2018-04-25

Performance and Portability of Cloud-based Computer vision: A Software Case Study

Salehian-Dardashti, Soheil

UNIVERSITY OF CALGARY

Performance and Portability of Cloud-based Computer vision: A Software Case Study

by

Soheil Salehian-Dardashti

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA
April, 2018

Abstract

The proliferation of visual sensors in terms variety and magnitude, has commenced an era where rich extensive visual data can be extracted and analyzed with the goal of a better understanding of the world around us. The forefront of this visual data analysis, is the field of computer vision that traditionally has been utilized in centralized systems but in recent years, has experienced an evolution towards a distributed cloud environment. The objective of this work is to analyze the software engineering processes that allow a rapid conversion of a centralized computer vision system to a cloud based distributed one. The work entails in developing the centralized algorithm in the specific realm of driver fatigue detection, analyzing its properties and designing an engineering methodology along with implementing the conversion from centralized to distributed computer vision.

The focus of the thesis is on the implementation and real world metrics that represent the challenges in performance and portability attributes of most distributed computer vision systems. The proposed methodology's merit is demonstrated through the blink detection use case study. Using requirement gathering techniques, the metrics are defined and compared at different stages of development to ensure minimal effort in porting the blink detection system. A generalizable message based architecture is proposed and implemented for a subclass of computer vision algorithms as a stepping stone for future analysis of other computational patterns that are associated with computer vision algorithms and systems.

Acknowledgements

Firstly, I would like to thank Dr. Smith for being a first believer in my abilities to bring me to his lab and Dr. Moussavi's teachings in doing great work. But most importantly, I am immensely grateful to Dr. Far for his unconditional support and patience in my journey to this point. Without the support of my wife Arlette, this milestone would not have been possible nor as enjoyable.

Soheil Salehian-Dardashti

Calgary

March 2018

Contents

Acknowledgements ii

List of Figures **viii**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Objectives 3

1.3 Thesis Outline 3

1.4 Thesis Contributions 4

2 Background **6**

2.1 Computational Models in Computer Vision 6

2.1.1 A Taxonomy of Patterns 7

2.1.2 Centralized Computer Vision 13

2.1.3 Computer Vision in The Cloud 15

2.1.4 Data vs. Algorithm: Perspectives On Performance Metrics . . 21

2.2 Software 21

2.2.1 Software Architecture 22

2.3 Quality Attributes 29

2.3.1 Modifiability General Scenarios 30

2.3.2 Modifiability Tactics 30

2.3.3 Design for Modifiability 31
2.4 QA: Performance 32
2.4.1 Performance General Scenario 33
2.4.2 Performance Tactics 34
2.4.3 Design for Performance 34
2.5 System's Quality Attributes 35
2.6 AMQP Message Based Architectures 36
2.7 RESTful Application Program Interfaces (APIs) 38
2.8 Concurrency in CV Pipelines 41

3 Use Case Algorithm 45
3.1 Introduction 45
3.2 Methods 47
3.2.1 Algorithm Overview 47
3.2.2 Face Detection 48
3.2.3 Eye Band Detection 52
3.2.4 Canny Edge Detection 53
3.2.5 Contour Extraction & Analysis 55
3.2.6 Ellipse Fitting 57
3.2.7 Global Thresholding Of Negative 59
3.2.8 Histogram Analysis 60
3.2.9 Merging of Results 60
3.3 Real-time Implementation 61
3.4 Algorithmic Results 62
3.5 Algorithm Discussion 63
3.6 Algorithm Conclusion & Future Work 65
3.7 Algorithm's Properties As A Use Case 66

4 Methodology: System Development 68
4.1 System's Scope & Requirements 69

4.1.1 Requirements Gathering 69
4.1.2 Metrics 72
4.1.3 Basic Portability (BP) 77
4.1.4 Weighted Portability (WP) 78
4.1.5 Service Communication Portability (SCP) 79
4.1.6 Portability/Performance Profiling 79
4.2 Languages & Tools 80
4.2.1 C++: OpenCV 80
4.2.2 Golang: Built-in System Concurrency & Error Handling . . . 84
4.3 System Design 85
4.3.1 Components 85
4.3.2 Life of a CV Request 87
4.4 Development Work-flow 89
4.5 Validation 92

5 Experiments & Results 94
5.1 Experiment Design 94
5.1.1 Experimental Data 95
5.1.2 Instrumentation for Performance 96
5.1.3 Portability Analysis 97

6 Conclusions & Future Work 104
6.1 Result Analysis 105
6.1.1 Guidelines 106
6.2 Future Work 108

A Software Architecture 110
A.1 Structures in Software Architecture 110
A.2 Benefits of Software Architecture 111
A.2.1 SA Abstraction 111

A.3 Importance of Software Architecture 112
A.4 Structure Views as Form of Training 113
A.5 Effects of SA Beyond the Software System 113
A.6 Software Architecture Activities 113
A.6.1 Quality Attributes 114
A.6.2 Requirements to Quality Attributes 114
A.6.3 SA Perspective on Quality Attributes 114
A.6.4 QA Scenarios 115
A.6.5 QA Requirements 115
A.6.6 Embedding QAs Into Design Decisions 116
A.7 QA: Availability 117
A.7.1 Techniques Against Failure 117
A.7.2 Availability General Scenario 118
A.7.3 Availability Tactics 119
A.8 QA: Interoperability 120
A.8.1 Reasons to Use Interoperability 121
A.8.2 Interoperability General Scenarios 121
A.8.3 Interoperability Tactics 121
A.8.4 Standards and Role of Semantics 122
A.8.5 Design for Interoperability 122
A.9 QA: Modifiability 123
A.9.1 Modifiability General Scenarios 124
A.9.2 Modifiability Tactics 125
A.9.3 Design for Modifiability 126
A.10 QA: Performance 127
A.10.1 Performance General Scenario 127
A.10.2 Performance Tactics 128
A.10.3 Design for Performance 128
A.11 QA: Security 129

A.11.1 Security General Scenario 130
A.11.2 Security Tactics 131
A.11.3 Design for Security 131

Bibliography **133**

List of Figures

1.1 The three pillars of the intelligent transportation system super project. . 2

2.1 Taxonomy of Patterns of Computation courtesy of [1]. Highlighted sections in pink refer to the most relevant to computer vision and this work. . . 10

2.2 A general computer vision pipeline representation. 11

2.3 The three elements of a CV cloud computing system with different type of users and camera settings. 16

2.4 Difference of scaling strategies with cloud computing and distributed systems. 42

2.5 The main paradigms of frameworks built for machine learning in the cloud. Image courtesy of [2] 43

2.6 Some of the companies that currently offer computer vision as a service APIs in the cloud. Note: The logos have hyperlink enabled. 43

2.7 Basic elements of an AMQP message based system. Image courtesy of [3] 44

3.1 Flow chart representation of the proposed blink detection algorithm. Please note the two parallel sub-algorithms and the combination of their results. 49

3.2 Some of the Haar features for edge, center and line detection. Please note that using only the horizontal, vertical and one orthogonal features all other orientation may be constructed. 50

3.3 Results of face (blue) and eye band (green) ROI detection using Haar-classifiers. The classifier has worked well in in identification of the eye ROI regions while the eyes are closed in (b). 52
3.4 Results of canny edge detection using $lowThreshold = 60$ and $maxThreshold = 140$ for both open and closed cases. Please note that while closed, the detector has only detected the shadow area around the eyes. 55
3.5 Results of the ellipse fitting procedure using discriminant of $\theta = 10°$ and $contourArea = 20$ pixels. The red and yellow line are the ellipse fitting procedure while the green box is the rectangular fit for orientation analysis. 58
3.6 Results of global thresholding, $\tau = 185$ on the negative of frames for both open and closed cases. Please note that the thresholding method has performed well in detecting the eyelid covering the eye but the areas around the eyes are still present. 59
3.7 Results of global thresholding, $\tau = 185$ on the negative of frames for both open and closed cases. Please note the decrease of the number high intensity pixels in (b). 60
3.8 Comparions of our Neon optimized RGB to gray scale conversion compared to native OpenCV C++ performance. 62

4.1 Methodology for homogeneous performance metrics generation in distinct environments. 84
4.2 Basic high level architecture and components of the developed system. . 88
4.3 The development work-flow utilized in this project. 93

5.1 Basic portability (BP) of the final developed system and the contribution of each component. 98
5.2 Weighted portability (WP) of the final developed system and the contribution of each component. 100
5.3 Service Communication Portability (SCP) of the final developed system and the contribution of each component. 100

5.4 The tracking of BP for each component during the development process. 102

5.5 The tracking of WP for each component during the development process. 103

5.6 The tracking of SCP for each component during the development process. 103

List of Tables

2.1 Timeline of Computer Vision Based on Performance Enchantment Technologies. 14

2.2 Relevant quality attributes to the system design of this work with their corresponding computer vision context. Based on previous experience in building computer vision systems (emperical) in the intelligent system proposed, and to limit the scope of this work, portability and performance were chosen as the critical requirements. 35

2.3 Common REST methods on resources in a RESTful web service. 40

3.1 Results of proposed blink detection algorithm in different sequences of frames in moderate ($s1-s5$), high ($s6-s9$) and low illumination ($s11-$s12) conditions. Accuracy and precision of each sequence and the total average has been shown. 64

4.1 Proposed system's functional requirements and their impact on system design. 71

4.2 An example of Severity of each element for 76

4.3 The service communication portability contribution from each system component. 79

5.1 Experimental data used and their desirable properties for the purpose of this work. 96

5.2 Performance metrics generated as part of the experiments. Results are the mean for each frame of the video set. 97

5.3 The type of costs during the analysis of the basic portability metrics tracking at the final stage of development. 98

Chapter 1

Introduction

1.1 Motivation

The field of computer vision is rapidly transforming in the recent years. With the advancement of high performance hardware and efficient software engineering techniques, the classic cases of computer vision algorithms solely being used in single machines are evolving. This evolution in distributed computer vision is a response to the strong performance requirements of high volume and high velocity visual sources that can be from smart phones to dedicated high resolution cameras. Although, distributed systems are becoming a relatively mature area of research, the application of computer vision and distributed systems has not been well studied.

Being part of a larger research project at the Intelligent Software Systems Laboratory (with focus on intelligent transportation systems), the issue of transforming software systems from embedded centralized systems to a distributed cloud systems became the inspiration of this work. At the core of the intelligent software system was the exploration of sound software engineering methodologies that allow seamless porting from the embedded to cloud. The combination of performance constraints of embedded devices and the regulatory roadblocks in installing new hardware in vehicles were part of the reasoning in pursuing a cloud based system.

Figure 1.1 depicts the interaction of other systems of this cluster of projects

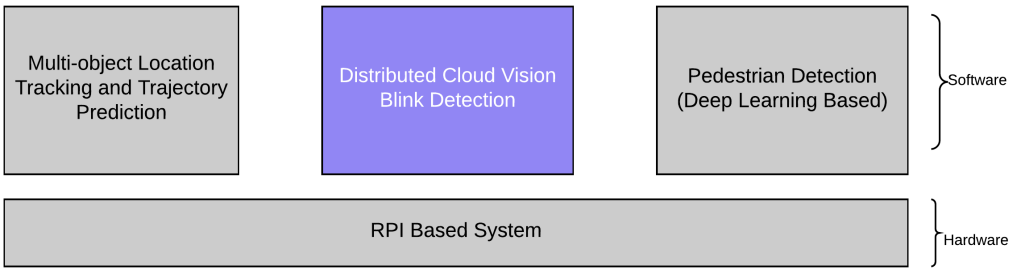


FIGURE 1.1: The three pillars of the intelligent transportation system super project.

that the work is part of. As it can be observed, the two other systems are in the development process with the goal of integration of a multi sensor, multi environment larger project in the future and hence a uniform and repeatable software engineering process is required.

At the time of this writing there has been a handful of work on the process of building distributed computer vision or general machine learning systems. Keutzer et al. [4] analyzed the taxonomy of patterns in machine learning computations and defined the requirements necessary to build systems that can provide the computational complexity. In other works the challenges of building distributed video processing cloud based systems were explored [5, 6]. With the advances of artificial intelligence and big data processing systems, the topic system design have become common place in both the academia [7] and industry (refer to Figure 2.6).

However often times, the challenges of the development process are not documented and analyzed. Often, the algorithms are developed first and then are passed on to software engineers to be rebuilt in production quality infrastructure. This has proven to be inefficient and studying the efficient methodology to improve this porting is the motivation of the following work. By studying the use case of a computer vision algorithm and documenting the process from building the algorithm, to porting to a distributed software architecture and finally validating its performance in the cloud, it is the hope of the author to explore the implications of the transformation from classic to distributed computer vision. Also, it is worth emphasising that although

there are currently distributed systems in the market, the documentation of software engineering methodologies that help in building such systems is rather lacking. This works intends to explore and propose a process for the cases where using already built cloud systems does not suffice.

1.2 Thesis Objectives

The main objective of this thesis is to explore and propose the methodologies in transforming a classic (centralized) computer vision algorithm to a distributed cloud based system. The main focus was in analyzing the work flow in terms of software design and architecture that improves the process for both computer vision scientists and software engineers. Although, the findings of the study can not be generally applicable to all computer vision algorithms, the work's other objective is to maintain the common problem constraints in many computer vision systems in order to study the properties of computer vision algorithms in distributed cloud systems. Achieving this objective through this work, will allow a closer look at the practical design and architectural decisions that are inherent in building a computer vision software engineering projects.

1.3 Thesis Outline

This thesis intends to explore the end to end process of building algorithms and transforming them into production ready distributed systems in the context of an intelligent transportation software system.

Chapter 2 describes the background information necessary in the field of computer vision with specific emphasis on the patterns of distributed systems in machine learning systems. In the second half of the chapter, the software engineering background such as the quality attributes and software architecture concepts are detailed.

In Chapter 3, the blink detection algorithm is described in detail with the intention

of building the necessary background in the properties that are desirable for the purpose of this work as a use case.

Chapter 4 outlines the system development methodology based on requirements and their metrics, the tools used to implement the system and the development work flow used to use instrumented metrics in guiding the building process. The chapter concludes with validation and acceptance criteria of the system based on the requirements gathered earlier.

Chapter 5, describes the experiments that were conducted to verify that the acceptance criteria has been met. This chapter also includes the experimental data and responses of the system based on the criteria previously defined.

Chapter 6 concludes the thesis with the analysis of the overall results of the experiments, the contributions of the methodology proposed and the future work associated with the possible next phase of this research.

There are two appendix chapters on software architecture background information and the testing methodology that was used during the process of experimentation in this work to both serve as possible references for the reader.

1.4 Thesis Contributions

The thesis explores the performance and portability concerns when transforming a centralized computer vision algorithm to a distributed system. The principle contributions are summarized as:

1. At the time of this writing, there are only a handful of existing studies on the development process of computer vision systems that have properties of near real-time and interoperable in a sensor fusion setting. This work attempts at discussing a methodology for developing a distributed computer vision system from an already existing developed algorithm using performance and portability metrics. In the opinion of this work, a metric driven approach enables robust tracking of the design and development trajectory of the various elements of

such systems as near performance requirements and the type of sensors used in the system evolve.

2. Findings on how a metric driven approach can be utilized for software engineering project management and resource allocation for future similar computer vision projects. Specifically, the benefits of tracking performance and portability in the study of the evolution of as a software project.

3. A novel blink detection algorithm (as part of a larger safety connected vehicle system) was developed to be the use case scenario of the process of transformation and scaling. The algorithm's real-time and multi sensor interoperability requirements were some of the deciding factors for the selection as a use case for this work.

4. A fully functional system was designed and developed (10k+ lines of code) during the analysis. The distributed computer vision system was developed using state of the art containerization, message passing and highly concurrent cloud based technologies in order to demonstrate the feasibility of the development process proposed. This would serve as documentation to explore the generalization of the approach for other types of machine learning and computer vision systems in future studies.

Chapter 2

Background

Computer vision systems are complex with a high requirement of understanding of a mixture of scientific fields. Furthermore, a solid background of their computational patterns, variety of deployment platforms (both hardware and software), and application context play a critical role in analyzing computer vision (CV) systems. In the following chapter, a brief overview of the three mentioned pillars of CV system design are presented. First, a taxonomy of of the most predominant computation models of computer vision is presented that opens the discussion to two fundamental deployment strategies of current computer vision algorithms. The second part of the chapter describes software concepts and tools that required in building a software based system proposed. The chapter concludes with a detailed description of the use case for the system which is a novel fatigue detection algorithm developed for the purpose of this study.

2.1 Computational Models in Computer Vision

This section discusses the major computational aspects associated with CV algorithms and serves as an overview of some of the inherent patterns of computation in CV algorithms. Studying patterns of computation has shown to be an effective way to build both software and hardware systems to resolve applications in many fields that

require high performance such as computer vision [8]. With the current trends in an explosion of visually rich data sources, the need for more computationally intensive systems are a reality. Hence, the study of patterns of CV workloads, such as the work of Chen et al. [9], has become an important area of research in computer vision.

2.1.1 A Taxonomy of Patterns

A design pattern is a generalizable solution of a reoccurring software problem that captures the essential basics of the solution for repeatable use or in many cases analysis. In order to understand the computational issues associated with computer vision algorithms in both singular and distributed forms, an analysis of computational patterns are a good starting point. Computational patterns categorize the levels and classes of computation present in an application, and in many cases, are generally distinguished based on their type of parallelism and system design implementation. In general, the patterns can be categorized what is computational (concurrent/parallel strategy patterns) and implementation patterns related to the data structures and their interaction in the system [1]. By describing these two types of pattern categories, a solid foundation of the nature of computer vision algorithms is established. Leveraging the work of Keutzer et al. [4] and as an extension of [1], this work will explore and build a taxonomy of patterns mostly relevant to task of moving a classic single machine CV algorithm or centralized CV algorithm and transforming it to distributed computer vision algorithm in the cloud.

Specifically, a CV application's computation may be be classified in two classes: structural patterns and computational patterns. This non-mutually exclusive classification, is based on both the algorithmic structure and the key computational aspects of the application. The *structural patterns* are more concerned with the high level organization of the computational elements similar to architecture styles as discussed in [10]. On the other hand, computational patterns are concerned with the type of computation that the application is mostly responsible for and define the various

type of computation that carried at each computational units of the application.

For instance, the general representation of pipes-and-filter is a common structural pattern that still requires a main computational model such as *dense linear algebra model* to complete the dichotomy of a CV application. It is important to mention that in many cases, both structure and computational patterns are present in serial and parallel applications [4], and are therefore sufficiently general to be applied to both centralized and distributed CV. Keutzer et al [4] designed a general pattern language for the purpose of describing the majority types of software centric workloads. In [1], Sundram looked to apply this pattern language to the field of computer vision with the intent of exploring the parallelization problem space. In the following section the most relevant patterns associated with centralized and distributed computer vision is explained in this context. Figure 2.1 summarizes the general taxonomy of these patterns with the highlighted areas as being the most relevant to explore for the background purposes of this work.

2.1.1.1 CV Structural Patterns

Some of the structural patterns used in the computer vision system proposed in this work are:

1. **Layered systems:** In this type of problem, the computer vision problem can be seen as natural evolution of layers on top of each other with the higher layers depending on the successive lower levels. The layered system pattern solution is to enforce separation of concerns. Only adjacent layers interact and each layer interaction is using interfaces [4]. This structure pattern allows for free change and high maintainability of the algorithm in terms of prototyping new computations as CV services [11].

2. **Map reduce:** For a general set of problems where there is a high level data parallelism and independence the data can be divided into many smaller parts, computed and the final result is an aggregation of the smaller computations.

The pattern identifies two essential steps by first mapping the independent sets of data and then in the second stage the results of each of the sets are reduced for the aggregation. The paradigm has become a standard in many computer vision applications in the cloud with large independent data sets where horizontal scaling to different machines in a distributed fashion are quite possible [12]. As it will be observed in the architecture section, a map reduce inspired pattern is used to divide the computer vision input data for the distributed CV architecture.

3. **Pipe-and-filter:** As one of the most common patterns in both computer vision and image processing, this pattern solves problems that are characterized by flow of data with transformations through each modular phase of computation. The computational elements (filters) are connected together using data communications mechanisms (pipes). The next section will describe this pattern in more depth as it is the foundation of processing portion of the proposed distributed CV system.

2.1.1.2 A Closer Look At Pipes-and-filters

This work defines a computer vision algorithm as a constructed model that through various transformations of an initial received visual data, extracts explicit meaningful descriptions that can be used to enhance visual perception [13]. The said transformations or "filters" can be observed as a series of pipes and filters similar to the pipes and filters software architecture pattern [14]. Based on this definition, the general CV pipeline is shown in Figure 2.2 based on [15].

Although describing the ever diverse set of of CV algorithms as a "pipeline" may be somewhat simplistic, the simple behaviours and interactions between pipes and filters assists in understanding the implementation of these highly complex models as software systems. As it will be observed, the pipeline concept will be used extensively coupled with concurrency patterns in the system development chapter of this work.

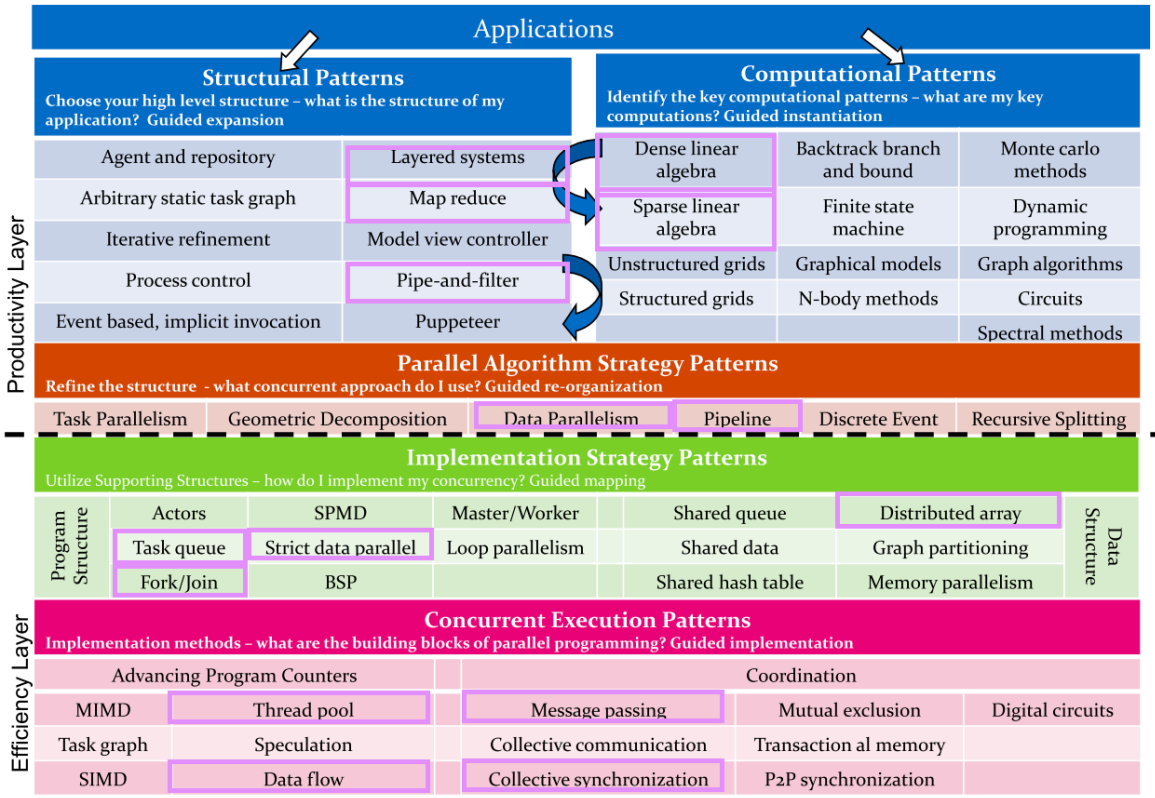


FIGURE 2.1: Taxonomy of Patterns of Computation courtesy of [1]. Highlighted sections in pink refer to the most relevant to computer vision and this work.

2.1.1.3 CV Computational Patterns

The computational patterns used in the proposed CV system can be categorized as:

1. **Dense linear algebra:** As expected, computational patterns associated with linear algebra are the most common form in computer vision. Dense linear algebra patterns resolve a rather small set of CV problems associated with linear operations to matrices and vectors with mostly non-zero elements. Due

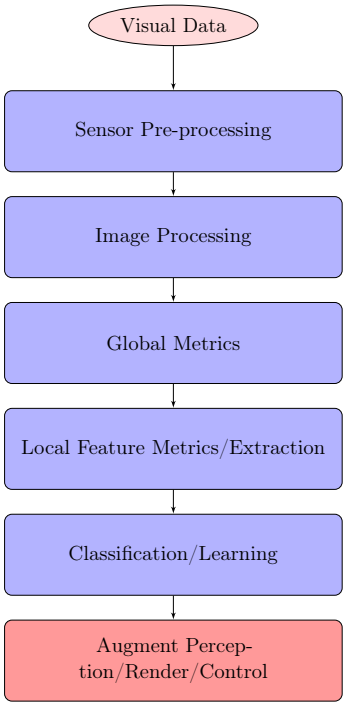


FIGURE 2.2: A general computer vision pipeline representation.

to regularity of the data access patterns these problems are well suited for pre-fetching to a single machine. Support vector machines (SVMs) and BLAS type problems are some of the use cases of this computational pattern.

2. **Sparse linear algebra:** More common than the dense counterpart, sparse linear algebra computations are used in linear solvers in object recognition, 3D reconstruction and other machine learning problems [1].

2.1.1.4 CV Concurrency/Parallelism Patterns

The patterns for optimizing performance of the CV algorithms can be described as follows:

1. **Data parallelism:** A set of problems where a single transformation/operation can be applied to each element of a data structure. For instance filtering, fast Fourier transform and binarization can be applied uniformly at each pixel

without a specific sequence. Exploiting this pattern, performance optimizations may be made to the CV algorithm.

2. **Pipeline:** This pattern looks at the data as a stream with transformations to apply to each element of the data. A strong pattern for concurrency, pipelines can be established using a set of stages in an assembly-line fashion. The depth of the pipeline is a critical factor in the amount concurrency achieved.

2.1.1.5 CV Concurrency/Parallelism Implementations

This section describes some of the patterns related to concurrency and parallelism that are part of implementation patterns. In other words, they are concerned with the implementation details of the previous patterns discussed in a software architecture. The implementation of the distributed CV system uses these patterns to enhance scalability and performance quality attributes discussed in the software section of the chapter.

1. **Task queue:** For problems that can be broken down into variable length executed tasks that need to run concurrently, the task queue is an effective implementation. The task queue's structure as a distributed queue is used extensively in order to scale the feeds of processing units. For more information on the details please refer section 2.6 of this chapter.

2. **Thread Pool**: A form of load-balancing, thread pooling allows for the reuse of threads without destruction and creation overheads. They are normally used in conjunction with task queues to achieve a better load distribution from the queue.

3. **Message passing:** A form of synchronization between tasks, this pattern allows communication between concurrent tasks without the need of shared memory or special hardware structures [4]. The message based architecture in section 2.6 explores the use of this pattern in more depth.

2.1.2 Centralized Computer Vision

Computer vision is a mix discipline of computer science, electrical engineering and artificial intelligent for the purpose of developing "machines that see". The field's conception came as part of Larry Robert's thesis at MIT in early 1960's, by exploring possibilities of extracting 3D geometrical information from 2D images. CV research continued in the field by solving low level vision problems (image processing tasks) such as edge/contour detection in the 70's [16]. The advancments of the fields followed by pattern recognition techniques and the incorporation of artificial intelligence in the late 70's mostly present in the defense sector. Marr et al. [17] was a pioneer in studying and using inspiration from the human vision system for the development of 3D computer vision systems for scene understanding. With the introduction of smaller video cameras in the mid 80's, studying motion became an important topic in computer vision research [18]. It was not until early 2000's where CV became much more accessible due to the performance of desktop computers andat the same time, the introduction of open source library OpenCV [19]. With high performance embedded vision systems, the same algorithms and techniques used on desktop only a few years back, were able to be used in mobile and small form factors in real world settings. As it can be observed in Table 2.1, the field's advancement has been historically strongly characterized by a follow up of strong advancement in the computing field. In other words, when there is a strong performance paradigm shift, new advancement and milestone follow in the CV field. As expected this has to do with the critical performance aspect of the CV systems. With the current revolution of deep learning in CV which has been primarily enabled by advancements in Graphic Processing Units (GPUs) [20], there is a strong trend of using scalable cloud systems for many machine learning tasks including CV [21], which will be revisited in more detail in the next section.

While CV technologies have undergone a rapid evolution, the design work flow of building real world systems have seen little change. The classical work flow for

Table 2.1: Timeline of Computer Vision Based on Performance Enchantment Technologies.

1960s	Larry Robert's Thesis on 3D geometry.
1970	Low Level Vision: Birth of Imaging Sensors.
1978	Marr's Bottom Up Approach: Stereo Vision.
1990s	CV and Motion: Processing Video.
2000s	OpenCV: Multicore and democratization.
2013	Deep Learning: GPUs.
Present	CV in the cloud: Distributed Cloud Computing for CV.

building computer vision algorithms has been historically a two step process. This process involves a general prototyping and porting flow: first, the algorithm scientist builds the algorithm in a desktop environment that allows rapid prototyping. Then with the help of a software engineer, the algorithm is transformed and ported over to the platform of choice. This two step process allows the algorithm scientist to work on a much higher level of abstraction without the implementation complexities of the software engineering and hardware details [22]. This dichotomy allows the software/hardware engineer to focus on a model specification that makes design trade-offs in size, energy consumption, performance and cost while the algorithm is being designed and validated at the same time.

This design methodology is used extensively for many computer vision algorithms in the embedded vision field which is the one most common form of centralized CV. In terms of drawbacks, the remote prototyping of algorithms require repeated validation stages and any changes to the system may require revalidating in both algorithm accuracy and general embedded constraint requirement analysis in order for the system to be production ready.

2.1.3 Computer Vision in The Cloud

An explosion of visual data is happening on the web. More than 72 hours of video is being uploaded every minute to Youtube and this same trend is being observed in other engineering and scientific fields where collection of video data is inherent such as astronomy and remote sensing [23]. With the proliferation of a variety of visual sensors and cameras, designing robust CV algorithms that can be portable on different platforms is becoming an active area of research in both the academia and the industry. In many of these cases, with the expansion of Internet of Things (IoT), these connected devices are becoming connected to the web and hence a new exciting design dimensionality has been opened in the field. Partitioning the computation of these devices locally vs. in the cloud has become an interesting research problem. On one hand, access to the performance advantages of the cloud can be beneficial to a CV system but at the same time, the reliability of the system may be dependant on the connections to the web and its corresponding technologies. With the connection of video capable sensors to the internet, smart phones capable of recording high quality video providing them to the cloud, a natural progression has become a trend to process this visual data in the cloud. It is the goal of this section to describe the state of the art in the use of cloud computing infrastructures, some of the advantages and challenges and in general and some of the differentiating factors of developing algorithms in such distributed environments.

Cloud computing use of distributed systems is considered to be part of disruptive technologies that can push existing CV algorithms to a new level performance. For the purposes of this work, CV cloud computing refers to both the ability to deliver a CV application as a service over the Internet and the infrastructure that entails both hardware/software that make providing this service possible [24]. The three basic components of most cloud computing system is shown in Figure 2.3.

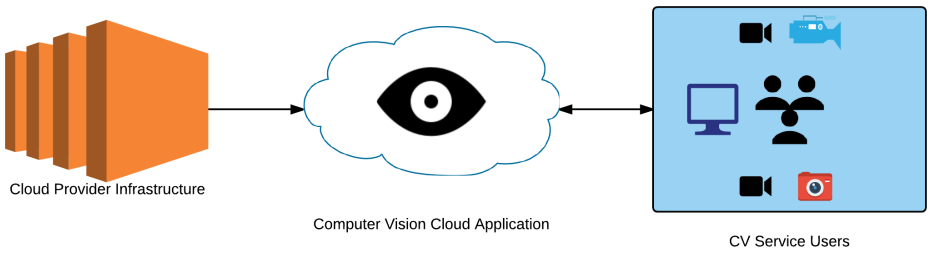


FIGURE 2.3: The three elements of a CV cloud computing system with different type of users and camera settings.

2.1.3.1 Vertical vs. Horizontal Scaling

The strong advantage in performance comes from the ability to scale algorithms as the data set grows (both in terms of quantity and size) with minimal modifications to the algorithm itself. A closer look at how scalability has been handled in the field of CV suggests that prior to cloud computing, a vertical scaling or "scale up" approach was required to deal with the performance demands of a CV system. This demand has been partly due a larger data set with larger data units (higher resolutions, higher number of frames, etc.) and more demanding algorithms (higher number of iterations, learning features, and generally more complexity). The "scale up" approach is based on upgrading the software/hardware components of a system to have higher performance.

Until recent years, vertical scaling was possible as performance of single cores were being doubled every year and half due to Moore's Law [25]. However, as the single chip performance deliveries slowed down, high performance applications such as computer vision have had to look into a different approach in scalability. Using cloud computing and elastic resources of distributed system, high performance applications can be horizontally scaled or "scaled out" where more cores and machines are mapped to do the computation instead of upgrading to a higher performance machine [25]. Figure 2.4 demonstrates these distinct approaches to scaling.

2.1.3.2 Previous Work

Although the application of cloud computing to computer vision is in its infancy stages, there has been considerable interest in the past years in designing distributed computer vision systems with cloud computing. Harnessing the enormous computing resources of cloud computing on demand is an attractive proposition in many fields including CV. The following section explores the existing literature on CV cloud computing.

Karimaa et al. [5] and Neal and Rahman [6] explored the implementation and

challenges of a cloud based video surveillance system. Similarly, in [26] the design of a scalable video recording system in the cloud was discussed. In particular, the main challenges and design points of cloud systems for video processing was explored in detail. A distributed system file system was used to resolve storage of video data which is reoccurring theme in surveillance system that are always-on.

In recent years, there has been a number of general purpose frameworks for distributed computing that are used for machine learning and computer vision tasks. In terms of these general frameworks, there has been strong efforts by Batra et al. [27] in democratizing computer vision algorithms through a web based application programming interface(API). In their work, the authors tackle the computer vision algorithm problems that can be inherently represented by cyclic graphs. These problems include image stitching, image classification and face recognition in images. The focus of their effort has been on building a general framework that can be integrated into robust machine learning distributed systems such as GraphLab [21] while the user of the algorithm only sends the image data. Batra et al. argue that computer vision researchers solve the same cloud computing infrastructure problems over and over and by the time these infrastructure problems are solved, a great amount of researcher resources have been spent not on the actual algorithm development. At the time of this writing most of the efforts has been on single images CV and not video processing.

Another set of frameworks that have been implemented to tackle the computer vision cloud computing problem are based on the MapReduce paradigm. A counter point to graph based problems, MapReduce problems have strong inherent data parallelism and unlike graph problems, they exploit this data parallelism by dividing the images/frames into smaller pieces and distribute them across many machines. The advantage here is that the relationship between images/frames are not as critical and therefore they can be merged back in the final step with relative ease. An open source framework implementation of MapReduce is Hadoop [28] and researchers at the University of Virginia have been successful at develoloping an abstraction level

knowns as the the HIPI framework [29] which serves as an interface for developing OpenCV based algorithms and batch processing using the Hadoop infrastructure. In their work, Sweeny et al. [12] argued that using the HIPI abstraction, the algorithm developer does not need to learn the rather complex details of setting up a Hadoop system. By first combining a large set of images into one large file and transforming into a similar input Hadoop form, HIPI is able to do large batch processing using algorithms developed in OpenCV.

Similarly, Yan and Huang [7] used Hadoop to build a large scale image processing engine for general CV applications. By using the streaming programming model of Hadoop, the system is able to do video processing with a high level of performance advantages. Based on the results, they were able to have a 2X speedup in very large data sets for a face detection algorithm that was run sequentially for comparison. In [30], leveraging Hadoop's storage capabilities, the authors were able to build a remote sensing image analysis system but due to limitations on the languages supported (only Java), computational performance was a shortcoming. Yamamoto [31], designed and implemented a cloud based architecture by developing a Ruby interface with Hadoop's mapper function in order to batch process a large data set of video files. In almost all these cases, the type of processing was mostly in a batch processing setting and the complexities of using the Hadoop ecosystem still required extra work by the algorithm developer.

A mapping of the landscape of CV cloud computing research suggests the presence of two paradigms: the first of which is utilizing data parallelism via the MapReduce patterns and the second using graph parallelism for problems that are do not have data parallel solutions. Figure 2.5 visually summarizes the differences of these two paradigms.

It is worth of note that the industry is also seeing a surge of computer vision as a service companies that are using CV in the cloud in order to make possible for many systems to use the algorithms and the corresponding application program interface (API). Figure 2.6 shows some of the companies that provide their CV APIs. Such

CV APIs allow for interoperable algorithms to integrate into other software systems much of the same way other APIs function. Interoperability is discussed in more detail in the software architecture section of this work.

2.1.3.3 Design Challenges

The scalable infrastructure, processing algorithms and web delivery mechanisms (such as an API) is what can be defined as cloud computer vision. By virtue of this definition, cloud computer vision has certain challenges that are distinct than the other deployment platforms for CV previously discussed. The following is an overview of some of these challenges associated unique to building a cloud computer vision system. Some of the cloud infrastructure challenges include [26]:

- **Data acquisition strategies:** The type of image, meta data and algorithm features being sent to the cloud (partitioning of algorithm responsibilities)
- **Cloud resource allocation for real time performance:** Resource management to ensure both design constraints and performance deadlines are met.
- **Algorithm processing:** Handling computer vision processing issues such frame rate, codec, algorithm speeds, concurrency and synchronization between CV workers.
- **Debugging and monitoring of operations:** Instrumentation and monitoring mechanisms to be able to debug in case of both algorithm and computational performance issues.
- **Storage mechanism:** how the images/videos, feature metadata are needed to be stored, what meta data
- **Security and privacy:** General security and privacy concerns associated with a public/private/hybrid cloud infrastructure.

2.1.4 Data vs. Algorithm: Perspectives On Performance Metrics

CV algorithm performance is dominated by the nature of the input/output data (data based metrics) and the computational complexities of the algorithm (algorithm based metrics) itself [1]. Some of the metrics strongly associated with computer vision system performance and their definition is described as:

1. Image resolution (RES): A data based metrics that defines the basic image/frame unit data of the system as the number of pixels on the grid.

2. Frames per second (FPS): Another data based metric identifying the number of frames that system is able to receive and process successfully at each second.

3. Memory read/write bandwidth (MemRead/MemWr): Algorithm based metrics that are associated with to the in memory usage of the image data by the algorithm.

4. Memory footprint (MemFoot): Another memory based metrics based on the algorithm's computational needs in keeping data in memory.

5. Memory access patterns (MemPatt): Algorithm based metric that monitors on what percentage of the algorithm memory interactions are writes and what percentage is reads at what stages.

6. CPU cycles/pixel (Cycle/Pixel): A macro metric defining the cost of each pixel in terms of CPU cycle.

7. Input output measure (IO): The metric defining disk related operations.

2.2 Software

In this section, an overview of the quality attributes of the proposed system and the corresponding set of software tools to fulfill those requirements is described. This includes an introduction to the quality attributes critical to designing the distributed

computer vision software system proposed, need for RESTful APIs, concurrency in real time web applications. Due to the nature of this work, the most relevant quality attributes specific to the system designed is explored in this section. For a more comprehensive overview of quality attributes, please refer to the Appendix chapter A.

2.2.1 Software Architecture

Software architecture's definition has been a topic of discussion since the early 70s. Specifically, a common misconception of the definition is that the architecture is solely based on early or major design decisions of the software system. Software architecture may be defined as:

"The software architecture of a system is the set of structures **needed to reason** about the system, which comprise software elements, relationship among them, and properties of both. [33]"

This definition emphasizes on the strong reasoning aspect of the discipline which implies the study of software architecture as an enabler in order to make sound software engineering tradeoffs at the design stage. This definition also suggests that architecture composes of a set of software structures that as a whole, define the behaviour of the system and its final attributes. Prior to discussing the uses of software architecture (SA) a more in depth understanding the building structures of an architecture is required which is discussed next.

2.2.1.1 Structures in Software Architecture

Structures in the software architecture context can be generally defined into three distinct categories:

1. Implementation units or *modules*: Modules assume computational responsibility as implementation units which can be classes, layers or mere division

of functionality. Modules are static and allow analysis on dependency of the internal software system such as inheritance and/or generalization relationships.

2. Dynamic units or *components/connectors*: Components/connector structures define how the system is to be structured with run-time behaviour (components) and interactions (connectors). These dynamic structures which can be services, peers, or servers allow for analysis of run-time behaviour of the software systems such as performance, security and availability [33].

3. Mapping units to non-software components or *allocation structures*: Allocation structures embody decisions on how the software system interacts with its environment such as CPUs, file systems, networks and development teams. Using allocation structures, the non-software environment interaction with the system can be viewed and analyzed in detail. Such through analysis is essential in the deployment of distributed systems.

2.2.1.2 Benefits of Software Architecture

Some of the benefits of a systematic approach to software architecture was briefly discussed in the previous section. In this section additional benefits are described in more detail.

2.2.1.3 SA Abstraction

SA can be viewed as an abstraction to cope with the complexities of today's software system design. SA consists of a number of structures and each structure includes elements (being a module or a component) and relations among those elements which contribute to the complexity. Often such elements have public and private properties and the job of SA is to abstract away the internal private properties that has no effect outside the element itself in order to look at the software system with elements, composition and relations in a holistic form [33]. SA abstraction inherently allows

for an explicit definition of the behaviour of the system and requires many aspects of the behaviour to be included and documented.

2.2.1.4 Importance of Software Architecture

Software architecture affects two important aspects of the system's quality attributes: initial quality attributes that become built-in to the system and secondly prediction of quality attributes. There are other important factor that contribute to the need for a through design of the software architecture. Some of these secondary factors include:

- SA as a reasoning tool for initial decisions and any change to the system implementation down the stream.
- SA allows stronger communication between stakeholder based on a single vocabulary and discussion point.
- SA puts forward the road map and constraints necessary for a high quality implementation
- Re-usability of software architecture (beyond element re-usability) has many positive consequences on time-to-market and maturity of new systems being designed.

These aspects and many others are the primary reasons why any software project regardless of its size and complexity benefits immensely from a architecture design stage.

2.2.1.5 Structure Views as Form of Training

The software architecture's ability to describe how elements interact with each other can be used as first introduction to the system effectively [33]. As a common reference point, the architecture views can serve to answer such inquires such as what elements do what, team assignment to system parts mapping (normally as part of the module

views). Conceptual views are effective in explaining on how the system is suppose to function and accomplish the task.

2.2.1.6 Effects of SA Beyond the Software System

Although there is a strong impact on the software system by SA, there are other ripple effects caused by SA. One of such "bi-directional" relationships is the effect of architecture on the organization structure. As Conway's Law eloquently described this phenomena where the "organizations that design systems...are constrained to produce designs which are copies of the communication structures of these organizations."

2.2.1.7 Software Architecture Activities

Often the SA activities can be summarized as:

1. Making the business case for the system
2. Exploring the architectural significant requirements
3. Software architecture development
4. Documenting and communication of the developed software architecture
5. SA evaluation/analysis
6. Implementation/testing of the system based on developed SA
7. Implementation conformance assurance to original SA

2.2.1.8 Quality Attributes

Quality attribute (QA) is defined as a "measurable ore testable property of a system that is used to indicate how well a system satisfies the needs of its stakeholders. [33]". Understanding the critical quality attributes that affect a software system's overall behaviour is an important criteria that the architect must poses. Software

architecture's support of quality attributes is via the mapping of the system's functionality to architectural structures (modular, conceptual, and allocation).

2.2.1.9 Requirements to Quality Attributes

A software may include various requirements but only some of these requirements can be designed into the system as quality attributes. Such requirements are simply known as quality attribute requirements which are distinct from requirements that describe what the system must do or how it must behave or functional requirements. Functionality of the system has the strongest relationship to the architecture itself because it is the ability of the system to perform the task that the system was designed to do. However it is important to note that functionality does not dictate architecture as the same functionality may have an endless number of corresponding architectures.

2.2.1.10 SA Perspective on Quality Attributes

Quality attributes have been of a much debated and researched topic in software engineering since the 1970s. As it was previously mentioned the mapping from requirements to quality attributes is an early task that is accomplished by the architect. However similar mappings to the software architecture has proven to be difficult due to the following issues:

1. Quality attributes solely are not testable and their definitions are not contextualized. For instance, a system may be robust to a a class of faults and brittle to other types.

2. Often there is a vast overlap among quality attributes in the system and their interactions make the job of the architecture a complex one. For example, a security breach to the system can be seen as both an aspect of availability and security.

3. Each attribute has its set of vocabulary and terminology which add complexity to the mapping of all important quality attributes to the architecture itself.

2.2.1.11 QA Scenarios

The concept of *quality scenarios* intends to resolve some of these issue by incorporating methods in order to quantitatively verify that a quality attribute exists and in which context. QA scenarios allow to divide QAs into attributes that are the property of the system at run-time (i.e. performance) and attributes that are property of the development of the system (i.e. modifiability).

2.2.1.12 QA Requirements

In order to discuss quality attributes in a homogeneous fashion, the following 6 characteristics are defined for a QA scenario:

1. *Sources of stimulus:* This is the event which arrives at the system and generates a form of stimulus.
2. *Stimulus:* The condition that requires a form of response from the system.
3. *Environment:* This describes the conditions at which the stimulus occurs, in other explicitly describing the context.
4. *Artifact:* The subsystem or system as whole that is stimulated.
5. *Response:* The activity of system as the result of receiving the stimulus.
6. *Response measure:* The Quantifiable measurement of the response of the system.

2.2.1.13 Embedding QAs Into Design Decisions

As previously mentioned, an important aspect of software architecture is its ability to apply sound design choices in order to come up with a collective set of engineering

tradeoffs of the quality attributes of the system that fulfills while fulfilling all business requirements. This collection of design decisions may be categorized as:

- Allocation of responsibility: It involves the identification of basic system functions, infrastructure to satisfy quality attributes and their allocation through static and run-time elements using modules and components.
- Coordination model: The mechanisms that collectively allow the interaction among software elements. Identifying which elements need communication, properties of the coordination, and defining communication mechanism are some of the design decisions.
- Data model: The interpretation of artifacts of system-wide interest or data as a collection may be refered to as a data model. Decisions such as, data abstractions, meta-data compilation, and data organization are associated with the data model.
- Management of resources: Decisions such as identification of resources and their limitations, system element to resource mapping, resource saturation analysis are part of this category.
- Mapping among architectural elements: An architecture provides mapping of element among each other in a given structure mapping but also the mapping of software elements to environment elements such as CPUs. The assignment of run-time elements, data model to data stores are included in this category.
- Binding time decisions: Such decisions establish the scope, the point in the life cycle and the mechanism of achieving variation for the rest of the decisions.
- Choice of technology: Deciding on which technology are available to realize the decisions of other categories, tools availability and the extent of familiarity of the team belong to this category.

2.3 Quality Attributes

The first important aspect of any software design flow is to understand the requirements and constraints of the problem at hand. This task also known as requirement analysis, involves defining some of the quality attributes (QA) that the design must balance through out both architecture design and implementation phases. In other words, QAs are non-functional requirements that describe and capture on how functionalities are achieved in the software system.

2.3.0.1 QA: Modifiability

Modifiability is an important quality attribute as studies have shown that the majority cost of a software system is often after it has been released. Its main concern is about the type of changes and cost/risk of those changes to the software system. This quality attribute intends to answer the following four important questions:

1. Which part of the system can change? This is important to the architect in terms of changes to platform, environment, quality attributes and capacity of the system.

2. What is the likelihood of the change? The architect decides on which changes are more probable given the current and future outlook of the software system.

3. When is the change made and who makes it? The time of change with respect to the product life cycle and the person making the change is important to modifiability. Specifically, changes can be made during development time (for example to the under-the-hood mechanisms of the system) or they can happen during usage of the system (for example a change to the graphic user interface).

4. What is the estimated cost of the change? Making a system more modifiable involves two types of costs: cost to introduce mechanisms to make it more modifiable and the cost of the modification using the mechanisms. This is

a tradeoff scenario, as designing the system to be over modifiable involves upfront costs that are many not be necessary. On the other hand, less design for modifiablility up front may expose the design to higher cost later on as the mechanisms were not in place.

2.3.1 Modifiability General Scenarios

- Sources of stimulus: Either the developer, system administrator or end-user who specifies the change.
- Stimulus: The specification of making the change such additional/removal of functionality, defect removal, or change to other quality attributes.
- Artifact: Refers to what specifically will be changed in terms of components/-modules, platform, etc.
- Environment: This specifies when the change is to be made for example: design time, compile time, or run-time.
- Response: Change is made, test it, followed by deploying it.
- Response measure: Money or time can be misleadingly difficult as measures for change, other measure can be extent of the change or the number of new defects that have been introduced as a consequence of the change.

2.3.2 Modifiability Tactics

The objective of tactics for modifiability is to control the complexity of making changes with both time and cost constraints. Three major ca of modifiability that allow classification of tactics are:

1. *Size of a module:* Splitting of modules into smaller modules that reduce the cost and impact of making changes.

2. *Coupling:* Reducing the strength of coupling between two modules allow for reduced cost at the time of change of those two modules.

3. *Cohesion:* Improving cohesion by removing non-cohesive responsibilities or reassigning these responsibilities to another module that has more cohesiveness with said responsibility can also improve modifiability.

The second aspect of modifiability tactics is concerning on establishing when the changes must occur with respect to the project life cycle to have optimum modifiability. This is also known as *binding time of modification* which incorporate the idea that a modifiable system is one that accommodates modifications late in the life cycle (the tradeoff cost is preparation of the architecture for late binding).

2.3.2.1 The Tactics

- Reduce the size: *split module*
- Increase Cohesion: *increase semantic coherence*
- Reduce Coupling: *encapsulate, use an intermediary, restrict dependencies, refactor, abstract common services*
- Defer Binding: *compile time parameterization, aspects, configuration time binding, runtime binding: registration, dynamic lookup, startup time binding, plug-ins, publish subscribe, shared repositories, polymorphism*

2.3.3 Design for Modifiability

- Allocation of responsibility: For each category of change, determine responsibilities that need modification or impacted, keep the responsibilities that get impacted similarly in the same module(s).
- Coordination model: Determine which functionality can change at runtime that would affect coordination, ensure these changes are to a minimal number of

modules. Use coordination models for impacted modules that reduce coupling, defer bindings or restrict dependencies.

- Data model: Determine determine which data abstractions will be modified, determine what happens with the data in terms of creation, initialization, persistence, manipulation , translation or destruction. Ensure an allocation of data abstraction that minimize the number and severity of modification to the abstractions.

- Management of resources: Determining what resource limits will change and the encapsulation of resource managers.

- Mapping among architectural elements: Determine the mapping time of elements, execution dependencies, assignment of data to databases.

- Binding time decisions: Determine latest time at which the change will be made, defer binding mechanisms for the right times, cost analysis of mechanisms, only few binding choices.

- Choice of technology: Is the technology assisting in easier modification, testing and/or deployment? choose technology that supports the most likely modifications.

2.4 QA: Performance

Performance is the ability of the software system to fulfill timing requirements in terms of interrupts, messages and request from users and other systems. It is a fundamental attribute of any designed system and one that has seen a vast amount of attention prior and during the design process. To achieve correctness in a system there is usually performance requirements which can be explicitly or implicitly defined. A deep understanding of performance requirements is essential to a well designed software system.

2.4.1 Performance General Scenario

A performance general scenario can be described as the arrival of event(s), a response from the system to that event via consumption of some form of system resource. These events may arrive in a predictable form, a probabilistic distribution or in an unpredictable fashion. Each type of event requires different approaches to fulfill the performance requirements of the system.

The response of the system may be measured via:

- *Latency*: The time interval between arrival of stimulus and the system response.
- *Deadlines in processing*: The checkpoints at each level of the system in order to generate the final response.
- *Throughput*: Number of transactions the system can process during a specified time interval.
- *Jitter*: The allowable variation in latency of the response.
- *Missed events*: The number of missed events due to saturation.

Using the above response measurements, the general scenario for performance can be concluded to be:

- Sources of stimulus: Internal and/or external sources of event arrivals.
- Stimulus: Event arrivals in either periodic, stochastic, or sporadic.
- Artifact: Refers to what specifically will be changed in terms of components/-modules, platform, etc.
- Environment: The various operational modes of the system such as normal or overload.
- Response: System's processing of the events arriving.
- Response measure: One or many of the measurements discussed previously.

2.4.2 Performance Tactics

Two contributors of the timing requirements that the performance often intends to fulfill are processing time and blocked time. Performance tactics allow the architect to control theses two times in order to effectively make engineering tradeoff with the end result of a performance oriented software system. Processing time consumes resources that inherently takes time. Handling resources is an important factor in controlling processing time. On the other hand, the blocked time is due to contention of needed resources, availability of resources or a dependency on previous computations. Therefore the tactics for performance can be categorized as:

1. Control resource demand: *manage sampling rate*, *limit event response*, *prioritize events*, *reduce overhead*, *bound execution time*, *increase resource efficiency*

2. Manage resources: *increase resources*, *introduce concurrency*, *maintain multiple copies of computation/data*, *bound queue sizes*, *schedule resources*

2.4.3 Design for Performance

- Allocation of responsibility: Determine system's responsibility with time-critical requirements and possible bottlenecks, responsibilities dealing with control oriented threads or scheduling of resources and other control mechanisms such as buffer and queues.

- Coordination model: Determine elements that need coordination with appropriate communication mechanisms to support concurrency and can capture different type of events.

- Data model: Determine which parts of the data model are heavily used or have time-critical responses and ensure that at those levels of data abstractions whether multiple copy of data, partition of data, or addition of data oriented resources may benefit complying with those requirements.

- Management of resources: Determine time-critical resources in order to prioritize and implement efficient scheduling and locking mechanisms.
- Mapping among architectural elements: Identify heavy load points and use co-location, efficient assignment of processing capacity, concurrency driven with the use of smart thread control mechanisms that minimize occurrence of bottlenecks.
- Binding time decisions: Determine the necessary binding time for each element, additional overhead introduced by late binding mechanisms.
- Choice of technology: Choose a technology that allows for control of scheduling policies, priorities for reducing demand and allocation of processors according to the performance requirements of the system to be designed.

2.5 System's Quality Attributes

Table 2.2, summarizes the critical QA mostly relevant to system design of this work.

Quality Attribute	**CV Context**
Scalability	Scalable CV workers
Interoperability	Operations with non-CV systems & services
Portability	Centralized & Distributed CV
Performance	Consistent CV oriented metrics

Table 2.2: Relevant quality attributes to the system design of this work with their corresponding computer vision context. Based on previous experience in building computer vision systems (emperical) in the intelligent system proposed, and to limit the scope of this work, portability and performance were chosen as the critical requirements.

A distributed CV software system in the cloud often allows for engineering trade-offs to be made in the context of prevalent QAs. Although in many cases the problem domain also has a strong effect on which QAs are more critical to the system. Often, performance is one of the most critical QAs in this realm where the system must react in a timely manner to its visual input and satisfy its computer vision functionality.

Performance as a QA can be defined as: a metric that states the amount of work an application must perform in specific amount of time while maintaining correct operational accuracy. Some of the ways performance is measured can be throughput, response time and deadlines. In the context of the CV problems, depending on the real time requirements of the system, performance takes the center stage in building computer vision algorithms that can accomplish the vision task. In the following chapters, metrics of performance specific to use case of this work will be explored in more depth.

A QA that is quite important in designing a distributed CV system is scalability. Building an architecture that can scale well by leveraging elastic resources of the cloud is an important aspect of this domain. As it will be observed in section 2.6, using a messaging protocol, CV workers can function independently in a distributed fashion. This allows for horizontal scaling of the processing units of the computer vision system which improves the scalability of the system.

A distributed cv software system that is designed to operate with other cloud services is defined as interoperable. Interoperability is a non functional requirement of a software architecture that is able to manage a wide range of services with relatively small effort in integration. Chapter 4 will explore in more detail as to why designing a RESTful API is important in enabling the system to be interoperable with other services and software system.

2.6 AMQP Message Based Architectures

In this section, the reasons behind having a messaging based system and its basic concepts is explored. One of the web's foundational technologies that power the communication is the Hypertext Transfer Protocol (HTTP), which is a stateless protocol by design. Statelessness is a property that has allowed web technologies to be built as simple as possible and scale easier. However, in recent years, there has been an increasing requirements of building systems that need to have state

transfered among them. This original dichotomy is which makes messaging and message passing systems an active area of research [34].

At a high level, a messaging system interacts can be decomposed of two actors: a publisher that constructs a message and a consumer that reads the message off the system with many different topologies possible. Therefore, the responsibilities of a messaging system can be summarized as [34]:

1. Route data from point A to point B
2. Handling asynchronous communication between actors
3. Queueing and buffering until delivery
4. Decoupling of publishers and consumers in a distributed fashion
5. Load balancing and scalable
6. Enhancing reliability, data visibility and interoperability

Specific to a distributed CV system, decoupling various CV workers from the source of image data is a critical in achieving interoperability and scalability. The asynchronous nature of arrival of data means that the algorithmic portion of the system may also need to respond in asynchronous fashion. As mentioned in previous sections, distributed CV systems are often heterogeneous systems where different system elements may be based on different technologies, tools and languages. Therefore interoperability advantages of a messaging system become attractive propositions in the design of heterogeneous CV systems.

For the purposes of incorporating messaging into to the architecture design of this work, an overview of the Advanced Message Queueing Protocol (AMQP) is discussed. AMQP is a simple but powerful asynchronous messaging protocol with the goal of increasing interoperability in a high traffic and distributed environment [35]. AMQP is a programmable protocol in that it allows its entities and routing schemes to be defined in the application layer rather than previously successful protocols of

the web. This allows application developers to be able to have fine grained control over and hence great interoperability. An abstracted view of the AMQP model of communication can be defined as the following: "messages are published to exchanges, which are often compared to post offices or mailboxes. Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers either deliver messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand." [3]

It is worth mentioning that AMQP's high targeted quality of service in terms of reliability, persistance, delivery to multiple consumers and preventing multiple and duplicated consumptions at high traffic speeds, makes the protocol an excellent option for computer vision applications. The implementation details of incorporating an AMQP based message broker system into the architecture of this work is explored in chapter 4.

2.7 RESTful Application Program Interfaces (APIs)

With the explosion of sources of data and the consequent data variation, designing software systems has become a challenge. These sources may be as varied as mobile devices with variety of sensors, to desktop computers and embedded devices such as internet of things. To cope with many of these heterogeneity issues, using modularity and encapsulation principles that is widely known as service oriented architecture (SOA), processing based software systems expose part of their input system using common patterns and abstractions to serve these various sources of data with minimal duplication of efforts. The use of this abstraction interface has become widely known as Application Program Interface (API) and it's the main focus of this section.

Specifically, an API can be defined as an abstraction of a software component with regards to its operations, inputs and outputs while defining the allowed interaction mechanisms from other components without compromising the consistency of the interface [36]. SOA allows various building blocks (or services), to interact with each

other using APIs which enhances the development and maintenance of the system.

Software systems often operate on *resources*, that can be text, web pages, or videos. In general an API's main responsibilities is to enhance interoperability of actions on these resources with responsibilities as the following:

1. Retrieve a request of an action from the requester(s)
2. Keep track of requests and requester(s)
3. Expose the functionality required for the action to be completed as per in the software system (provider)
4. Retrieve the results of the action once it's completed and send a response back to the requester(s)

As it can be observed, an API is required to interface well with both the client(s) and the software system that handles the action. With the proliferation of APIs on the web, certain standardized abstractions have shown to solve the main interoperability concerns. One of such best practice abstractions is known as Representational State Transfer (REST). REST can be defined as an architectural style based on the most common constraints of a web service in the SOA paradigm [37] where using simple behaviours, allows for the organization of interactions between independent systems of services that often communicate through the HTTP protocol. RESTful services have the following features [38]:

- Resource representations: they define abstract representation of the main resources that need to be operated on and their relationships similar to data modeling in databases. The format of the representation is not constrained by REST.
- Request and response: the messages that are communicated between client(s) and server(s) which often are handled by an API. This is normally via the HTTP protocol.

- Uniform Resource Identifier(URI): utilized to address the resource in a human readable form.
- Uniform Interface: a set of methods (known as verbs) that define the interactions with the resource. Table 2.3 defines some of the more important verbs.
- Stateless: a RESTful service does not maintain application state which means each request is treated independent of others. This is to reflect HTTP's design for simplicity.
- Links between resources: The resource representation usually contains the relation to other resources
- Caching: the mechanism to store the generated results for faster responses to the same request in the near future.

Method	**Operation performed by the provider**
`GET`	Read a resource
`POST`	Update resource if already exists
`PUT`	Insert a new resource (or update)
`DELETE`	Delete a resource
`OPTIONS`	List allowed operation on a resource
`HEAD`	Return only the response headers but not body

Table 2.3: Common REST methods on resources in a RESTful web service.

The design of a computer vision distributed system in the cloud requires the use of RESTful APIs in order to maintain service for the large number of distinct requesters. Therefore, any vision sensor system, capable of making REST request on the HTTP protocol can be served by the CV RESTful API without maintainability or interoperability issues. It is worth mentioning that most of the CV service APIs described in Figure 2.6 are designed to be RESTful. In chapter 4, an in-depth look at the implementation details of this work's RESTful API for retrieving vision data from multiple sources is carried out.

2.8 Concurrency in CV Pipelines

Running machine learning applications such as CV on the web is becoming a reality thanks largely due to advances in both distributed computing and concurrent systems. Machine learning applications in the cloud have to cope with increasing demand and scale in terms of feature sizes, data and storage. It is the objective of this section to summarize some of the important aspects of concurrency in the intersection of image processing and web applications.

Concurrent programming is distinct from sequential programming in that several stream of operations may execute at their own schedule while there is an overlap of execution. Each single operation is sequential but operations are happening simultaneous in one form or another. Concurrency improves performance of a system in there fundamental ways according to Cantril et al. [39]: by reducing the latency of executed operation, by hiding the latency of blocked operations such as disk access and I/O, and finally increasing throughput by completing more operations simultaneously.

Within the CV context, concurrency is inherent to any distributed CV system due to the fact that each CV "worker", must execute the CV algorithm on visual data simultaneously while this execution happens in different machines. Based on the discussion on pipeline patterns in section, a concurrent pipeline is a series of stages connected by channels where at each stage a group of operations are performed prior to moving to the next pipeline stage. In a concurrent pipeline operation the following activities are performed [40]:

1. Receive relevant algorithmic values into an inbound channel
2. The CV function is performed on the data and the transformed values are produced
3. The values are sent downstream using an outbound channel

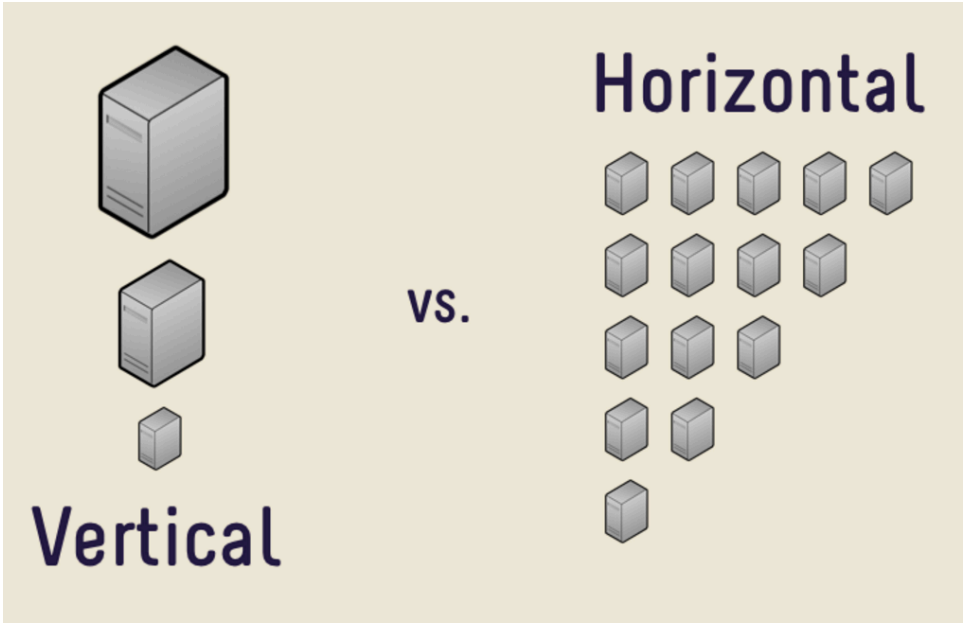


FIGURE 2.4: Difference of scaling strategies with cloud computing and distributed systems.

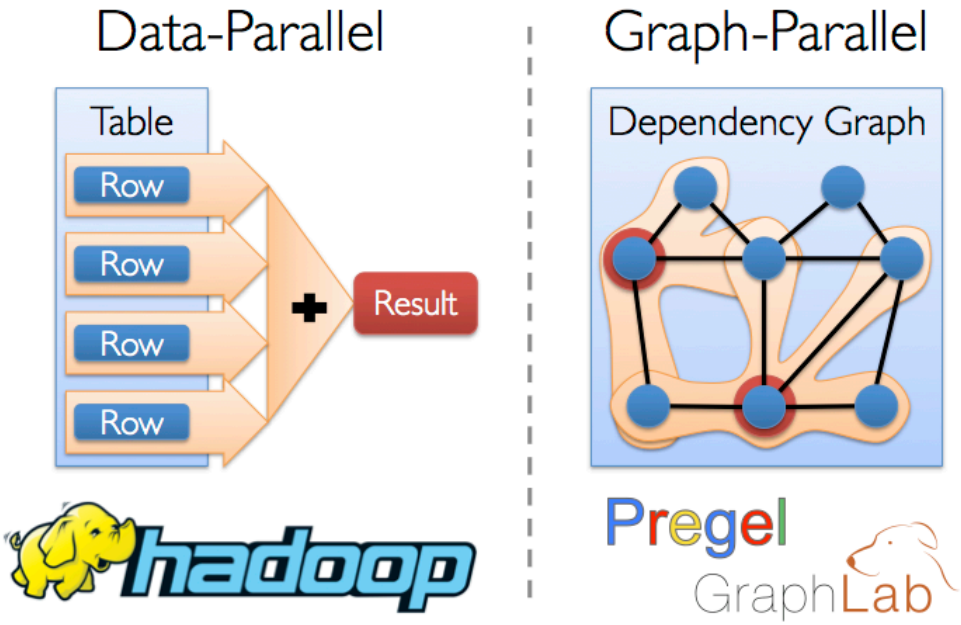


FIGURE 2.5: The main paradigms of frameworks built for machine learning in the cloud. Image courtesy of [2]

FIGURE 2.6: Some of the companies that currently offer computer vision as a service APIs in the cloud. Note: The logos have hyperlink enabled.

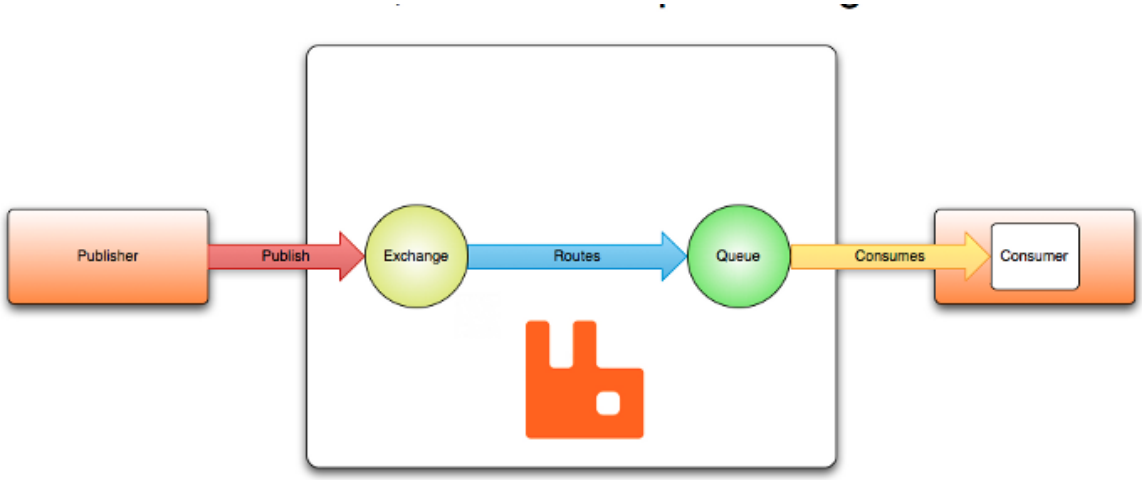


FIGURE 2.7: Basic elements of an AMQP message based system. Image courtesy of [3]

Chapter 3

Use Case Algorithm[1]

3.1 Introduction

The alarming number of traffic accidents due to driver fatigue accounts for more than half of all truck collisions in the United States [42]. Diminished levels of attention caused by fatigue, increases response time while in more severe cases it may result in short lapses of sleep by the driver. Research has shown that after 2-3 hours of constant driving, fatigue plays an important factor in slowing decision making and perception of the driver of the vehicle. More recently, a study by the National Sleep Foundation in the US showed that in their study, more than 51% of adult drivers with drowsy symptoms had driven a vehicle and 17% had momentarily fallen asleep while driving [43]. It is estimated that 1,200 deaths and 76,000 injuries are due to fatigue induced accidents annually. Due to the recent attention to fatigue related crashes, fatigue detection systems have become an active area of research.

There has been substantial work on characterizing driver fatigue based on various models. Dinges et al. [44] showed that physiological signals such as the electroencephalography (EEG) and the electro-cardiogram (ECG) can be used to measure fatigue. Other less intrusive methods using physical information of the vehicle combined with the patterns of driving has also been researched extensively with

[1]The following chapter is heavily based on the published conference paper by Salehian et al. [41]

limited success[45]. Although the most accurate results have been reported using physiological instrumentation, such systems are not practical as they require initial setup which is a hassle for the driver [46, 47]. With non intrusiveness constraints, another category of methods that has become an active research area is non-intrusive online monitoring of the driver using computer vision. In this category, "visual cues" such as gaze, head movement, and eye blink rate is tracked in order to accurately estimate the state of the driver. These computer vision techniques aim to extract visual fatigue related characteristics in real time using image/video processing. For instance, Boverie et al. [48] developed a system to correlate eyelid movement to estimate the degree of vigilance of the driver. While others such as Ueno et al. [45] looked at methods to measure the degree of openness of the eyelids to make the fatigue characterization. The majority of such early studies involved strictly controlled environments, lighting conditions and line of sight for the extraction process to work properly. Recent clinical research on the effectiveness of blinks as strong indicators of fatigue [49], make blink detection a strong candidate that is the basis of the following work.

Although most of previous work has focused on the ability of the vision system to correctly detect blinks in various lighting conditions, in real world software vision systems there are response time requirements that dictate the effectiveness of such systems. Studies by Muttart [50] have quantitatively examined the driver response times based on various real conditions on the road. Their conclusion suggested that there is a variety in response time in drivers that is obviously correlated to driver speed and conditions. Therefore, it is of note to mention that the response time of a developed computer vision system should be minimized as much as possible to account for this variation in driver population and allow for sufficient time of reaction in real settings. From the moment the system has the image data this time is labelled *processing time* and is required to respond in less than an estimated 900ms. It is the focus of the following work to not only develop the algorithm but enhance in its performance on an implementation in an embedded system with real

time constraints.

In the presented chapter, an eye blink detection algorithm is proposed using machine learning and image processing techniques in an effort to enhance the robustness of blink detection as an important part of a driver fatigue monitoring system. The contribution of this work includes two complimentary algorithms that exploit different information in each image/frame in order to arrive at a more robust estimation of the driver blink rate along with their concurrent implementation on an embedded system achieving real time requirements. The remainder of the paper is organized as follows: Section II provides a detailed explanation of the proposed algorithm methodology. The implementation details and optimization required to enhance performance of the vision system is described in Section III. The results of the critical steps of the algorithm and the validation methodology is presented in Section IV. A discussion of the results of the algorithm in a video processing setting is part of Section V and finally, conclusions on effectiveness and limitations of our approach along with future work is outlined in section VI.

3.2 Methods

In the following section, the eye blink algorithm is described with a detailed discussion on key sections of the algorithm such as: face detection, eye detection, pre-processing of the region of interest (ROI), shape analysis, complimentary histogram analysis method and combination of their outputs. The algorithm was designed using a set of real-time video captures in various lighting conditions for robustness verification.

3.2.1 Algorithm Overview

The high level flow of the proposed algorithm, initiates with detecting the face area using Haar-features. These features are extracted using a Haar classifier that has been trained with frontal face images. Once the face area is located, a second classifier using similar Haar-features finds the eye band area for both eyes. This eye band area

is the ROI that will be processed by two separate eye blink detection methods. In one method, called pipeline A, the gray scale image of the initial frame is used as input. Then, the edges within the ROI are detected using the canny edge detector with a 3×3 Gaussian blur filter to remove noise. This edge detection works well because there is a strong contrast between the iris and the choroid.

The algorithm proceeds by doing contour analysis, where the various large contours of the ROI are examined further. Due to the elliptical shape of the eyes while being open, an ellipse fitting is performed to identify the eyes as open. Upon closure of the eyes, the number of ellipses, corresponding to each eye, in the image reduces greatly which indicates that a blink may have happened.

However our experimentation shows that contour information may not be sufficient in robustly detecting the blinks. Therefore we have chosen to implement a second computationally efficient method that concurrently enhances the detection outcome.

In parallel, a secondary method (pipeline B) looks at non spatial information of the ROI. This second method, takes the negative of the frame and uses a simple threshold to globally threshold high pixel values (which will include the surrounding areas around the eyes and the eye structure). The histogram of such binary image has a bimodal shape with two impulses. It can be observed that blinking reduces the number of white pixels because momentarily the eyelids will cover the eyes and there is a shift of pixels from the high end of the histogram to the low end which is detected by the algorithm. The fusion of the results of contour/shape analysis with histogram analysis allows for the detection of eye blinks.

A flowchart representation of the algorithm overview is presented in Fig. 3.1. The various stages of the algorithm is explained in more detail in the following sections.

3.2.2 Face Detection

The following section presents the learning-based method used in detecting the face area in our blink detection algorithm. Learning based methods use training samples

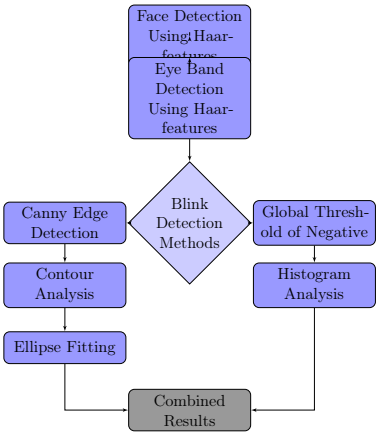


FIGURE 3.1: Flow chart representation of the proposed blink detection algorithm. Please note the two parallel sub-algorithms and the combination of their results.

in combination with statistical and machine learning models that have have shown to be effective in detecting facial features [51]. One main advantage of learning methods is their ability to adopt to various scenarios given adequate and large training sets. Variety of lighting conditions, driver demographics and other features specific to the driving of the vehicle can be included in the training set in order to increase accuracy and robustness of the face detection process.

Viola et al. [52] proposed a set of features named Haar-like features due to similarity to Haar wavelet basis functions. Their algorithm has gained popularity due its robust and computationally efficient property for object detection specifically in the face detection domain. Haar-like features use the change in contrast of adjacent rectangular groups of pixels instead of the pixel's own intensity values. The variance between the neighbourhoods surrounding the pixel are used to identify areas of high and low intensity values. Different number of grouping of such basis functions based on their variance can result in detecting different types of features such as edge, line or center-surround features [53].

A common set of Haar features are shown in Fig. 3.2.

The simplicity of these features allow for scaling and therefore scale-invariant detection of face region in the frame. Viola et al. [52] showed that for a rather small image, the total number of such elementary features is in the order 180,000 which

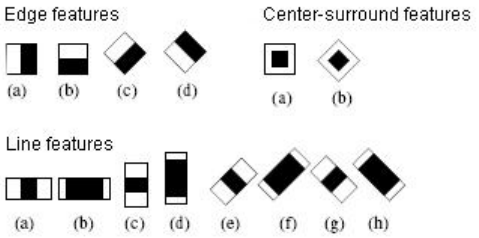


FIGURE 3.2: Some of the Haar features for edge, center and line detection. Please note that using only the horizontal, vertical and one orthogonal features all other orientation may be constructed.

may be impractical to calculate [53]. However, for accurate object detection, they noted that not all features are required. By transforming the image into what they called an "integral image", any of the haar features is able to be computed at any scale in constant time. The construction of the features is initiated by generating the integral image. The integral image intermediate representation (iI) of original image (I) at *x,y* contains the sum of pixels above and to the left at *x',y'* which can be formally defined as:

$$iI(x,y) = \sum_{x' \leq x, y' \leq y} I(x',y')$$

Then the cumulative row sum *s(x,y)* is:

$$s(x,y) = s(x,y-1) + i(x,y)$$

Then the integral image can be re-written in terms of the cumulative row sum as:

$$iI(x,y) = iI(x-1,y) + s(x,y)$$

Using this simple technique in generating the integral image, all the combination of rectangle feature sets may be constructed which makes feature generation computationally efficient. These features are sensitive to presence of edges, bars, and simple structures with only horizontal, vertical and diagonal orientation [52]. Specifically in the case of face detection, it was noted that some features are more

effective than others based on exploiting the property of the region of the eyes that is often darker than the region of the nose and cheek (with a higher variance in the eye region) and similarly darker region of the eyes from the bridge of the nose.

The effect of different lighting conditions are important in a variance based method and therefore during the training it was addressed by a variance normalization procedure defined as:

$$\sigma_w^2 = \mu_w^2 - \frac{1}{N}\sum p_w^2$$

Where the variance σ^2 of window w is defined in terms of mean of the window (μ_w) and the sum of squared pixels p of window w. The summation is calculated using the integral image procedure previously described. It is important to note that such normalization in the training procedure is inherently needed in the detection phase as well.

With the feature generation phase complete, a learning method is required in order to perform a classification function. The AdaBoost learning algorithm is used for both tasks of feature selection and the training of the classifier [54]. Using the Adaboost weak learner procedure, each classifier can only depend on a single feature and cascading of such classifiers allow for a robust method for scale invariant object detection.

A large reduction in the number of non-contributing features, and its excellent generalization performance allows AdaBoost to be used in a cascaded format that forms the cascade classification of haar-features. The cascading of classifiers allows for training each classifier using AdaBoost and adjusting each classifier's threshold and weights to minimize false negatives using error minimization.

The final strong classifier can be formally described as:

$$h(x) = \begin{cases} 0 & \text{if} \sum_{t=1}^{T} \alpha_t h_t \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 1 & \text{otherwise} \end{cases}$$

(a) Open eyes ROIs (b) Closed eyes ROIs

FIGURE 3.3: Results of face (blue) and eye band (green) ROI detection using Haar-classifiers. The classifier has worked well in in identification of the eye ROI regions while the eyes are closed in (b).

The results have proven to have a high accuracy rating and a subsequent better performance as the cascading continues. This makes the haar-like cascaded classifier method ideal for the face detection task of our real time blink detection algorithm.

3.2.3 Eye Band Detection

The eye band detection method to identify the eyes in each frame uses the same methodology as the face detection mechanism via Haar-like features and AdaBoost combination. Beyond the technical aspects of the classifier mechanism, for the eye detection few considerations are worth mentioning:

- By finding the face region in each frame, the ROI for eye detection becomes smaller and the performance of the classifier increases dramatically.
- A separate training set for eyes is used to train the classifier. In the case of this work in its current form, a pre-trained classifier was used which performed adequately for the eye detection task.
- The decision was made to detect both eyes as an "eye band" as the trained classifier performed best when both eyes were facing the camera.

Fig. 3.3 demonstrates the results of simultaneous face and eye detection for frames corresponding to both open and closed eyes using the described procedure.

3.2.4 Canny Edge Detection

Edge detection is an important procedure and a first step in identifying objects of interest in the image. Once the ROI has been identified, edge detection allows for structural analysis inside the ROI which in the case of this work is the eye band detected in the previous stage. There are a number of edge detectors that may be used depending on the desired structural properties. In the proposed algorithm, the popular Canny edge detector algorithm [55] was selected due to its following characteristics:

1. Robust detection: in the blink detection application, the probability of detecting real edges need to be quite high despite high noise levels in each frame.

2. Computationally inexpensive: due to the real time nature of the application, high performance was a secondary but important deciding factor.

3. Step edges: the strong variance between the eye region and skin (both horizontally and vertically) can be characterized as step edges which the canny algorithm was originally designed for [55].

The Canny edge detection algorithm can be summarized in the following steps:

- **Smoothing**: The edge detection procedure is initiated by preprocessing the gray scale frame using a Gaussian filter. The smoothing filter, acts as a low pass filter which reduces high frequency noise with some cost of blurring edges. In the case of our algorithm, a 3×3 Gaussian filter was used to make this tradeoff.

- **Finding gradients**: In this step, the canny algorithm looks for strong directional second order derivatives in both horizontal and vertical directions to find the gradients of each pixel in the image. This is accomplished by applying the Sobel operators. Then a measure of edge strength at each pixel is calculated

by using a euclidean distance measure to find the gradient magnitude, $|G|$, at each pixel which can be defined defined as:

$$|G| = \sqrt{G_x^2 + G_y^2}$$

where G_x and G_y are gradients in x and y directions respectively. The gradient direction θ is also calculated for each edge strength and is defined as:

$$\theta = \tan^{-1} \frac{|G_y|}{|G_x|}$$

where θ is rounded to represent horizontal,vertical or two diagonal directions.

- **Non-maximum suppression**: It is important to note that the edge strength of the smoothed image will not guarantee to find real edges and its output often includes all strong directional changes in the image. To detect real edges, Canny proposed a search mechanism that compares each edge strength with its neighbours based on its direction. It was shown that given the image gradients, the gradient magnitude assumes a local maximum at its direction. In other words, at each pixel, p, the edge strength $|G_p|$ is suppressed if its magnitude is not greater than the magnitude of the two neighbours in direction θ_p. Using this method, the algorithm is able to discard pixels that have been marked as edge but do not fall on the real edge in the previous stage.

- **Hysteresis double thresholding:** large intensity gradients often correspond to stronger edges but it is difficult to apply a single threshold to discriminate between "weak" and "strong" edges. Therefore, the edge detection algorithm uses the method of hysteresis which makes the assumption that important edges are along a continuous curve. Hysteresis uses a two step thresholding procedure by applying a high threshold ($maxThreshold$) to mark out strong edges first which are known to be genuine. Given the directional information of the gradients, a low threshold ($lowThreshold$) is applied to find edges that

(a) Open eyes Canny output (b) Closed eyes Canny output

FIGURE 3.4: Results of canny edge detection using $lowThreshold = 60$ and $maxThreshold = 140$ for both open and closed cases. Please note that while closed, the detector has only detected the shadow area around the eyes.

have lower gradient intensities. The explanation is that some of these weak edges are due to noise or other variations in each frame. Upon using the low threshold, weak edges that are connected to strong edges are marked as genuine edges.

The resulting output of this stage, is a binary image that identifies all the corresponding edges in the eye band area. Fig. 3.4 shows the effect of the operator in open and closure sample cases described previously. It is worth mentioning that the low threshold for the method was proven to be critical in detecting important edge structure between the eye choroid and the pupil. A severely low threshold would mark many details of the eye band as edges which was sub-optimal for the blink detection algorithm. The thresholds were chosen empirically based on the training set frames during development.

3.2.5 Contour Extraction & Analysis

By finding the strong edges using the previous operation in the eye band region, the structural information of the ROI is ready for further analysis in detecting the eye regions. The goal of this proposed stage is to find an approximation of all the contours that are present in each frame. The following section explains in detail how contour extraction from the edge information is accomplished and the corresponding analysis that is carried out on each contour.

The contour extraction operation used in our algorithm is based on the work of Suzuki et al. [56] where a topological analysis is done on the contours found by what

is known as "border tracing" based on earlier work by Rosenfeld et al.[57] and the utilization of Freeman's chain codes [58].

A summary of the border tracing algorithm concept is as follows:

1. Assuming a binary image, search from top left until a region pixel, p, is reached. Each contour pixel has a defined dir_p which denotes the direction of the previous move along the border to the current pixel p (the direction can based on either 4-connected or 8-connected chain coding).

2. In a 3×3 neighbourhood, of current pixel p, search in a counter clockwise fashion for a pixel, p_{new}, that has the same value as the original pixel p. the next boarder pixel is effectively p_{new}.

3. Repeat previous steps until the starting pixel, p, is traversed again.

4. The set of pixels from p up to p_{new-2} is the contour pixel set.

Once all contours are traced using the above algorithm, the operation proposed by Suzuki et al. derives a sequence of coordinates on each contour and constructs a topological ordering of such coordinates. It was shown that using such technique, both outer contours and inner contours (holes) can be effectively labeled and the topological analysis can lend to accurate categorization and discrimination of enclosing contours vs. inner contours. In the case of our blink detection algorithm, due to the large size of both eye regions in the band area, it was desired to analyze the outer contours in each frame. The proposed algorithm exploits the fact that during the blinking motion, larger contours of the eye will be deformed and disappear rapidly and hence extracting the contours is a first step in monitoring the blinking. Once the contours are extracted, the proposed algorithm proceeds by calculating the area of each contour for further analysis. It was observed that discriminating the large contours (such as large reflections due to sever lighting conditions) or smaller extracted contours (due to poor edge extraction) based on area threshold was an effective method in keeping

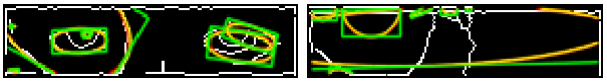
only contours related to the eye region. The area threshold is adaptive and based on the size and resolution of the eye band frame.

3.2.6 Ellipse Fitting

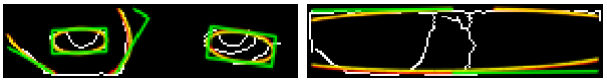
Once the corresponding contours to the eye region (the choroid and iris sections) have been identified, shape analysis is the next stage of the proposed algorithm. In this section, the ellipse fitting procedure and some of the assumptions and criteria of the analysis is discussed in more detail. The intuition behind the procedure is based on a *priori* that the eye region contours have an elliptical shape. Fitzbibbon et al. [59] evaluated various methods of fitting data to conic sections based on assumptions of isotropic normally distributed noise and incomplete contours. Their work is of particular interest in our application due to its analysis of performance in terms of algorithm complexity and computation in the task of ellipse fitting using least-squares as a distance metric. Fitzbibbon et al. showed that based on their experimentation evaluation of the popular conic fitting algorithms with strong variants of noise, orientation, and occlusion, that least-squares based algorithm of statistical distance (also known as BIAS [60]) has a good tradeoff between performance and accuracy in fitting contour points to an ellipse even with the presence of outliers due to high noise and discontinuities.

Although a detailed description of least-squares algorithm using statistical distance is beyond the scope of this work, the general procedure can be summarized as:

1. Given a set of 2-D data points $P = \{\boldsymbol{x_i}\}_{i=1}^{n}$, where $\boldsymbol{x_i}$ is a pixel in the image.
2. There exists a family of curves $C(a)$ parameterized by vector $\boldsymbol{a}$.
3. The distance from each $\boldsymbol{x}$ to each curve $C(\boldsymbol{a})$ is defined as $\delta[C(\boldsymbol{a}), \boldsymbol{x}]$.
4. Then $\boldsymbol{a_{min}}$ by minimizing the error function $\epsilon^2 = \sum_{i=1}^{n} \delta[C(\boldsymbol{a}), \boldsymbol{x_i}]$, corresponds to the curve $C(\boldsymbol{a_{min}})$ which in turn the best fitting curve.



(a) Open eyes all ellipses (b) Closed eyes all contours



(c) Open eyes all ellipses (d) Closed eyes all contours

FIGURE 3.5: Results of the ellipse fitting procedure using discriminant of $\theta = 10°$ and $contourArea = 20$ pixels. The red and yellow line are the ellipse fitting procedure while the green box is the rectangular fit for orientation analysis.

5. The error function maybe improved by subtracting a bias (based on noise levels) which enhances the performance of the algorithm [59].

During the shape analysis stage of the blink detection algorithm (implementation in OpenCV), all fitted ellipses had to meet a discrimination criteria to be included for further stages. The orientation of the fitted ellipses become important in distinguishing eyes from other ellipses. It can be observed that contours of the eye region has an orientation close to the horizontal line with some varying angle, θ, assuming that the driver is not orienting their head substantially. Choosing an appropriate threshold on θ allows the algorithm to only include ellipses resembling the contours of the eye region (θ was empirically found to be 10°). Another discriminant which was used to avoid fitting all possible contours was the contour area. Fig. 3.5 shows the results of ellipse fitting with and without constraints in both open and eyes closed cases. Please note that the fitted small ellipse in Fig. 3.5 (b) has been discarded using a contour area threshold based on the resolution of the frame in Fig. 3.5 (d). The effect of orientation discriminant can be observed in between Fig. 3.5 (a) and (c).

With the shape analysis in place, this pipeline of the algorithm (pipelline A) is able to classify the eyes in the frame as open or closed. During the blinking motion, the monitored number of allowable ellipses at each frame has a reduction of 2 or more which will indicate that the eyes have closed and therefore the frame can be marked accordingly.

(a) Open eyes global threshold (b) Closed eyes global threshold

FIGURE 3.6: Results of global thresholding, $\tau = 185$ on the negative of frames for both open and closed cases. Please note that the thresholding method has performed well in detecting the eyelid covering the eye but the areas around the eyes are still present.

Although this pipeline works in most test cases, it was observed that due to variations of sampling from different experimented cameras and the variability in lighting conditions, the pipeline requires a complementary synchronization mechanism to overcome some of these shortcomings. A second complimentary pipeline (pipeline B) was developed to run in parallel which will be discussed in the following section.

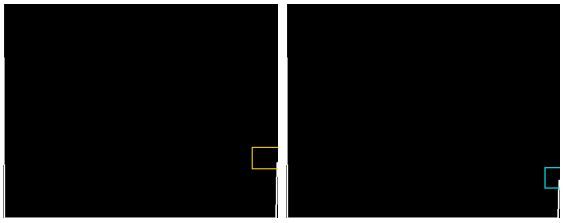
3.2.7 Global Thresholding Of Negative

To independently complement the shape/contour analysis pipeline discussed in the previous sections, pipeline B was developed using a rather simple thresholding technique. It was observed that during the blinking motion, when the eyelids cover the choroid and iris there is a change in the number of pixels that represent the skin. By exploiting this idea, a global threshold on the negative of the gray scale frame is used to detect the eye region and its approximate surroundings. The global thresholding is formally defined as [58]:

$$g(m,n) = \begin{cases} 0 & \text{if} f(m,n) \leq \tau \\ 255 & \text{if} f(m,n) > \tau \end{cases}$$

where the resulting binary image, $g(m,n)$, is based on the global thresholding operation on the original gray scale image $f(m,n)$ with threshold τ. This procedure renders robust results with prior knowledge of the lighting conditions and range of skin pixel values.

Fig. 3.6 shows the effect of global thresholding on the negative of each frame.



(a) Open eyes global threshold (b) Closed eyes global threshold

FIGURE 3.7: Results of global thresholding, $\tau = 185$ on the negative of frames for both open and closed cases. Please note the decrease of the number high intensity pixels in (b).

3.2.8 Histogram Analysis

Using the resulting binary image, simple histogram analysis is used to monitor two groups of pixels in the image. One group belongs to the background and the other group is the approximate eye region. During the blinking motion, there is a substantial change in the number of high intensity pixels and a comparison of this number with the previous frame has shown to be robust in the preliminary results of this work. Using this difference, $\%d$, the frame is classified as closed only if the difference is greater and equal to 20% of the previous frame's number of high intensity pixels (found empirically). Fig. 3.7 shows the sudden drop of the number of high intensity pixels during the blinking motion.

3.2.9 Merging of Results

In this final stage of the algorithm, the results of the shape analysis/elliptical fitting (pipeline A) and the results of global thresholding (pipeline B) is merged for each frame. If results from both pipelines match to be closure, the frame is classified as a detected blink. If there is a discrepancy among the pipelines, the result is marked as an open eye.

3.3 Real-time Implementation

In this section some of the details of the implementation of the proposed algorithm is presented. As previously mentioned, strict real time requirements are present in order for the system to fulfill the minimization of *processing time* of 900 ms and allow for slowest human driver response time in the process. The embedded system chosen for this work was the Nvidia Jetson TK1 which includes an ARM Cortex A-15 dual core processor and a Nvidia GPU for embedded vision applications on the Ubuntu 14.04 L4T platform. Although our development environment was similar on Linux 14.04 on x64 machine, during the porting process on the target embedded system, there were optimization to be considered. With the latency constraint of *processing time* being 900 ms it was important to optimize CPU code.

One of the main steps in the algorithm that is used by both pipeline A and pipeline B extensively was the color to gray scale conversion. With the target platform being an ARM based CPU with ARM's NEON [61] capabilities which allows for single instruction multiple data (SIMD) operations, the approach chosen was to enhance gray scale conversions using NEON intrinsic instructions.

The optimized implementation of gray scale conversion using NEON intrinsics is shown below:

```
void neon_rbg_gray (uint8_t * __restrict dest,
uint8_t * __restrict src, int numPixels)
{
  int i;
  // 8x8 Neon registers are filled
  // Red channel multiplier
  uint8x8_t rfac = vdup_n_u8 (77);
  // Blue channel multiplier
  uint8x8_t gfac = vdup_n_u8 (151);
  // Green channel multiplier
  uint8x8_t bfac = vdup_n_u8 (28);
  int n = numPixels / 8;

  // Conversion in 8 pixel chunks
  for (i=0; i < n; ++i)
  {
    uint16x8_t  temp;
```

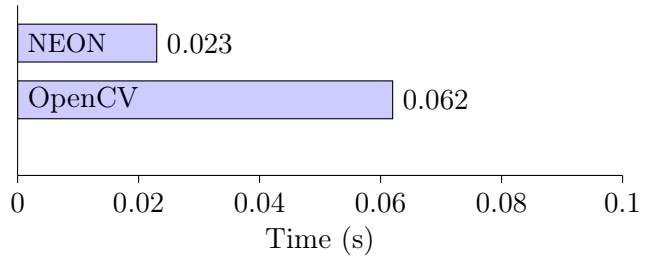


FIGURE 3.8: Comparions of our Neon optimized RGB to gray scale conversion compared to native OpenCV C++ performance.

```
    uint8x8x4_t rgb  = vld4_u8 (src);
    uint8x8_t result;

    temp = vmull_u8 (rgb.val[0],      bfac);
    temp = vmlal_u8 (temp,rgb.val[1], gfac);
    temp = vmlal_u8 (temp,rgb.val[2], rfac);

    result = vshrn_n_u16 (temp, 8);
    vst1_u8 (dest, result);
    src  += 8*4;
    dest += 8;
  }
}
```

Fig 3.8 shows some of the earliest results with comparison to OpenCV's native C++ implementation. As it can be observed a reduced factor of 2.5x allows for the system to meet the latency requirements of *processing time* of 900ms.

3.4 Algorithmic Results

The following section outlines the results obtained through experiments of this work. The proposed algorithm was developed and visually validated using live video processing of a 720p webcam at 20 frames per second using the OpenCV C++ APIs. For validation of this work, our preliminary validation procedure was carried out from a set of sequence of frames picked from various recordings in different lighting conditions and a small sample of subjects. The sampling included a relatively even distribution of the three possible cases of: open eyes, closure motion, and closed eyes. As it is evident, the algorithm initially needs to detect the face and

eyeband ROI region accurately and reliably prior to moving on to the later stages for blink detection. By constraining the distance away from the camera and the head orientation of the driver, it was found that the Haar-feature classifier performs well based on our qualitative analysis of frames through out development and on the sequence frames used for validation.

The distribution of the sequences which can be observed in Table 3.1, are as follows: sequences $s1 - s5$ were based on moderate illumination conditions, $s6 - s9$ were realized in highly illuminated and $s10 - s12$ were in low illuminated conditions. The results including the accuracy and precision of each sequence have been based on the assumption that the eyeband ROI detection procedure has been successful (as the algorithm will not carry on if the eye band ROI is not detected). True positives (TP) include all cases that both eyes were closed and the system was able to detect the blink. The cases of detected false blinks when they were either open or the eye band ROI was not even detected, is labeled as false positives (FP). The percentage of inability of the algorithm to detect closure while detecting the eyeband ROI region is included under the false negatives category (FN). The number of actual positives (AP) was measured by repeated review of each sequence via only visual inspection at this time.

A noteworthy detail of the implementation of the algorithm is that if for any reason the eye ROI is not found, the algorithm stays idle without exiting the program. The detection resumes as normal once the ROI is found, which helps in eliminating non relative frames in the validation procedure. However as it was discussed previously, in almost all cases 100% of frames were included due to both the highly robust performance of the face/eye detection stages and also the controls of the experiments.

3.5 Algorithm Discussion

The preliminary results of the validation of the algorithm, show promise of the proposed complimentary approach of using shape analysis in parallel with histogram

Table 3.1: Results of proposed blink detection algorithm in different sequences of frames in moderate ($s1-s5$), high ($s6-s9$) and low illumination ($s11-$s12) conditions. Accuracy and precision of each sequence and the total average has been shown.

Sequence #	Frames	AP	TP	FP	FN	Accuracy	Precision
s1	242	16	15	1	0	0.9375	0.9375
s2	200	12	12	0	0	1.0000	1.0000
s3	193	8	8	0	1	0.8889	1.0000
s4	300	4	3	1	2	0.5000	0.7500
s5	275	21	19	2	1	0.8634	0.9048
s6	250	14	12	2	9	0.5218	0.8571
s7	220	9	8	1	6	0.5333	0.8889
s8	244	11	9	2	8	0.4737	0.8182
s9	207	7	5	1	1	0.7142	0.7143
s10	190	10	6	4	2	0.5000	0.6000
s11	202	12	9	3	2	0.6428	0.7500
s12	248	14	13	1	3	0.7647	0.9286
Average						**0.6942**	**0.8458**

analysis. This is apparent in the accuracy and precision results in sequences $s1 - s5$ of Table 3.1, where the small sample of participants in moderate and consistent lighting conditions. However, it was observed that both the canny edge detector and global thresholding methods are sensitive to both highly illuminated environments (observed in sequences $s6 - s8$) and low illumination conditions (sequence $s10 - s11$).

Specifically, the accuracy of sequences $s6 - s8$ has decreased substantially in comparison to the moderate lighting conditions of the first set of sequences. The analysis shows that this might be due to the performance of the canny edge detector. It was observed that the low threshold and maximum threshold during its hysteresis phase, needed to be adjusted manually in order to improve the performance of the edge detection in low illumination conditions. To improve on this shortcoming, histogram equalization of the original gray scale was applied which has helped in reducing low illumination effects. The preliminary results of inclusion of the histogram equalization operator is shown in sequence $s12$ for reference with improvements in accuracy with a slight negative effect on precision. Further validation is required to concretely conclude the effect of the operator.

For high illumination cases, the histogram threshold had to be re-adjusted as it was observed that due to high illumination around the eyes, the thresholding results would include a larger area in the proximity of the eyes and hence a higher number of high illuminated pixels. This higher number effects the histogram analysis which means that the difference between the open eye frame and a closed eye frame may be smaller than the predefined difference $\%d$ set in average illumination cases. The results show degradation of performance in sequences $s10 - s11$. To improve on the robustness of choosing $\%d$, a normalization factor may be used. The ratio of the number of high pixels to low pixels seem to have eliminated this issue in controlled conditions resulting in performance improvements especially to precision as it can be seen in the results for $s9$. However extensive validation is required to analyze the full effect of this correction.

In terms of computional performance, our implementation allows to meet the minimum driver response time requirements of the real time system the algorithm is designed for. More extensive profiling shows other areas of improvements such as using the GPU to enhance the performance of edge detection using the Canny edge detector.

3.6 Algorithm Conclusion & Future Work

This work presented a blink detection algorithm based on two complimentary, but independent approaches using shape and histogram analysis. The monitoring of the driver's blink patterns was performed in near real time using efficient image and computer vision techniques. The preliminary results in terms of total accuracy and precision rates indicate that the current approach can be useful in monitoring blink detection for fatigue. The algorithm requires training in more varying lighting conditions in order to be robust. Future work will include improvements to the image acquisition system such as using an infra red camera and also additional preprocessing techniques such as gamma correction or histogram equalization. Furthermore the

inclusion of adaptive methods in the edge detection step and also in the global thresholding stages will be part of the continuation of this work. Using similar techniques in identifying other visual cues such as facial expressions and yawning may enhance the accuracy of a well defined driver fatigue detection in the future.

It is worth mentioning that a larger benchmark study where the performance of the proposed algorithm is analyzed against other blink detection algorithms on a benchmark dataset will be a path to extending this work.

3.7 Algorithm's Properties As A Use Case

The choice of a sufficiently complete use case for software engineering research is important. This is because the constraints of many software engineering problems may not be well exposed and it was the intention of this work to utilize a real world algorithm that that author has developed in order to fully understand the complexities of porting a centralized CV system to a distributed one.

The chosen algorithm for the larger task of studying the process of building distributed CV systems had the following desirable properties:

1. Performance intensive: The blink detection algorithm is resource intensive which means that there is a real need for a distributed system. This is an important characteristic as with performance tracking certain performance metrics of the centralized and distributed versions of the system can be compared and analyzed.

2. Highly prarallizable in the cloud: The algorithm's main property needs to be able to ran across many machines (or containers), in the cloud in order to analyze the complexities of distributed systems.

3. Real world application: A real world application means that there are real software engineering requirements that need to be met while building the

system and performance and portability are scarce resources in the project budget.

4. Novel: The algorithm is newly developed and there are no other implementation of the algorithm in the cloud yet. It was the intention of this work to analyze as close to a real world scenario of when there are no existing distributed versions of the algorithm.

Chapter 4

Methodology: System Development

The main objective of the thesis is to explore the process of porting a computer vision algorithm on a singular machine to a distributed cloud architecture. Hence, the process and methodology of building a distributed system to achieve this objective, is a critical aspect of exploring the issues, nuances, and differing methodologies of developing computer vision systems. The methodology for this work is of a quantitative experimental nature where by building the system, instrumenting and tracking the system's properties at each development step, the work-flow, architecture and implementation roadblocks are examined in a use case scenario. This process

In this chapter, the development of a distributed computer vision system from the inception of the use case blink detection algorithm (Chapter 1) up to the implementation of the first iteration of the system is described. An in depth look at the process and methodology to build such system will be the main cornerstone of this work which is the exploration of building computer vision systems that are distributed and cloud based from the ground up. The chapter initiates with a closer look at which set of requirements are integrated into the implementation. Next, a look at metrics, show how these requirements are tracked in both the classic algorithm development

work flow and the distributed system. A discussion on languages and frameworks explores more concrete and specific concepts introduced in the background chapter. A look at components of the system, role of messaging in the architecture and following an end-to-end flow of a message through the system is discussed. The chapter is concluded with the documented work-flow that will enhance portability and iteration speeds in developing a distributed CV system from a centralized version.

4.1 System's Scope & Requirements

With the objective of building a distributed computer vision system from an already existing centralized computer vision algorithm, system requirements were a crucial factor in the design and implementation process. The important non-functional requirements of a distributed CV system described in section 2.3 and the more detailed version in Appendix 2.2.1 are based on the different design constraints and system demands associated with the problem domain of CV. Although these requirements are not generalizable, they are often a part of many distributed machine learning and computer vision system in the cloud.

In the process of defining both functional and non-functional requirements of the proposed distributed CV system, it became obvious that due to the nature of the research question at hand, a set of corresponding quantitative metrics were required to be defined and tracked during the implementation of the system. The metrics' initial contribution was to monitor the success of porting from a single machine system to a distributed message based architecture. However, it became apparent the trajectory of these metrics during the distributed system development became an area of opportunity in maintaining the quality attributes defined by the requirements.

4.1.1 Requirements Gathering

Understanding the problem domain associated with the system to be designed, is a crucial step prior to gathering the requirements. In other words, the problem domain

describes what problem the system is designed to solve. Software architecture of most systems can only focus on solving specific problems effectively. In the case here, transforming a central CV algorithm into a distributed cloud based system where many clients of the algorithms can connect and use the algorithm in a distributed and seamless fashion is the problem domain. By exploring the problem domain, proper requirement gathering and analysis was an effective step in the design process of the CV system.

Requirement analysis as either a formal process or an informal one, is part of the design methodology of most successful software systems [62]. In most of such systems, the design constrains of the system are explored which assist in the evaluation of the system's final properties. In this work, requirements were selected based on the background research on distributed machine learning system and concerns of a similar project for fatigue detection of truck drivers by an industrial partner in the energy sector. The system's requirements at a high level are:

(R1) The system is distributed in nature with each computer vision task is atomic and independent

(R2) The computer vision processing is solely done in the cloud

(R3) The system is able to handle a large number of concurrent visual sources and clients

(R4) Blink detection should be completed within a certain period of time that is acceptable for fatigue detection.

(R5) Every frame of video should be analyzed by the system without unprocessed frames

(R6) The system's algorithmic accuracy should be very similar to the centralized version

Requirement	Impact on system design
The system will incorporate distributed system	Distributed system constraints with CV constraints
All processing must be done in the cloud	distributed cloud architecture
Many types of visual data sources can send data	Need for a distributed system with API layer
Blink detection should not be lagging more than a few minutes	Performance oriented software design
Every frame needs to analyzed for an accurate blink rate	Ensure frame fidelity via message architecture

Table 4.1: Proposed system's functional requirements and their impact on system design.

In the requirement analysis of the system, it was inherent to analyze which requirements have the most impact on the system design and development. Prioritizing requirements is important because often resolving conflicts between requirements is part of the design of the system and core properties of the system is at stake in the balancing of opposing requirements.

Although most of the requirements of the system remained intact during the design process, their impact factor and level of contribution varied as the project became more mature. An iterative approach to requirements analysis was carried out in order to make sound design and implementation decisions. For instance, the decision to limit all of the processing on the cloud side was based on allowing to serve various visual data sources with the same system, and allow for provisioning of the processing resources. However, other requirements such as a consistent development environment for both single machine and distributed system was in conflict with similar accuracy and performance in the distributed cloud version.

Table 4.1 and 2.2 summarize the general functional and non-functional (quality attributes) requirements associated with the distributed CV system. The last column describes the contributions and impacts of the requirement to the system's design. These contributions were used as guidelines in the early stages of the design of the proposed system. Throughout the next few sections some of the design and implementation tactics in order to fulfill some of these requirements is described.

4.1.2 Metrics

As previously described, the methodology used in exploring the research question is quantitative and experimental. The goal was to actively track how the transformation of the system from singular to distributed may effect the design and requirements. In order to accomplish the tracking of the metrics, the role of metrics was essential in quantitatively performing profiling and experiments during the system design process.

The metrics that were defined in the development process are categorized as:

1. Performance metrics: These metrics track both the computational and algorithmic performance of the system in both centralized and distributed designs.

2. Portability metrics: These define how portable different components of the system are to collectively be able to allow for the transformation between singular and distributed. Furthermore, the work establishes a "portability budget" throughout the transformation process.

The following sections discusses the metrics generation process of the developed system.

4.1.2.1 Performance Metrics

Performance metrics quantify the success or failure of achieving a set of particular computing tasks. In the context of this work, there are two sets of computing tasks that is of interest. First, is the execution of the blink detection algorithm which is present in both centralized and distributed systems which is defined as *algorithmic* performance. The second is the end-to-end performance metrics that define the application's response from the time the input data is received all across the system until the final results are received by the client (the system that has initiated the request). This system based metric, assists in quantifying the performance of a

complete task which has shown to be valuable in the design and implementation of the system [63].

In the developed system of this research, the algorithmic performance metrics are based on metrics used in profiling many computer vision algorithm computations [15]. These metrics are defined as:

- **Pre- and post- memory consumption (MemC)**: This metric tracks the memory consumption for both before and after a single call is made to analyze the frame by the algorithm. Memory is one of the most important resources to manage in any CV system. CV applications are normally demanding in terms of the memory because not only the size of the images, but the total frames, number of times pixels are read from memory [64]. Hence, this metric was generated and instrumented since the very early stages of the project. The implementation details to achieve this, is discussed in section 4.1 in more detail.

- **Frame Resolution (Res)**: The average resolution of frames being processed by the processing portion of the system (algorithm). As mentioned previously, the frame resolution has a direct impact on performance because the frame is transformed and moved in the vision pipeline extensively for various processing needs particular to the algorithm. This means that the average resolution of frames that are being processed is an indicator of how the algorithm is performing in terms of input/output characteristics and memory bandwidth.

- **CPU Cycles per call (CpuCy)**: The number of CPU cycles for each call to the algorithm. This is a secondary metric to compare algorithm performance between the centralized and distributed systems. In theory this metric should be very similar in both cases.

- **CPU Cycles per pixel (CpuCy/Pix)**: The cost of processing each pixel in terms of CPU cycles after one frame is processed. This metric is a granular measure of how the system is performing the processing portion in both

versions. As it discussed in Chapter 5, it is a critical metric in tracking system requirements related to the response of the system to input.

It is worth mentioning that all of the metrics are based on the processing performance of the system on a single frame. This granularity based on an atomic processing units was chosen in order to be able track and compare performance between centralized single machine version of the system and the processing done in the distributed CV system. In the case of the use case blink detection algorithm, it was desirable to not only define some of the metrics but be able to generate them independent of algorithm's environment being centralized or distributed. The instrumentation of these metrics were a strong reason in choosing the frameworks and languages which is explored fully in section 4.2.

4.1.2.2 Portability Metrics

A software unit exhibits portability across various environments to the degree that the cost of transport and adoption to the new environment is much less than the cost of redeveloping it [65]. Although it would be an ideal proposition to have completely portable software units, the task of building software with zero cost of transport between environments is a challenge and practically never possible. The portability metrics define the degrees of portability of the system with relations of porting and/or redevelopment costs.

In general portability is a crucial quality attribute of computer vision systems because the testing and validation of these systems are a large portion of costs to the redevelopment. Any drastic changes to the system due redevelopment means there will be revalidation many parts of the system. Hence, tracking portability of the system in both quantitative and qualitative forms using metrics was one of the focus points of development and methods. In this section, the portability metrics that assist in guiding the development of the proposed CV system from the centralized environment to a distributed cloud environment is explored. In [66],

Mooney et al. defines a cost estimation analysis approach in measuring portability in the development cycle. Based on their work, measuring the portability of our system or the degrees of portability (DP) is defined as:

$$DP = 1 - \frac{cost\_to\_port}{cost\_to\_redevelop} \tag{4.1}$$

A higher degree of portability would be closer to 1 and therefore it is important to minimize the cost of redevelopment and porting as much as possible and keep DP high. The redevelopment cost specific to blink detection algorithm use case is characterized as redevelopment of the algorithm, repeated fine tuning and finally revalidation. The porting costs associated with the project can be characterized as the cost of building software collaterals and tools in order to build porting mechanisms of the algorithm from centralized to distributed system, testing the system in the new environment and the comparison analysis among the two environments.

However, during the development process, the degrees of portability metric alone was found to be too coarse and subjective for the objectives of this work. Taking inspiration from the domain of process aware distributed software systems, where portability is a critical quality attribute, a suite of objective and detailed metrics were defined. In [67], Lenhard et al. propose a continuous inspection and integration system that instruments and measures portability using a vast test suite and instrumentation code. The quality model used to define the granular metrics is based on the ISO/IEC 25010 Quality Model [68]. The model establishes that most software systems have parts that are directly portable(defined as "adaptability") and parts that require new code to be written (defined as "replaceability"). The standard describes the ease of installation in the various platforms using the refined quality attribute "installability". Based on these three refined quality attributes that contribute to the portability of a distributed software system, more subjective metrics were defined based on [69]. Furthermore, the portability metrics used are mathematically validated using theoretical techniques in both measurment theory

Elements	Language	Platform	# of Supported Environments
Frame grabber	C++	OpenCV	Windows/Embedded/Linux/Android/iOS (5)
CV Clients	Golang	Multiple	Windows/Embedded/Linux/Android/iOS (5)
API Server	Golang	Docker	Linux (1)
CV Worker	Golang	Docker	Linux (1)
CV Processor	Golang/C++	Docker	Linux (1)
CV Algorithm	C++	OpenCV	Linux (1)

Table 4.2: An example of Severity of each element for

and constraint validity by Lenhard et al. [70].

Lenhard argues that degrees of portability using lines of code calculations is not representative of the true portability issues in real distributed system. Degrees of portability captures the adaptability of the system but not the replaceability attributes. Hence, they propose the addition of degrees of severity, D_sev, which is defined by the degrees that languages and platforms can be ported based on the list of environment that support them. For instance a large severity indicates that the language of the component or software system is supported only by few environments and therefore making it harder to port the component or system. More subjectively, a list of supported and unsupported environments can be generated based on the languages and platforms used in the computer vision system to accurately estimate the degrees of severity [70]. A severity matrix table such as Table 4.2 can be generated which details the severity matrix of each element of the centralized system including languages and platforms that may be ported to in the future.

By quantifying the degrees of severity based on number of supported environments in the life cycle of the project, The following subjective metrics were defined:

1. **Basic Portability (BP):** It can be thought as the degrees of portability (DP) but measured in Lines of Code (LOC).

2. **Weighted Portability (WP):** The weighted sum of the contributions of each element of the system based on their degrees of severity.

3. **Service Communication Portability (SCP):** The effects of change in the communication mechanisms between elements such as message structure or interfaces measured in LOC and the number of distinct service communication protocols.

4.1.3 Basic Portability (BP)

As with definition of most metrics in software engineering, the challenge lies in instrumenting them through out the software development life cycle. The cost estimation is heavily dependant on the problem domain and system being developed. On one hand building portable software systems are dependant on implementation details that break portability barriers. On the other hand, portability metrics are not generalizable due to the high variability in the problem domain and system architectures [66]. Furthermore, in many cases the redevelopment cost estimations are based on future results of the system when the system is being maintained. Therefore, for the purposes of this work, to effectively measure the basic portability of the system the estimation was done at the system component level for two reasons:

1. In order to observe the costs at the interfaces of critical components at each iteration development of the system.

2. Assess the portability impact of each component prior to strong coupling to the system.

Based on the described approach to estimating, the following costs were incorporated into the BP metric and tracked for each new component of the system that was being developed through the development life cycle:

(C1) Consistent development environments: the cost of building a development environment that works seamlessly in both centralized and distributed realms. This cost include languages and frameworks used. (porting cost)

(C2) Consistent visual inputs across both systems: the cost of building collateral input components that allow the same visual inputs such as a video frame with consistent formatting, video encoding and compression to be used in both realms. (porting cost)

(C3) Algorithm adjustments and tuning: the cost of adjusting and fine tuning the algorithm after it has been ported to the new system. (redevelopment cost)

(C4) Validation: The cost of revalidating the CV algorithm in the new environment due to new constraints and dissimilarities. (redevelopment cost)

(C5) Testing: The amount of tests that need to be written as safety net in order to ensure consistent system behaviour across the two environment. This may include a variety of tests that need to be written as the porting is happening including integration tests. (porting cost)

Each cost is evaluated at component level and the corresponding degree of portability based on equation 4.1 is recorded for further analysis discussed in Chapter 5.

4.1.4 Weighted Portability (WP)

Based on the definition of severity [67] and the elements of the system shown in Table 4.2, WP can be mathematically described:

$$WP(S) = \sum_{i=1}^{el} C_{el}(el_i) \tag{4.2}$$

where for each elementel_i of the system S , the element complexity of each element C_{el} based on the number of supported environments outlined in the Table 4.2.

Elements	Communication Protocals	Communication Observables
Frame grabber	REST, AMQP(2)	ImageCapture(1)
CV Clients	AMQP(1)	Multiple(2)
API Server	REST,AMQP(2)	MessagePassing(1)
CV Worker	AMQP(1)	MessagePassing(1)
CV Processor	AMQP(1)	MessagePassing(1)
CV Algorithm	System Shell(1)	OpenCV ImageCalls(1)

Table 4.3: The service communication portability contribution from each system component.

4.1.5 Service Communication Portability (SCP)

This metric based on the work of Lenhard et al [67] focuses on tradeoffs between portability and communication among the services and elements of the computer vision system. This is a variation of the WP metric but instead focuses on the complexity of the communication protocals and obervable behaviour among the various elements. SCP can be defined as:

$$SCP(S) = \sum_{i=1}^{el} Protocal_{el}(el_i) \tag{4.3}$$

Where for each element el_i of the system S , the element's number of communication protocal or communication complexity $Protocal_{el}$ is taken into account for the service communication portability.

4.1.6 Portability/Performance Profiling

At the initial stages of of development, a main objective to define the portability budget in order to make the right type of engineering trade-offs between a highly portable system, high performance and engineering time to build. In the software reliability engineering domain, Musa et al [71] proposed profiling methodology to track the reliability of a software system using operational profiles and reliability demonstration charts. Similarly in this work, the portability and performance metrics were captured and plotted in a similar profiling chart in 3D to track how well the

system is maintaining the right balance of performance and portability quality attributes quantitatively. The profiles allows the engineering team to prioritize different quality attributes as the development continues throughout the project. The results of the graphs are discussed in more detail in the next chapter.

4.2 Languages & Tools

In the previous section, the requirements and some of the metrics were discussed. The choice of languages and frameworks have a direct impact on both the performance and portability metrics defined. The instrumentation and strategies to generate those metrics is the main subject of this section. This discussion takes a look at the choice of languages and tools with their effect on the metrics and some of the implementation details in the instrumentation code that hopefully will be of value in developing similar systems. The section also explores some of the general criteria for choosing tools and languages in building a distributed CV system in the cloud.

4.2.1 C++: OpenCV

In section 4.1.2.1, the instrumented performance metrics of the system was outlined. These metrics allow an iterative comparison of the design process throughout the porting of the system. In order to achieve a consistent method of instrumenting the code without having to write separate instrumentations in each of the environment, C++, and in specific the Open Computer Vision library (OpenCV) was selected to build the algorithm. OpenCV is a C++ library widely used in many computer vision algorithms where both real time performance and portability is important. Written in C/C++, it leverages the portability of the language due to the fact that the same C/C++ code can be compiled on many systems. This is mainly possible, because the code is transformed to machine language with the help of the compiler and there is no need of extra layer of abstractions (such as the Java Virtual Machine). This also means that the programs written in C++, have a fine grained control on memory

management based on the need of the programmer. In other words, the programmer has full access to the memory and resource management of the algorithm. This choice of tooling, allowed to build instrumentation code around the algorithm that is native to the machine code that is running the algorithm.

As an implementation detail, the goal was to be able to observe the memory consumed when OpenCV data structures were being allocated and de-allocated. This meant that the instrumentation code had to be added at the OpenCV layer. Below is the code snippet based on work by [72] that allows this type of instrumentation of memory consumption and facilitates generating memory related performance metrics:

```
1 class MemoryManager
2 {
3 public:
4     typedef std::map<void*, size_t>     AllocationTable;
5     typedef std::lock_guard<std::mutex> LockType;
6
7    ...
8
9     MemorySnapshot makeSnapshot()
10    {
11        LockType guard(mAllocMutex);
12
13        MemorySnapshot snapshot;
14        snapshot.peakMemoryUsage = mPeakMemoryUsage;
15        snapshot.peakMemoryUsageSinceLastSnapshot =
              mPeakMemoryUsageSinceLastSnapshot;
16        snapshot.allocatedMemory = mCurrentMemoryUsage;
17        snapshot.allocationsCount = mAllocationsCount;
18        snapshot.deallocationsCount = mDeallocationsCount;
19        snapshot.liveObjects = mAllocationsCount -
              mDeallocationsCount;
20
```

```
21          mPeakMemoryUsageSinceLastSnapshot = 0;
22
23          return std::move(snapshot);
24      }
25  private:
26
27      MemoryManager()
28          : mCurrentMemoryUsage(0)
29          , mPeakMemoryUsage(0)
30          , mPeakMemoryUsageSinceLastSnapshot(0)
31          , mAllocationsCount(0)
32          , mDeallocationsCount(0)
33      {
34      }
35
36  private:
37      std::mutex      mAllocMutex;
38      AllocationTable mAllocatedMemory;
39
40      size_t          mCurrentMemoryUsage;
41      size_t          mPeakMemoryUsage;
42      size_t          mPeakMemoryUsageSinceLastSnapshot;
43
44      size_t          mAllocationsCount;
45      size_t          mDeallocationsCount;
46
47      static std::once_flag  mInitFlag;
48      static MemoryManager * mInstance;
49  };
50
51  ...
52
```

```
53 MemorySnapshot memorySnapshot()
54 {
55     return std::move(MemoryManager::Instance().makeSnapshot());
56 }
```

The code snippet is the memory instrumentation class that allows generation of all memory related performance metrics. As it can be observed at line 9, the *makeSnapshot* method uses mutual exclusion locks to look at memory usage (line 14), the usage since the snapshot was initiated (line 15) and allocated memory and deallocated memory counts (lines 16 and line 18 respectively). This means that for the memory consumption metric MemC, there is access to pre- and post-memory consumption by simply passing the algorithm's main function to this method. For the complete version please refer to [73].

Similarly, to generate the other performance metrics, low level assembly C code allows to capture the number of CPU cycles each call to the main algorithm function takes. The snippet below can be called before and after the call to the main algorithm function to find the CPU cycles:

```
1 int64_t rdtsc() {
2   unsigned int lo,hi;
3  __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
4  return ((uint64_t)hi << 32) | lo;
5 }
```

Hence using these two instrumentation code snippets, one at the OpenCV middleware layer and one at the low level assembly level, we can calculate the four metrics identified in section 4.1.2.1 to be tracked through the development life cycle as the algorithm is ported without any adjustments for different environments. This means that each distributed agent that is executing the algorithm will generate these metrics automatically and an averaging aggregation process will give the metrics for the whole system. Figure 4.1 demonstrates the homogeneous nature of instrumentation. As it will be discussed in section 4.3.2, these metrics are generated and sent back as

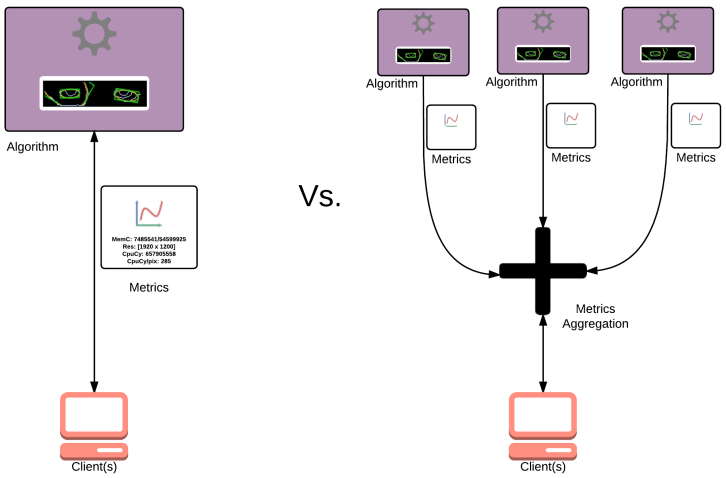


FIGURE 4.1: Methodology for homogeneous performance metrics generation in distinct environments.

part of the result of blink detection messages to the client and aggregated result is shown at the client level.

4.2.2 Golang: Built-in System Concurrency & Error Handling

The choice of the main language for the infrastructure of a distributed cloud system is often based on design constraints, performance criteria and specific features of the system. Golang has emerged as one of few important languages for writing server and infrastructure code bases. Designed by Google to tackle the infrastructure challenges of millions of request and server side processing of many Tera bytes of data, Golang is picking up momentum in this domain due to design of concurrency and error handling mechanisms. These two aspects were very important to the success of building the CV distributed system of this work. The simplicity and explicit nature of Golang enables the software system to gracefully handle many requests over the network without the complexity of network middleware software. Specifically to the system developed, Golang was chosen as the main infrastructure language due to the following:

- Built-in concurrency for pipe-and-filter pattern: As one of the main patterns

used through the system to build the CV processing, building robust concurrent abstractions were important.

- Strong idioms in building RESTful services: Building a large scale API server in order to handle many vision source clients.

- Thread safety for an atomic message passing architecture: In a message passing system that is concurrent, the atomicity of messages with guarantees that messages are consumed only once was important because concurrency bugs may result in wrong calculations of the blink rate.

4.3 System Design

In the following section, the architecture of an implemented distributed CV system is presented. The discussion of this section is initiated with the various components of the system, some of the implementation details with regards to the requirements and metrics gathered previously. The abstract high level architecture sets the path on how these components interact as the life of a request for the CV service is followed throughout the system. The integration of the algorithm in the system and the validation of the methodology used to build the system conclude this section's contributions to the work.

4.3.1 Components

The developed CV system consists of four major components: *API server*,*CV client*, *workers*, and *processors*.

The **API server** is an HTTP server which receives the HTTP requests of various sources of visual data and sets up the process of sending these requests to the CV client. The API server is the public facing part of the system and therefore handles different type of requests. The API server is RESTful and stateless and can handle a large number of clients simultaneously.

The **CV clients** are in charge of the communication between the API server and the RabbitMQ broker. They receive the HTTP requests handled by the API server and puts them on the RabbitMQ messages to be carried to the messaging exchange. This involves publishing an AMQP message with the proper header content type and callback queue information. The CV client is also responsible for aggregating metrics (as described in section 4.2.1) and along with the algorithm results once the other components have operated on the message. It is also the responsibility of the CV client to acknowledge the completion of the processing to the API server to be communicated with the externals of the system who have requested for the CV service.

The **CV workers** are designed as responders to the exchange where they handle the messages to be send to the processors. The worker's main responsibilities include load balancing, downloading the frames necessary and overall orchestrating the flow of requests to messages to be processed by the processors.

The **CV processors** are a distributed set where each one responds to messages being generated from workers. These processors hold a binary copy of the algorithm that performs blink detection and metrics generation. They work concurrently and out of order to process each independent frame of the video. Once a processor finishes with one frame (and corresponding performance metrics), it sends a result message back to the workers to be handled and communicated to the API server. The processors use go-routines [74] and channels enhance concurrent performance as discussed in section 4.2.2.

The patterns utilized amongst components are both consistent and simple. The pipes-and-filters pattern is used extensively to allow effective concurrency in the tasks responding to messages. Using the messaging based architecture of RabbitMQ enables asynchronous processing of messages that are independent (requirement $R1$ in section 4.1.1). Messages are atomic and can not be lost during the process which covers the requirement $R5$. As it can be observed, each type of component has a single responsibility which enables the system to be composable in building different

types of CV algorithms. Figure 4.2 demonstrates the high level relationship amongst the four major components of the developed system.

The general process to build each component was as follows:

1. Write the minimum logic to be able to connect to the RabbitMQ messaging broker
2. Instrument the messages that are communicated from the component to the broker and vice versa.
3. Write the actual logic for the component to complete its responsibility.
4. Integration tests between two or more finished components to ensure messages are being responded to properly and the transformations and metrics are taking effect.

4.3.2 Life of a CV Request

In this section the functionality of the developed system is examined. The goal is to emphasize the distributed interaction of components to deliver the results of the blink detection algorithm previously detailed.

The system's functionality can be summarized as part of the lifetime of a single CV request as follows:

1. The user(s) sends an HTTP request to the API server to find inquire about blink detection in the video at hand.
2. The API server receives this CV request.
3. It processes it and hands it off to a CV worker.
4. The CV worker transforms the CV request into data structures specific to the system's algorithmic needs.

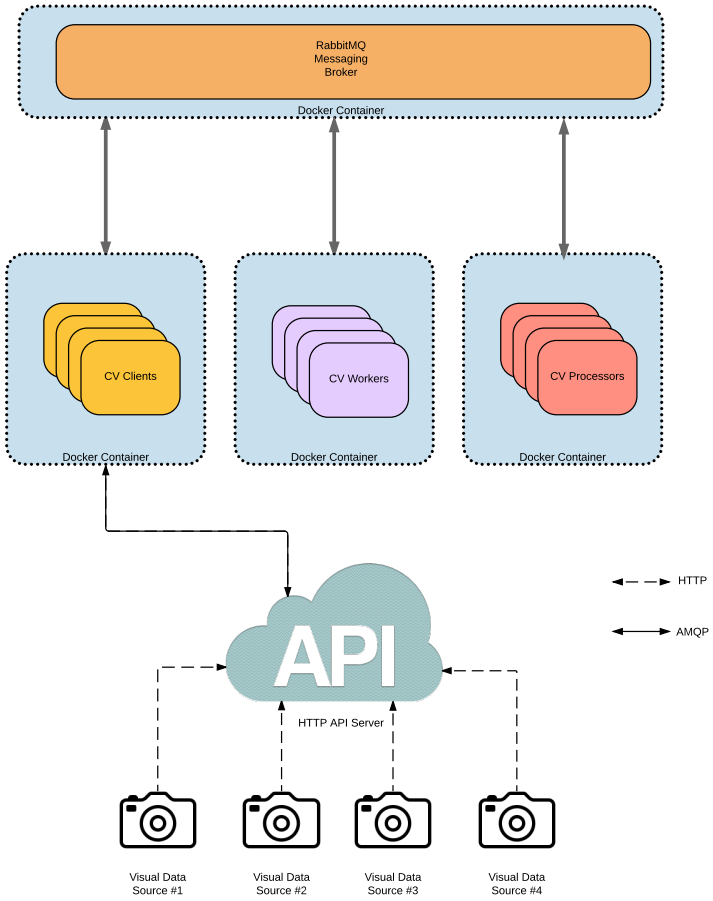


FIGURE 4.2: Basic high level architecture and components of the developed system.

5. The Cv worker registers all queues and remote procedure calls necessary for other components and connects to the RabbitMQ broker.

6. A message is published by the CV worker to the broker which handles it to the proper exchanges and queues.

7. CV workers are standby ready to consume messages in the appropriate queues.

8. CV workers grab each frame of the video (either by downloading or if it's included as part of the message).

9. Each CV worker knows about a specific temporary call back queue that CV clients are listening on.

10. The CV workers then publish messages back to the exchange where CV processors are listening.

11. Each CV processor instance consumes one message, runs the algorithm on the proper frame and generates the necessary performance metrics.

12. The results are stored in the same CV request but with fields that were empty before.

13. A message is generated by the CV processor that has finished its job and this message is consumed by one of the workers that is idle.

14. The message is sent to the appropriate call back queue that was set by the client.

15. The client returns the results and hands it off to the API server to serve the user.

16. The API server serves back the results/metrics to the client that had the original request.

4.4 Development Work-flow

The process of building software systems is as critical to the success of the delivery of the system as overcoming the technical challenges [75]. As one of the cornerstones of this work, documenting the development work-flow at various stages of the project was important. In this section, the overview of the proposed work-flow for moving from a centralized CV algorithm into a distributed CV system in the cloud is discussed.

The process, summarized in Figure 4.3, was initiated by building the algorithm development environment. Based on remote prototyping principles in embedded

system developed explored in [76], the environment setup has to be as close as possible to the environment of the distributed version of the system. As explored in section 4.2.1, OpenCV and C++ were used in a Linux docker container to avoid revalidation. With the intuition that many copies of these containers were going to be used in the distributed system, the algorithm was developed, statically compiled and linked to be run in a single docker container. The performance and portability metrics were generated after each step during the final algorithm development process.

Once the algorithm was ready and evaluated in terms of computer vision performance as discussed in Chapter 1, the focus was shifted on the infrastructure code base to enable distributed networking of the system. Here the components were designed and the process of component design explained in section 4.2 was followed. These services were designed to be included in a docker container. For instance, each CV worker or CV client components are instantiated to be a single Linux container. The CV processor container is merged and integrated with algorithm container to keep a similar interface as other containers. As each component type was built and containerized, portability metrics were estimated.

Each component container was unit tested for functionality and then integrated using the Docker methodology of composition (docker-compose [77]). Once the CV processor containers were connected to the other components, metrics generation and evaluation was the next step.

Performance and portability metrics generated upon completion of the integration were compared to the centralized version to assess the effectiveness of the methodology. Any change to the components meant, re generating metrics to ensure the system requirements are met as the porting was happening. The final stage of work-flow was to write end-to-end system validation tests to ensure all containers/components are performing correctly.

The code snippet below documents an integration test to demonstrate the composability of the design and integration testing methodologies used in this work. It is worth mentioning that the system is also being load tested with 50000 frames being

requested to be analyzed for any given integration test.

```
1 // Integration test from RPCclient all the way to CV processor with
      generated performance metrics
2 func TestCvRpcClientIntegration(t *testing.T) {
3         // Test frame URL
4         testImageUrl :=
5         "http://fullhdpictures.com/wp-content/uploads/2015/04/
              testFrame.jpeg"
6
7         // Setting up rabbitmq broker for test purposes
8         rabbitConfig := rabbitConfigForTests()
9
10        // Instantiateing a CV worker
11        cvWorker, err := NewCvRpcWorker(rabbitConfig)
12        if err != nil {
13                log.Info("TEST ", "err: ", err)
14        }
15        // Running the CV worker
16        cvWorker.Run()
17
18        // Instanantiating a CV client
19        cvClient, err := NewCvRpcClient(rabbitConfig)
20        if err != nil {
21                log.Info("TEST ", "err: ", err)
22        }
23
24        // Make sure there are no errors propagated up to this point
25        assert.True(t, err == nil)
26
27        // Generated the request 50000 times to put the system under
              stress of 5000 frames
28        for i := 0; i < 50000; i++ {
```

```
29                  var processorChain []string
30                  // Link the processor to the blink detection
                        algorithm
31                  processorChain = append(processorChain, "DetectEyes"
                        )
32                  // Setup a CV HTTP request
33                  cvRequest := CvRequest{ImgUrl: testImageUrl,
                        ProcessorChain: processorChain}
34                  // Call the image decode via the client
35                  decodeResult, err := cvClient.DecodeImage(cvRequest)
36                  if err != nil {
37                          log.Info("TEST ", "err: ", err)
38                  }
39                  // Again check to see there are errors
40                  assert.True(t, err == nil)
41                  // Log the results
42                  log.Info("TEST ", "decodeResult ", decodeResult)
43                  // If the result of algorithm was an Error a test
                        failure should occur
44                  assert.Equal(t, decodeResult.Output, "Error")
45          }
46 }
```

4.5 Validation

In order to validate if the developed system has fulfilled all functional and non-functional requirements, the following acceptance criteria scenarios were established and evaluated:

- ✓ The system is capable of managing more than 10+ clients simultaneously requesting for the CV blink detection algorithm.

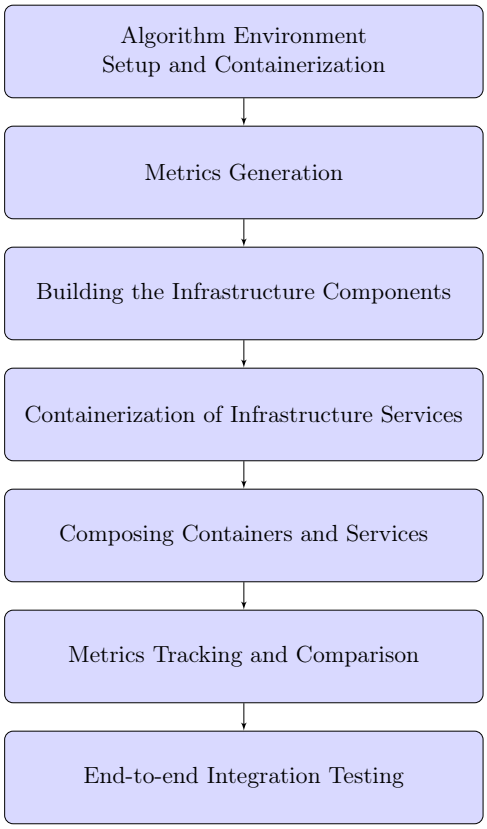


FIGURE 4.3: The development work-flow utilized in this project.

✓ These requested videos are able to be run on the centralized system and the distributed system for comparison purposes.

✓ The Performance metrics are visible to the requester and are within a delta of 10% in terms of blink detection algorithmic accuracy.

✓ The distributed system's performance of the number of frames processed per second (FPS), is not any worst than 70% of the FPS for the centralized single machine system.

With the evaluation of these criteria, the workflow and architectural design choices are validated for the specific case of the blink detection CV algorithm. Chapter 5 explores the experimentation results, metrics and validation outcomes of this process.

Chapter 5

Experiments & Results

The main focus of this investigation is to propose a software design and implementation work flow for building distributed computer vision systems. The goal is to reduce the portability and performance costs in building such systems by implementing metrics. In this investigation, as part of an empirical software engineering research, a set of circumstances and environments are established in order to conduct a *case and exploration study*. The exploration study assists in analyzing the proposed methodology and architecture for transforming a centralized CV system, in the application of blink detection, into a distributed cloud CV system in a practical setting where a controlled experiement of multiple studies was not possible.

This chapter aims at defining the experiment design process and protocol, the execution of the experiments at various phase of development based on the metrics defined in the previous chapters and finally presenting the results associated with each experiment phase.

5.1 Experiment Design

In controlled experiments, a case study problem is defined in order to constrain and control the scope of the experiment. The developed blink detection CV algorithm's properties allow the experiment to be organically constrained which is valuable when

dealing with empirical software engineering research.

The design of experiments carried as part of the validation of the architecture has the following essential elements:

- Experimental data: The visual data used for the experiments on both the centralized and distributed version of the system. A uniform data across both systems allows for control over algorithmic accuracy while elimination the bias in cases where one system would naturally suite the video set better.

- Environments: As one of the important aspects of this work, the experiment environment have direct impact on the analysis of the research questions. Choosing appropriate environments were based on having the cloud environment to deploy the system on and execute the experiments.

- Instrumentation for performance metrics: A uniform set of C/C++ instrumentation calls that are used in both systems to ensure a fair comparison. This instrumentation code gathers the performance metrics The instrumentation implementation was discussed throughly in the previous chapter.

- Reporting and Analysis for portability metrics: At critical points of the project such as considerable milestones in the development of the distributed version of the system, portability metrics of the methodology are analyzed and recorded as part of a consistent report.

- Experiment execution: As one of the main goals of the designed system was to fulfill a special use case of distributed computer vision, the execution of experiment in order to assess the validity of the flow and design of the system had to resemble the use case closely.

5.1.1 Experimental Data

A brief overview of the visual data used in the experimentation of this work is outlined with the goal of reporting the desirable properties of visual data experiments.

These properties can be categorized as architectural context and high performance requirements.

The videos chosen for the experiment had to exercise the important part of the architectures and be real use cases of the requirements of the systems built. Hence videos acquired from different types of visual sensors (phone cameras and dashboard cameras) were used to fulfill the heterogeneous nature of the requirements for the sources of data.

On the other end, the resolution and frame rates of the data chosen had to be sufficiently high to require a distributed system in the cloud in order to explore the complete spectrum of issues while building a distributed cloud CV system.

Table 5.1, describes the nature of the dataset and how they have the desirable properties mentioned previously.

Video Set	Source	Duration(min)	Properties
V1	iPhone 6	2:00	high resolution, low framerate
V2	iPhone 6	1:00	high resolution, low framerate
V3	iPhone 6	2:00	low resolution, high framerate
V4	iPhone 6	2:00	low resolution, high framerate
V5	iPhone 6	2:00	low resolution, high framerate
V6	webcam	2:00	low resolution, high framerate
V7	webcam	2:00	low resolution, high framerate
V8	dash camera	2:00	low latency requirements

Table 5.1: Experimental data used and their desirable properties for the purpose of this work.

5.1.2 Instrumentation for Performance

As described in section 4.1.2.1 and 4.2.1, the instrumentation code was ran as part of the algorithm by making low level calls to acquire the metrics without any post experimentation work. The inclusion of instrumentation as part of the implementation, or design for instrumentation, allowed faster feedback cycles in the design process in a more data driven approach to designing the system.

Table 5.2, summarizes the results of the performance metrics of the dataset. For a detailed description of each of the metrics, please refer to section 4.1.2.1.

Video Set	Res	PreMemC (e+06)	PostMemC (e+06)	CpuCy(e+06)	CpyCy/Pix
V1	1280x720	3.31	22.5	399	433.46
V2	1280x720	3.23	21.9	378	410.2
V3	920x480	2.47	18.4	207	468.7
V4	920x480	2.44	17.9	209	473.3
V5	920x480	2.65	18.2	203	459.7
V6	920x480	2.18	14.4	193	442.3
V7	920x480	2.20	12.5	188	410.8
V8	1280x720	3.53	25.9	395	434.0

Table 5.2: Performance metrics generated as part of the experiments. Results are the mean for each frame of the video set.

As the performance instrumentation results show, the system is able to mange many different types of visual inputs in a distributed form without significant performance costs. The CpyCy/Pix metric shows clear consistency in performance even though the main operations of the system seem to be memory bound. It is worth of note that the system performs equally in high and low frame rate which is due to the system's atomic view of each frame in such a message based architecture using RabbitMQ.

5.1.3 Portability Analysis

As discussed in section 4.1.2.2, portability metrics were tracked at the system component level in order to assess the impact of transforming the centralized version of the system to the distributed CV system in the cloud. Using portability profiling techniques described in section , the portability metrics are tracked. As each critical system component was designed the portability metrics for that component and other existing component is evaluated. The components, outlined in section 4.2, are not explicitly present in the centralized version but some of their functionalities are implemented across the functions. Therefore, the decision was made to utilize the design milestone of each component as an opportunity to track the portability

Cost ID	Short Description	Type	Final BP(SLOC)
C1	Development environment cost	Porting cost	709
C2	Visual inputs cost	Porting Cost	80
C3	Algorithm adjustments cost	Redevelopment Cost	5
C4	Revalidation cost	Redevelopment Cost	14
C5	System testing cost	Porting Cost	57

Table 5.3: The type of costs during the analysis of the basic portability metrics tracking at the final stage of development.

metrics described in Section 4.1.2.2 of the are summarized in Table 5.3. A set of figures summarize the results obtained in this portability analysis at the final stage of devlopment. For a comprehensive description of the metrics, please refer to the end of section 4.1.2.2. The section is concluded with a snapshot look of the portability metrics during the development of the project.

Table 5.3 outlines the basic portability (BP) based on each cost measured at the end of the project in SLOC.

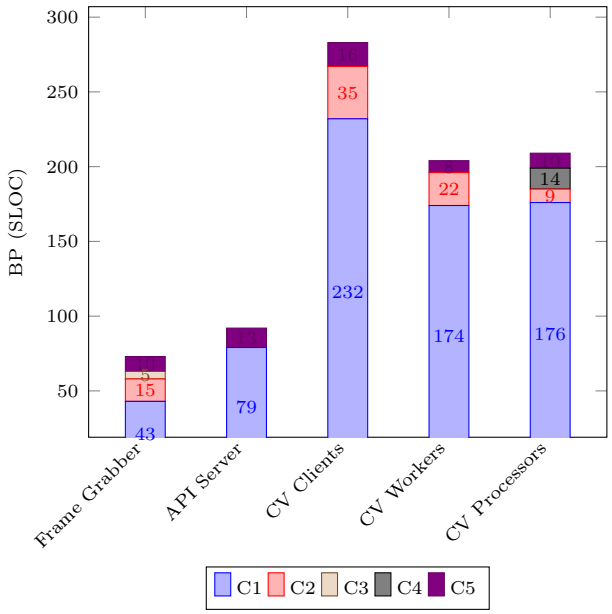


FIGURE 5.1: Basic portability (BP) of the final developed system and the contribution of each component.

The results shown in Table 5.1, show the contribution of each macro level cost to the component instrumented using the BP metric. Analysis shows that in terms of

degrees of portability (DP), CV Clients is the weakest link. A large portion of the cost was associated with this component. This is to be expected as the CV Clients had to be written for the purposes of the scratch, had to support a number visual sources and is the backbone of the processing system in the backend. It can also be observed that the algorithm adjustment costs (C3) were minimal which indirectly lowers the revalidation costs (C4) as well. By encapsulating the algorithm as a microservice, C4 is solely affecting the CV processors components. This secondary effect contributed to a continuous integration (CI) methodology of fine tuning the algorithm, running the validation and deploying the system online.

The BP analysis was more useful using continuous integration/inspection meaning guiding the development process by analysing BP metrics as code is committed. Due to relative ease of generating BP, it was found to be a much better approach in building for portability than a post-mortem approach where metrics are generated and analyzed after portability issues or certain milestones of the software project. With a high enough granularity and in larger software projects, the results also may assist in resource allocation and project ETAs.

As outlined in Section 4.1.4, the weighted portability takes into account the severity and the supported environments of the computer vision system that is missing from our previous analysis of BP. Table 5.2 outlines the WP measured in SLOC at the final stage of development.

Analysis of the results in Table 5.2 indicates that the largest contribution to the portability of the system is still CV clients. This is due to the fact that the component not only has a large code base (depicted in the previous analysis of BP), but also requires to support a number of environments based on Table 4.2. It can be observed that Frame Grabber is another critical part of the system with a higher WP than API Server component. API Server is supported in the Docker environment which reduces their effect on the portability of the system. In fact, the components that were able to be supported in a docker container environment, their contribution to the portability budget was minimized to a large extent.

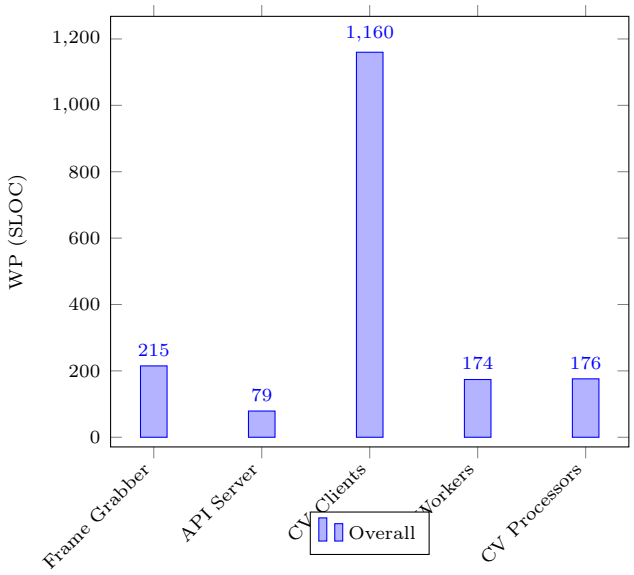


FIGURE 5.2: Weighted portability (WP) of the final developed system and the contribution of each component.

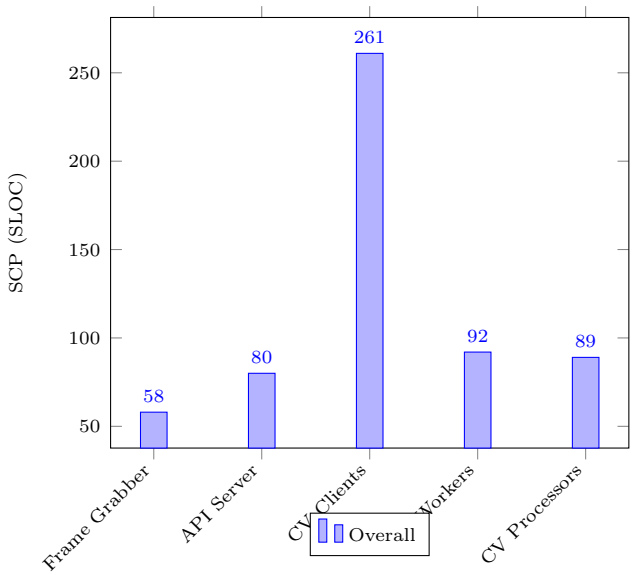


FIGURE 5.3: Service Communication Portability (SCP) of the final developed system and the contribution of each component.

As outlined in Section 4.1.5, the service communication portability takes into account the complexity of change in the intra service communication of the computer vision system components. Table 5.3 outlines the WP measured in SLOC at the final stage of development using Table 4.3.

The results of generating SCP metrics indicate the same trends observed with CV Clients during the BP and WP analysis. CV Clients is a central pieces of communication logic between the various components and is in the top group of components that require to support multiple protocols and communication observables. The similarity of the structure of the CV Workers and Processors indicate that perhaps a single abstract component may reduce the portability budget and help in having a higher degrees of portability, lower total severity. Although due to the different communication protocols and observables, its SCP should remain the same in a consolidation scenario and will be a sum for the newly abstracted component. In general, the analysis of the SCP can assist on how resilient is the system to changes to newer communication protocols and observable patterns or behaviours as more features are added to the computer vision system.

5.1.3.1 Portability Analysis During Development

One of the advantages of automatic generation of metrics for portability is the ability to inspect and track the portability budget as code is being written and committed to the repository. In this section, the trajectory on the three previously defined portability metrics through the development stage of the project is discussed.

Figure 5.4 shows the contribution of each component at different commit days as the project code base was solidifying. Albeit the coarse nature of BP, the analysis uncovers some insights into the development of our specific computer vision system. It can be observed that certain components had minimal changes of codebase after their inspection. These include CV Workers and the Frame Grabber components. Conversely other components, specifically CV Clients, were contributing to the BP of the system in many different phases of the project. In other words, CV Clients can be identified as a critical component in any portability decisions.

A more granular view of the system's portability trajectory can be observed in Figure 5.5. The weighted nature of WP, shows that certain components that were identified as relatively sensitive to portability (such as CV workers or CV

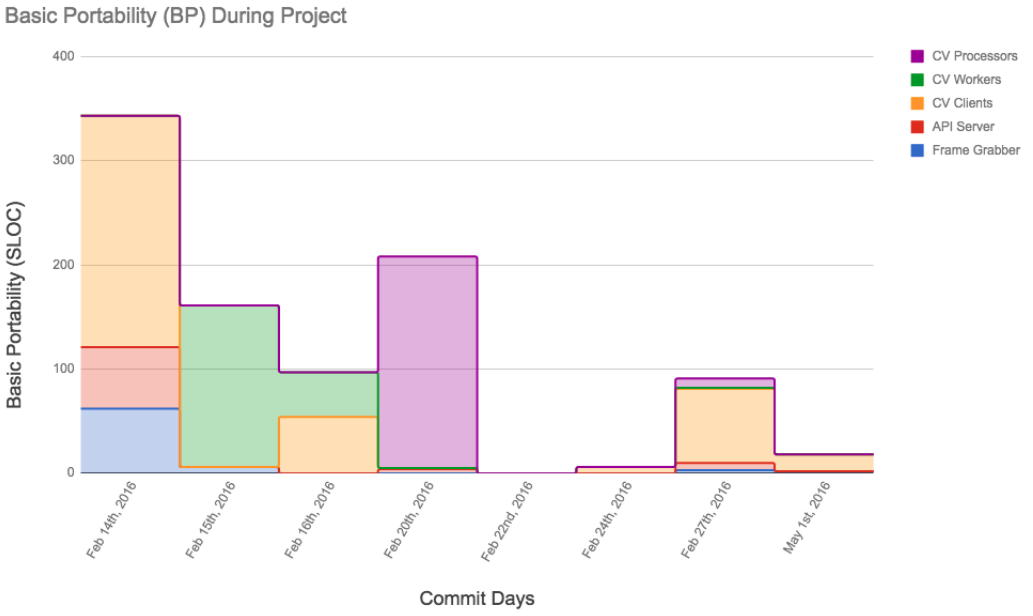


FIGURE 5.4: The tracking of BP for each component during the development process.

Processors) in the previous BP analysis, are in fact not portability "points of failure". The domination of the CV Clients is much more evident during the WP analysis. Perhaps a breakdown of the component is necessary to reduce its strong effects on the portability quality attribute of the system.

Lastly, we can observe the effect of service communication among the components during development in Figure 5.6. SCP analysis during commit days shows that almost in all cases the communication contributions are established while the code is finalized for that component. This result is in line with the observation that most of the communication protocols and observable patterns of a system does not change throughout the project life cycle.

In closing, the diagrams of portability can be used as a guide in allocating engineering resources to maintain high quality attributes and main engineering tradeoffs of the desired system. They may be also used as historical references for similar computer vision projects in the future.

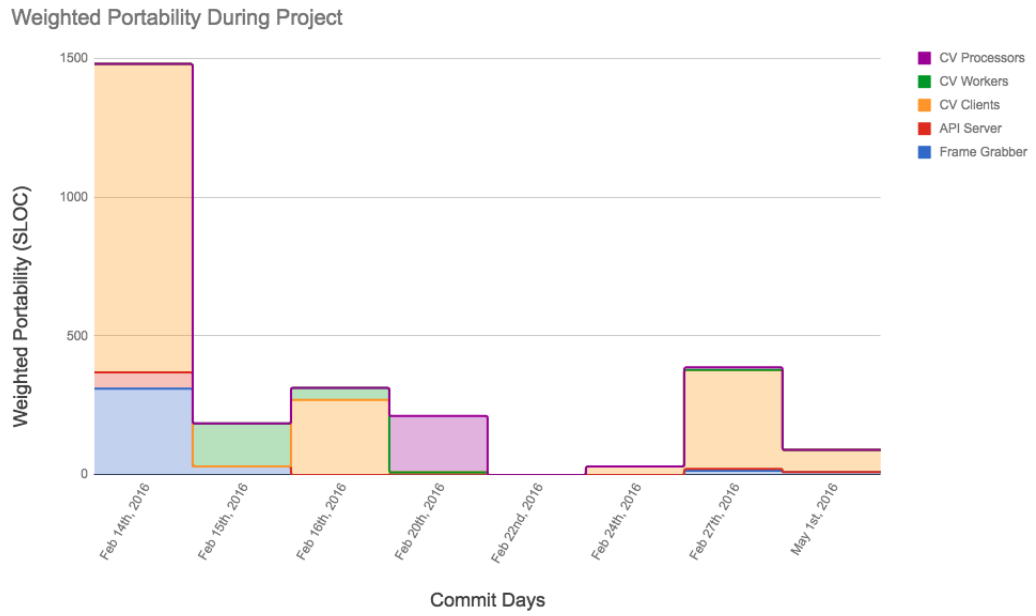


FIGURE 5.5: The tracking of WP for each component during the development process.

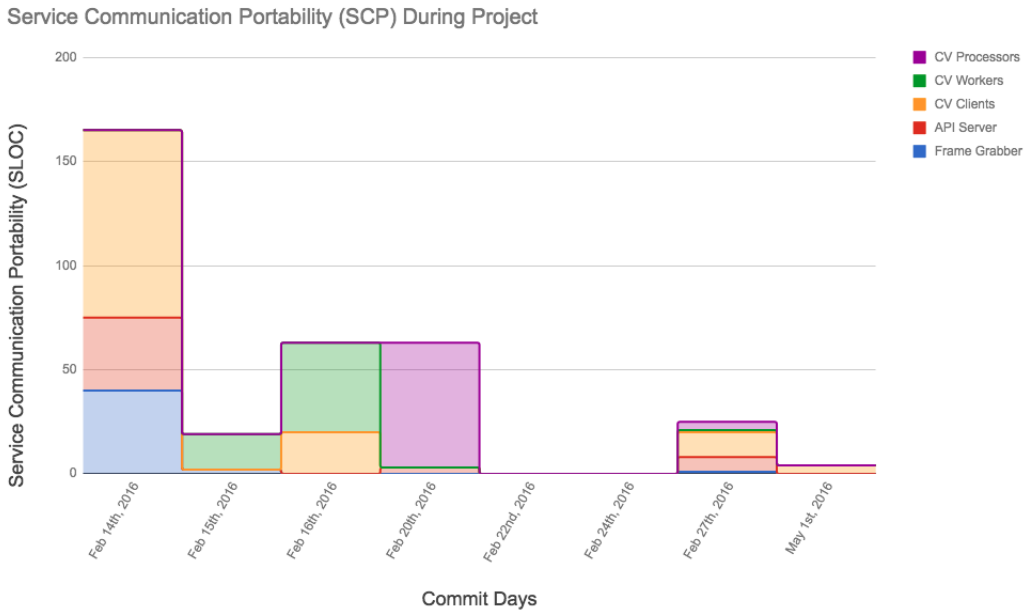


FIGURE 5.6: The tracking of SCP for each component during the development process.

Chapter 6

Conclusions & Future Work

The main objectives of this thesis was to examine the methodology required to build a distributed CV system given a centralized classic computer vision algorithm as part of a larger intelligent transportation system. A blink detection algorithm was built from scratch with the appropriate properties to examine this active area of research in distributed systems and computer vision.

With the use case study, software requirements were gathered and acceptance criteria for the final system was established. The work flow to build and implement the system using various software architecture techniques were reviewed as part of the methodology of the work. Based on the acceptance criteria, performance and portability metrics were tracked during the implementation phases of the project and experiments were ran in order to validate that the system in fact has met the requirements. The main contribution of this work as a workflow process and software engineering management were studied in detail.

In the closing chapter of this work, a summary of the final results of this exploraratry study and based on the acceptance criteria is presented. The chapter concludes with a suggested direction to build future aspects of the project is reviewed.

6.1 Result Analysis

In order to validate and verify that the system built meets the requirements set in the initial gathering stages, the acceptance criteria was examined against the results of the experiments. As discussed in the previous chapter, utilizing metrics it is possible to assess how well the system met the criteria.

In terms of performance acceptance criteria, a 8+ simultaneous visual input sources were connected to the distributed system and as it was observed in the experiments, the system was able to serve all of these sources without any degradation. The same visual input data was used between both the centralized and distributed versions as control in the experiments. In terms of accuracy, the same algorithm was able to run in both systems and therefore very minimal difference in algorithmic accuracy was observed. This was due to the choice of containerizing the algorithm as part of the CV Processors component.

As one of the criteria specified a degradation budget of 30% in terms of frames per second, the centralized version was able to process many more frames per second than the distributed version of the system. On average, the difference in frames were more than 40%. This large delta could be explained partly due to the overhead of the communication. Specifically, as each component is sending the frames via the AMQP protocol to the next component, this means that large overhead is introduced in the case of visual sources with high resolution frames. It can been observed that the system was not able to perform the processing and failed to deliver the required frames per second latency even though in terms of throughput the distributed system had minimal performance degradation. Having a frame data store and sending identifier across the messages may improve the latency of the system dramatically. There results of the experimentation shows that there are a category of the results that can be generalizable and others that require a large sample of projects. For instance, the instrumentation and metric definition for performance showed that when the problem is able o parallelized in such a way where multiple frames can be

independently processed allows for a message passing system such as RabbitMQ to handle high memory and high CPU tasks without degradation.

The portability analysis at each milestone of a critical component, showed the majority of the portability costs were associated with building the API server and CV Workers. This was expected as these part of the systems were needed to be developed for the distributed version in the cloud. In contrast, other parts of the system such as CV Processors, had high portability by leveraging the advantages associated with the containerization of the algorithm without large modification to the original code base. Hence, revalidation effort were almost non existent as there were very few instances of fine tuning necessary. Based on the experience of porting the algorithm, it would be beneficial to implement the algorithm inside a container environment in order to avoid costs of building new environments. This would reduce the porting costs to a distributed version dramatically. Furthermore, a continuous deployment of the system to the cloud allowed to test the system in a real distributed setting.

However there is a number of limitations to the proposed approach in the work. Firstly, the scope of the work is based on designing and building an end-to-end computer vision system including proposing and validating a new algorithm in the fatigue detection domain. However, this means that a statistical analysis of the effectiveness of the methodology proposed is required to show quantitative generalizability across other types of computer vision and machine learning systems. Second, the work's key quality attributes were only performance and portability whereas in many real life systems there may be additional quality attributes that may be in play that may limit the findings of the thesis.

6.1.1 Guidelines

The analysis of an end-to-end computer vision software system from problem inception, through requirements and all the way to a complete development had secondary

benefit of allowing for observation of some of the recurring patterns that will be discussed in this section as guidelines for future reference of other machine learning software engineering projects. Some of the guidelines include:

1. **Early algorithm selection/design**: This stage of the design of the algorithm is critical to the ability to scale to a distributed system. Strongly parallizable machine learning pipelines can help in distributing the workload to many machines in a horizontal fashion. Similar to other software engineering tasks, an early design and vision for the algorithm pipeline is a prerequisite for the success of the project.

2. **Containerization**: The ability to containerize the maximum number of components early and throughout the process assists in collaboration in the later stages but also improves the portability of the system. The experience of this work shows that the use of technology controls the portability risks and reproducibility of the results required during the porting process.

3. **Maximum capture of data**: Capturing data is inherent in the ability to quantitatively make architectural and design decisions. These tasks include high daily commit rates, investment in instrumentation code, and maintaining the results in version control. The analysis of the results in either post-mortem or periodic fashion with generated data from the system helps in guiding the development process immensely.

4. **Automated validation suite**: Revalidation costs are often high in computer vision and machine learning development. The ability to run and compare automated validation tests enables the development team to be more agile. It also shows what commits are breaking the validation results before a large time investment in design decisions that may hinder the ability of the system to produce the best results possible.

5. **Continuous portability analysis**: Similarly to validation, portability needs to be quantified and analyzed at the critical points of the project. Some of these critical points may include when a new component is being introduced, when abstractions are replacing legacy parts of the system and when new requirements are being added during the development.

6.2 Future Work

The system developed for the purpose of this work is not without areas of improvements. As a prototyped system, the latency properties are yet to be within the range of requirements to be deployed in a real life setting. Performance optimizations such as reducing overhead of messaging by using a frame store may be part of the next iteration of the system. The methodology proposed will need to be also validated with various computer vision systems in order to establish sufficient generality. In particular, a closer look at different paradigms of computer vision algorithms (based on the taxonomy of patterns discussed in this work) which can be beneficial in comparing which type of algorithms are well suited to be ported to the developed system or other distributed systems.

In terms of solving the general fatigue detection problem, given the system's interoperable API, other sources of data processing may be integrated to work together in resolving fatigue detection. These sources may include other indicators of the driver's vehicle and other vehicles in the vicinity for true connected intelligent transportation systems.

The development and metrics driven approach to building a computer vision system will require a statistical analysis on more than one system. Given the quantitative nature of the portability and performance metrics, similar machine learning systems may be analyzed to assess the generalizability of the approach as a foundation of more successful computer vision and machine learning software projects. This means that the development of more automated tools and test may assist in

studying a larger number of software projects without the need to understand the full architectural and design decisions beforehand. Additionally, a through analysis of multiple key quality attributes that have stronger opposing characteristics will be beneficial in refining the methodology of development computer vision systems.

Appendix A

Software Architecture

Software architecture's definition has been a topic of discussion since the early 70s. Specifically, a common misconception of the definition is that the architecture is solely based on early or major design decisions of the software system. Software architecture may be defined as:

"The software architecture of a system is the set of structures **needed to reason** about the system, which comprise software elements, relationship among them, and properties of both. [33]"

This definition emphasizes on the strong reasoning aspect of the discipline which implies the study of software architecture as an enabler in order to make sound software engineering tradeoffs at the design stage. This definition also suggests that architecture composes of a set of software structures that as a whole, define the behaviour of the system and its final attributes. Prior to discussing the uses of software architecture (SA) a more in depth understanding the building structures of an architecture is required which is discussed next.

A.1 Structures in Software Architecture

Structures in the software architecture context can be generally defined into three distinct categories:

1. Implementation units or *modules*: Modules assume computational responsibility as implementation units which can be classes, layers or mere division of functionality. Modules are static and allow analysis on dependency of the internal software system such as inheritance and/or generalization relationships.

2. Dynamic units or *components/connectors*: Components/connector structures define how the system is to be structured with run-time behaviour (components) and interactions (connectors). These dynamic structures which can be services, peers, or servers allow for analysis of run-time behaviour of the software systems such as performance, security and availability [33].

3. Mapping units to non-software components or *allocation structures*: Allocation structures embody decisions on how the software system interacts with its environment such as CPUs, file systems, networks and development teams. Using allocation structures, the non-software environment interaction with the system can be viewed and analyzed in detail. Such through analysis is essential in the deployment of distributed systems.

A.2 Benefits of Software Architecture

Some of the benefits of a systematic approach to software architecture was briefly discussed in the previous section. In this section additional benefits are described in more detail.

A.2.1 SA Abstraction

SA can be viewed as an abstraction to cope with the complexities of today's software system design. SA consists of a number of structures and each structure includes elements (being a module or a component) and relations among those elements which contribute to the complexity. Often such elements have public and private properties and the job of SA is to abstract away the internal private properties that has no

effect outside the element itself in order to look at the software system with elements, composition and relations in a holistic form [33]. SA abstraction inherently allows for an explicit definition of the behaviour of the system and requires many aspects of the behaviour to be included and documented.

A.3 Importance of Software Architecture

Software architecture affects two important aspects of the system's quality attributes: initial quality attributes that become built-in to the system and secondly prediction of quality attributes. There are other important factor that contribute to the need for a through design of the software architecture. Some of these secondary factors include:

- SA as a reasoning tool for initial decisions and any change to the system implementation down the stream.
- SA allows stronger communication between stakeholder based on a single vocabulary and discussion point.
- SA puts forward the road map and constraints necessary for a high quality implementation
- Re-usability of software architecture (beyond element re-usability) has many positive consequences on time-to-market and maturity of new systems being designed.

These aspects and many others are the primary reasons why any software project regardless of its size and complexity benefits immensely from a architecture design stage.

A.4 Structure Views as Form of Training

The software architecture's ability to describe how elements interact with each other can be used as first introduction to the system effectively [33]. As a common reference point, the architecture views can serve to answer such inquires such as what elements do what, team assignment to system parts mapping (normally as part of the module views). Conceptual views are effective in explaining on how the system is suppose to function and accomplish the task.

A.5 Effects of SA Beyond the Software System

Although there is a strong impact on the software system by SA, there are other ripple effects caused by SA. One of such "bi-directional" relationships is the effect of architecture on the organization structure. Conway's Law eloquently described this phenomena where the "organizations that design systems...are constrained to produce designs which are copies of the communication structures of these organizations."

A.6 Software Architecture Activities

Often the SA activities can be summarized as:

1. Making the business case for the system
2. Exploring the architectural significant requirements
3. Software architecture development
4. Documenting and communication of the developed software architecture
5. SA evaluation/analysis
6. Implementation/testing of the system based on developed SA
7. Implementation conformance assurance to original SA

A.6.1 Quality Attributes

Quality attribute (QA) is defined as a "measurable ore testable property of a system that is used to indicate how well a system satisfies the needs of its stakeholders. [33]". Understanding the critical quality attributes that affect a software system's overall behaviour is an important criteria that the architect must poses. Software architecture's support of quality attributes is via the mapping of the system's functionality to architectural structures (modular, conceptual, and allocation).

A.6.2 Requirements to Quality Attributes

A software may include various requirements but only some of these requirements can be designed into the system as quality attributes. Such requirements are simply known as quality attribute requirements which are distinct from requirements that describe what the system must do or how it must behave or functional requirements. Functionality of the system has the strongest relationship to the architecture itself because it is the ability of the system to perform the task that the system was designed to do. However it is important to note that functionality does not dictate architecture as the same functionality may have an endless number of corresponding architectures.

A.6.3 SA Perspective on Quality Attributes

Quality attributes have been of a much debated and researched topic in software engineering since the 1970s. As it was previously mentioned the mapping from requirements to quality attributes is an early task that is accomplished by the architect. However similar mappings to the software architecture has proven to be difficult due to the following issues:

1. Quality attributes solely are not testable and their definitions are not contextualized. For instance, a system may be robust to a a class of faults and brittle to other types.

2. Often there is a vast overlap among quality attributes in the system and their interactions make the job of the architecture a complex one. For example, a security breach to the system can be seen as both an aspect of availability and security.

3. Each attribute has its set of vocabulary and terminology which add complexity to the mapping of all important quality attributes to the architecture itself.

A.6.4 QA Scenarios

The concept of *quality scenarios* intends to resolve some of these issue by incorporating methods in order to quantitatively verify that a quality attribute exists and in which context. QA scenarios allow to divide QAs into attributes that are the property of the system at run-time (i.e. performance) and attributes that are property of the development of the system (i.e. modifiability).

A.6.5 QA Requirements

In order to discuss quality attributes in a homogeneous fashion, the following 6 characteristics are defined for a QA scenario:

1. *Sources of stimulus:* This is the event which arrives at the system and generates a form of stimulus.

2. *Stimulus:* The condition that requires a form of response from the system.

3. *Environment:* This describes the conditions at which the stimulus occurs, in other explicitly describing the context.

4. *Artifact:* The subsystem or system as whole that is stimulated.

5. *Response:* The activity of system as the result of receiving the stimulus.

6. *Response measure:* The Quantifiable measurement of the response of the system.

A.6.6 Embedding QAs Into Design Decisions

As previously mentioned, an important aspect of software architecture is its ability to apply sound design choices in order to come up with a collective set of engineering tradeoffs of the quality attributes of the system that fulfills while fulfilling all business requirements. This collection of design decisions may be categorized as:

- Allocation of responsibility: It involves the identification of basic system functions, infrastructure to satisfy quality attributes and their allocation through static and run-time elements using modules and components.
- Coordination model: The mechanisms that collectively allow the interaction among software elements. Identifying which elements need communication, properties of the coordination, and defining communication mechanism are some of the design decisions.
- Data model: The interpretation of artifacts of system-wide interest or data as a collection may be refered to as a data model. Decisions such as, data abstractions, meta-data compilation, and data organization are associated with the data model.
- Management of resources: Decisions such as identification of resources and their limitations, system element to resource mapping, resource saturation analysis are part of this category.
- Mapping among architectural elements: An architecture provides mapping of element among each other in a given structure mapping but also the mapping of software elements to environment elements such as CPUs. The assignment of run-time elements, data model to data stores are included in this category.
- Binding time decisions: Such decisions establish the scope, the point in the life cycle and the mechanism of achieving variation for the rest of the decisions.

- Choice of technology: Deciding on which technology are available to realize the decisions of other categories, tools availability and the extent of familiarity of the team belong to this category.

A.7 QA: Availability

Availability refers to the property of software that it is ready to carry out its task when necessary. This general definition may extended to "the ability of the system to repair faults such that cumulative service outage period doesn't exceed a required value over a specified of time [33]". Availability as a quality attribute is mainly considered with minimizing service outage time by mitigating faults. Therefore in order to build a highly available system requires a complete understanding of the nature of the failures that can arise during the operation of the system. In reliability software engineering, there is an important distinction between *fault* and *failure*: A failure's cause is the fault. Such distinction allows discussion on strategies such as auto-repair where a failure in the system is masked to the user or repaired quickly enough which eliminates the occurrence of an actual fault.

A.7.1 Techniques Against Failure

In order to have a robust highly-available system in safety-critical systems, a through planning, design and analysis using a variety of techniques is essential. These techniques will allow for a better understanding of expected type of failures, their severity and an overall picture of what is required of the system. The realm of reliability engineering is vast however some of the most prominent techniques are discussed in the next sections.

A.7.1.1 Hazard Analysis

This technique uses a catalogue of hazards that may occur during the various operation modes of the system. These categorizes which vary in severity can range

from *catastrophic* to *hazardous* to *major* and finally *no effect*. Further analysis of the probability of each category maps the possible failures that may occur and allows the architecture and design team to focus on building strategies embedded into the system to eliminate faults from such possible failures.

A.7.1.2 Fault Tree Analysis

In this technique the various states of the system is analyzed to find the various ways that undesired states can occur. The graphical representation of the fault tree allow for static analysis using graph algorithms to identify single points of failure and also the probability of failures can be combined into one and traced back to a parent event that is contributing the most. The advantage of this technique is that it can be used at design time and run-time analysis. This allows various fault detection and recovery mechanism to be initiated upon analysis of the failures at run-time.

A.7.2 Availability General Scenario

The general scenario for availability is identified based on previously defined QA scenarios as follows:

- Sources of stimulus: Either internal and external origins.
- Stimulus: Either omission, crash, timing and/or incorrect response.
- Artifact: The resource that is most needed for the system to be highly available.
- Environment: The state of the system at the time of the failure or fault.
- Response: Either detection, recovery, logging, degrading mode with less performance and/or becoming unavailable until repair concludes.
- Response measure: It can be measured as availability percentage, time to detect fault or time to repair it, and/or time intervals that the system must be available.

A.7.3 Availability Tactics

These tactics may be categorized into three: fault detection, fault recovery, and fault prevention. Each category is described in more detail in the following section.

A.7.3.1 Fault Detection

Some of the detection tactics include:

- *Monitor*
- *Heartbeat*
- *Time stamp*
- *Sanity checking*
- *Condition monitoring*
- *Voting*

A.7.3.2 Fault Recovery

Some of the recovery tactics include:

- *Active redundancy (hot spare)*
- *Passive redundancy (warm spare)*
- *Spare (cold spare)*
- *Exception handling*
- *Rollback*
- *Software upgrade*
- *Retry*
- *Ignore faulty behaviour*

- *Degradation*
- *Reconfiguration*
- *Shadow*
- *State re-synchronization*
- *Escalating restart*
- *Non-stop forwarding*

A.7.3.3 Fault Prevention

Some of the detection tactics include:

- *Removal from service*
- *Transactions*
- *Predictive model*
- *Exception prevention*
- *increase competence set*

A.8 QA: Interoperability

Interoperability is the ability of a group of different or similar systems to usefully exchange information via interfaces in some particular context. A prior knowledge of the interface(s) of an external system allows software architecture tactics to incorporate this quality attribute into the system. In some cases without this knowledge, the Interoperability attributes of the design can be enhanced by designing generic enough interfaces that other systems can exchange information with with little effort.

A.8.1 Reasons to Use Interoperability

There are two scenarios that require Interoperability as a quality attribute in the system:

1. The system-to-be-designed provides a server to a mixture of known and unknown systems that may be added in the future.

2. When additional capabilities are added to an existing set of systems where these systems need to interoperate closer in order to provide such capabilities.

A.8.2 Interoperability General Scenarios

- Sources of stimulus: A system's initiation of request to interoperate.

- Stimulus: The request to exchange information.

- Artifact: The collection of systems that interoperate.

- Environment: Discovery of systems wishing to interoperate at runtime or known prior to runtime.

- Response: The request results in exchange of information. This information is understood by the receiving system both synthetically and semantically. (A rejection scenario can be viewed as response as well.)

- Response measure: The amount of information exchanged correctly or the percentage of correct rejection responses.

A.8.3 Interoperability Tactics

The tactics involved to incorporate Interoperability in the system involves a set of controls that allow a correct handle of exchange information request among the systems and services. In this context service is defined as a set of capabilities that can be accessed via an interface.

A.8.3.1 Locate Tactics

A tactic known as *Discover service* is where a a known service searches in a directory of services for the desired service based on name type, location or other attributes. This tactic is a runtime only tactic.

A.8.3.2 Interface Management Tactics

Another tactic which is part of the more interface oriented group of tactics is known as *Orchestrate*, where a control mechanism coordinates, manages and sequences the exchange among particular services. This tactic is used in complex interactions in complex tasks. The mediator design pattern is well suited for this type of interface management.

The second tactic in this category is the *Tailor interface*, where by adding capabilities such as translation of data or removing capabilities such as hiding particular functions from trusted users the exchange interface is capable of maintaining Interoperability. The decorator or wrapper pattern is most suited for this type of tactic.

A.8.4 Standards and Role of Semantics

It is important mention that standards alone will not be sufficient to guarantee Interoperability as the vague and open-ended nature of them allows for customization that make Interoperability between two systems using the same standard but from different vendors a complex task. Therefore an architecture first approach is needed prior to identifying which standards are best suited to that architecture.

A.8.5 Design for Interoperability

- Allocation of responsibility: Determine responsibilities of the system-to-be-designed that need Interoperability

- Coordination model: Ensure the coordination mechanisms meet other critical quality attributes such as performance for instance with regards to traffic volumes.
- Data model: Determine the syntax and semantics of the data abstractions to be exchanged.
- Management of resources: Ensure Interoperability does not exhaust critical system resources.
- Mapping among architectural elements: Ensuring processors external can communicate via network mechanisms while meeting security, availability and performance criteria.
- Binding time decisions: Identifying which systems at which instances of time will be interoperable with the use of proper response policies for both acceptance and rejection.
- Choice of technology: Visibility at the interface boundaries is critical in this context. Consider technologies that have built-in support for Interoperability with sufficient amount of control.

A.9 QA: Modifiability

Modifiability is an important quality attribute as studies have shown that the majority cost of a software system is often after it has been released. Its main concern is about the type of changes and cost/risk of those changes to the software system. This quality attribute intends to answer the following four important questions:

1. Which part of the system can change? This is important to the architect in terms of changes to platform, environment, quality attributes and capacity of the system.

2. What is the likelihood of the change? The architect decides on which changes are more probable given the current and future outlook of the software system.

3. When is the change made and who makes it? The time of change with respect to the product life cycle and the person making the change is important to modifiability. Specifically, changes can be made during development time (for example to the under-the-hood mechanisms of the system) or they can happen during usage of the system (for example a change to the graphic user interface).

4. What is the estimated cost of the change? Making a system more modifiable involves two types of costs: cost to introduce mechanisms to make it more modifiable and the cost of the modification using the mechanisms. This is a tradeoff scenario, as designing the system to be over modifiable involves upfront costs that are many not be necessary. On the other hand, less design for modifiablility up front may expose the design to higher cost later on as the mechanisms were not in place.

A.9.1 Modifiability General Scenarios

- Sources of stimulus: Either the developer, system administrator or end-user who specifies the change.

- Stimulus: The specification of making the change such additional/removal of functionality, defect removal, or change to other quality attributes.

- Artifact: Refers to what specifically will be changed in terms of components/-modules, platform, etc.

- Environment: This specifies when the change is to be made for example: design time, compile time, or run-time.

- Response: Change is made, test it, followed by deploying it.

- Response measure: Money or time can be misleadingly difficult as measures for change, other measure can be extent of the change or the number of new defects that have been introduced as a consequence of the change.

A.9.2 Modifiability Tactics

The objective of tactics for modifiability is to control the complexity of making changes with both time and cost constraints. Three major ca of modifiability that allow classification of tactics are:

1. *Size of a module:* Splitting of modules into smaller modules that reduce the cost and impact of making changes.
2. *Coupling:* Reducing the strength of coupling between two modules allow for reduced cost at the time of change of those two modules.
3. *Cohesion:* Improving cohesion by removing non-cohesive responsibilities or reassigning these responsibilities to another module that has more cohesiveness with said responsibility can also improve modifiability.

The second aspect of modifiability tactics is concerning on establishing when the changes must occur with respect to the project life cycle to have optimum modifiability. This is also known as *binding time of modification* which incorporate the idea that a modifiable system is one that accommodates modifications late in the life cycle (the tradeoff cost is preparation of the architecture for late binding).

A.9.2.1 The Tactics

- Reduce the size: *split module*
- Increase Cohesion: *increase semantic coherence*
- Reduce Coupling: *encapsulate, use an intermediary, restrict dependencies, refactor, abstract common services*

- Defer Binding: *compile time parameterization, aspects, configuration time binding, runtime binding: registration, dynamic lookup, startup time binding, plug-ins, publish subscribe, shared repositories, polymorphism*

A.9.3 Design for Modifiability

- Allocation of responsibility: For each category of change, determine responsibilities that need modification or impacted, keep the responsibilities that get impacted similarly in the same module(s).
- Coordination model: Determine which functionality can change at runtime that would affect coordination, ensure these changes are to a minimal number of modules. Use coordination models for impacted modules that reduce coupling, defer bindings or restrict dependencies.
- Data model: Determine determine which data abstractions will be modified, determine what happens with the data in terms of creation, initialization, persistence, manipulation , translation or destruction. Ensure an allocation of data abstraction that minimize the number and severity of modification to the abstractions.
- Management of resources: Determining what resource limits will change and the encapsulation of resource managers.
- Mapping among architectural elements: Determine the mapping time of elements, execution dependencies, assignment of data to databases.
- Binding time decisions: Determine latest time at which the change will be made, defer binding mechanisms for the right times, cost analysis of mechanisms, only few binding choices.
- Choice of technology: Is the technology assisting in easier modification, testing and/or deployment? choose technology that supports the most likely modifications.

A.10 QA: Performance

Performance is the ability of the software system to fulfill timing requirements in terms of interrupts, messages and request from users and other systems. It is a fundamental attribute of any designed system and one that has seen a vast amount of attention prior and during the design process. To achieve correctness in a system there is usually performance requirements which can be explicitly or implicitly defined. A deep understanding of performance requirements is essential to a well designed software system.

A.10.1 Performance General Scenario

A performance general scenario can be described as the arrival of event(s), a response from the system to that event via consumption of some form of system resource. These events may arrive in a predictable form, a probabilistic distribution or in an unpredictable fashion. Each type of event requires different approaches to fulfill the performance requirements of the system.

The response of the system may be measured via:

- *Latency*: The time interval between arrival of stimulus and the system response.
- *Deadlines in processing*: The checkpoints at each level of the system in order to generate the final response.
- *Throughput*: Number of transactions the system can process during a specified time interval.
- *Jitter*: The allowable variation in latency of the response.
- *Missed events*: The number of missed events due to saturation.

Using the above response measurements, the general scenario for performance can be concluded to be:

- Sources of stimulus: Internal and/or external sources of event arrivals.
- Stimulus: Event arrivals in either periodic, stochastic, or sporadic.
- Artifact: Refers to what specifically will be changed in terms of components/-modules, platform, etc.
- Environment: The various operational modes of the system such as normal or overload.
- Response: System's processing of the events arriving.
- Response measure: One or many of the measurements discussed previously.

A.10.2 Performance Tactics

Two contributors of the timing requirements that the performance often intends to fulfill are processing time and blocked time. Performance tactics allow the architect to control theses two times in order to effectively make engineering tradeoff with the end result of a performance oriented software system. Processing time consumes resources that inherently takes time. Handling resources is an important factor in controlling processing time. On the other hand, the blocked time is due to contention of needed resources, availability of resources or a dependency on previous computations. Therefore the tactics for performance can be categorized as:

1. Control resource demand: *manage sampling rate*, *limit event response*, *prioritize events*, *reduce overhead*, *bound execution time*, *increase resource efficiency*
2. Manage resources: *increase resources*, *introduce concurrency*, *maintain multiple copies of computation/data*, *bound queue sizes*, *schedule resources*

A.10.3 Design for Performance

- Allocation of responsibility: Determine system's responsibility with time-critical requirements and possible bottlenecks, responsibilities dealing with control

oriented threads or scheduling of resources and other control mechanisms such as buffer and queues.

- Coordination model: Determine elements that need coordination with appropriate communication mechanisms to support concurrency and can capture different type of events.

- Data model: Determine which parts of the data model are heavily used or have time-critical responses and ensure that at those levels of data abstractions whether multiple copy of data, partition of data, or addition of data oriented resources may benefit complying with those requirements.

- Management of resources: Determine time-critical resources in order to prioritize and implement efficient scheduling and locking mechanisms.

- Mapping among architectural elements: Identify heavy load points and use co-location, efficient assignment of processing capacity, concurrency driven with the use of smart thread control mechanisms that minimize occurrence of bottlenecks.

- Binding time decisions: Determine the necessary binding time for each element, additional overhead introduced by late binding mechanisms.

- Choice of technology: Choose a technology that allows for control of scheduling policies, priorities for reducing demand and allocation of processors according to the performance requirements of the system to be designed.

A.11 QA: Security

Security is the system's ability to protect data and other characteristics from unauthorized access without degrading access capabilities to authorized people or systems. Security can be characterized into six concepts as the following:

1. Confidentiality: Protection from unauthorized access.
2. Integrity: Protection from unauthorized manipulation of data.
3. Availability: Property that makes the system available for legitimate use even under security attacks.
4. Authentication: Verification of identities of the parties requesting a transaction from the system.
5. Non-repudiation: Property that guarantees the sender/receiver can not deny having sent or received interaction messages with the system.
6. Authorizing: Granting privileges to perform a task to certain user or group.

A.11.1 Security General Scenario

Similar to techniques in modelling faults for a system availability analysis using fault tree analysis, an "attack tree" is a threat modelling tool in order to identify weakest points of the system. A general scenario for security can be summarized as:

- Sources of stimulus: Unknown or previously identified human or system as the source of the attack.
- Stimulus: An unauthorized attempt to view or manipulate data or other system services known as attack.
- Artifact: The target of the attack which can be system data and/or services on vulnerable components of a system.
- Environment: The attack can happen during different operational modes of the system.
- Response: Protection of data/services from unauthorized access and manipulation, tracking of all behaviours with non-repudiation and availability of legitimate use of the system during attacks.

- Response measure: Time elapsed before an attack is detected, recovery time, amount of venerable data to a particular data are some of the measurements.

A.11.2 Security Tactics

In order to incorporate security into a software system, it is important to remember a physical security situation. IN general, there are four categories of tactics for security:

- Detect: *detect intrusion*, *detect service denial*, *verify message integrity*, *detect message delay*.
- Resist: *Identify actors*, *authenticate actors*, *authorize actors*, *limit access*, *limit exposure*, *encrypt data*, *separate identities*, *change default settings*.
- React: *revoke access*, *lock computer*, *inform actors*.
- Recover: *majority of availability tactics apply*.

A.11.3 Design for Security

- Allocation of responsibility: Determine which parts of the system need to be secure with efficient allocation to authenticate, authorize, grant access, record activity, recover from attacks and verify using checksums/hash values.
- Coordination model: Mechanisms in place with encryption of data and monitoring of transmission for activity.
- Data model: Determine the sensitivity of different data levels.
- Management of resources: Determine which resources need authorization, record access data or resources, detect resource exhaustion due to attacks.
- Binding time decisions: Determine late-bound components that may be venerable, use security certificates to access and manage late-bound services and data.

- Choice of technology: Choose a technology that allows for control of security measure in order to detect, resist, react and recover from possible attacks to security critical components of the system.

Bibliography

[1] Narayanan Sundaram. Making computer vision computationally efficient. 2012.

[2] Graphx: Unifying graphs and tables. `https://amplab.github.io/graphx/`. Accessed: 2016-01-02.

[3] Amqp 0-9-1 model explained. `https://www.rabbitmq.com/tutorials/amqp-concepts.html`. Accessed: 2016-01-22.

[4] Kurt Keutzer, Berna L Massingill, Timothy G Mattson, and Beverly A Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, page 9. ACM, 2010.

[5] Aleksandra Karimaa. Video surveillance in the cloud: Dependability analysis. In *Proceeding of the 4th International Conference on Dependability (DEPEND 11)*, pages 92–95, 2011.

[6] David Neal and Syed Rahman. Video surveillance in the cloud? *The International Journal of Cryptography and Information Security (IJCIS)*, 2(3), 2012.

[7] Yuzhong Yan and Lei Huang. Large-scale image processing research cloud.

[8] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[9] Trista P Chen, Dmitry Budnikov, Christopher J Hughes, and Yen-Kuang Chen. Computer vision on multi-core processors: Articulated body tracking. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1862–1865. IEEE, 2007.

[10] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[11] Mircea-Florin Vaida, Valeriu Todica, and Marcel Cremene. Service oriented architecture for medical image processing. *International Journal of Computer Assisted Radiology and Surgery*, 3(3-4):363–369, 2008.

[12] Chris Sweeney, Liu Liu, Sean Arietta, and Jason Lawrence. Hipi: a hadoop image processing interface for image-based mapreduce tasks. *Chris. University of Virginia*, 2011.

[13] Andreas Schierwagen. Vision as computation, or: Does a computer vision system really assign meaning to images? In *Integrative Systems Approaches to Natural and Social Dynamics*, pages 579–587. Springer, 2001.

[14] Alexandre R Francois. Software architecture for computer vision: Beyond pipes and filters. Technical report, DTIC Document, 2003.

[15] Scott Krig. Vision pipelines and optimizations. In *Computer Vision Metrics*, pages 313–363. Springer, 2014.

[16] TS Huang. Computer vision: Evolution and promise. *1996 CERN SCHOOL OF COMPUTING*, page 21, 1996.

[17] David Marr and A Vision. A computational investigation into the human representation and processing of visual information. *WH San Francisco: Freeman and Company*, 1982.

[18] Mubarak Shah. Guest introduction: the changing shape of computer vision in the twenty-first century. *International Journal of Computer Vision*, 50(2): 103–110, 2002.

[19] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[20] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th international conference on machine learning*, pages 1337–1345, 2013.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5 (8):716–727, 2012.

[22] Sankalita Saha and Shuvra S Bhattacharyya. *Design methodology for embedded computer vision systems*. Springer, 2009.

[23] Etienne Fluet-Chouinard, Bernhard Lehner, Lisa-Maria Rebelo, Fabrice Papa, and Stephen K Hamilton. Development of a global inundation map at high spatial resolution from topographic downscaling of coarse-scale remote sensing data. *Remote Sensing of Environment*, 158:348–361, 2015.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[25] Welcome to the jungle. `http://herbsutter.com/welcome-to-the-jungle/`. Accessed: 2015-10-30.

[26] M Anwar Hossain. Framework for a cloud-based multimedia surveillance system. *International Journal of Distributed Sensor Networks*, 2014, 2014.

[27] Harsh Agrawal, Clint Solomon Mathialagan, Yash Goyal, Neelima Chavali, Prakriti Banik, Akrit Mohapatra, Ahmed Osman, and Dhruv Batra. Cloudcv: Large-scale distributed computer vision as a cloud service. In *Mobile Cloud Visual Media Computing*, pages 265–290. Springer, 2015.

[28] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[29] Hipi: Hadoop image processing interface. `http://hipi.cs.virginia.edu/about.html`. Accessed: 2015-12-30.

[30] Mohamed H Almeer. Hadoop mapreduce for remote sensing image analysis. *International Journal of Emerging Technology and Advanced Engineering*, 2(4): 443–451, 2012.

[31] Muneto Yamamoto and Kunihiko Kaneko. Parallel image database processing with mapreduce and performance evaluation in pseudo distributed mode. *International Journal of Electronic Commerce Studies*, 3(2):211–228, 2013.

[32] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[33] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012. ISBN 0321815734, 9780321815736.

[34] Introduction to rabbitmq. `http://www.rabbitmq.com/resources/google-tech-talk-final/alexis-google-rabbitmq-talk.pdf`. Accessed: 2016-01-22.

[35] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, (6):87–89, 2006.

[36] Application programming interface. `https://en.wikipedia.org/wiki/Application_programming_interface`. Accessed: 2016-01-22.

[37] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[38] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[39] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, 2008.

[40] Go concurrency patterns: Pipelines and cancellation. `https://blog.golang.org/pipelines`. Accessed: 2016-01-27.

[41] Soheil Salehian and Behrouz Far. Embedded real time blink detection system for driver fatigue monitoring. 2015.

[42] Qiang Ji, Zhiwei Zhu, and Peilin Lan. Real-time nonintrusive monitoring and prediction of driver fatigue. *Vehicular Technology, IEEE Transactions on*, 53 (4):1052–1068, 2004.

[43] WW Wierwille. Overview of research on driver drowsiness definition and driver drowsiness detection. In *Proceedings: International Technical Conference on the Enhanced Safety of Vehicles*, volume 1995, pages 462–468. National Highway Traffic Safety Administration, 1995.

[44] DF Dinges and MM Mallis. Managing fatigue by drowsiness detection: Can technological promises be realized? In *International Conference On Fatigue and Transportation, 3RD, 1998, Frementle, Western Australia*, 1998.

[45] Hiroshi Ueno, Masayuki Kaneda, and Masataka Tsukino. Development of drowsiness detection system. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 15–20. IEEE, 1994.

[46] M Kaneda, H Iizuka, H Ueno, M Hiramatsu, M Taguchi, and M Tsukino. Development of a drowsiness warning system. In *Proceedings: International*

Technical Conference on the Enhanced Safety of Vehicles, volume 1995, pages 469–476. National Highway Traffic Safety Administration, 1995.

[47] Susumu Saito. Does fatigue exist in a quantitative measurement of eye movements? *Ergonomics*, 35(5-6):607–615, 1992.

[48] S Boverie, A Giralt, JM Lequellec, and A Hirl. Intelligent system for video monitoring of vehicle cockpit. Technical report, SAE Technical Paper, 1998.

[49] Robert Schleicher, Niels Galley, Susanne Briest, and Lars Galley. Blinks and saccades as indicators of fatigue in sleepiness warnings: looking tired? *Ergonomics*, 51(7):982–1010, 2008.

[50] Jeffrey W Muttart. Quantifying driver response times based upon research and real life data. In *3rd International Driving Symposium on Human Factors in Driver Assessment, Training, and Vehicle Design*, volume 3, pages 8–29, 2005.

[51] Artem A Lenskiy and Jong-Soo Lee. DriverâĂŹs eye blinking detection using novel color and texture segmentation algorithms. *International Journal of Control, Automation and Systems*, 10(2):317–327, 2012.

[52] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[53] Phillip Ian Wilson and John Fernandez. Facial feature detection using haar classifiers. *Journal of Computing Sciences in Colleges*, 21(4):127–133, 2006.

[54] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[55] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[56] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1): 32–46, 1985.

[57] Azriel Rosenfeld and Avinash C Kak. *Digital picture processing*, volume 2. Elsevier, 1982.

[58] Rangaraj M Rangayyan. *Biomedical image analysis*. CRC press, 2004.

[59] Andrew W Fitzgibbon, Robert B Fisher, et al. A buyer's guide to conic fitting. *DAI Research paper*, 1996.

[60] Ken-ichi Kanatani. Statistical bias of conic fitting and renormalization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(3):320–326, 1994.

[61] Chirag Pujara, Anurag Modi, G Sandeep, Shilpa Inamdar, Deepa Kolavil, and Vidhu Tholath. H. 264 video decoder optimization on arm cortex-a8 with neon. In *India Conference (INDICON), 2009 Annual IEEE*, pages 1–4. IEEE, 2009.

[62] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[63] Prithwish Basu, Wang Ke, Thomas DC Little, et al. Metrics for performance evaluation of distributed application execution in ubiquitous computing environments. In *Workshop on Evaluation Methodologies for Ubiquitous Computing at Ubicomp*, volume 1, 2001.

[64] Trista P Chen, Horst Haussecker, Alexander Bovyrin, Roman Belenov, Konstantin Rodyushkin, Alexander Kuranoc, and Victor Eruhimov. Computer vision workload analysis: Case study of video surveillance systems. *Intel Technology Journal*, 9(2), 2005.

[65] James D Mooney. Bringing portability to the software process. *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[66] James D Mooney. Issues in the specification and measurement of software portability. In *poster session at the 15th International Conference on Software Engineering*, 1993.

[67] Jörg Lenhard. Portability of process-aware and service-oriented software.

[68] Iso/iec 25010, July 2017. URL `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`.

[69] Jorg Lenhard and Guido Wirtz. Measuring the portability of executable service-oriented processes. In *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, pages 117–126. IEEE, 2013.

[70] Jörg Lenhard. Improving process portability through metrics and continuous inspection. In *Advances in Intelligent Process-Aware Information Systems*, pages 193–223. Springer, 2017.

[71] John D Musa. Operational profiles in software-reliability engineering. *IEEE software*, 10(2):14–32, 1993.

[72] Hacking opencv for fun and profit, January 2016. URL `http://computer-vision-talks.com/hacking-opencv-for-fun-and-profit/`.

[73] February 2016. URL `https://github.com/SoheilSalehian/opencv`.

[74] Go concurrency, February 2016. URL `http://dancallahan.info/journal/go-concurrency/`.

[75] James J Jiang, Gary Klein, Hsin-Ginn Hwang, Jack Huang, and Shin-Yuan Hung. An exploration of the relationship between software development process maturity and project performance. *Information & Management*, 41(3):279–288, 2004.

[76] Mohammad Ayoub Khan, Saqib Saeed, Ashraf Darwish, and Ajith Abraham. *Embedded and Real Time System Development: A Software Engineering Perspective: Concepts, Methods and Principles*, volume 520. Springer, 2013.

[77] Docker compose documentation, February 2016. URL `https://docs.docker.com/compose/`.