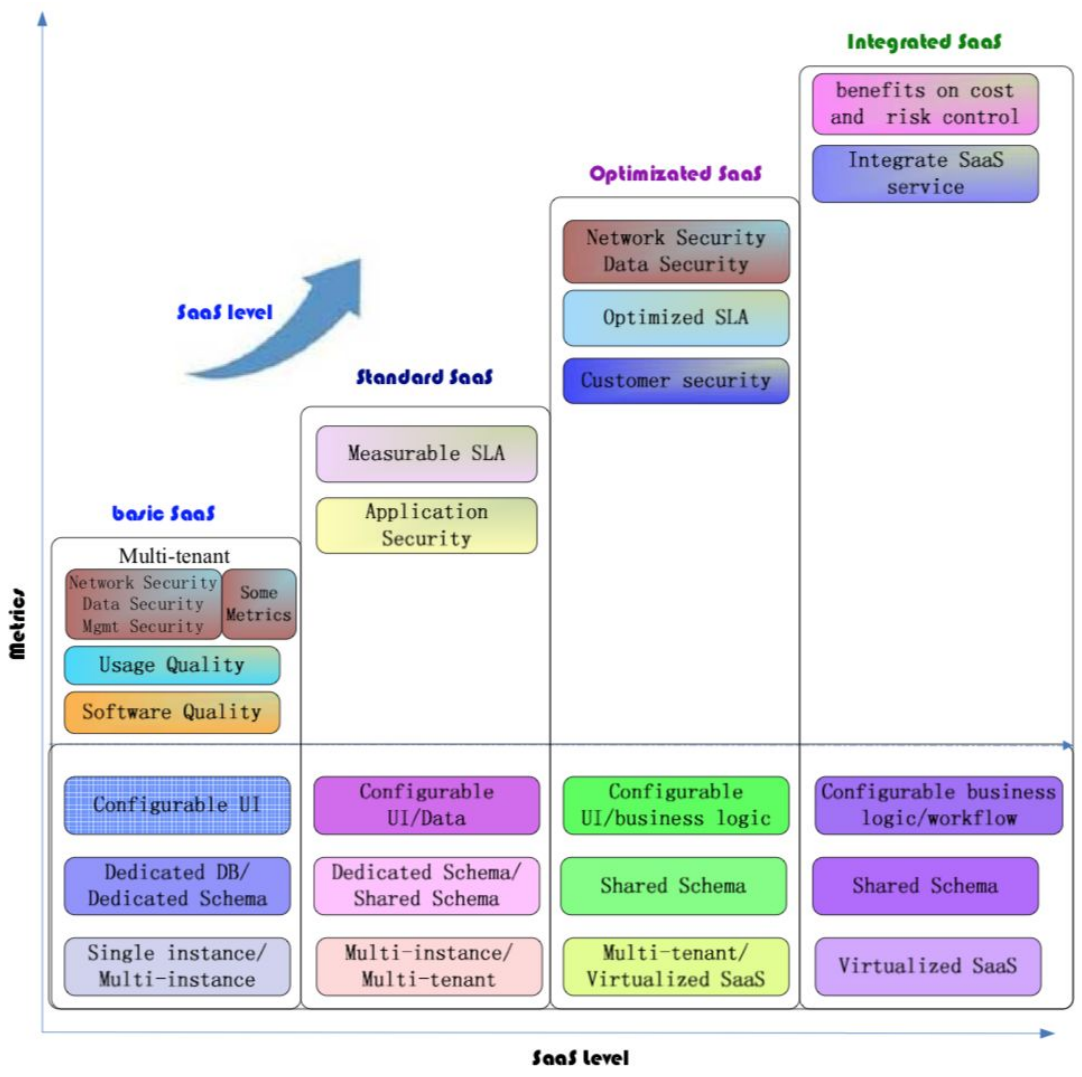

Integrated SaaS
benefits on cost and risk control
Integrate SaaS service
Optimizated SaaS
Network Security Data Security
Optimized SLA
Customer security
SaaS level
Standard SaaS
Measurable SLA
Application Security
basic SaaS
Multi-tenant
Network Security Data Security Mgmt Security
Some Metrics
Usage Quality
Software Quality
Metrics
Configurable UI
Dedicated DB/ Dedicated Schema
Single instance/ Multi-instance
Configurable UI/Data
Dedicated Schema/ Shared Schema
Multi-instance/ Multi-tenant
Configurable UI/business logic
Shared Schema
Multi-tenant/ Virtualized SaaS
Configurable business logic/workflow
Shared Schema
Virtualized SaaS
SaaS level

Figure 2.3.2: Evaluating Model of SaaS Service (Wen, 2013)

This model decomposes quality into a number of metrics. These metrics fall under a the following factors:

1. **Security:** Security is considered a primary concern for the customer, and is therefore included by the author of this work as a high-level category. They decompose security further into five categories: Customer Security, Application Security, Network Security, Data Security, and Management Security. Each of these categories are further decomposed into metrics.
2. **Quality of Service (QoS):** The authors divide QoS into three categories: Quality of Platform (QoP), Quality of Application (QoA), and Quality of Experience (QoE). Each of these categories are further decomposed into metrics.

3. **Software Quality Metrics:** The final quality, software quality metrics, refers to the general quality of the software. Here the authors use the model specified in ISO/IEC 25010:2011, reviewed later in this section.

Each of these metrics decomposed from the factor apply to a specific component level (SaaS Platform, Application, or Customer). The metric mapping is illustrated in Figure 2.3.3.

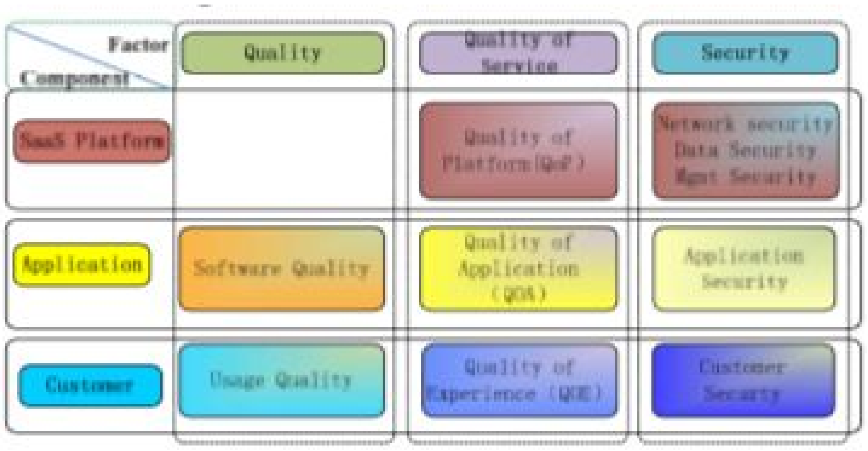


Figure 2.3.3: Quality Model of SaaS Service (Wen, 2013)

Miguel, et al. provide a summary of popular software quality models. These models are described below.

McCall Model

The McCall model, also known as the General Electric (GE) Model also used logical decomposition to define software features and metrics. The authors see software quality as being with respect to the perspective of a stakeholder, so they chose to decompose from three perspectives: Product Operation, Product Review, and Product Transition. These were then decomposed into features, which were decomposed into metrics. This decomposition can be seen in Figure 2.3.4, below.

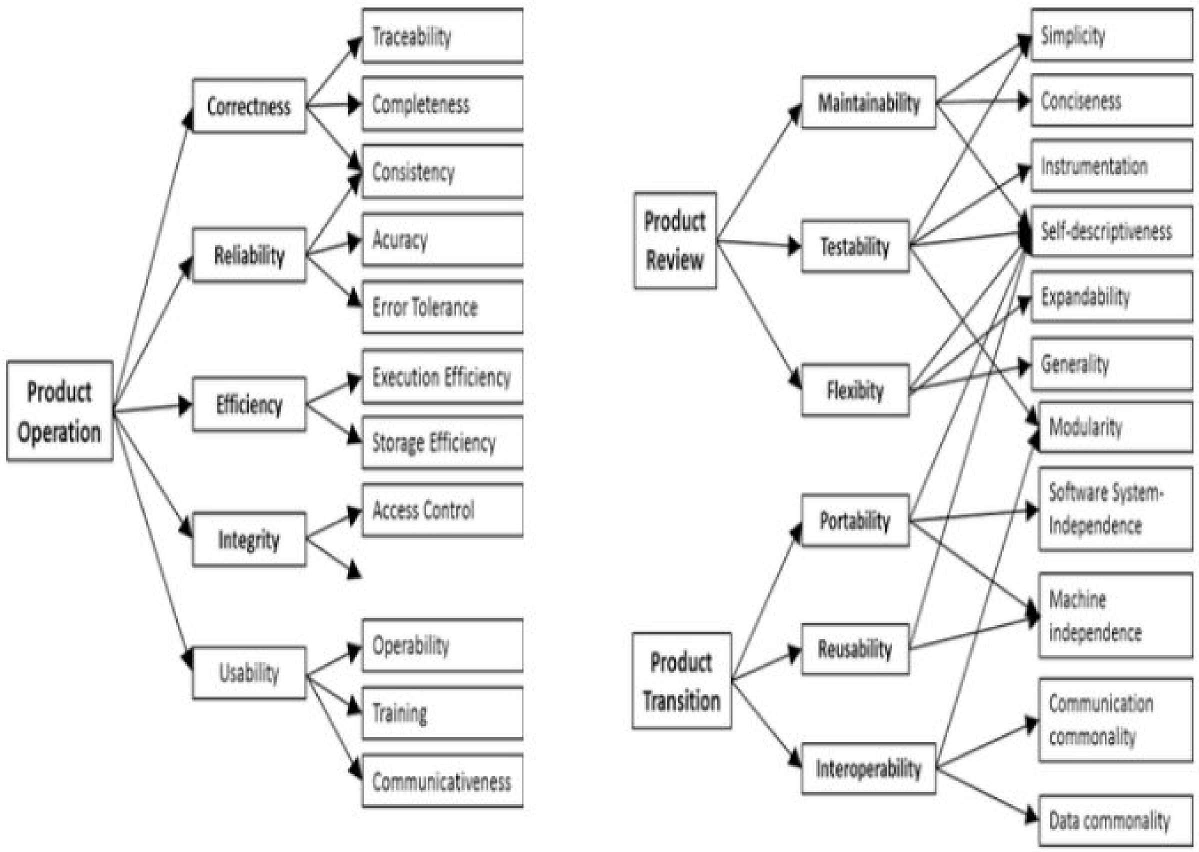


Figure 2.3.4: McCall Quality Model (Miguel, 2014)

Each metric from this model has a yes/no question associated with it. Answering these yes/no questions provides an estimate of software quality.

Boehm Model

The Boehm Model is a highly-cited software quality model based on logical decomposition from the general concept of quality. Quality is decomposed into three high-level factors: Portability, Utility, and Maintainability. These are decomposed into features where are decomposed into metrics. Figure 2.3.5 shows this model.

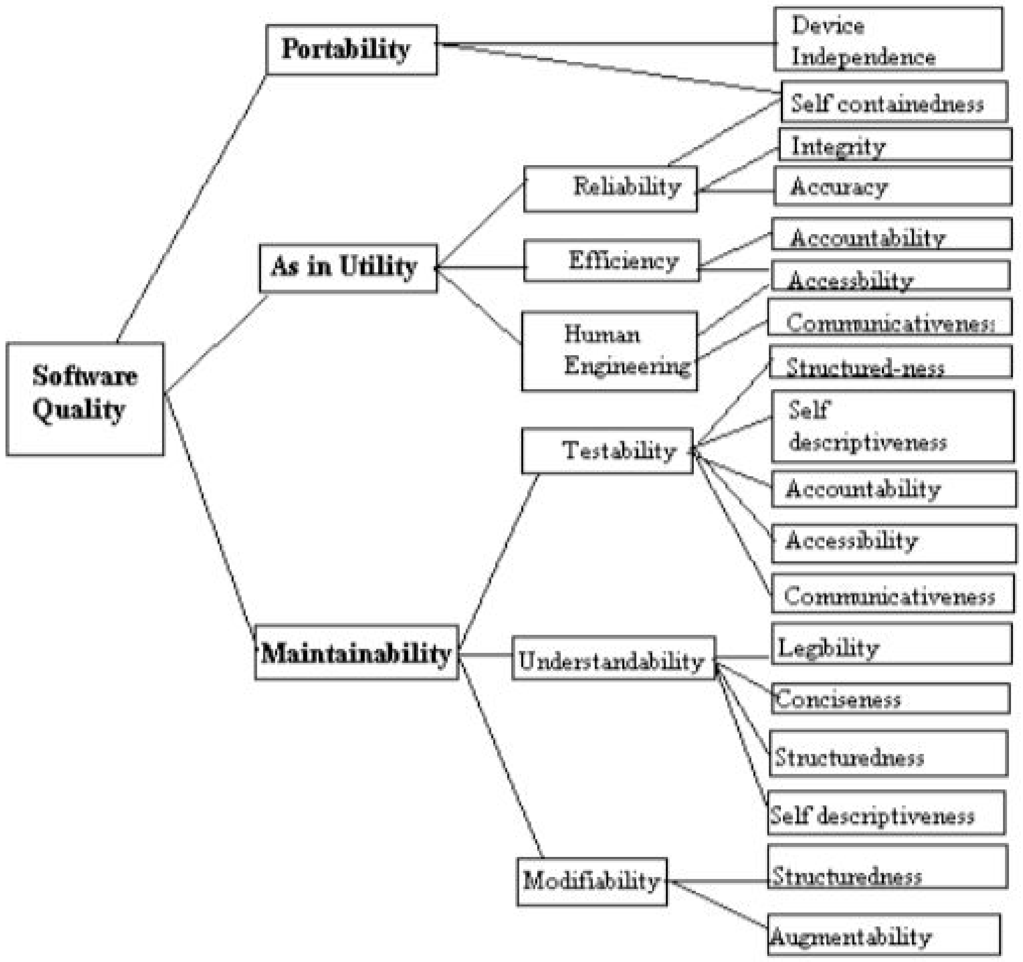


Figure 2.3.5: Boehm Model (Dubey, 2012)

Dromey Model

Like the McCall Model, the Dromey Model asserts that quality is a function of perspective. As such it begins logical decomposition with the concept of *implementation*. Implementation is decomposed into three characteristics, which are decomposed into sub-characteristics. These sub characteristics are never decomposed to a level where it can be used in practice, but this model serves the basis of others. The model is included below in Figure 2.3.6.

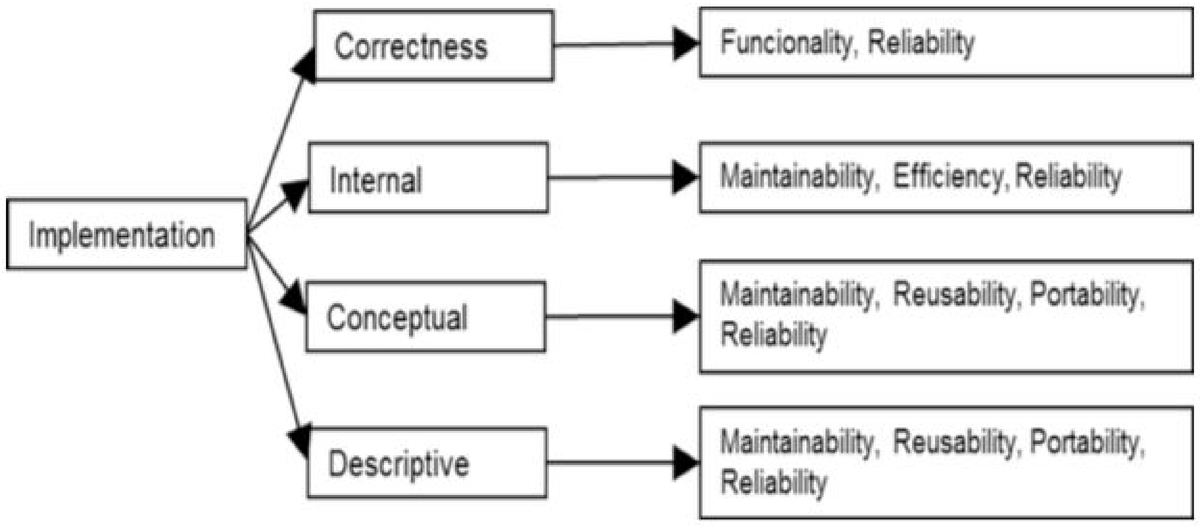
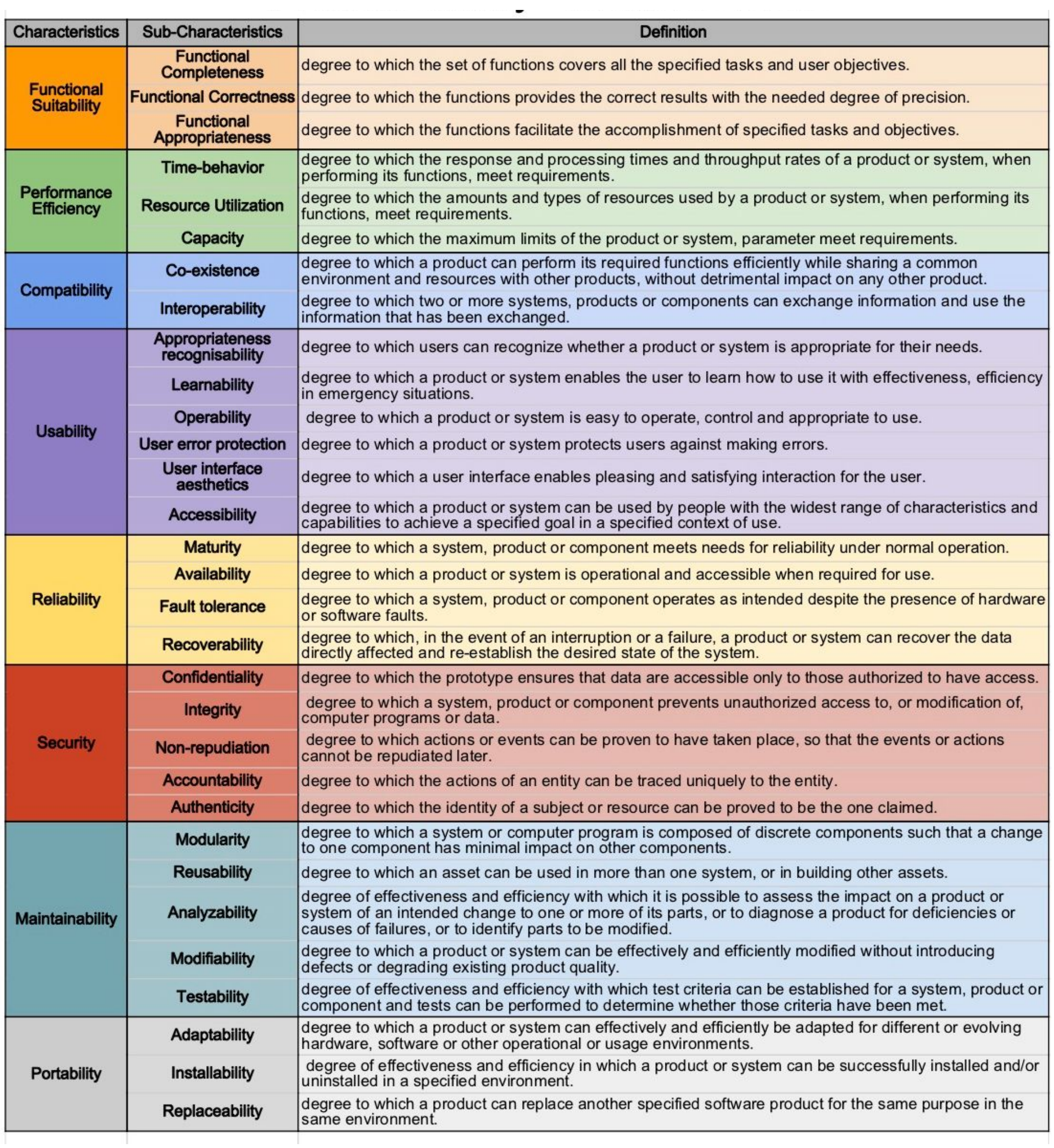

Implementation
Correctness
Internal
Conceptual
Descriptive
Funcionality, Reliability
Maintainability, Efficiency, Reliability
Maintainability, Reusability, Portability, Reliability
Maintainability, Reusability, Portability, Reliability

Figure 2.3.6: Dromey Model (Miguel, 2014)

ISO 25010 Model

In 2011, the International Organization for Standardization (ISO) released an updated Standard on System and software quality models (ISO, 2011). This was an update of the ISO 9126 Model, which was based on the Dromey and Boehm Models (Miguel, 2014). This standard divides quality into product (See Figure 2.3.7) and use (See Figure 2.3.8) quality categories, each further decomposed into characteristics and sub-characteristics.

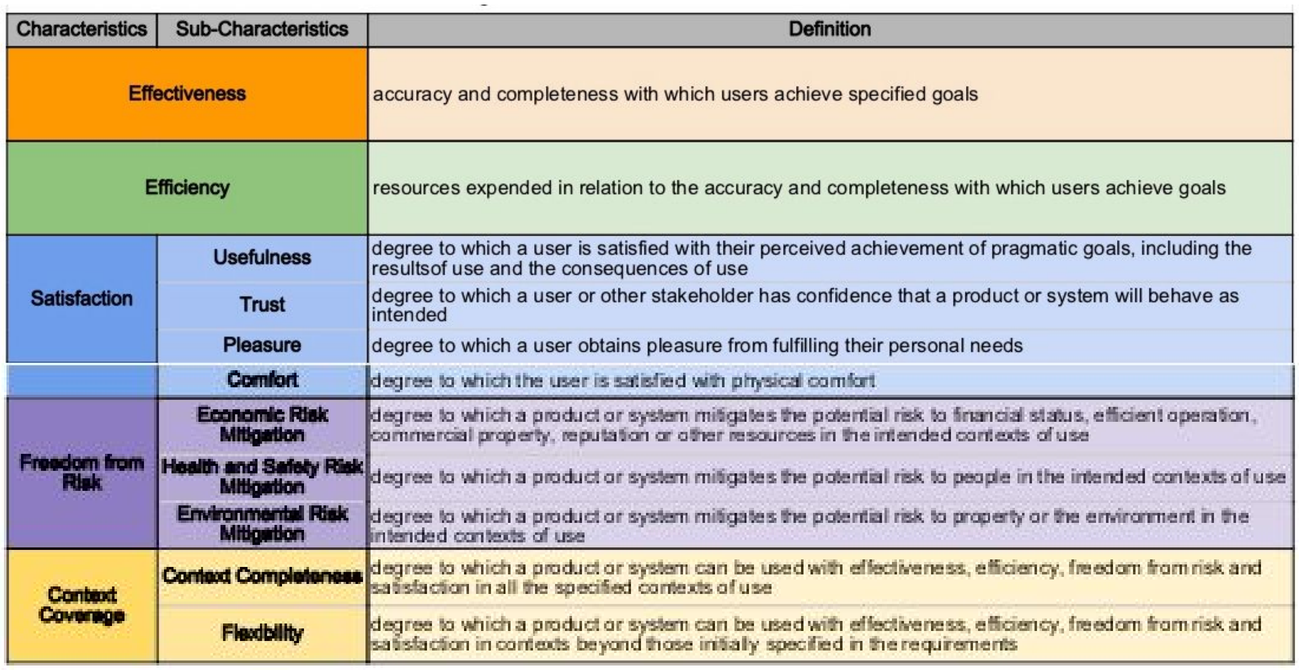


Characteristics	Sub-Characteristics	Definition
Functional Suitability	Functional Completeness	degree to which the set of functions covers all the specified tasks and user objectives.
	Functional Correctness	degree to which the functions provides the correct results with the needed degree of precision.
	Functional Appropriateness	degree to which the functions facilitate the accomplishment of specified tasks and objectives.
Performance Efficiency	Time-behavior	degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
	Resource Utilization	degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
	Capacity	degree to which the maximum limits of the product or system, parameter meet requirements.
Compatibility	Co-existence	degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
	Interoperability	degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.
Usability	Appropriateness recognisability	degree to which users can recognize whether a product or system is appropriate for their needs.
	Learnability	degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations.
	Operability	degree to which a product or system is easy to operate, control and appropriate to use.
	User error protection	degree to which a product or system protects users against making errors.
	User interface aesthetics	degree to which a user interface enables pleasing and satisfying interaction for the user.
	Accessibility	degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.
Reliability	Maturity	degree to which a system, product or component meets needs for reliability under normal operation.
	Availability	degree to which a product or system is operational and accessible when required for use.
	Fault tolerance	degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
	Recoverability	degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.
Security	Confidentiality	degree to which the prototype ensures that data are accessible only to those authorized to have access.
	Integrity	degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
	Non-repudiation	degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
	Accountability	degree to which the actions of an entity can be traced uniquely to the entity.
	Authenticity	degree to which the identity of a subject or resource can be proved to be the one claimed.
Maintainability	Modularity	degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
	Reusability	degree to which an asset can be used in more than one system, or in building other assets.
	Analyzability	degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
	Modifiability	degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
	Testability	degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
Portability	Adaptability	degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
	Installability	degree of effectiveness and efficiency in which a product or system can be successfully installed and/or uninstalled in a specified environment.
	Replaceability	degree to which a product can replace another specified software product for the same purpose in the same environment.

Figure 2.3.7: Product Quality - ISO/IEC 25010 (Universidade de São Paulo, 2011)

Product Quality (Figure 2.3.7, above) is divided into eight “characteristics” these characteristics closely map to what are being called NFR’s in this document (See Figure 2.5.1). These “characteristics are further divided into 31 sub-characteristics which map to what is called properties in this document.

Similarly, Use quality is divided into five characteristics, and 11 sub-characteristics. Each of these are defined in Figure 2.3.8.



Characteristics	Sub-Characteristics	Definition
Effectiveness		accuracy and completeness with which users achieve specified goals
Efficiency		resources expended in relation to the accuracy and completeness with which users achieve goals
Satisfaction	Usefulness	degree to which a user is satisfied with their perceived achievement of pragmatic goals, including the resultsof use and the consequences of use
	Trust	degree to which a user or other stakeholder has confidence that a product or system will behave as intended
	Pleasure	degree to which a user obtains pleasure from fulfilling their personal needs
	Comfort	degree to which the user is satisfied with physical comfort
Freedom from Risk	Economic Risk Mitigation	degree to which a product or system mitigates the potential risk to financial status, efficient operation, commercial property, reputation or other resources in the intended contexts of use
	Health and Safety Risk Mitigation	degree to which a product or system mitigates the potential risk to people in the intended contexts of use
	Environmental Risk Mitigation	degree to which a product or system mitigates the potential risk to property or the environment in the intended contexts of use
Context Coverage	Context Completeness	degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in all the specified contexts of use
	Flexibility	degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in contexts beyond those initially specified in the requirements

Figure 2.3.8: Use Quality - ISO/IEC 25010 (Universidade de São Paulo, 2011)

Each of these related research products provide their own interpretation of software quality. The advantages and disadvantages of these are explored in Section 2.4. This will serve the basis of our initial quality model, as described in Chapter 4.

2.4. Advantages and Disadvantages

The advantages and disadvantages of the Software Quality Models summarized in Section 2.3 are explored in Table 2.4.1.

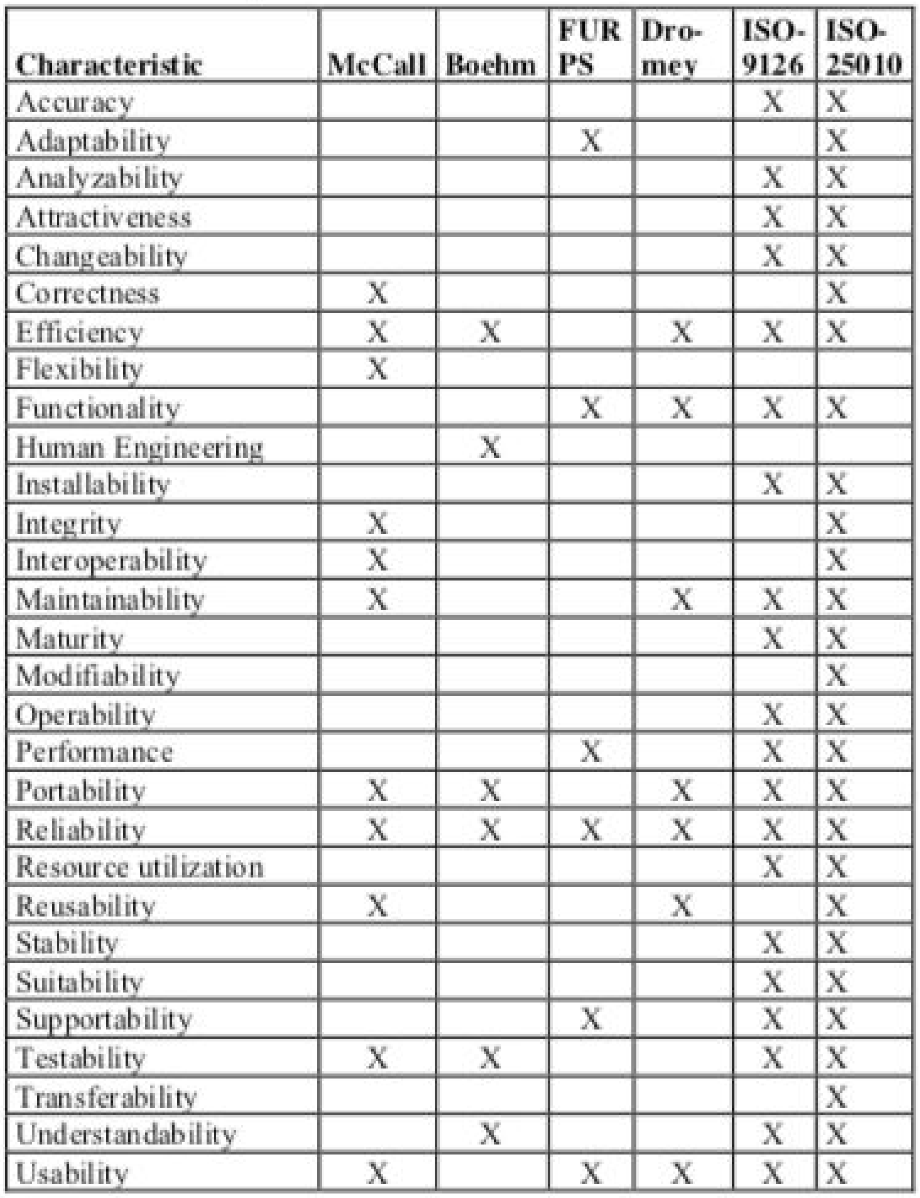
Table 2.4.1: Advantages and Disadvantages of Past Research

Related Research	**Advantages**	**Disadvantages**
Bogner Maintainability Quality Model (Bogner, 2017)	- Define a software quality model for services and microservices - Used a logical decomposition, layered approach - Defined meaningful quantifiable metrics - Detailed approach	- Only consider maintainability - No experimental analysis, only based on literature survey and intuition of authors - Missing testability metrics, which is considered an important part of maintainability
Wen Model (Wen, 2013), (Banerjee, 2014)	- Explicitly targeting services - Includes full cross-section of quality metrics	- Metrics are not quantifiable - Does not go into great detail on the “software quality” metric, which is of most interest to us.

McCall Model (McCall, 1977)	- Considers relationships between quality characteristics and metrics (Miguel, 2014) - Includes full cross-section of quality metrics	- Binary- based on answers of yes/no. No resolution of quality - Does not consider functionality - Not targeted towards services or microservices
Boehm Model (Boehm, 1978)	- Includes full cross-section of quality metrics - Additional layers to decomposition	- Not targeted towards services or microservices
Dromey Model (Dromey, 1995)	- Based in concept of perspective	- Not targeted towards services or microservices - Theoretical model- no discussion of application
ISO 25010 Model (ISO, 2011)	- Includes full cross-section of quality metrics - Thorough and complete - Built on the knowledge of practitioners	- Not targeted towards services or microservices

Miguel, et al. provide a summary of the characteristic (NFR) covered by each of the models they summarized. These are included in Table 2.4.2, below

Table 2.4.2: Comparison of Basic Models (Miguel, 2014)



Characteristic	McCall	Boehm	FUR PS	Dro-mey	ISO-9126	ISO-25010
Accuracy					X	X
Adaptability			X			X
Analyzability					X	X
Attractiveness					X	X
Changeability					X	X
Correctness	X					X
Efficiency	X	X		X	X	X
Flexibility	X					
Functionality			X	X	X	X
Human Engineering		X				
Installability					X	X
Integrity	X					X
Interoperability	X					X
Maintainability	X			X	X	X
Maturity					X	X
Modifiability						X
Operability					X	X
Performance			X		X	X
Portability	X	X		X	X	X
Reliability	X	X	X	X	X	X
Resource utilization					X	X
Reusability	X			X		X
Stability					X	X
Suitability					X	X
Supportability			X		X	X
Testability	X	X			X	X
Transferability						X
Understandability		X			X	X
Usability	X		X	X	X	X

2.5. Solution Definition

The solution in this effort will consist of a quality model for microservices and a microservice demonstration. This quality model will be developed from combined published observations of software engineers and through the author's own investigation and intuition. The solution development process is described further in Chapter 4.

We propose using a logical decomposition approach like those used in (Bogner, 2017). The top layer of the decomposition tree is the common non-functional requirements most important to microservice developers: Reusability, Flexibility, and Maintainability. The second layer is a set of properties that fulfill those NFRs. The final layer is a set of metrics. Each metric is unique and quantifiable.

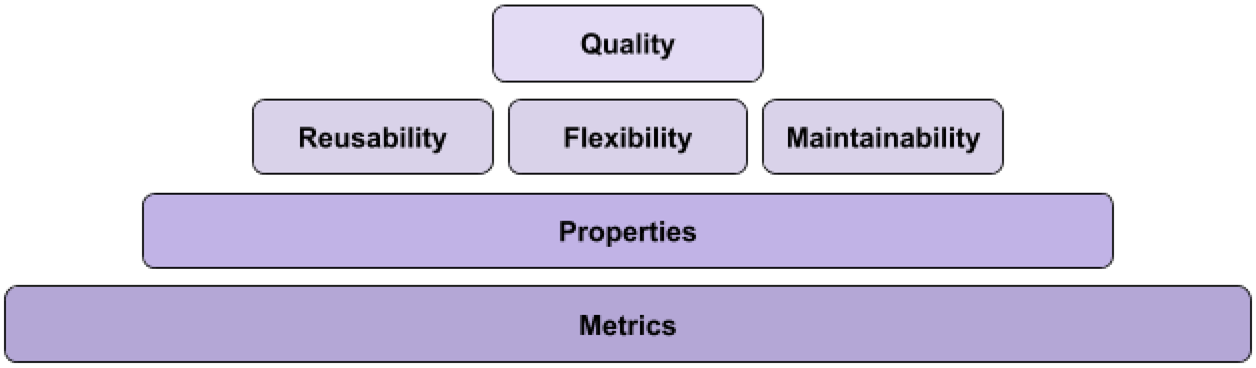


Figure 2.5.1: Quality Model Layers

Assessing the quality of a solution is done by combining the metric scores. Each metric for a property is combined and divided by the number of properties to get a property score. This is repeated for each NFR, and finally for the final quality score, as seen in the below pseudocode:

```
Quality_Score = 0
for each NFR:
     NFR_Score = 0
     Property_Count = 0
     for each property in NFR:
          Property_Score = 0
          Metric_Count = 0
          For each metric in property:
               Property_Score += Metric_Score
               Metric_Count++
          Property_Score /= Metric_Count
          NFR_Score += Property_Score
          Property_Count++
     NFR_Score /= Propert_Count
     Quality_Score += NFR_Score
```

2.6. Solution Unique Elements

The team’s quality model is unique in two ways:

1. It is targeted specifically to microservices.
2. They will iterate, improve, and adapt historical quality models based on our their experience form implementing a test/demonstration microservices.

2.7. Solution Strengths

This team’s quality model is strong in two ways:

1. It is targeted specifically to microservices.
2. It is based on both experience of past engineers, but also experimental observations.

In conclusion, this chapter defined the problem, described the different use cases and why a solution is needed. Then the team looked at related research and determined their advantages and disadvantages in creating a quality model. Based on the related research the team proposed the solution definition, how the solution is unique and the strength this solution has compared to other research.

3. Hypothesis

As described in the preceding sections, there are a number of general software quality models in existence. There is, however, no comprehensive quality model targeted specifically towards the microservice architecture.

The authors' hypothesis is the following:

Existing general software quality models are incomplete for assessing microservice quality and can be improved

In order to test this hypothesis, the goals of this project are the following:

1. Define a quality model for Microservices
2. Identify best practices for microservice design that meet the
3. Demonstrate these best practices with example quality microservices

The methodology the team will use to meet these goals is defined in the following chapter.

4. Methodology

To meet the goals enumerated in Chapter 3, the following steps will be followed:

1) Generating and Collecting Input Data,
2) "Solving the Problem",
3) Generating Output, and
4) Testing our Output.

Each of these are described in detail in the following sections. The project will culminate in a software demonstration of quality microservice design and inter-microservice communication.

The team's schedule can be seen in Gantt Chart form in Figure 4.1. Each of these activities are described in greater detail in the following sections.

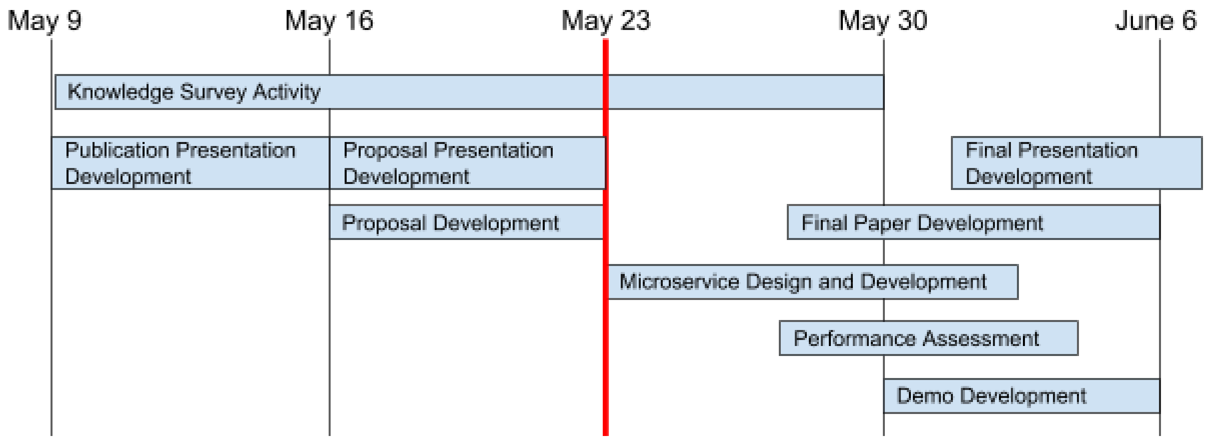


Figure 4.1: Team Gantt Chart

4.1. Generating and Collecting Input Data

For this effort, there are three sources of input data that will be drawn from:

1) The experience and lessons learned of skilled practitioners, scientists, and engineers.
2) Our own professional engineering intuition.
3) Experimental observations.

From these sources, the team will derive an understanding of what makes a quality microservice. This effort is divided into two activities: knowledge survey and microservice design and development. These activities are summarized in the following subsections.

Knowledge Survey Activity

The goal of this activity is to collect the experience and lessons learned of skilled practitioners (Source 1 from the above list). The team will collect this information from sources such as publications in professional journals, books, and conference proceedings from professional societies (e.g., IEEE, ACM); discussions with technical experts; videos tutorials and lectures; and online sources (e.g., websites from trusted sources). Information will also be collected by investigating the design of open-source microservices from inline repositories and content management sources such as Github or Gitlab. Extra attention will be given to services known to be of high-quality and those created by respected institutions.

The results of this activity will be collected, organized, and documented in the final presentation. The results of this activity will also be used to create an initial “quality model” that will be iterated upon in the microservice design and development activity.

This activity began upon team assignment on May 9th, 2018. The results of the team’s initial efforts in this activity are summarized in Chapters 1 and 2.

Microservice Design and Development Activity

The goal of activity is to improve our understanding of quality design of microservices through experimental observation. The team will design and develop simple example microservices. The design of these microservices is described in Section 4.2. The quality of these microservices will be evaluated using our existing quality model (as described in Sections 4.3,4.4), and the quality model will be updated based on the team’s observations. The results of this activity will also be documented in the final project paper.

4.2. How to solve the problem

As described in Section 4.1. A number of simple microservices will be designed and develop to mature the quality model created from the knowledge survey activity. The following subsections describe this in greater detail. The *Algorithm Design* Subsection describes the types of microservice examples to be developed and their requirements. The *Language Used* and *Tools Used* Subsections describes the choice of language and the tools or modules used to create these microservices, respectively.

Algorithm Design

The team identified the following properties that we desire of our example microservices in order for this activity to be effective:

1) **Function Variety:** That the services represent a cross section of the functions appropriate for microservices.
2) **Interface Variety:** That the services represent a cross section of the interface architectures appropriate for microservices
3) **Interdependence:** That some of the microservices use each other. This is necessary to ensure consideration of inter-microservice interactions is included in the quality model

Based on these criteria, the following microservices were selected to represent our *example microservice set*. **One** of these microservices will be selected to be implemented.

1. Carbon footprint estimation and tracking service(s)

The purpose of this service(s) is to provide an estimate of a person's carbon footprint based on their behavior. In reality, there are many factors that contribute to an individual's impact (Berkeley Student Environmental Resource Center, 2014). Understanding an individual's impact helps those individuals identify how they can reduce their footprint. The idea for this service comes from the SCU Mobile Application Development Course Project *Impact Estimator* (Impact Estimator Github Repository, 2018)

This service was chosen because it is a stateless service (the server does not retain any information) and because it is a complex service that will need to be broken down to interdependent microservices (fulfilling criteria 3).

The informal requirement for this service are identified to be the following:
The service shall produce an estimate of an individual's carbon footprint for each of the following categories: Travel, Food, Household, Utilities, Products, Services.

This is, of course, not a complete formal requirement set (and in itself is a compound requirement), but this level of rigor is considered appropriate for this exercise. The context of this service can be seen in the below figure.

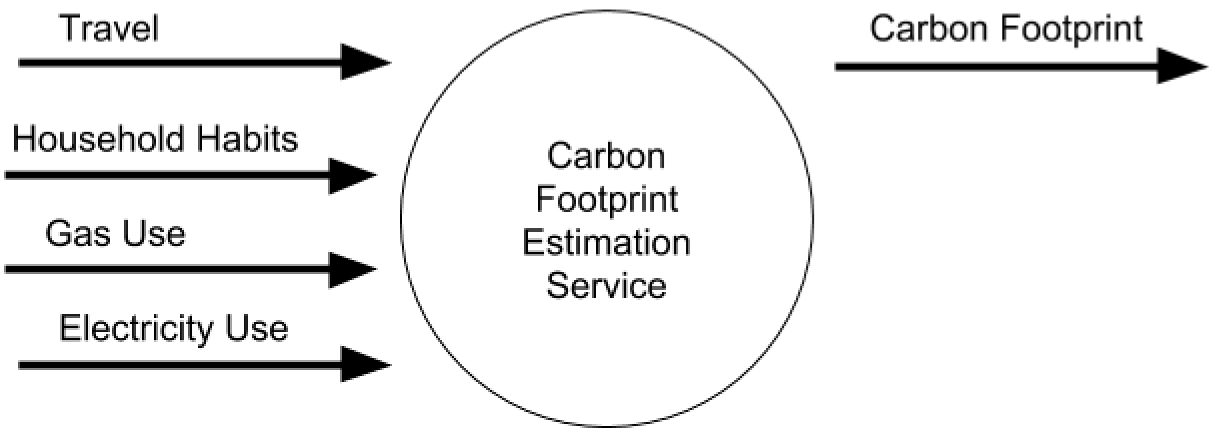


Figure 4.2.1: Impact Tracking Service Context

The actual design of this service/these services will depend on the results of the initial service model identification activity.

2. Time tracking and task estimation service

The second service identified is a time tracking and task estimation. The purpose of this service is twofold:

1. Tracking time that individuals spend on tasks (e.g., 30 minutes to create a function to do x…)
2. Estimating time required for a task. A combined architecture like this will allow managers to estimate effort required for a required task, based on real historical trends for similar tasks.

This service was chosen because it is 1) a complex service with multiple discrete functions that will likely have to be broken into sub-tasks (microservices), and 2) it is a stateful task (it requires the service to maintain certain information).

The informal requirements for this service are identified to be the following:

1. The service shall accept details about a task performed and the time taken to perform that task from individual users
2. The service shall produce a summary of work conducted by an individual on request
3. The service shall produce an estimate of the time required to complete a given task based on the received task details (see requirement 1) on request.

This is, of course, not a complete formal requirement set, but this level of rigor is once again considered appropriate for this exercise. The context of this service can be seen in the below figure.

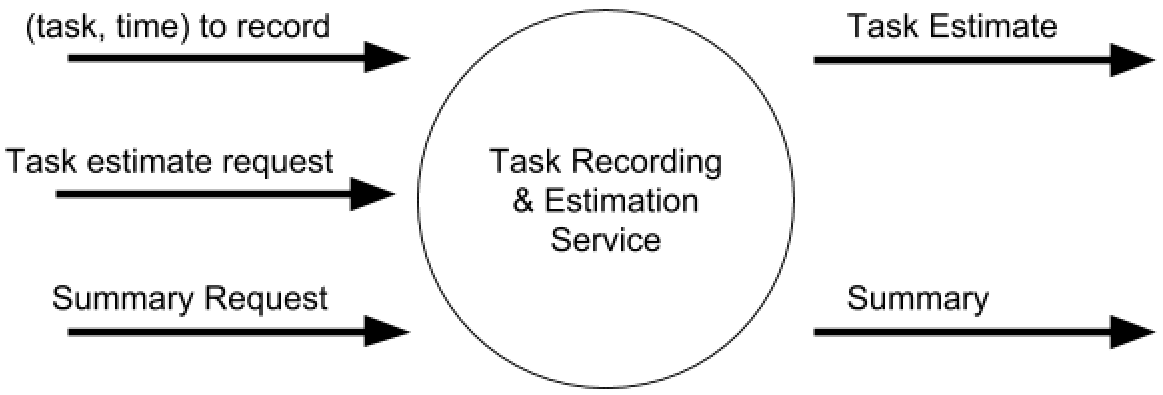


Figure 4.2.2: Time Tracking Service Context

The actual design of this service/these services will depend on the results of the initial service model identification activity.

Language Used

The team chose *Python* for the programming language for this project. Python was chosen because it is is a very high-level language with several modules supporting web interfaces. Python is a strong language for short-turnaround prototyping projects where speed is not critical, like this one.

Tools Used

The team expects to heavily utilize the *Flask* python module (Pocoo, 2018) and many of its extension modules. *Flask* is a BSD licensed python framework for web-services. Flask is used by many popular websites, like LinkedIn (Sanders, 2014) and Pinterest (Cohen, 2015).

4.3. How to Generate Output

Output from the Microservice Design and Development Activity, outlined in Section 4.2, will come in two forms: 1) team observations and thoughts and 2) quality model metrics.

The first form of output, team observation and thoughts, is the more qualitative of the two. The team will record their thoughts and observations about the strengths and weaknesses of particular design choices and methodologies during development and following the completion of the microservices. Special care will be taken to record thoughts about the degree to which the non-functional requirements (NFRs) in the quality model are met.

The second form of output, quality model metrics, is evaluated upon completion of the microservice. The microservice will be scored by the metrics in the initial quality model. The results of this scoring will be recorded and used again in Section 4.4.

4.4. How to Test Against Hypothesis

The purpose of this phase of the effort is to evaluate the effectiveness of the quality model and revise, as-needed. The scores and observations from Section 4.3 will be reviewed. From this review, the team will make an evaluation of the completeness and appropriateness of the identified quality model, revising where appropriate. The results of this activity will be recorded in the final paper and presentation.

In conclusion, this chapter has described the methodology that the team will use in solving the problem with quality in microservices. It also describes how input data is collect and how output data is generate. to prove that the solution is valid this section also describes how testing will be conducted against the hypothesis.

5. Implementation

This chapter describes how the solution was implemented based on the findings described in Chapter 4. Section 5.1 defines the quality model that will solve the problem. Section 5.2 describes how the quality model is evaluated and corrected through the development of the ‘Time Tracking and Task Estimation Service’.

5.1. Quality Model

The first step was the building of an initial quality model. This model was produced based on the investigation of existing software quality models, and the application of the intuition of the developers.

As described in Chapter 4, the authors are using a layered quality model, where quality is decomposed into NFRs of interest to microservices. These NFRs are further decomposed into attributes, which are decomposed into specific metrics. This structure is illustrated in Figure 1.1.1.

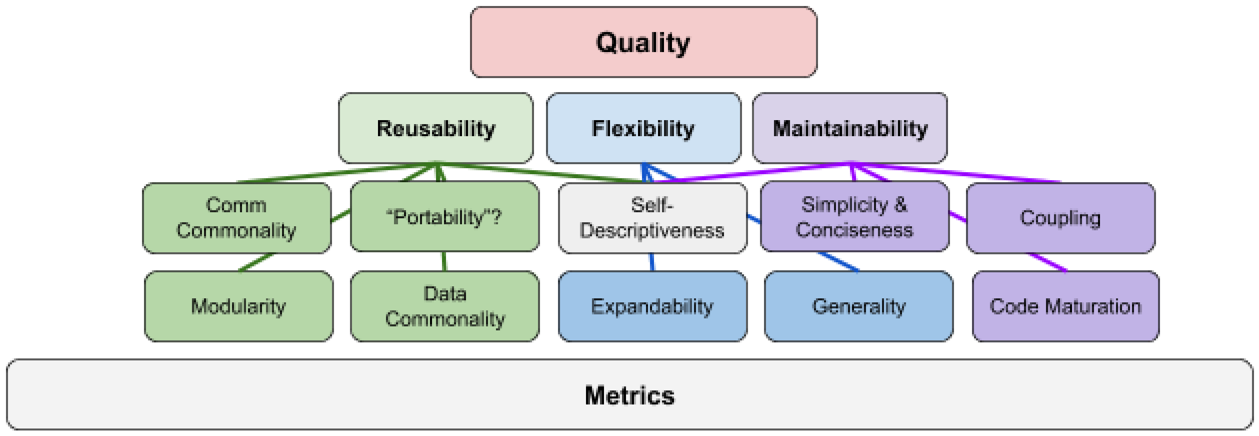


Figure 5.1.1: Microservice Quality Model

The metrics for each attribute are introduced in Table 5.1.1, below, and further described in the following sections.

Table 5.1.1: Microservice Quality Metrics

	NFR	Attribute	Metrics
QUALITY	**Reusability**	**Modularity**	Functional dependency
		Comm Commonality	Total communication protocols
		Data Commonality	Total data type conversions
		Loosely Coupled	Requirement documentation
		Self-Descriptiveness	Comment Ratio
	Flexibility		
	Maintainability		
	Flexibility	**Generality**	Number of module reference by other modules/total modules
	Maintainability	**Simplicity & Conciseness**	Total Response of Service (TRS) (Bogner, 2017)
			Halstead's Measure (Halstead, 1977)
		Coupling	Absolute Importance of the Service (AIS) (Bogner, 2017)
			Absolute Dependence of the Service (ADS) (Bogner, 2017)
			Services Interdependence in the System (SIY) (Bogner, 2017)
		Code-Maturation	Comment Ratio (CR) (Bogner, 2017)
			Clone Coverage (CC) (Bogner, 2017)
			Test Coverage (TC) (Bogner, 2017)

Reusability

The ability to reuse code for development. Reusability increases the consistency efficiency on development by reusing code, be able to run existing tests. The benefits come from avoiding cost and duplication of work.

- **Modularity:** Modularizing microservices to be in the component level. When used in large systems microservices are easier to implement, and problem determination.
  - Potential function modularization ratio (Potential Ratio)
    - $PR = [Potential\ Modularization\ of\ Function/[Total\ Modules]$
  - Modules dependent on others
    - List the modules
    - Determine if dependency can be removed
  - Ratio of dependent modules
    - $DM = [Number\ of\ Dependent\ Modules]$ / $[Total\ Modules]$
- **Communication Commonality:** The communication protocol that is used by the microservice through the network. Communication commonality simplifies problem determination errors caused by the network. It also allows the module to be reusable as the communication protocol will be common.
  - Communication protocol documented and used.
    - Yes/No
  - Number of communication protocol used.
- **Data Commonality:** The input and output data from the microservice is the same type. This allows microservices to be reusable as anyone using the microservice knows what data type is required thus avoiding data format conversion which adds another level of complexity.
  - Documentation describing data format used
    - Yes/No
  - Is there more than one data format used for input and output
    - Yes/No
  - Ratio of data format type conversion to the total modules that allows input and output
    - $DC = [Number\ of\ Data\ Format\ Conversion]$ / $[Total\ Input\ and\ Output\ Modules]$
- **"Loosely Coupled"?:** The capability of the microservice to be accessed from any device without little to no manual steps required by the user. Allows the microservice to not be dependent on hardware or software requirements.
  - Documentation describing what requirements are needed
    - Yes/No

    - Are there hardware restrictions?
        - Yes/No
    - Are there dependencies on third party libraries?
        - Yes/No
- **Self-Descriptiveness:** Each module should be self descriptive through the modules name, and comments should be precise with describing the modules inputs, outputs and function so that the user can understand how reusable the module is.
    - Variable names that are acronyms or shortened.
        - List them
        - Change to readable name when possible
    - Comments describing what a module does.
        - $NC = [Number\ of\ Comments\ per\ Module] / [Total\ Lines\ of\ Code\ per\ Module]$

Flexibility

The ability to be able to switch technology without disrupting other parts/components of a system. This give developers or users the ability to choose their development language as long as the microservice's input and outputs use a common data source.

- **Generality:** The generalization of modules where it is stripped from uniqueness so that anyone is able to use the module for their needs.
    - Number of module reference by other modules/total modules
    - Can all modules be called independently
- **Self-Descriptiveness:** Each module should be self descriptive through the modules name, and comments should be precise with describing the modules inputs, outputs and function so that the user can understand how flexible the module is.
    - Variable names that are acronyms or shortened.
        - List them
        - Change to readable name when possible
    - Comments describing what a module does.
        - $NC = [Number\ of\ Comments\ per\ Module] / [Total\ Lines\ of\ Code\ per\ Module]$

Maintainability

Boehm, et al. describes maintainability as the "Effort required to locate and fix an error in an operational program" (Boehm).

Below is a list of the maintainability attributes and metrics chosen for this Quality Model:

- **Simplicity & Conciseness:** “Those attributes... that provide for implementation of the functions in the most understandable manner. (Usually avoidance of practices which increase complexity)” and “implementation of a function with a minimum amount of code” (Boehm)
    - Total Response of Service (TRS): “For each operation- weighted sum of operations/local methods called” (Bogner, 2017)
    - Halstead's Measure (Halstead, 1977)

$$N_0 = N_1 + N_2$$

Where $N_1 =$ total number of operators

and $N_2 =$ total number of operands

$$N_C = n_1 log_2(n_1) + n_2 log_2(n_2)$$

Where $n_1 =$ total number of unique operators

and $_2 =$ total number of unique operands

The Halstead metric is defined by

$$1 - \frac{|N_C - N_0|}{N_0}$$

- **Coupling:** ““The degree or indication of the strength of interdependencies and interconnections of a service with other services.” (Bogner, 2017)
    - Absolute Importance of the Service (AIS) (Bogner, 2017): “The [fraction] of clients that invoke one operation from the service”
    - Absolute Dependence of the Service (ADS) (Bogner, 2017): “The [fraction] of services that service S depends on”
    - Services Interdependence in the System (SIY) (Bogner, 2017): “The [fraction] of services that are bi-directionally dependent on each other”
- **Code-Maturation:** “The degree of technical proficiency and consistency of the code base of [a service]” (Bogner, 2017)
    - Comment Ratio (CR) (Bogner, 2017): Determines how much description is added for each module.

$$CR = 1 - [Lines\ of\ Quality\ Comments] / [Total\ Lines]$$

    - Clone Coverage (CC) (Bogner, 2017): Amount of duplicated code

$$CC = [Duplicated\ Lines\ of\ Code] / [Total\ Lines]$$

    - Test Coverage (TC) (Bogner, 2017): Shows how thorough the microservice was tested.

$$TC = 1 - [Lines\ Executed\ in\ Tests] / [Total\ Lines]$$

- **Self-Descriptiveness:** Each module should be self descriptive through the modules name, and comments should be precise with describing the modules inputs, outputs and function so that the user can understand how flexible the module is.

- Variable names that are acronyms or shortened.
    - List them
    - Change to readable name when possible
- Comments describing what a module does.
    - $NC = [Number\ of\ Comments\ per\ Module] / [Total\ Lines\ of\ Code\ per\ Module]$

Estimating Quality

Overall microservice quality is assessed using the following algorithm:

```
Quality_Score = 0
for each NFR:
      NFR_Score = 0
      Property_Count = 0
      for each property in NFR:
            Property_Score = 0
            Metric_Count = 0
            For each metric in property:
                  Property_Score += Metric_Score
                  Metric_Count++
            Property_Score /= Metric_Count
            NFR_Score += Property_Score
            Property_Count++
      NFR_Score /= Propert_Count
      Quality_Score += NFR_Score
```

5.2 Software Design

In order to evaluate and improve upon the teams quality model, the team chose an example microservice to build (as described in Section 4.2). The team chose the second example, the *Time Tracking and Task Estimation Service.* The design of this microservice is described in this section.

Goals

The goals of this software are the following:

1. Provide reports of work done by a user
2. Provide estimates for new tasks

Design

To meet these two goals, the team undertook a design activity. A use case diagram for this software can be found below, in Figure 5.2.1. Here there are two users modeled: the developer who logs time, and the manager who generates reports of work done, and uses the data to estimate the time required to complete new tasks.

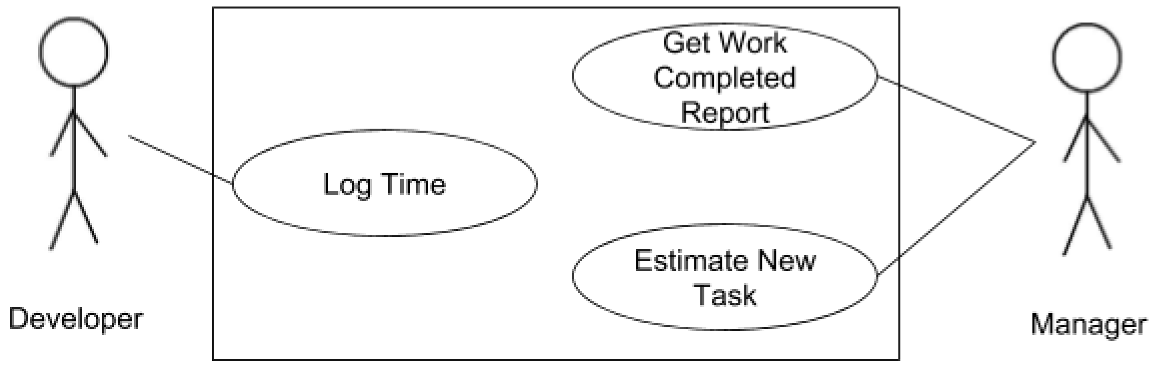


Figure 5.2.1: Overall Use Case Diagram

To accomplish the actions described in the use case diagram, the following design was used. This design divides the service into a number of microservices, each accomplishing a portion of the complete task. This design is illustrated in Figure 5.2.2. This design allows developers to create new services reusing the microservices below. For example, a developer who wants to create auditing, or timesheet software might reuse the work recorder microservice.

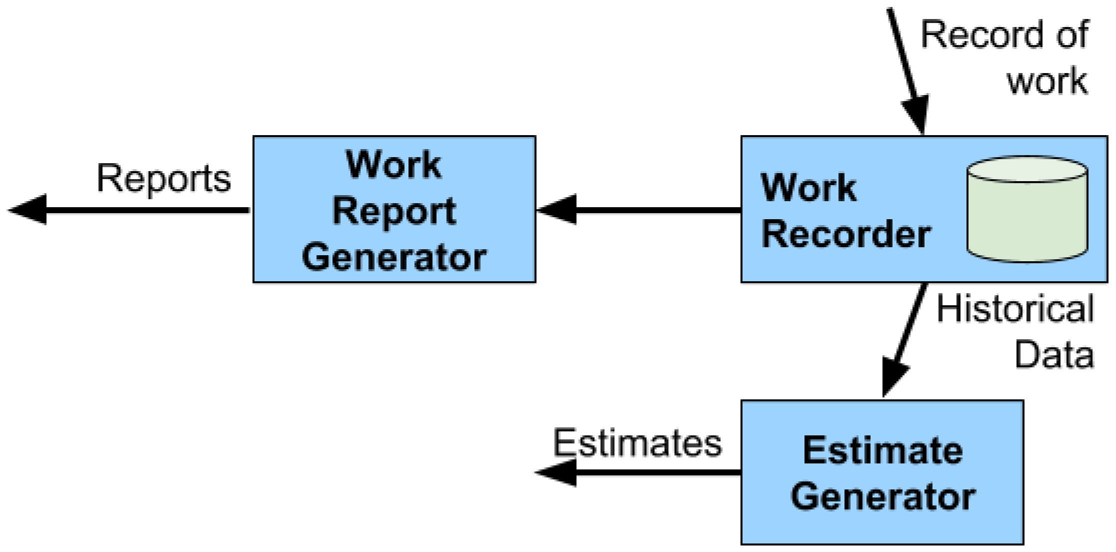


Figure 5.2.2: Time Tracking and Task Estimation Service Design

To implement this, the team created a package for each microservice. First a private github repository was created to store the software. Two scripts were created to run the services. The first, *setup_env.sh* sets up the python environment using virtualenv, and installs the required package (described in *requirements.txt*). This only has to be run once when the user first clones the repository. The second script *startup.sh* starts each of the four microservices in its own terminal window (using xterm) and with its own port number. See Appendix C for the source code. The file structure can be seen below:

```
setup_env.sh            -- Script to setup python environment
run_demo.py             -- Python script to run demo
requirements.txt        -- Python requirements
startup.sh              -- Script to start microservices
work_report_generator/  -- Work Report Generator Package
estimate_generator/     -- Estimate Generator Package
work_recorder/              -- Work Recorder Package
README.md                   -- Readme file (Markdown)
```

The assigned port numbers for each microservice are described in Table 5.2.1, below:

Table 5.2.1: Microservice Port assignment

Microservice	**Port Number**
Estimate Generator	12001
Work Report Generator	12004
Work Recorder	12003

The setup for each package can be seen below. Each package (microservice) has a file called *__init__.py*. This file is run by Flask when the microservice is startup. *__init__.py* describes the API for the microservice. Each microservice also includes a subpackage for models used by the microservice. Other files might be included to accomplish specific functions of that microservice

```
__init__.py             -- Module API
model/                  -- Any models used by the module
```

The code for each microservice can be found in Appendix D-G. The design of the individual packages are described in the following subsections.

Estimate Generator

The estimate generator will provide an estimated time to develop(TDEV) based on the assumed single line of code in thousands (KSLOC), complexity factor both provided through the request by a user and the historical data factor obtained through the work recorder service.

The historical data factor is important in estimating the TDEV because it adds the organizations adjustment factor based on historical data. The work recorder keeps a record of actual TDEV so it will also have the KSLOC and complexity factor. When a request is sent to the estimate generator service, it will then send a request to the work recorder service. The work recorder service then obtains the actual KSLOC and complexity factor on historical data and compute the historical data factor through the internal COCOMO class.

The figure below show a use case for the estimate generator.

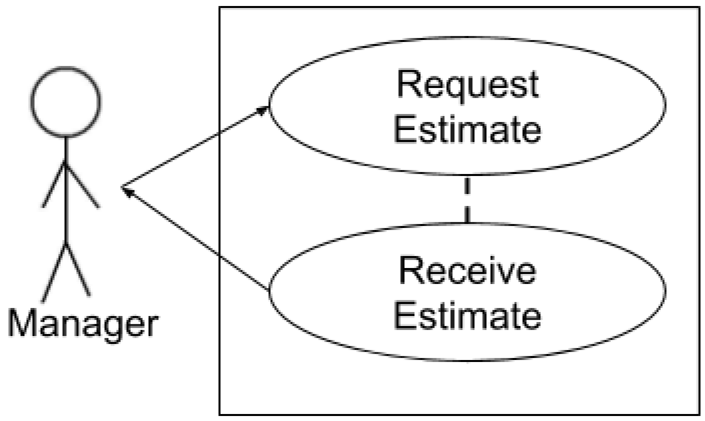


Figure 5.2.3: Estimate Generator Use Case Diagram

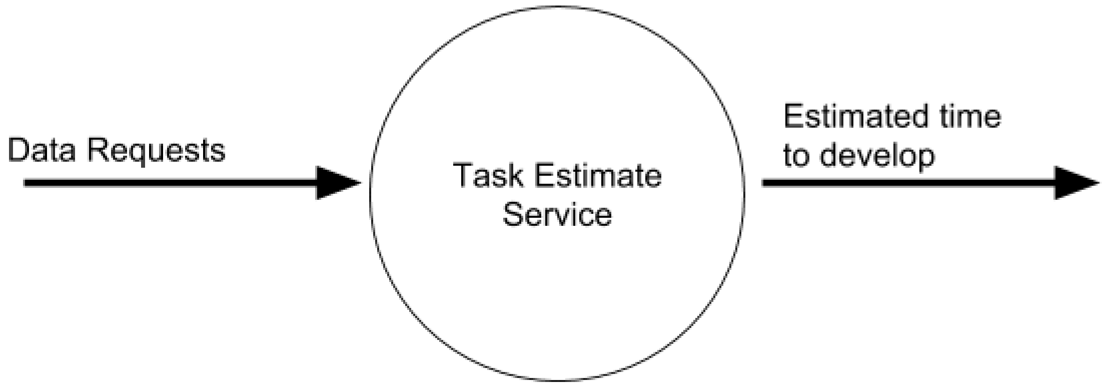


Figure 5.2.4: Estimate Generator Context Diagram

The API URIs for this package are described in Table 5.2.2

Table 5.2.2: Estimate Generator REST API Specification

URIs	GET	POST	PUT	DELETE
/api/	API Description --- < 200 (OK) `{"versions":"v1"}`			
/api/v1/	Version 1 Description --- < 200 (OK) `{"status":"OK","message":"Work Recorder API version 1.0.0","response":null}"`			

/api/v1/TDEV	Get estimated time to develop with historical factor
	--- 	
	> 200 (OK)
	`{"TDEV" : { "ksloc" : 100, "scalefactor" :{ "PREC" : ["sf"], "FLEX" : ["sf"], "RESL" : ["sf"], "TEAM" : ["sf"], "PMAT" : ["sf"]" }}}`
	--- 	
	< 200 (OK)
	`{"TDEV":null}`			

Work Recorder

The work recorder is tasked with maintaining a record of performed work to be used by other microservices. A use case diagram and context diagram for the work recorder can be found below in Figure 5.2.5 and 5.2.6, respectively.

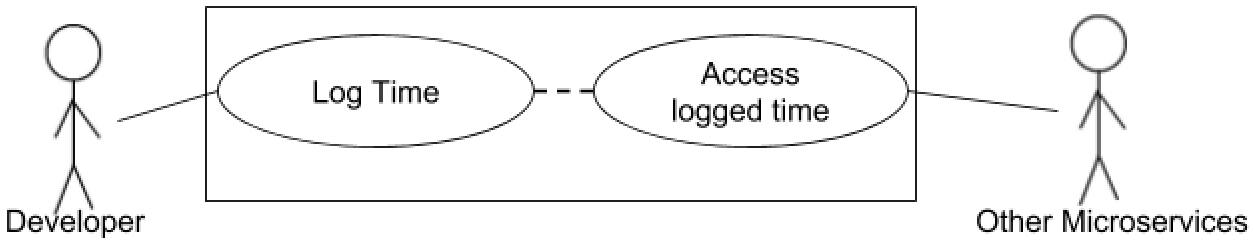


Figure 5.2.5: Work Recorder Use Case Diagram

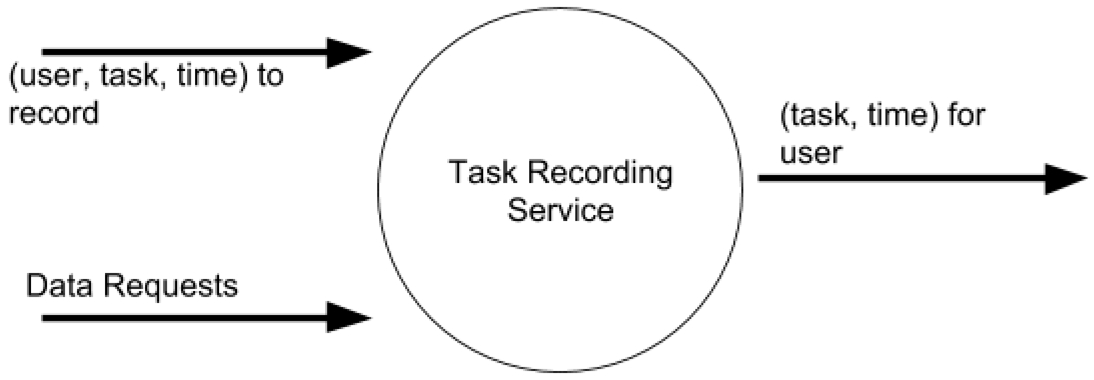


Figure 5.2.6: Work Recorder Context Diagram

Per the context diagram, the Task Recorder will receive records of tasks performed by a user, and receive requests for recorded information. The Task Recorder, on receipt of a request, will release information about performed tasks.

The API URIs for this package are described in Table 5.2.3.

Table 5.2.3: Work Recorder REST API Specification

URIs	GET	POST	PUT	DELETE
/api/	API Description --- < 200 (OK) `{“versions”:”v1”}`			
/api/v1/	Version 1 Description --- < 200 (OK) `{"status":"OK","message":"Work Recorder API version 1.0.0","response":null}"`			
/api/v1/users	Get list of users --- < 200 (OK) `[[username],...]`			
/api/v1/users/ [uname]	Get summary for user --- < 200 (OK) `{"username":[uname],"num_tasks":#,"first_task":[date],”average_tasks_per_day":#,"tasks":[#,#,#,#]}`			

/api/v1/users/ [uname]/ tasks	Get all tasks for user
	--- 	
	< 200 (OK)
	`[{"task_id":[taskid],"date":[date],"SLOC":[SLOC],"TDEV",[TDEV]},...]`
		Add a new task for user
		--- 	
		>
		`{"SLOC":[SLOC],"TDEV":[TDEV]}`
		--- 	
		< 201 (CREATED)
		`[taskid]`		

/api/v1/users/ [uname]/ tasks/ [taskid]	Get task specified by [taskid]
	--- 	
	< 200 (OK)
	`{"date":[date],"SLOC":[SLOC],"TDEV",[TDEV]}`
			Update task record
			--- 	
			>
			`{"date":[date],"SLOC":[SLOC],"TDEV",[TDEV]}`
			--- 	
			< 200 (OK)
			`[jobid]`
				Remove task specified by taskid from record
				--- 	
				< 204 (NO CONTENT)

The design of the python package implementing the API described in Table 5.2.3 is illustrated in Figure 5.2.7. There are two models used by the API to accomplish a task-a *user* model and a *task*. The work records and users are managed by the database class (*database.py*). The API is implemented in __init__.py which gets used by Flask to implement the REST API.

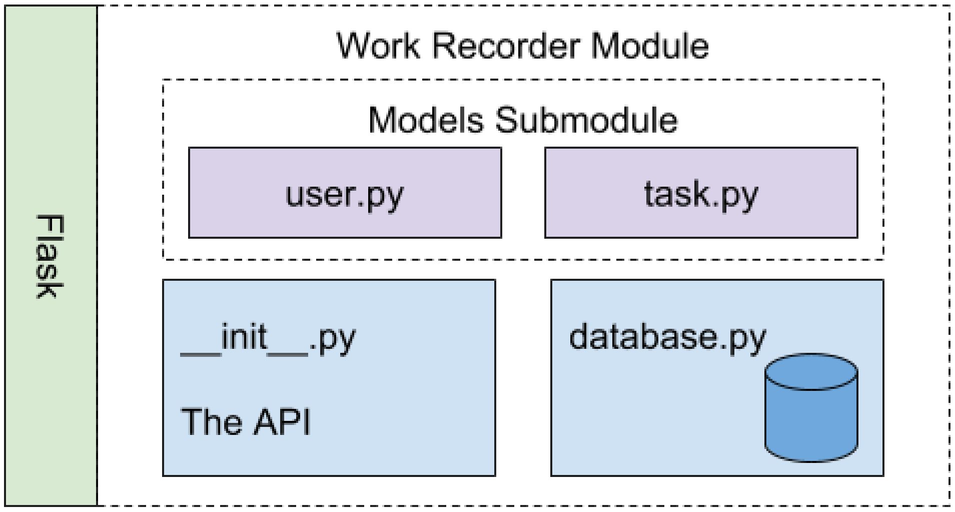


Figure 5.2.7: Work Recorder Design

Task Report Generator

The Task Report Generator is tasked with generating report of work performed for managers or other users. A use case diagram and context diagram for the work recorder can be found below in Figure 5.2.8 and 5.2.9, respectively.

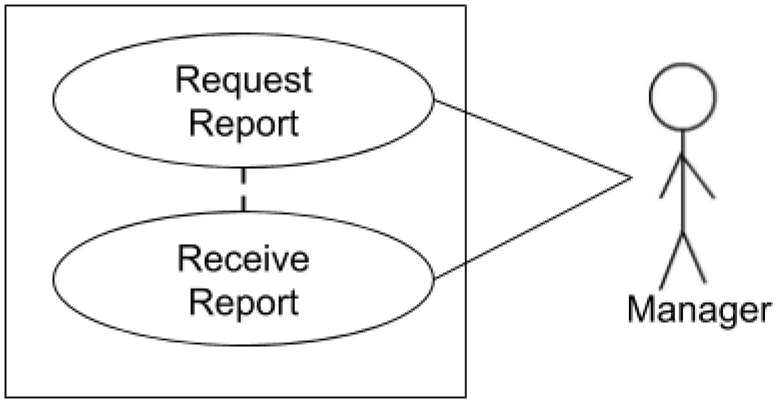


Figure 5.2.8: Use Case Diagram

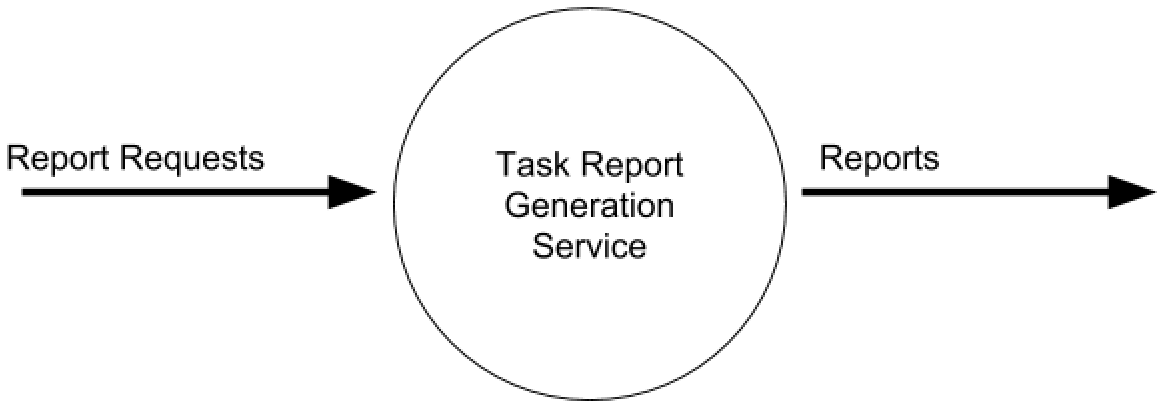


Figure 5.2.9: Context Diagram for Task Report Generator

Per the context diagram, the Task Report Generator will receive requests for reports. The Task Report Generator, on receipt of a request, will generate and return the requested report.

The API URIs for this package are described in Table 5.2.4.

Table 5.2.4: Work Report Generator REST API Specification

URIs	**GET**	**POST**	**PUT**	**DELETE**
/api/	API Description ——— < 200 (OK) `{“versions”:”v1”}`			
/api/v1/	Version 1 Description ——— < 200 (OK) `{"status":"OK","message":"Work Recorder API version 1.0.0","response":null}"`			
/api/v1/report?start=[date]&end=[date]	Get report for all users for dates between start and end ——— HTML Report			
/api/v1/report/user/[userid]?start=[date]&end=[date]	Get report for specified user (userid) for dates between start and end ——— HTML Report			

The design of the python package implementing the API described in Table 5.2.4 is illustrated in Figure 5.2.10. There are three models used by the API to accomplish a task- a *user* model, a *work_record*, and the *report*. The API is implemented in __init__.py which gets used by Flask to implement the REST API.

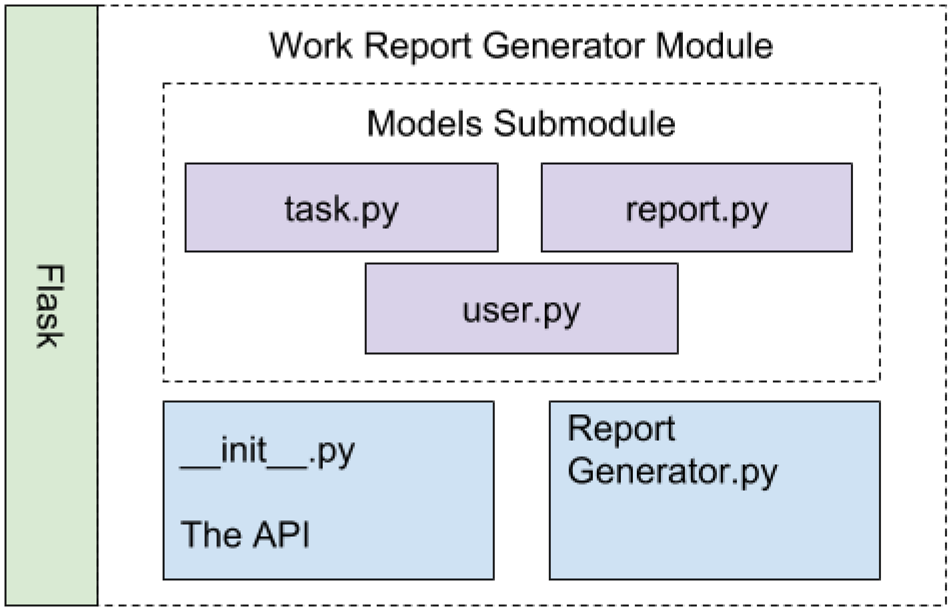


Figure 5.2.10: Work Report Generator Design

This chapter described the Quality Model and the metrics that will be used to measure Reusability, Flexibility and Maintainability to determine if each of the measures meets the standards of the Quality Model. Then the design of the Task Estimation microservice was described with the Quality Model standard applied to it. In the next chapter the outcome of the solution will be described.

6. Data Analysis and Discussion

This chapter is going to describe the data analysis of the solution and then discuss on improvements for the future. In Section 6.1, the authors will describe their observations on the experience when developing the microservice while using the Quality Model standard. Section 6.2 will analyze the metrics from the Quality Assessment and how it compares to the Quality Model Standard. Section 6.3 will describe the the metrics in comparison to the hypothesis. Lastly in Section 6.4, the authors will discuss on what can be done in the future to improve the Quality Model.

6.1. Output Generation

Developer Observations

The primary output for this effort was the recorded observations of the primary developers. These observations were made during the assessment of existing models, the creation of a hybrid model, and the creation and evaluation of the example services described in Chapter 5.

These raw, unfiltered observations were collected below:

- Comments by itself is not a meaningful metric. Some comments have more significance or quality than others. Unnecessary comments can reduce the quality of a software
- Creating meaningful variable or function names must be done be understanding the audience that will use the microservice. Some of the names can cause confusion to the average user.
- Agreeing on data commonality early in the design of the microservice helped developers understand what type of data each microservice needed.
- Modularity of functions occurred naturally. When the design is finished early then the developers have a greater understanding on what needs to be developed and know how to modularize the functions.
- Communication between developers is important to keep each other in check on the Quality Model
- Expandability should be an important characteristic- Is missing from quality model
- Some metrics are duplicated, can be combined

Quality Assessment of Example Services

Table 6.1.1: Quality Assessment for Estimate Generator

	Metrics	Results
QUALITY	Potential function modularization ratio (Potential Ratio) $PR = [Potential\ Modularization\ of\ Function/[Total\ Modules]$	No, modularized
	Modules dependent on others ● List the modules ● Determine if dependency can be removed	Dependent on organizations scale factor from Work_Recorder Cannot remove
	Ratio of dependent modules $DM = [Number\ of\ Dependent\ Modules] / [Total\ Modules]$	1/13
	Communication protocol documented and used?	Yes
	Number of communication protocol used.	1 (REST)
	Documentation describing data format used?	Yes
	Is there more than one data format used for input and output?	No
	Ratio of data format type conversion to the total modules that allows input and output $DC = [Number\ of\ Data\ Format\ Conversion] / [Total\ Input\ and\ Output$	0
	Documentation describing what requirements are needed?	Yes
	Are there hardware restrictions?	No
	Are there dependencies on third party libraries?	Yes(Flask, request)

	Variable names that are acronyms or shortened.
	● List them
	● Change to readable name when possible
		r = request
		e = error
		TDEV = time to develop
		Ksloc = thousand single line of code
		SE = scaled effort

	Comment Ratio (CR) (Bogner, 2017) $CR = 1 - [Lines\ of\ Quality\ Comments] / [Total\ Lines]$	34/184
	Number of module reference by other modules/total modules	1/4
	Can all modules be called independently	No

	Total Response of Service (TRS) (Bogner, 2017) *For each operation- weighted sum of operations/local methods called*	N/A
	Halstead's Measure (Halstead, 1977)	N/A
	Absolute Importance of the Service (AIS) (Bogner, 2017) *The [fraction] of clients that invoke one operation from the service*	2/12
	Absolute Dependence of the Service (ADS) (Bogner, 2017) *The [fraction] of services that service S depends on*	2/12
	Services Interdependence in the System (SIY) (Bogner, 2017) *The [fraction] of services that are bi-directionally dependent on each other*	0
	Clone Coverage (CC) (Bogner, 2017) $CC = [Duplicated\ Lines\ of\ Code] / [Total\ Lines]$	0
	Test Coverage (TC) (Bogner, 2017) $TC = 1 - [Lines\ Executed\ in\ Tests] / [Total\ Lines]$	1

Table 6.1.2: Quality Assessment for Work Recorder

	Metrics	**Results**
QUALITY	Potential function modularization ratio (Potential Ratio) $PR = [Potential\ Modularization\ of\ Function/[Total\ Modules]$	No, modularized
	Modules dependent on others ● List the modules ● Determine if dependency can be removed	__init__ is dependent on the models and database, and the database on the models
	Ratio of dependent modules $DM = [Number\ of\ Dependent\ Modules]\ /\ [Total\ Modules]$	2/4
	Communication protocol documented and used?	Yes
	Number of communication protocol used.	1 (REST)
	Documentation describing data format used?	Yes
	Is there more than one data format used for input and output?	No
	Ratio of data format type conversion to the total modules that allows input and output $DC = [Number\ of\ Data\ Format\ Conversion]\ /\ [Total\ Input\ and\ Output\ I$	0
	Documentation describing what requirements are needed?	Yes
	Are there hardware restrictions?	No
	Are there dependencies on third party libraries?	Yes (Flask)
	Variable names that are acronyms or shortened. ● List them ● Change to readable name when possible	No
	Comment Ratio (CR) (Bogner, 2017) $CR = 1 - [Lines\ of\ Quality\ Comments]\ /\ [Total\ Lines]$	9/252
	Number of module reference by other modules/total modules	5/16
	Can all modules be called independently	No
	Total Response of Service (TRS) (Bogner, 2017) *For each operation- weighted sum of operations/local methods called*	N/A
	Halstead's Measure (Halstead, 1977)	N/A
	Absolute Importance of the Service (AIS) (Bogner, 2017) *The [fraction] of clients that invoke one operation from the service*	5/16

	Absolute Dependence of the Service (ADS) (Bogner, 2017) *The [fraction] of services that service S depends on*	5/16
	Services Interdependence in the System (SIY) (Bogner, 2017) *The [fraction] of services that are bi-directionally dependent on each other*	0
	Clone Coverage (CC) (Bogner, 2017) $CC = [Duplicated\ Lines\ of\ Code] / [Total\ Lines]$	0
	Test Coverage (TC) (Bogner, 2017) $TC = 1 - [Lines\ Executed\ in\ Tests] / [Total\ Lines]$	1

Table 6.1.3: Quality Assessment for Work Report Generator

	Metrics	Results
QUALITY	Potential function modularization ratio (Potential Ratio) $PR = [Potential\ Modularization\ of\ Function]/[Total\ Modules]$	No, modularized
	Modules dependent on others ● List the modules ● Determine if dependency can be removed	__init__ is dependent on the three models. This cannot be removed
	Ratio of dependent modules $DM = [Number\ of\ Dependent\ Modules] / [Total\ Modules]$	1/4
	Communication protocol documented and used?	Yes
	Number of communication protocol used.	1 (REST)
	Documentation describing data format used?	Yes
	Is there more than one data format used for input and output?	No
	Ratio of data format type conversion to the total modules that allows input and output $DC = [Number\ of\ Data\ Format\ Conversion] / [Total\ Input\ and\ Output$	0
	Documentation describing what requirements are needed?	No
	Are there hardware restrictions?	No
	Are there dependencies on third party libraries?	Yes (Flask, request)

	Variable names that are acronyms or shortened.
	● List them
	● Change to readable name when possible
		r = request
		Sloc = software lines of code
		Tdev = time to develop

	Comment Ratio (CR) (Bogner, 2017) $CR = 1 - [Lines\ of\ Quality\ Comments] / [Total\ Lines]$	1/139
	Number of module reference by other modules/total modules	3/12
	Can all modules be called independently	No
	Total Response of Service (TRS) (Bogner, 2017) *For each operation- weighted sum of operations/local methods called*	N/A
	Halstead's Measure (Halstead, 1977)	N/A
	Absolute Importance of the Service (AIS) (Bogner, 2017) *The [fraction] of clients that invoke one operation from the service*	3/12

	Absolute Dependence of the Service (ADS) (Bogner, 2017) *The [fraction] of services that service S depends on*	3/12
	Services Interdependence in the System (SIY) (Bogner, 2017) *The [fraction] of services that are bi-directionally dependent on each other*	0
	Clone Coverage (CC) (Bogner, 2017) $CC = [Duplicated\ Lines\ of\ Code] / [Total\ Lines]$	0
	Test Coverage (TC) (Bogner, 2017) $TC = 1 - [Lines\ Executed\ in\ Tests] / [Total\ Lines]$	1

6.2. Output Analysis

The evaluation of the quality model identified some strengths and weaknesses of our example service. The identified strengths include the independence and consistency. The identified weaknesses include the comments and tests. For the most part the authors agree with the results of the analysis. The software could certainly be improved with the addition of tests (which the authors ran out of time to develop), and some comments. But they feel that there are some of the quality metrics are incomplete, missing (e.g., expandability), or redundant. For example, adding comments by itself is not a meaningful metric. Some comments have more significance or quality than others. Unnecessary comments can reduce the quality of a software.

There are certainly metrics that are more important for microservices than they are for the average software project. That by itself demonstrates the value of a microservice-targeted quality model.

6.3. Compare Output against Hypothesis

As described in Chapter 3, the author's hypothesis is the following:

Existing general software quality models are incomplete for assessing microservice quality and can be improved

The authors investigated general software quality models and the published works of expert engineers on microservice design and best practices. They used this to develop their own microservice quality model. They developed example microservices, recording their observations as they did so, and partially evaluated the software according to the model.

Through their own engineering judgement and observations, the authors were able to evaluate their model, and evaluate the completeness of general software quality models. They found that there are critical software attributes specific to microservices that are missing from general software models. From this evaluation they have evaluated their hypothesis as confirmed.

6.4. Discussion

After creating the quality model standard and implementing an example microservice with those standards included the authors discussed their observations. Then a quality assessment was done on the example services to determine how it compared to the quality model. The output of the quality assessment was analyzed and then compared it to the hypothesis. Lastly the authors discussed where improvements could have been made.

The authors do not present their software quality model as a complete or correct software quality model for microservices, instead they present it as an iteration on generic software quality models as applied to microservices, complete enough to confirm the hypothesis that general software quality models can be improved upon for microservices.

7. Conclusions and Recommendations

In this chapter the authors will summarize their research and findings. From that they have determined recommendations that future developers can do for this study.

7.1. Summary and Conclusions

For this project, the authors explored the best practices and tradeoffs in microservice design. They first conducted a literature survey, collecting the thoughts and practices of experts. The authors adapted these to create their own microservice quality model.

The authors then produced their own example microservices based on the specified, recording their impressions and thoughts (Section 6.1) as they did. They evaluated their microservices based on their quality model (also Section 6.1) and analyzed the results to adjust their quality model.

Through their own engineering judgement and observations, the authors were able to evaluate their model, and evaluate the completeness of general software quality models. They found that, though generic software quality models have their use, there are critical software attributes specific to microservices that are missing. For this reason it is important that developers and managers consider architecture-specific, language-specific, and tool-specific attributes when evaluating the quality of a software product.

7.2. Recommendations for Future Studies

A quality model is most effective when it is objective and quantifiable. The authors recommend that future developers continue to iterate on the work done in this project to improve the objectivity and quantifiability of the identified metrics. This could include resolving the comments mentioned in Chapter 6. Additionally, evaluation of the quality model at a service-level could bring additional insights.

There are certainly metrics that are more important for microservices than they are for the average software project. This could be represented in future work through a weighting factor on the NFRs. Currently, each NFR is considered equally.

Bibliography

Alvaro, A., Almeida, E. S., & Meira, S. R. L. (2005). Towards a software component quality model. In Submitted to the 5th International Conference on Quality Software. Chicago

Banerjee, S., & Jain, S. (2014). A survey on Software as a service (SaaS) using quality model in cloud computing. International Journal of Engineering and Computer Science, 3(01), 3598-3602.

Berkeley Student Environmental Resource Center. (2014, March 4). Prioritize Your Environmental Impact: A Guide You've Never Seen Before. Retrieved May 19, 2018. serc.berkeley.edu/prioritize-your-environmental-impact-a-guide-youve-never-seen-before

Boehm, B. W., Brown, J. R., & Kaspar, H. (1978). Characteristics of software quality. Chicago

Bogner, J., Wagner, S., & Zimmermann, A. (2017, September). Towards a practical maintainability quality model for service-and microservice-based systems. In Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings (pp. 195-198). ACM.

Bourque, P, & Fairley, R.E., eds. (2014). Guide to the Software Engineering Body of Knowledge, Version 3.0. IEEE Computer Society. www.swebok.org.

Cloud Academy. (2015, Nov 2015). Microservices architecture: advantages and drawbacks. Retrieved May 19 2018 from https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

Cohen, Steve. (2015, Jan 7). What challenges has Pinterest encountered with Flask? https://www.quora.com/What-challenges-has-Pinterest-encountered-with-Flask/answer/Steve-Cohen?srid=hXZd

Dharmendra Shadija, Mo Rezai, Richard Hill (2017). Microservices: Granularity vs. Performance. Retrieved May 20, 2018 from https://arxiv.org/abs/1709.09242

Dromey, R. G. (1995). A model for software product quality. IEEE Transactions on software engineering, 21(2), 146-162.

Dubey, S.K & Soumi Ghosh & Ajay Rana. (2012). "Comparison of Software Quality Models: An Analytical Approach," International Journal of Emerging Technology and Advanced Engineering, Volume 2, Issue 2, pp 111-119

Halstead, M. (1977), Elements of Software Science, Elsevier Computer Science Library, N.Y.,

IEEE. (2012). P730TM/D8 Draft Standard for Software Quality Assurance Processes, IEEE.

IEEE Standards Coordinating Committee. (1990). IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. CA: IEEE Computer Society, 169. Chicago

Impact Estimator Github Repository. (2018). Retrieved May 19, 2018, from https://github.com/teubert/impact-estimator

ISO/IEC. (2001). ISO/IEC 9126-1. Software engineering -- Product quality -- Part 1: Quality model.

ISO. (2011). ISO/IEC 25010:2011. Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.

Kryvtsov, Aleksey (2016, March 19). Service approach for development Rest API in Symfony2 .https://www.slideshare.net/savchenko1/symfony2-rest-api-59772368

Kshirsagar, Mahesh M. (2016, November 13) Containers in Enterprise, Part 2 : DevOps. https://blogs.msdn.microsoft.com/maheshkshirsagar/2016/11/13/containers-in-enterprise-part-2-devops/

Lee, J. Y., Lee, J. W., & Kim, S. D. (2009, December). A quality model for evaluating software-as-a-service in cloud computing. In Software Engineering Research, Management and Applications, 2009. SERA'09. 7th ACIS International Conference on (pp. 261-266). IEEE. Chicago

Massachusetts Institute of Technology. (2008, April 29). Carbon Footprint Of Best Conserving Americans Is Still Double Global Average. *ScienceDaily*. Retrieved May 19, 2018 from www.sciencedaily.com/releases/2008/04/080428120658.htm

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). Factors in software quality. volume i. concepts and definitions of software quality. GENERAL ELECTRIC CO SUNNYVALE CA.

Miguel, J. P., Mauricio, D., & Rodríguez, G. (2014). A review of software quality models for the evaluation of software products. arXiv preprint arXiv:1412.2977.

Pocoo. (2018). Flask. Retrieved May 19, 2018, from http://flask.pocoo.org

Sanders, Rachel (2014). Developing Flask Extensions. *PyCon*. Retrieved May 19, 2018, from https://www.youtube.com/watch?v=OXN3wuHUBP0#t=46

Universidade de São Paulo (USP). (2011). Product Quality - ISO/IEC 25010. Retrieved May 20, 2018, from edisciplinas.usp.br/pluginfile.php/294901/mod_resource/content/1/ISO%2025010%20-%20Quality%20Model.pdf

Wasson, M., Celarier, S. (2017, November 28) Microservices architecture style. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Wen, P. X., & Dong, L. (2013, September). Quality model for evaluating SaaS service. In Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on (pp. 83-87). IEEE.

Appendix A: Glossary of Terms

API	“Application programming interface”
Configurability	“The ability of the component to configurable.” (Alvaro, 2005)
DevOps	“Development Operations”
Extendability	The ease of adding additional capabilities to a software
Flask	A Python Module for web-services
Flexibility	The ease of adapting a software to perform a different task
Interoperability	“Attributes of software that bear on its ability to interact with specified systems.” (ISO, 2011)
Maintainability	“The degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications” (ISO, 2011) , (ISO/IEC, 2001)
Microservice	A small service with light interface that performs small task. Microservices are combined to perform larger, more complicated tasks.
Modularity	“The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.” (ISO, 2011)
Non Functional Requirements (NFR)	Requirements that do not relate to a function of the product. Also called *-ilities* based on their common use of the *-ility* suffix
Reliability	“The degree to which the software product can maintain a specified level of performance when used under specified conditions.” (ISO, 2011) , (ISO/IEC, 2001)
REST	“Representational State Transfer”
Reusability	“The degree to which an asset can be used in more than one software system, or in building other assets” (ISO, 2011)
Robustness	“The degree to which an executable work product continues to function properly under abnormal conditions or circumstances.” (ISO, 2011) (Dromey, 1995)
Quality Model	"The set of characteristics, and the relationships between them that provides the basis for specifying quality requirements and evaluation” (ISO/IEC, 2001)
Quality of Service (QoS)	The ability of a software service such as a SaaS to meet demands on the system

Scalability	“The ease with which an application or component can be modified to expand its existing capabilities. It includes the ability to accommodate major volumes of data.” (Dromey, 1995) (Alvaro, 2005)
Security	“The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure” (ISO, 2011)
Software As-A-Service (SaaS)	Software architecture where software functionality is provided to users through a web api or browser
Testability	“The degree to which the software product enables modified software to be validated” (ISO, 2011)
Usability	The ease with which the software can be used

Appendix B: ReadMe

Test-Microservices

These are a set of microservices for the SCU Cloud Computing Class

Getting Started

To setup your environment run the script *setup_env.sh*

Starting the microservices

To startup the microservices run the script *startup.sh*. NOTE: This script uses xterm, so make sure you have xterm forwarding enabled by adding the *-X* flag to your ssh command when connecting.

Appendix C: Project Source Code

setup_env.sh

```
/opt/python-3.4/linux/bin/python3 -m virtualenv env
source env/bin/activate
pip install -r requirements.txt
```

requirements.txt

```
click==6.7
Flask==1.0.2
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
Werkzeug==0.14.1
```

startup.sh

```
export FLASK_ENV=development
source env/bin/activate

export FLASK_DEBUG=1

export FLASK_APP=estimate_generator
xterm -T 'Estimate Generator' -e flask run --port 12001 &

export FLASK_APP=estimate_report_generator
xterm -T 'Estimate Report Generator' -e flask run --port 12002
&

export FLASK_APP=work_recorder
xterm -T 'Work Recorder' -e flask run --port 12003 &

export FLASK_APP=work_report_generator
xterm -T 'Work Report Generator' -e flask run --port 12004 &
```

Appendix D: Estimate Generator

__init__.py

```
from flask import Flask, abort, request
from estimate_generator.cocomo import Cocomo
from estimate_generator.org_factor_generator import OrgFactorGenerator

# Meta Data
__name__    = 'Estimate Generator'
__version__ = '1.0.0'
__summary__ = 'A microservice for calculating time to develop in months'

app = Flask(__name__)
cocomo = Cocomo()
estimate_record = OrgFactorGenerator()

@app.route('/api', methods=['GET'])
def versions():
    '''Get supported api versions'''
    return "{\"versions\":[\"v1\"]}"

@app.route('/api/v1', methods=['GET'])
def v1():
    '''Get details of version 1'''
    return "{\"status\":\"OK\",\"message\":\"Estimate Generator API version
1.0.0\",\"response\":null}"

@app.route('/api/v1/TDEV', methods=['POST'])
def generate_TDEV():
    ''' Obtain time to develop based on anticipated single lines of code in
thousands and complexity factor'''
    app.logger.debug(request.args)
    req_json = request.get_json()

    if 'TDEV' in req_json:
        data = req_json['TDEV']
        ksloc = data['ksloc']
        scalefactor = data['scalefactor']
        TDEV = cocomo.getTDEV(ksloc, scalefactor)

        org_factor = estimate_record.get_org_factor()

        org_TDEV = round((TDEV*org_factor),3)
    return "{\"status\":\"OK\",\"message\":\"Obtained time to
develop\",\"response\":" + str(org_TDEV) + "}"
```

cocomo.py

```
import json
import math

class Cocomo:
    '''Comomo ii equations used to determine estimated TDEV(time to
develop).
       User provides the complexity factors and single lines of code in
thousands

        A, B, C, D        -- Calibration variables based off COCOMO II.2000.
        nominalEAF        -- Nominal Effor Adjustment factor.
        scale_factor_json -- Scale factor values based off COCOMO
II.2000.'''

    def __init__(self):
        ''' Initializing Cocomo.'''
        self.A = 2.94
        self.B = 0.91
        self.C = 3.67
        self.D = 0.28
        self.nominalEAF = 1.0
        self.scale_factor = json.loads("""{"PREC": {"verylow": 6.2,"low":
4.96,"nominal": 3.72,"high": 2.48,"veryhigh": 1.24,"extrahigh": 0},
        "FLEX": {"verylow": 5.07,"low": 4.05,"nominal": 3.04,"high":
2.03,"veryhigh": 1.01,"extrahigh": 0},
        "RESL": {"verylow": 7.07,"low": 5.65,"nominal": 4.24,"high":
2.83,"veryhigh": 1.41,"extrahigh": 0},
        "TEAM": {"verylow": 5.48,"low": 4.38,"nominal": 3.29,"high":
2.19,"veryhigh": 1.1,"extrahigh": 0},
        "PMAT": {"verylow": 7.8,"low": 6.24,"nominal": 4.68,"high":
3.12,"veryhigh": 1.56,"extrahigh": 0}}""")

    # currently using nominal EAF for simplicity
    def getEffortPM(self, ksloc, E):
        '''Calculates the effort measured in person-month
            Returns value on effort '''
        effort = self.A * self.nominalEAF * math.pow(ksloc, E)
        return effort

    def getTDEV(self, ksloc, complexity_factor):
        '''Calculates time to develop based on ksloc and complexity
factor'''
        overall_scale_factor = 0.0

        for key,value in complexity_factor.items():
            overall_scale_factor += self.scale_factor[key][value]

        E = self.B + 0.01 * overall_scale_factor

        effort = self.getEffortPM(ksloc, E)
        SE = self.D + 0.2 * (E - self.B)
        TDEV = self.C * math.pow(effort, SE)
        TDEV = round(TDEV,3)
        return TDEV
```

org_factor_generator.py

```
import requests
from enum import Enum
import ast
work_recorder = 'http://127.0.0.1:12003/api/v1'

class Levels(Enum):
    '''Levels enumeration'''
    VERY_LOW    = 0
    LOW         = 1
    NOMINAL     = 2
    HIGH        = 3
    VERY_HIGH   = 4
    EXTRA_HIGH  = 5

class OrgFactorGenerator:
    '''Organizational Factor Generator class
      used to generate an estimate of the organization adjustment factor
      based on historical data. The work_recorder service is queried for
      historical data. The historical data is compared with estimates to
      generate the org adjustment factor.

      Instance variables:
        tasks   -- Records of what tasks from the historical record are
                   included in the calculation
        estimate_record
                -- Record of estimates by scaling factor'''
    class EstimateRecord:
        '''The estimate record
          Instance variables
            estimates    -- A 5-dimensional list of estimates, each dimension
                            is a scaling factor. Scaling factors are recorded
                            in the following order:
                PREC: Precedentness (similarity to previous jobs)
                FLEX: Development Flexibility
                RESL: Architecture/Risk  Resolution
                        (including thoroughness of risk management)
                TEAM: Team Cohesion
                PMAT: Process Maturity'''
        class Estimate:
            '''Estimate for a single set of possible scaling factors

              Instance variables:
                sum          -- sum of all estimates for those scaling
factors
                num_estimates-- Total number of estimates'''
            def __init__(self):
                '''Initialize estimate'''
                self.sum = 0
                self.num_estimates = 0

        def add_estimate(self, estimate):
                '''Add a single estimate'''
                self.sum = self.sum + estimate
                self.num_estimates = self.num_estimates + 1

        def get_estimate(self):
                '''Get estimate'''
                if self.num_estimates is 0:
                    return 1 # default
                else:
                    return self.sum/self.num_estimates

        def __init__(self):
            '''Initialize Estimate Record'''
            self.estimates = []
            for i in range(6):
                self.estimates.append([])
                for j in range(6):
                    self.estimates[i].append([])
                    for k in range(6):
                        self.estimates[i][j].append([])
                        for l in range(6):
                            self.estimates[i][j][k].append([])
                            for m in range(6):
                                Self.estimates[i][j][k][l].
                                    append(self.Estimate())

        def get_estimate(self, PREC=Levels.NOMINAL, FLEX=Levels.NOMINAL,
RESL=Levels.NOMINAL, TEAM=Levels.NOMINAL, PMAT=Levels.NOMINAL):
            '''Return an estimate for specific scaling factor values'''
            return
self.estimates[PREC.value][FLEX.value][RESL.value][TEAM.value][PMAT.value].get
_estimate()

        def add_estimate(self, estimate, PREC=Levels.NOMINAL,
FLEX=Levels.NOMINAL, RESL=Levels.NOMINAL, TEAM=Levels.NOMINAL,
PMAT=Levels.NOMINAL):
            '''Add an estimate to the record'''
          self.estimates[PREC.value][FLEX.value]
                 [RESL.value][TEAM.value][PMAT.value].add_estimate(estimate)

    def __init__(self):
        '''Initialize new OrgFactorGenerator'''
        self.tasks = set()
        self.estimate_record = self.EstimateRecord()

    def get_org_factor(self, factors = {}):
        '''Generate estimate of given factors'''
        try: # Check for updates
            r = requests.get('{}/users'.format(work_recorder))
            for username in ast.literal_eval(r.text):
               r = requests.get('{}/users/{}'.format(work_recorder, username))
               user_tasks = ast.literal_eval(r.text)['tasks']
               for task_id in user_tasks:
                  if task_id not in self.tasks:
                     r = requests.get('{}/users/{}/tasks/{}'.
                             format(work_recorder, username, task_id))
                        task = ast.literal_eval(r.text)

                        if task is not None:
                            # TODO(CT): Get estimate
                            estimate = task['SLOC']/2.5
                            for factor in task['Factors']:
                                task['Factors'][factor] =
                                   Levels(int(task['Factors'][factor]))
                            self.estimate_record.add_estimate(
                             task['TDEV']/estimate, **task['Factors'])
                        self.tasks.add(task_id)

            # Generate estimate
            return self.estimate_record.get_estimate(**factors)
        except:
            # On exception - don't use historical data
            return 1
```

Appendix E: Work Recorder

__init__.py

```python
from flask import Flask, abort, request
from work_recorder import database
from work_recorder.models.task import Task

# Meta Data
__name__     = 'Work Recorder'
__version__ = '1.0.0'
__summary__ = 'A microservice for maintaining a record of tasks completed'

app = Flask(__name__)
database = database.Database()

@app.route('/api', methods=['GET'])
def versions():
    '''Get supported api versions'''
    return "{\"versions\":[\"v1\"]}"

@app.route('/api/v1', methods=['GET'])
def v1():
    '''Get details of version 1'''
    return "{\"status\":\"OK\",\"message\":\"Work Recorder API version
1.0.0\",\"response\":null}"

@app.route('/api/v1/users', methods=['GET'])
def get_users():
    '''Get all users'''
    return str(database.get_users())

@app.route('/api/v1/users/<username>', methods=['GET'])
def user_summary(username):
    '''Get summary of specified user'''
    return str(database.get_user(username))

@app.route('/api/v1/users/<username>/tasks', methods=['GET','POST'])
def all_tasks_summary(username):
    '''Get:     get all tasks for specified user
       Post:    add new task
    '''
    if request.method == 'POST':
        app.logger.debug(request.args)
        req_json = request.get_json()
        sloc = req_json['sloc']
        tdev = req_json['tdev']
        app.logger.debug('u:{}, loc:{}, t:{}'.format(username, sloc, tdev))
        new_task = Task(username, sloc, tdev)
        for factor in new_task.factors.keys():
           if factor in req_json:
             new_task.factors[factor] = ScaleFactor(int(req_json[factor]))
        return str(database.add_task(new_task))
    else: # Get all tasks
        return_message = '['
        tasks = database.get_tasks(username=username)
        for task in tasks:
            return_message = return_message + str(task) + ','
        return return_message[:-1] + ']'

@app.route('/api/v1/users/<username>/tasks/<task_id>', methods=['GET','PUT',
'DELETE'])
def task_summary(username, task_id):
    '''get  -- Get details of specified task
       put  -- Update task
       delete -- Delete task

    '''
    if request.method == 'PUT':
        app.logger.debug(request.args)
        req_json = request.get_json()
        sloc = req_json['sloc']
        tdev = req_json['tdev']
        new_task = Task(username, sloc, tdev, int(task_id)
        for factor in new_task.factors.keys():
           if factor in req_json:
              new_task.factors[factor] = ScaleFactor(int(req_json[factor]))
        database.update_task(new_task))

        return task_id
    elif request.method == 'DELETE':
        if database.delete_task(int(task_id)):
            return ('', 204)
        else:
            return ('', 404)
    else: # Get all tasks
        return str(database.get_task(int(task_id)))
```

database.py

```
from work_recorder.models.user import User
from work_recorder.models.task import Task

class Database:
    '''Database Manager Class

      Manages a database of recorded tasks and users.

      Instance variables:
        tasks   -- All recorded tasks
        users   -- Any user who has submitted a task (map of
username:user)'''
    def __init__(self):
        '''Initialize database manager'''
        self.tasks = {} # task_id:task
        self.users = {} # username:user
        self._next_id = 0 # Private counter

    def assign_task_id(self):
        '''Assign a new task id and iterate counter
          Returns assigned task_id'''
        task_id = self._next_id
        self._next_id = self._next_id + 1
        return task_id

    def add_task(self, task):
        '''Add a new task to the database

          Arguments:
            task    -- Task to be added to database
          Returns task_id'''
        # Add Task
        task.task_id = self.assign_task_id()
        self.tasks[task.task_id] = task

        # Manage User
        if task.username not in self.users:
            # New User
            self.users[task.username] = User(task.username)
        self.users[task.username].tasks.append(task.task_id)

        # Return Task_id
        return task.task_id

    def update_task(self, updated_task):
        '''Update existing task with updated_task. Replaces
          existing task of same task_id

          arguments:
            updated_task    -- Task object to replace existing
          returns True if replace was successful'''
        if updated_task.task_id in self.tasks:
            self.tasks[updated_task.task_id] = updated_task
            return True
        else:
            # Task doesn't exist
            return False

    def delete_task(self, task_id):
        ''' Delete an existing task

          arguments:
            task_id -- Id of task to be deleted
          returns True if delete was successful'''
        if task_id in self.tasks:
            del self.tasks[task_id]
            return True
        else:
            # Task doesn't exist
            return

    def get_task(self, task_id):
        '''Get a task by task_id

          arguments:
            task_id -- Id of task to be returned
          returns task with id task_id, otherwise returns none'''
        return self.tasks.get(task_id)

    def get_user(self, username):
        '''Get a user by username

          arguments:
            username -- Username for user to be returned
          returns user with specified username'''
        return self.users[username]

    def get_users(self):
        '''Get all users
           returns list of users'''
        return [user for user in self.users]

    def get_tasks(self, username=None):
        '''Get tasks

          arguments:
            username    -- Name of user to get tasks for.
                           If none, returns all tasks
          returns task for user or all tasks'''
        if username is None:
            # Return all tasks
            return self.tasks
        else:
            # Return tasks for user
            return [self.get_task(task_id) for task_id in
self.get_user(username).tasks]
```

Appendix F: Work Report Generator Source Code

__init__.py

```python
from flask import Flask, request
import requests
import ast
from work_report_generator.models.task_report import TaskReport

# Meta
__name__    = 'Work Report Generator'
__version__ = '1.0.0'

app = Flask(__name__)
work_recorder = 'http://127.0.0.1:12003/api/v1'

@app.route('/api', methods=['GET'])
def versions():
    '''Return versions for api'''
    return "{\"versions\":[\"v1\"]}"

@app.route('/api/v1', methods=['GET'])
def v1():
    '''Return status of version 1'''
    return "{\"status\":\"OK\",\"message\":\"Work Report Generator API 
version {}\",\"response\":null}".format(__version__)


def _add_user_to_report(report, username):
    '''Add information for user to existing report'''
    r = requests.get('{}/users/{}/tasks'.format(work_recorder, username))
    for task in ast.literal_eval(r.text):
        report.add_task(task)
    r = requests.get('{}/users/{}'.format(work_recorder, username))
    report.add_user(ast.literal_eval(r.text))

@app.route('/api/v1/report', methods=['GET'])
def generate_report_for_all_users():
    '''Return html report of tasks for all users'''
    start = request.args['start']
    end = request.args['end']
    report = TaskReport(start, end)
    r = requests.get('{}/users'.format(work_recorder))
    for username in ast.literal_eval(r.text):
        _add_user_to_report(report, username)
    return str(report)

@app.route('/api/v1/report/user/<username>', methods=['GET'])
def generate_report_for_user(username):
    '''Return html report of tasks for single user'''
    start = request.args['start']
    end = request.args['end']
    report = TaskReport(start, end)
    _add_user_to_report(report, username)
    return str(report)
```

Appendix G: Models

user.py

```
from datetime import date # Used for today

class User:
    '''Model class for user

    Instance attributes:
        tasks      -- list of tasks belonging to user
        username   -- name of user
        first_task -- date of first task'''
    def __init__(self, username):
        '''Constructor for user model
            username-- name of user'''
        self.tasks = []
        self.username = username
        self.first_task = date.today()

    def __str__(self):
        '''Return string dict representation of user'''
        return '{{\'username\': \'{}\', \'num_tasks\': {}, \'first_task\':
\'{}\', \'average_tasks_per_day\': {}, \'tasks\': {}}}'.format(self.username,
self.num_tasks(), self.first_task, self.tasks_per_day(), str(self.tasks))

    def __eq__(self, other):
        '''Returns if users have same username'''
        return self.username == other.username

    def __ne__(self, other):
        '''Returns if users do not have the same username'''
        return self.username != other.username

    def add_task(self, task_id):
        '''Add a new task for user'''
        tasks.append(task_id)

    def num_tasks(self):
        '''Returns the number of tasks for user'''
        return len(self.tasks)

    def tasks_per_day(self):
        '''Return the average tasks per day for user'''
        return self.num_tasks()/((date.today() - self.first_task).days + 1)
```

task.py

```
from datetime import date # Used for today
from enum import Enum

class ScaleFactor:
    '''A scaling factor to describe job

        level   -- level of scaling factor (from Levels Enum)'''
    class Levels(Enum):
        '''Levels enumeration'''
        VERY_LOW     = 0
        LOW          = 1
        NOMINAL      = 2
        HIGH         = 3
        VERY_HIGH    = 4
        EXTRA_HIGH   = 5

    def __init__(self, level = Levels.NOMINAL):
        '''Initialize Scaling Factor
          level -- level of factor (default nominal)
                   can be passed in as Levels object or integer (0-5)'''
        if type(level) is int:
            self.level = ScaleFactor.Levels(level)
        elif type(level) is ScaleFactor.Levels:
            self.level = level

    def __str__(self):
        '''Return value (integer 0-5) as string'''
        return str(self.level.value)

class Task:
    '''Model class for task

        username   -- name of user owning task
        date       -- date of task
        SLOC       -- Lines of code
        TDEV       -- Time to develop (minutes)
        Factors    -- Scaling factors to describe job, each factor is
                      given a a level (see Levels Enum) or a corresponding
                      integer (0-5, where 5 is positive). The scaling
                      factors are stored as a dict. Each value is described
                      below:
                PREC: Precedentness (similarity to previous jobs)
                FLEX: Development Flexibility
                RESL: Architecture/Risk  Resolution
                        (including thoroughness of risk management)
                TEAM: Team Cohesion
                PMAT: Process Maturity'''

    def __init__(self, username, SLOC, TDEV, task_id = -1):
        '''Constructor for task
        username    -- name of user owning task
        SLOC        -- Lines of Code
        TDEV        -- Time to develop (minutes)'''
        self.task_id    = task_id
        self.username   = username
        self.date       = date.today()
        self.SLOC       = SLOC
        self.TDEV       = TDEV # Minutes
        self.factors    = {'PREC':ScaleFactor(), 'FLEX':ScaleFactor(),
            'RESL':ScaleFactor(), 'TEAM':ScaleFactor(), 'PMAT':ScaleFactor()}

    def __str__(self):
        '''Return str dict representation of task'''
        text =
'{{\"task_id\":{},\"date\":\"{}\",\"SLOC\":{},\"TDEV\":{},\"Factors\":{{'.fo
rmat(self.task_id, self.date, self.SLOC, self.TDEV)
        for factor in self.factors:
            text = text + '\"{}\":{},'.format(factor,
str(self.factors[factor]))
        return text + '}}'
```

task_report.py

```
from datetime import date

class TaskReport:
  '''TaskReport Class
        users   -- Users in report
        tasks   -- Tasks in report
        start   -- Start date of report
        end     -- End date of report'''
   usr_template = "<h3>{}</h3>{}<br/><br />{}<br /><hr>"
   usr_summary_template = "Total SLOC: {} <br/> Average SLOC/day: {}"
   usr_tbl_head_template = "<table><tr><th>Date</th><th>SLOC</th><th>Time
(min)</th></tr>"
   usr_tbl_row_template = "<tr><td>{}</td><td>{}</td><td>{}</td></tr>"

  def __init__(self, start, end):
    '''Initialize Task Report
            start   -- Start date for report
            end     -- Ending date for report'''
    self.users  = {}
    self.tasks  = {}
    self.start  = start
    self.end    = end

  def add_user(self, user):
    '''Add user to report'''
    self.users[user['username']] = user

  def add_task(self, task):
    '''Add task to report'''
    self.tasks[task['task_id']] = task

  def __str__(self):
    '''Return html report'''
    page = "<html><head><title>{} - {} Task
Report</title></head><body><h1>Work Report</h1><br />Generated: {}<br
/>Dates: {} - {}<br />Users: {}<br /><br />".format(self.start, self.end,
date.today(), self.start, self.end, list(self.users.keys()))
    for user_id in self.users:
      user = self.users[user_id]
      sloc = 0
      usr_tbl = TaskReport.usr_tbl_head_template
      for task_id in user['tasks']:
        task = self.tasks[task_id]
        sloc = sloc + task['SLOC']
        user_tbl
=usr_tbl+TaskReport.usr_tbl_row_template.format(task['date'], task['SLOC'],
task['TDEV'])
      usr_tbl + '</table>'
      usr_summary = TaskReport.usr_summary_template.format(sloc,
(sloc/user['num_tasks'])*user['average_tasks_per_day'])
      page = page + TaskReport.usr_template.format(user['username'],
usr_summary, usr_tbl)
    return page + "</body></html>"
```