HARD-REAL-TIME COMPUTING PERFORMANCE IN A CLOUD ENVIRONMENT

by

Alvin Cornelius Murphy
B.S. May 1991, Virginia Polytechnic Institute and State University
M.S. Jan 2007, George Mason University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

ENGINEERING MANAGEMENT AND SYSTEMS ENGINEERING

OLD DOMINION UNIVERSITY
October 2022

Approved by:

James D. Moreland, Jr. (Director)

Andres Sousa-Poza (Member)

Samuel Kovacic (Member)

Saikou Diallo (Member)

ABSTRACT

HARD-REAL-TIME COMPUTING PERFORMANCE IN A CLOUD ENVIRONMENT

Alvin Cornelius Murphy
Old Dominion University, 2022
Director: Dr. James D. Moreland, Jr.

The United States Department of Defense (DoD) is rapidly working with DoD Services to move from multi-year (e.g., 7-10) traditional acquisition programs to a commercial industry-based approach for software development. While commercial technologies and approaches provide an opportunity for rapid fielding of mission capabilities to pace threats, the suitability of commercial technologies to meet hard-real-time requirements within a surface combat system is unclear. This research sets to establish technical data to validate the effectiveness and suitability of current commercial technologies to meet the hard-real-time demands of a DoD combat management system. (Moreland Jr., 2013) conducted similar research; however, microservices, containers, and container orchestration technologies were not on the DoD radar at the time. Updated knowledge in this area is desired to inform future DoD roadmaps and investments. A mission-based approach using Mission Engineering will be used to set the context for applied research. A hypothetical yet operationally relevant Strait Transit scenario has been established to provide context for definition of experimental parameters to be set while assessing the hypothesis. System models federated to form a system-of-systems architecture and data from a cloud computing environment are used to collect data for quantitative analysis.

NOMENCLATURE

ADF	Australian Defense Forces
AFLCMC	Air Force Life Cycle Management Center
AI	Artificial Intelligence
AMI	Amazon Machine Images
AMQP	Advanced Message Queuing Protocol
AOC	Air Operations Center
API	Application Programming Interface
App	Application
AWS	Amazon Web Services
B	Bytes
C2	Command and Control
C4	Context, Container, Component, and Code
CaaS	Container as a Service
CCI	Commonwealth Cyber Initiative
CERN	European Organization for Nuclear Research
CLI	Command Line Interface
CMS	Combat Management Systems
COVA	Coastal Virginia
CPU	Central Processing Unit
Dev	Development

DevSecOps	Development, Security, Operations
DIUx	Defense Innovation Unit Experimental
DoD	Department of Defense
DSS	Decision Support System
E2E	End-to-End
EC2	Elastic Cloud Compute
EMSE	Engineering Management and Systems Engineering
GB	Gigabyte
Gbps	Gigabits per second
GPU	Graphics Processing Unit
gRPC	Google Remote Procedure Call
HPA	Horizontal Pod Autoscaler
HSE	Health & Safety and Environment
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Data Environment
IM	Information Management
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
K8S	Kubernetes
MB	Megabyte

MDC2	Multi-Domain Command and Control
µs	Microsecond
ms	Millisecond
mTLS	Mutual Transport Layer Security
NIST	National Institute of Standards and Technology
Ops	Operations
OS	Operating System
OTP	Open Telecom Platform
PaaS	Platform as a Service
PC	Personal Computer
P_n	Probability of Negation
REMUS	Radiation and Environmental Unified Supervision
REST	Representational State Transfer
RPC	Remote Procedure Call
RPPC	Related Process Per Container
s	Second
SA	Situational Awareness
SaaS	Software as a Service
SCADA	Supervision, Control, and Data Acquisition
SCI	Software Configuration Item
SDK	Software Development Kit
Sec	Security

SLA	Service-Level Agreement
SME	Subject Matter Expert
SOAP	Simple Object Access Protocol
SPP	Scala Platform Process
TAO	Tactical Actions Officer
TDS	Tactical Decision Support
USAF	United States Air Force
VM	Virtual Machine
WC	Weapon Coordinator
XML	eXtensible Markup Language
YAML	YAML Ain’t Markup Language

TABLE OF CONTENTS

Page

LIST OF TABLES.....viii

LIST OF FIGURES.....ix

Chapter

1 INTRODUCTION..... 12

1.1 THEORETICAL FORMULATION..... 15
1.2 PURPOSE..... 21
1.3 PROBLEM STATEMENT AND RESEARCH QUESTIONS..... 24

2 BACKGROUND OF THE STUDY..... 26

2.1 REVIEW OF PRIOR RESEARCH..... 26
2.2 LIMITATIONS OF EXISTING STUDIES..... 46

3 METHODOLOGY..... 48

3.1 METHODOLOGICAL APPROACH..... 48
3.2 CONCEPTUAL ANALYSIS FRAMEWORK..... 49
3.3 MISSION CONTEXT..... 56
3.4 EXPERIMENT DESIGN..... 58
3.5 SOFTWARE IMPLEMENTATION..... 60
3.6 COMPUTING INFRASTRUCTURE..... 68
3.7 TELEMETRY COLLECTION..... 72

4 RESULTS..... 76

4.1 LOAD DATA FILES..... 76
4.2 CONVERT DATA INTO USEABLE METRICS..... 76
4.3 EXPLORATORY DATA ANALYSIS..... 77
4.4 HYPOTHESIS TESTING..... 98

5 DISCUSSION..... 101

5.1 OVERVIEW OF FINDINGS..... 101
5.2 RESEARCH IMPLICATIONS..... 105
5.3 RESEARCH LIMITATIONS..... 106

6 CONCLUSIONS..... 107

6.1 PRIMARY CONTRIBUTIONS OF THIS STUDY..... 107
6.2 WIDENING THE SCOPE..... 107
6.3 SUGGESTIONS FOR FUTURE RESEARCH..... 109

REFERENCES..... 111

LIST OF TABLES

TABLE 1. THE TWELVE FACTORS 19

TABLE 2. SOLID PRINCIPLES 21

TABLE 3. DEVOPS ANSWERS TO MDC2 REQUIREMENTS 38

TABLE 4. LIMITATIONS OF EXISTING STUDIES 47

TABLE 5. DSS PROTOTYPE COMPUTE RESOURCES 69

TABLE 6. QUESTION BASED HYPOTHESIS RESULTS 102

TABLE 7. DSS PROTOTYPE APPLICATIONS MAPPED TO MISSION USE CASES 103

TABLE 8. RESEARCH APPLICABILITY TO KNOWN COMMERCIAL CONCERNS 108

LIST OF FIGURES

Figure 1. DevSecOps Software Lifecycle 12

Figure 2. Mission Engineering, Systems Engineering, and Computer Science Core Competencies 13

Figure 3. Research Approach 14

Figure 4. Monoliths and Microservices (Lewis & Fowler, 2014) 16

Figure 5. Virtual Machines vs. Docker Containers 18

Figure 6. Deadline represented with value functions 22

Figure 7. Combat Management System (CMS) Scheduling Example 24

Figure 8. Microservices Literature Review 26

Figure 9. Overview of master-slave and nested-container models (Amaral et al., 2015) 27

Figure 10. Time to create an increasing number of instances of virtual containers (base 2 log scale in both axes). Where the nested-container is a fully initialized parent plus one child (Amaral et al., 2015) 29

Figure 11. Test configurations described in (Amaral et al., 2015) 29

Figure 12. Network Throughput and Latency for Different Configurations of Client/Server Under Bare-Metal, Container, and Virtual Machine on a Single Host Machine (Amaral et al., 2015) 30

Figure 13. Network Throughput and Latency Evaluation for Different Configurations of Client/Server Under Bare-Metal, Container, and Virtual Machines Across Two Hosts (Amaral et al., 2015) 30

Figure 14. Tools Used to Support the Testing of Microservices (Sotomayor et al., 2019) 32

Figure 15. Data Model (Mayer & Weinreich, 2018) 34

Figure 16. System Characteristics (Bogner et al., 2019) 36

Figure 17. MDC2 Vision "Multi-Domain Command and Control Operating Concept," 2016 from (Bruza, 2018) 38

Figure 18. Kafka Architecture (Dobbelaere & Esmaili, 2017) 42

Figure 19. RabbitMQ (AMQP) Architecture (Dobbelaere & Esmaili, 2017) 44

Figure 20. RabbitMQ vs. Kafka Latency Results (Dobbelaere & Esmaili, 2017) 45

Figure 21. Conceptual Analysis Framework 49

Figure 22. Research Mind Map 56

Figure 23. Mission Scenario OV-1 and Kill Chain 58
Figure 24. Microservice Performance Cause and Effect Diagram (Ishikawa) 59
Figure 25. Experimental Prototype Software Context 60
Figure 26. DSS Prototype Use Cases (OMG, 2015) 61
Figure 27. DSS Service Interactions (UML Sequence Diagram) (OMG, 2015) 62
Figure 28. DSS Component Diagram 64
Figure 29. DSS Deployment Diagram 67
Figure 30. The Role of the Artifact Repository (Humble & Farley, 2011) 68
Figure 31. DDS Prototype Development, Integration, and Test Environment 69
Figure 32. Amazon Machine Image (AMI) CPU Details 71
Figure 33. CCI Research Environment (Pratt, 2022) 72
Figure 34. Jaeger Visualization 73
Figure 35. Collection of I/O Metrics within Jaeger 75
Figure 36. DSS Prototype Data Files 76
Figure 37. Combined Data Summary 77
Figure 38. Duration vs. Use Cases and Computing Platform Source 78
Figure 39. Duration vs. Use Cases with Mac Platform Removed 79
Figure 40. Duration Histogram with Use Case Indicated (i.e. Trace.name) 80
Figure 41. Duration vs. Use Case Boxplot 81
Figure 42. Q-Q Plot of Duration per Use Case 82
Figure 43. Trace-Route from Home Lab to OpenSky API 83
Figure 44. OpenSky Endpoint in Gretzenbach, Switzerland 83
Figure 45. Density Plot of Duration Clusters 85
Figure 46. BIC Plot of Components 86
Figure 47. Best BIC Values 86
Figure 48. Classification Plot of BIC Component Relationships 87
Figure 49. Internal Duration Boxplot 88

FIGURE 50. INTERNAL DURATION HISTOGRAM 89

FIGURE 51. EXTERNAL DURATION BOXPLOT 90

FIGURE 52. EXTERNAL DURATION HISTOGRAM 91

FIGURE 53. SHAPIRO-WILK NORMALITY TEST 92

FIGURE 54. SHAPIRO-WILK TESTING WITH PROCESSING DELAY 93

FIGURE 55. ORIGINAL INTERNAL DURATION 94

FIGURE 56. MODIFIED INTERNAL DURATION WITH PROCESSING DELAY 95

FIGURE 57. ORIGINAL EXTERNAL DURATION 96

FIGURE 58. MODIFIED EXTERNAL DURATION WITH PROCESSING DELAY 97

FIGURE 59. REVISED DATA "GLIMPSE" WITH THRESHOLD INDICATOR 98

FIGURE 60. T-TEST RESULTS FOR INTERNAL USE CASE DATA 99

FIGURE 61. BINOMIAL TEST (ALL USE CASE DATA) 100

FIGURE 62. BINOMIAL TEST (EXTERNAL USE CASE DATA) 100

FIGURE 63. MAPPING OF MEASURED DURATIONS TO INTERNAL USE CASES 105

1 INTRODUCTION

The Defense Department is pursuing an aggressive software development program focused on bringing automated software tools, services, and standards to DoD programs so that software applications can be created, deployed, and operated in a secure, flexible, and interoperable manner (AFCEA, 2019). The effort, referred to as DevSecOps, merges software development efforts with operations to increase speed of software delivery. Security is integrated across the effort using automated scripts to identify vulnerabilities during development and operations. The DevSecOps software life-cycle from (DoD, 2019) is depicted in Figure 1. The approach harnesses so-called software containers for deployment of software as microservices, a dedicated repository of code and solutions that is secure and compliant with the Federal Risk and Authorization Management Program, or FedRAMP, and National Institute of Standards and Technology (NIST) criteria. The platform also utilizes Kubernetes, the Google-designed open-source container orchestration tool for automatically deploying and managing software containers (AFCEA, 2019).

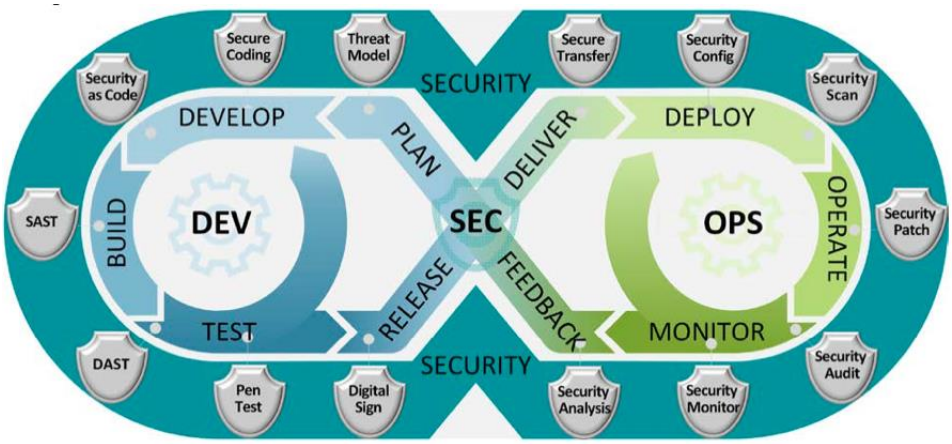

Figure 1. DevSecOps Software Lifecycle

While these technologies are well known and understood in commercial industry, they have not been used to date within surface combat systems. Prototype research is needed to verify and validate that it is possible to synthesize cloud computing and open-source technologies to realize a microservices architecture for a hard-real-time deterministic Combat Management System (CMS). An example of past surface combat system research to assess commercial technologies is provided in (Moreland, Sarkani, & Mazzuchi, 2014).

This research will use Mission and Systems Engineering techniques to design a relevant experimental prototype. Figure 2 summarizes the difference between mission engineering, system engineering, and computer science competencies; e.g. this research will not address detailed software design activities that would occur after the system engineering has translated mission requirements into system requirements ((OUSD(R&E), 2020), (ISO/IEC/IEEE, 2015), (INCOSE, 2015)).

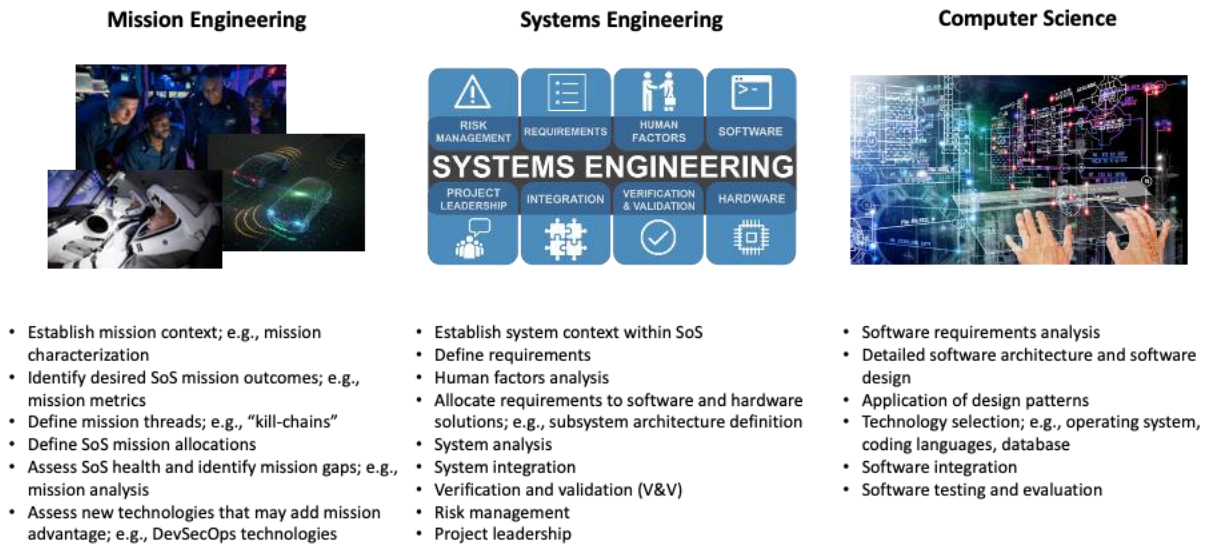

Figure 2. Mission Engineering, Systems Engineering, and Computer Science Core Competencies

Use cases will be used to define the prototype mission objects related to notional Decision Support System (DSS); e.g., weapon assessment, trial engage. DevSecOps reusable components (e.g., container) will be used to define system threads within a system-of-systems. Use of mission threads will enable the selection of container-based services needed to meet mission objectives without bringing along extra functional "baggage." The use of containers demonstrates an ability to quickly re-configure (e.g., composability) to meet emergent mission requirements beyond what was possible with a monolithic system.

The approach for this research is depicted in Figure 3.

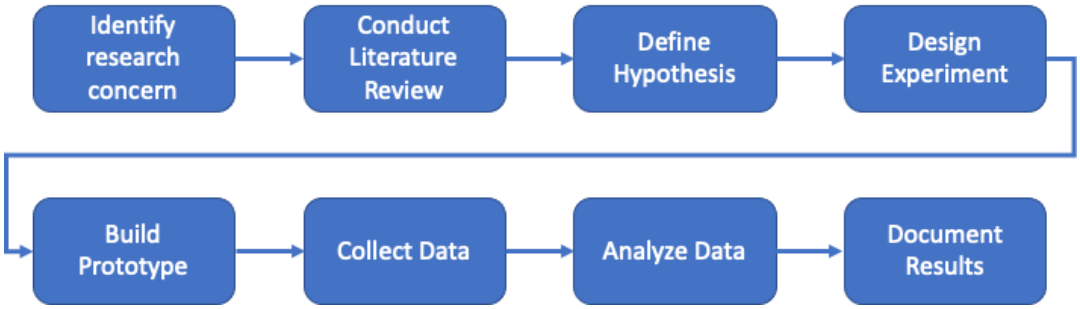

Figure 3. Research Approach

This document is organized as follows:

- Chapter 1 (Introduction) discusses the theoretical formulation behind the proposed research, discusses the purpose of this research, and identifies questions to be addressed.
- Chapter 2 (Background of the Study) presents a literature review and discusses limitations of past research.
- Chapter 3 (Methodology) discussed the methodology approach, articulates the mission engineering context, presents predictions and hypothesis, and discusses development and instantiation of the prototype test environment.

- Chapter 4 (Results) discusses the statistical analysis results.
- Chapter 5 (Discussion) provides and overview of findings and discusses research implications and limitations.
- Chapter 6 (Conclusions) presents contributions to the Engineering Management and Systems Engineering (EMSE) “body of knowledge” from this study and offers suggestions for future research.

1.1 THEORETICAL FORMULATION

(Grant & Osanloo, 2014) describes a theoretical framework as the foundation from which all knowledge is constructed (metaphorically and literally) for a research study. The theory that I will use is framed around the use of microservices and cloud native software principles to determine effectiveness and suitability of these technologies to meet the hard-real-time requirements of a CMS.

Microservices

A Microservices framework was first discussed in (Fernåndez-Villamor et al., 2010) as an attempt to simplify the process of defining service descriptions to push automatic service consumption in the semantic web. In this framework, the description task attempts to be improved by enabling reusability across service descriptions.

The current definition of microservices was formally introduced in 2014. (Lewis & Fowler, 2014) described microservices as an architectural style characterized around organization and business capability, automated deployment, intelligence in the endpoints, and

decentralized control of languages and data. Prior to the introduction of microservices, applications were considered to be monolithic. While the monolithic style has delivered successful software over the years, changes require rebuilding of the entire application prior to deployment of capability upgrades and bug fixes. (Lewis & Fowler, 2014) discusses that over time it's often hard to keep a good modular structure within monolithic applications, which makes it harder to keep changes from affecting multiple modules. Scaling to meet capacity demands requires scaling of the entire application rather than parts of it that require greater resource. A comparison of the two architectural styles is presented in Figure 4.

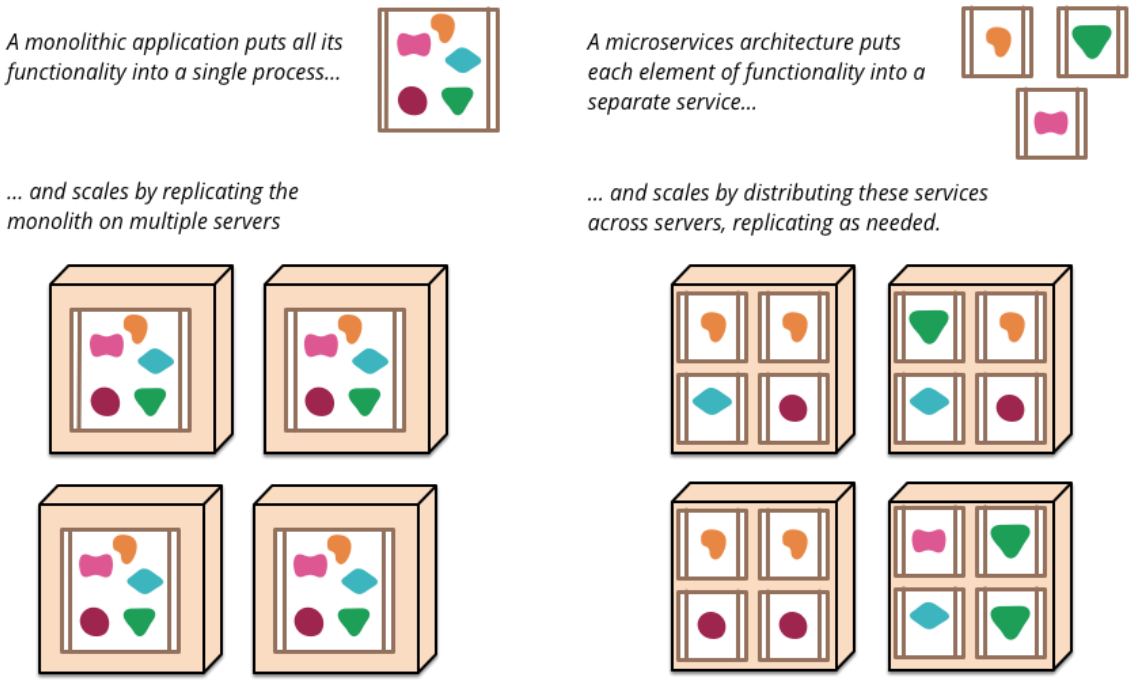

Figure 4. Monoliths and Microservices (Lewis & Fowler, 2014)

Virtual Machines vs. Containers

Microservices are typically deployed in Containers to enable rapid upgrades. However, it should be noted that the introduction of Virtual Machines was a technology enabler that led to the introduction of Containers. Figure 5 from (Janetakis, 2017) summarizes the difference

between the layers of a VM vs. a Container. Both use a server infrastructure that may be a desktop, laptop, or cloud provided server resource (e.g., Amazon). On top of the infrastructure is the operating system such as Mac OS, Windows, or Linux. The next layer introduces the differences. On a VM a Hypervisor is used to enable each VM to be its on self-contained computer running on the server infrastructure. By use of a Hypervisor, multiple guest operating systems (OS) may be run on the host machine. However, each guest OS require a full set of resource and library requirements. For example, if 3 guest OS instances are running and each is 700 MB, 3 times the server resources are required to support all 3 (e.g., 2.1 GB). Containers use a daemon to share a single instance base OS resource across multiple containers. Containers are packages as images that unique library resources as required by the individual applications. This enable starting of containers in milliseconds instead of minutes required to start up individual OS resources. However, each application running in the containers must be based upon the same base OS, e.g., Linux. Both approaches have differing use cases. For example, VMs enable isolation among full systems where's containers focus on isolation of applications to enable rapid deployment. This description is simplified by (Walsh & Duffy, 2015) in the Red Hat Container Coloring Book. Virtual machines can be thought of as houses that are self-contained with a standard set of amenities. You may have more infrastructure than is needed. Containers are like apartment where you are free to a pick a size that meets individual needs, e.g., studio apartment vs. a penthouse.

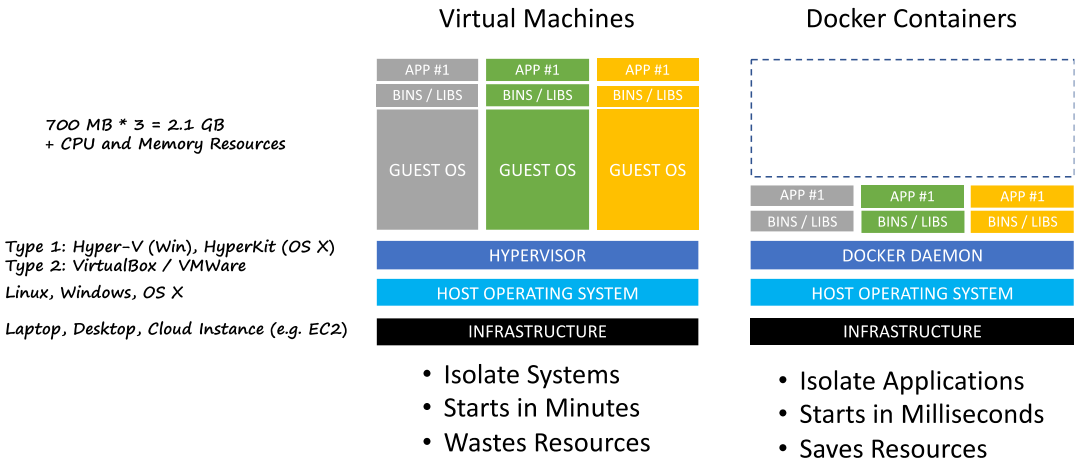

Figure 5. Virtual Machines vs. Docker Containers

Cloud-native Applications

(Gannon et al., 2017) provides an overview of the properties of a cloud-native application:

- Cloud-native applications operate at a global scale where application's data and services are replicated in local data centers so that interaction latencies are minimized. Consistency models are robust enough to give the user confidence in the integrity of the application.
- Cloud-native applications scale well with thousands of concurrent users while ensuring data synchronization and consistency.
- Cloud-native applications are built on the assumption that cloud infrastructure is fluid and failure is constant.
- Cloud-native applications are designed so that upgrade and test occur seamlessly without disrupting production.

- Security is not an afterthought in cloud-native applications where security is part of the underlying application architecture.

Twelve-Factor Application

The twelve-factor application methodology is complimentary to realizing cloud-native applications. The twelve-factor app is a methodology for building software-as-a-service applications that (Wiggins, 2017):

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments;
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;
- Minimize divergence between development and production, enabling continuous deployment for maximum agility;
- And can scale up without significant changes to tooling, architecture, or development practices.

The twelve factors from (Wiggins, 2017) are summarized in Table 1.

Table 1. The Twelve Factors

Factor	Description
I. Codebase	One codebase tracked in revision control, many deploys
II. Dependencies	Explicitly declare and isolate dependencies
III. Config	Store config in the environment
IV. Backing services	Treat backing services as attached resources
V. Build, release, run	Strictly separate build and run stages
VI. Processes	Execute the app as one or more stateless processes

Factor	Description
VII. Port binding	Export services via port binding
VIII. Concurrency	Scale out via the process model
IX. Disposability	Maximize robustness with fast startup and graceful shutdown
X. Dev/prod parity	Keep development, staging, and production as similar as possible
XI. Logs	Treat logs as event streams
XII. Admin processes	Run admin/management task as one-off processes

Reactive Manifesto

The Reactive Manifesto defines criteria for building systems that are more flexible, loosely coupled, and scalable. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance, and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in petabytes. Today's demands are simply not met by yesterday's software architecture. Systems built as reactive systems are easier to develop and amenable to change. Reactive Systems are significantly more tolerant of failure when failure does occur then meet it with elegance rather than disaster. Large systems are composed of smaller ones and therefore depend on the Reactive properties of their constituents. This means that Reactive Systems apply design principles so these properties apply at all levels of scale, making them composable (Bonér et al., 2014).

SOLID Principles

(Martin, 2017) discusses the SOLID principles that tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected. The goal of the principles is the creation of mid-level software structures that:

- Tolerate change,

- Are easy to understand, and
- Are the basis of components that can be used in many software systems

The term “mid-level” refers to the fact that these principles are applied by programmers working at the module level. They are applied just above the level of the code and help to define the kinds of software structures used within modules and components. The SOLID principles from (Martin, 2017) are summarized in Table 2.

Table 2. SOLID Principles

SOLID Principle	Description
SRP: The Single Responsibility Principle	An active corollary to Conway’s law: The best structure for a software system is heavily influenced by the social structure of the organization that uses it so that each software module has one, and only one, reason to change.
OCP: The Open-Closed Principle	Bertrand Meyer made this principle famous in the 1980s. The gist is that for software systems to be easy to change, they must be designed to allow the behavior of those systems to be changed by adding new code, rather than changing existing code.
LSP: The Liskov Substitution Principle	Barbara Liskov’s famous definition of subtypes, from 1988. In short, this principle says that to build software systems from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted one for another.
ISP: The Interface Segregation Principle	This principle advises software designers to avoid depending on things that they don’t use.
DIP: The Dependency Inversion Principle	The code that implements high-level policy should not depend on the code that implements low-level details. Rather, details should depend on policies.

1.2 PURPOSE

Prototype research in the surface combat system domain is needed to verify and validate that it is possible to synthesize cloud computing and open source technologies to realize a microservices architecture for a hard-real-time deterministic Combat Management System (CMS). (Abbott, 2017) discusses the severity of missing a hard, soft, and firm real-time deadline. Real-time is focused on getting the expected result given input “in time” for the

response to be useful, e.g., meeting a deadline. Missing a hard-real-time deadline can result in catastrophic mission failure. An example of past surface combat system research to assess hard-real-time with commercial technologies is provided in (Moreland et al., 2014).

Figure 6 from (Wang, 2011) illustrates the impact of missing a deadline in a real-time system. V(t) defines the value of making or missing a deadline. In soft real-time subfigure (a) if a deadline is missed the value of the data provided gracefully degrades over time. In firm real-time subfigure (b) if a deadline is missed, the data after the deadline is of no value. In hard essential real-time subfigure (c) if the deadline is missed, there is a known defined penalty (n). In hard critical real-time subfigure (d) if the deadline is missed, the realized disaster is of exponential unknown impact.

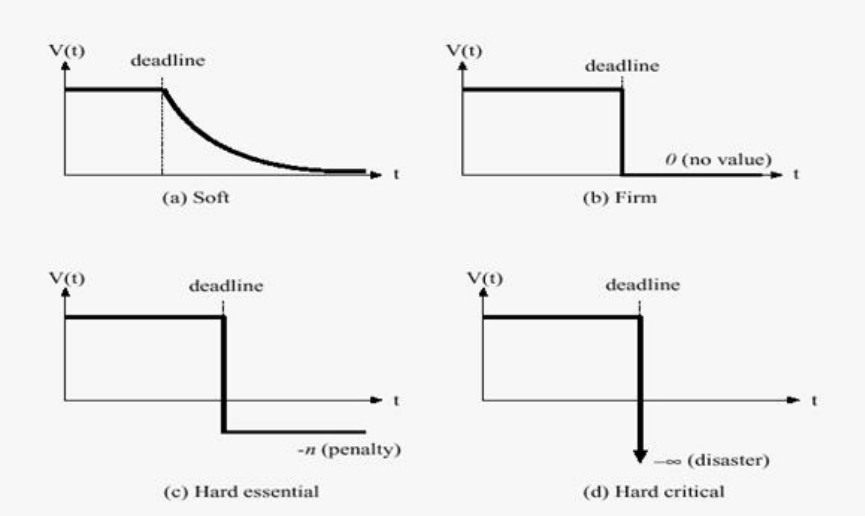

Figure 6. Deadline represented with value functions

In Figure 7 we use an end-to-end (E2E) mission analysis approach adapted from (Firesmith, 2019) to define a CMS use case to illustrate the impact of missing a real-time deadline. Mission threads are used to identify potential integration problems. In the scenario

depicted, the Weapon Coordinator software component requests effectiveness options from the integrated Laser, Missile, and Gun Controllers. The mission thread steps are as follows:

1. CMS operator enters threat criteria into the Tactical Decision Support (TDS) Software. Ordered engagements are designated as semi-auto; i.e. the Weapon Coordinator (WC) has weapon designation authority with operator command by negation.
2. Track data provided by the Information Management (IM) Software qualifies a track as a threat.
3. TDS orders Weapon Coordinator (WC) Software engagement of the threat.
4. WC asks all weapons for predicted effectiveness via associated Weapon Controllers (WCtrl). Each WCtrl is given X ms to respond.
5. Each WCtrl responds to WC within X ms with probability of kill/negation (Pn) and cost factor.
6. The WC selects the weapon based upon a Greedy Algorithm approach.
7. WC orders weapon engagement and reports status to TDS.
8. TDS provides Situation Awareness (SA) to the CMS operator prior to opening fire.

Let's assume that the hard-real-time deadline for controller responses is set to 75 milliseconds. The Weapon Coordinator would need to make a weapon selection decision prior to getting the Laser response in 100 milliseconds. A less effective, higher cost weapon would be selected and potentially waste a weapon that could be more effective in a future engagement.

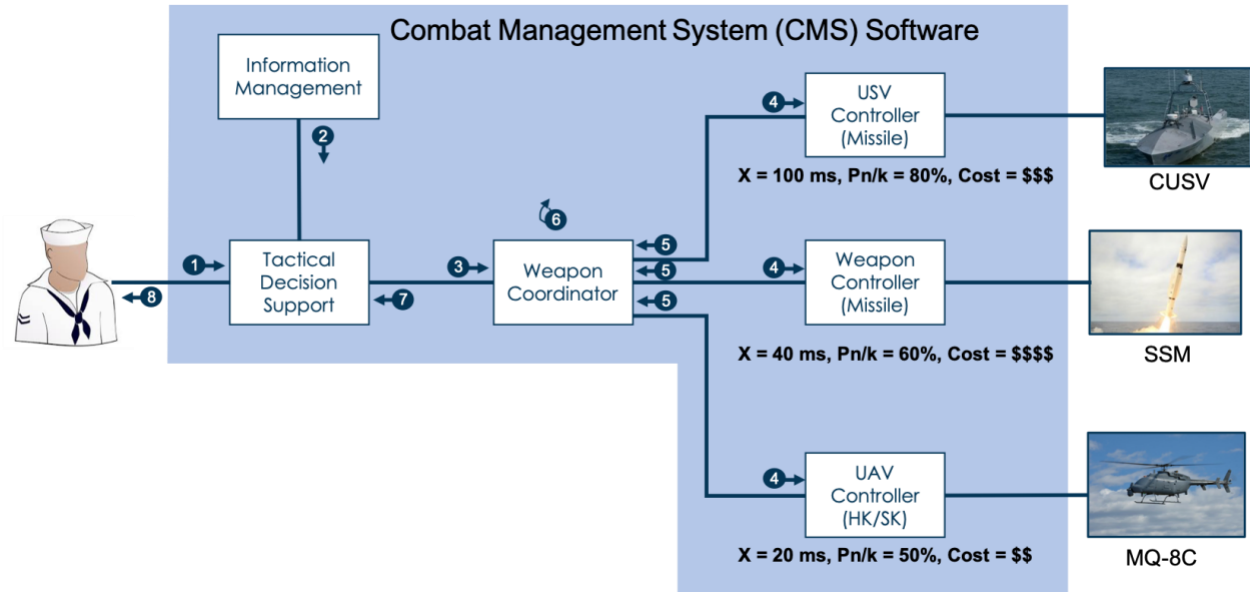

Figure 7. Combat Management System (CMS) Scheduling Example

1.3 PROBLEM STATEMENT AND RESEARCH QUESTIONS

While commercial technologies and approaches provide an opportunity for rapid fielding of capabilities to pace threats, the suitability of commercial technologies to meet hard-real-time requirements within a surface combat system is unclear. The following questions are posed based upon literature review.

Questions

The primary question is related to the deterministic nature of DevSecOps technologies.

Q1: Is it possible to synthesize cloud computing and open-source technologies to realize a microservices architecture for a hard-real-time deterministic Combat Management System (CMS)?

Additional questions are proposed based upon the use of mission analysis and specific technology implementation.

Q2: Does end-to-end (E2E) mission thread analysis increase SoS interoperability and reduce integration issues?

Q3: What benefits are gained from a microservice based DevSecOps approach?

Q4: Is the resultant architecture hard-real-time deterministic?

2 BACKGROUND OF THE STUDY

Several key publications from the existing body of knowledge are presented and research areas lacking sufficient study are identified to inform this research.

2.1 REVIEW OF PRIOR RESEARCH

Figure 8 provides an overview of literature identified as relevant to this research. Each article is organized by the research question identified in Section 1.3.

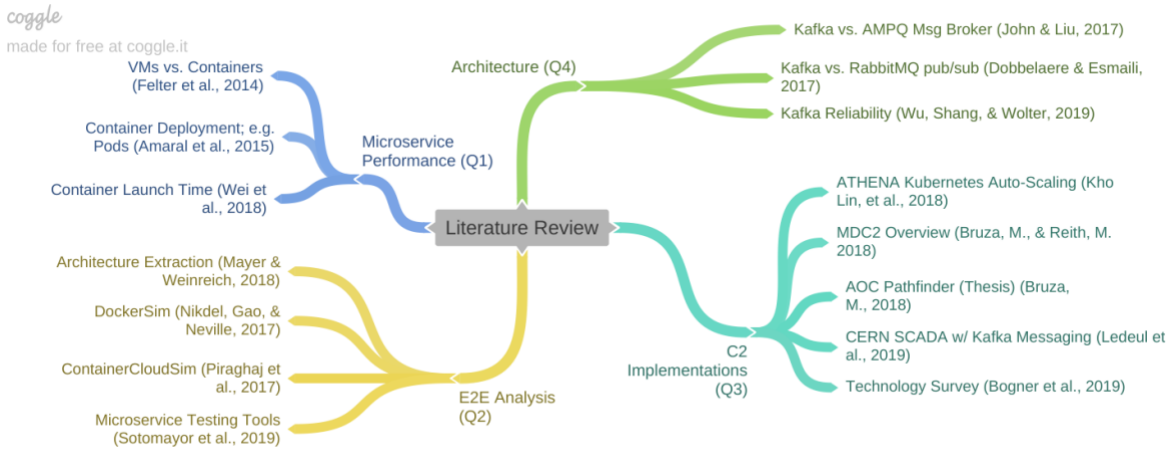

Figure 8. Microservices Literature Review

<u>Container Based Microservice Performance (Q1)</u>

Multiple studies have been conducted on cloud-based deployment and orchestration technologies. We shall focus on the performance of a container-based deployment of microservices (see Question 1).

(Felter et al., 2014) compares deployment of software in VMs versus containers and finds that Docker containers equal or exceed performance in every case tested. A suite of

workloads is used to stress the CPU, memory, storage, and networking resources. The authors suggest that their findings imply against implementation on IaaS using VMs and PaaS implemented as containers. A container-based IaaS can offer better performance. The authors assess the performance of Redis and MySQL in the different environments.

(Amaral et al., 2015) focuses on a deployment model where one process (or few related processes) is deployed per container. The deployment technique is referred to as Related Process Per Container or RPPC. The authors cite that two different approaches emerge for deployment of containers: master-slave or nested containers (see Figure 9).

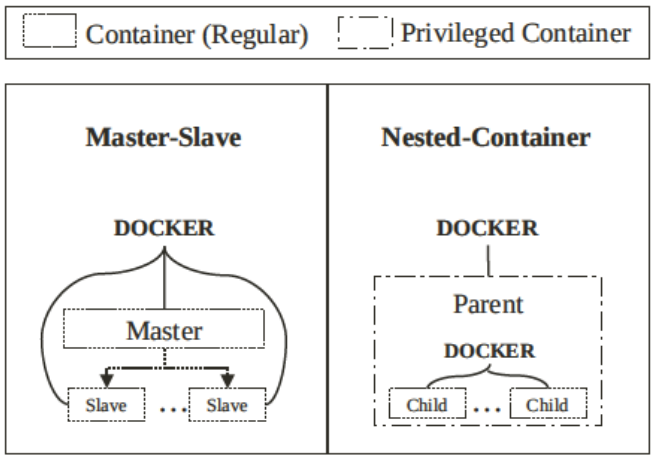

Figure 9. Overview of master-slave and nested-container models (Amaral et al., 2015)

The master-slave approach discussed by (Amaral et al., 2015) is composed of one container as the master coordinating other containers called slaves, in which application processing will be running. In this approach the master needs to track the subordinates' containers, help their communication, and guarantee that the slaves do not interact with other containers from a different master. The authors refer to the *master-slave* approach as "regular

container." In the *nested-container* approach, the subordinates' containers (the children) are hierarchically created into the main container (parent). The children run the application process and they are limited by the parent's boundaries. The nested-container approach might be easier to manage since all other containers are inside only one container. This approach may benefit from sharing the same memory, disk, and network; however, there may be an overhead penalty. The authors state that the nested-containers are inspired by the "pod" concept that is implemented by Google for better managing Docker containers (i.e. Kubernetes) (Google, 2020). The authors cite their research contribution as providing a benchmark analysis for container virtualization by implementing nested and master-slave containers hence comparing the performance against virtual machines. The authors compare (i) bare metal, (ii) regular containers (i.e. master slave), (iii) nested containers, and (iv) virtual machines (VMs).

(Amaral et al., 2015) noted that the creation of regular containers is the fastest deployment approach as expected, followed by nested containers, and VMs. However, the creation of a single nested-container has almost 8 times more overhead than the creation of one regular container, but the creation of nested-containers is still more than twice as fast as virtual machines. The authors attribute this nested container overhead to initialization of the Docker container host platform within the parent container, which involves loading an image stored locally on the host and creation of the child container itself. The image loading was taking an average of 6.2s (see Figure 10).

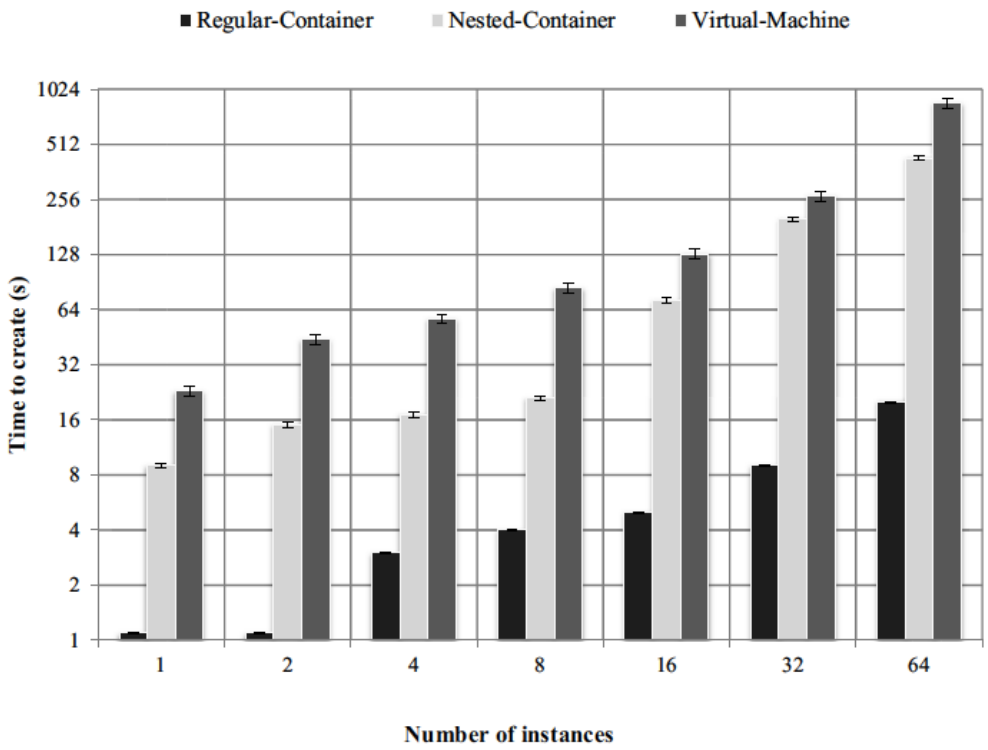

Figure 10. Time to create an increasing number of instances of virtual containers (base 2 log scale in both axes).

Where the nested-container is a fully initialized parent plus one child (Amaral et al., 2015)

(Amaral et al., 2015) assess network throughput and latency for single intra-host and host-to-host communication. Figure 11 depicts the configurations discussed by the authors. Results are summarized in Figure 12 and Figure 13.

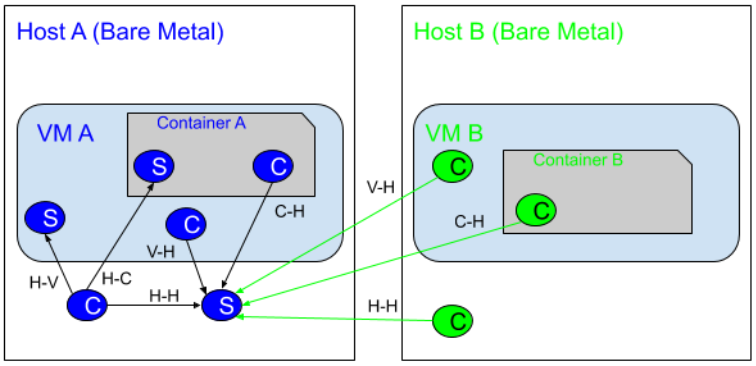

Figure 11. Test configurations described in (Amaral et al., 2015)

	Throughput			Latency		
(Client - Server)	Host-Network	Linux Bridge	Open vSwitch	Host-Network	Linux Bridge	Open vSwitch
Host - Host	35.71 Gbps σ=0.32	-	-	102.77 μs σ=0.95	-	-
Container - Host	35.13 Gbps σ=0.48	15.82 Gbps σ=0.36	16.01 Gbps σ=0.47	104.48μs σ=1.45	231.97μs σ=5.3	229.37μs σ=6.38
Host - Container	34.96 Gbps σ=0.63	15.96 Gbps σ=0.51	16.86 Gbps σ=0.35	105.0μs σ=1.94	230.17μs σ=7.35	217.76μs σ=4.63
Virtual machine - Host	-	8.64 Gbps σ=0.28	7.94 Gbps σ=0.69	-	424.92μs σ=14.09	465.53μs σ=43.57
Host - Virtual machine	-	9.24 Gbps σ=0.27	8.77 Gbps σ=0.55	-	397.53μs σ=12.08	420.14μs σ=27.09

Figure 12. Network Throughput and Latency for Different Configurations of Client/Server Under Bare-Metal, Container, and Virtual Machine on a Single Host Machine (Amaral et al., 2015)

	Throughput			Latency		
(Client - Server)	Host-Network	Linux Bridge	Open vSwitch	host-network	Linux Bridge	Open vSwitch
Host - Host	142.21 Mbps σ=8.64	-	-	25.97 ms σ=1.7	-	-
Container - Host	157.92 Mbps σ=1.06	154.51 Mbps σ=5.22	157.25 Mbps σ=3.95	23.29 ms σ=0.15	23.83 ms σ=0.82	23.40 ms σ=0.58
Virtual machine - Host	-	135.92 Mbps σ=6.77	136.92 Mbps σ=5.37	-	27.13 ms σ=1.31	26.5 ms σ=1.23

Figure 13. Network Throughput and Latency Evaluation for Different Configurations of Client/Server Under Bare-Metal, Container, and Virtual Machines Across Two Hosts (Amaral et al., 2015)

(Wei et al., 2018) notes that there are very few studies revealing the overheads, such as starting new containers in orchestration systems, such as Kubernetes. Though traditional Virtual Machines (VMs) can take on the order of minutes to launch, containers are much faster, and the launch times can be on the order of seconds. These overheads are typically considered to be negligible compared with the benefits of container-based systems; however, are they predictable?

(Wei et al., 2018) investigates these costs in a systematic study within a private cloud platform. The evaluation outlines a process for studies of this kind. Study results confirm that launch times of VMs are in the range of minutes, whereas containers typically only take seconds. However, these results also show that launch times for new containers do not always

scale linearly. Specifically, the authors discuss that a system organized by Minikube, a tool that eases local deployment of Kubernetes, introduces a penalty on launch times once the number of containers exceeds 80% of the maximum number of pods available for the cluster. This work demonstrates the presence of unexpected overheads and the need for a systematic infrastructure for testing deployments of containerized services at scale.

Architecture Analysis Tools (Q2)

Assessment of a microservice based E2E architecture will require automated testing tools as well as architecture documentation for detailed static analysis by CMS subject matter experts (SMEs). The following articles have been deemed relevant (see Question 2).

(Sotomayor et al., 2019) provides a comparison of tools used for runtime testing of microservice architectures (see Figure 14). There are several tools to support the testing of microservices, including tools to support different levels of testing (e.g., unit, integration, and system). The authors cite testing challenges due to the added complexity of network communications between collaborating services.

ContainerCloudSim is an extension of CloudSim developed by (Piraghaj et al., 2017) that provides support for modeling and simulation of containerized cloud environments. The simulation supports comparison of container scheduling and provisioning policies in terms of energy efficiency and SLA compliance. The authors provide an overview of similar efforts; however, previous efforts do not support modeling and simulation of containers in a cloud environment. ContainerCloudSim offers a Container as a Service (CaaS) model that consists of containerized cloud data centers, hosts, virtual machines, containers, and applications along

with their workloads. The authors demonstrated that ContainerCloudSim can support large scale CaaS simulations of up to 5,000 containers. They believe that their research will energize research in CaaS policies.

No.	*Name*	*Interface Method*	*Implementation*	*Platform*	*Test Cases Language(s)*	*Testing Objectives/ Support*	*Testing Strategies*
1	Appium [10]	WebDriver	IDE	iOS, Android	Any WebDriver compatible language	Regression, Exploratory	System
2	Docker Compose Rule [11]	HTTP Requests	Library	JVM	Java	Regression	Integration, System
3	Gatling [12]	HTTP Requests	Framework	JVM, SPP (Scala)	Java, Scala	Performance, Stress	Component, System
4	Gremlin [13]	HTTP Requests	Side-car + Proxy	Linux, Python Runtime Environment (PRE)	Linux Scripting Language, Python	End-to-End	Component, Integration, System
5	Hoverfly [5]	HTTP Requests	Side-car + Proxy	JVM, PRE, MacOS, Windows, Linux	Java, Python, Scripting Language	End-to-End	Component, Integration
6	JMeter [14]	HTTP Requests	IDE	JVM	Java	Functional, Regression	Component, Integration
7	JUnit [15]	Method call	Framework	JVM	Java	Functional, Regression	Unit
8	Minikube [16]	CLI	Container's Orchestrator	Linux, Windows, MacOS	Scripting Language	Test Harness	Component, Integration
9	NUnit [17]	Method call	Framework	.Net Framework	.Net Languages	Functional, Regression	Unit
10	Pact [18]	HTTP Requests	Framework	Cross-section of runtime systems	Any supported language	Regression, Component	Contract
11	Selenium IDE [19]	HTTP Requests	Web browser extension	Chrome, Firefox	Selenium Script	Regression, Exploratory	System
12	Spring Boot Starter Test [20]	Method Call	Framework	JVM	Java	Test Harness	Unit
13	Spring Cloud Contract [21]	HTTP Requests	Framework	JVM	Groovy, YAML	Regression, End-to-End	Contract
14	Telepresence [22]	Two-way proxy	CLI	Windows, Linux	Any	Test Harness	Component
15	Tsung [23]	HTTP Requests	Framework	OTP (Erlang)	XML	Stress	Component, System
16	Watir [24]	WebDriver	CLI	Ruby VM	Ruby	Regression	System

Figure 14. Tools Used to Support the Testing of Microservices (Sotomayor et al., 2019)

DockerSim is an extension of the iCanCloud platform developed by (Nikdel et al., 2017). DockerSim adds (i) a full container deployment and behavioral layer, (ii) full packet-level network and protocol behaviors, (iii) full packet-level scheduling behaviors, and (iv) a generic queuing network approach to modeling application-layer software as a service (SaaS) deployment. The authors cite that their approach adds a research capability to support scientific methods per-experiment control and repeatability tenants. Within existing cloud simulators (e.g., CloudSim, iCanCloud, etc.), a common choice has been made to trade-off network- and packet-level fidelity in favor of increased simulator performance. The authors

state that ContainerCloudSim does not seek to incorporate the (i)-(iv) capabilities discussed above, thereby, placing DockerSim in a better position to support research into timing sensitive cloud computing issues whereas ContainerCloudSim is better positioned to explore energy consumption issues.

(Mayer & Weinreich, 2018) presents an approach to extract and analyze the architecture based on a combination of static service information with infrastructure-related and aggregated run-time information. The agility of a microservices based architecture, which results from independent development and integration of new services, leads to continuous architectural changes. The authors propose a generic way to retrieve the necessary static and dynamic data from different distributed microservices, with the collected information combined in a central location. Close attention has been paid to continuously extracting the architecture over a long period of time. To support long-term analysis, an aggregation process has been established to condense the collected run-time information. This approach facilitates identification of design weaknesses, managing of service APIs, and management of scaling issues.

(Mayer & Weinreich, 2018) developed a data model for collection of architectural information (see Figure 15). As shown in the data model, architectural information is important at three different levels in a microservice-based software system: services, infrastructure, and interaction. The data model was optimized for use in a graph database that contains nodes which relate to directed edges. The authors point out that using a graph database, simple transitive queries can be written, for example, to load all services with which a specific service is communicating directly or indirectly. The authors surveyed 15 architects, developers, and

operations experts. The study participants identified information about service APIs, service interactions and dependencies, service version, the number and distribution of service instances, and system metrics as most important. Prototype dashboards were developed based upon these use case. The authors demonstrated feasibility of their approach to support automated generation of documentation and dashboards to support analysis, maintenance, and software development to add capability.

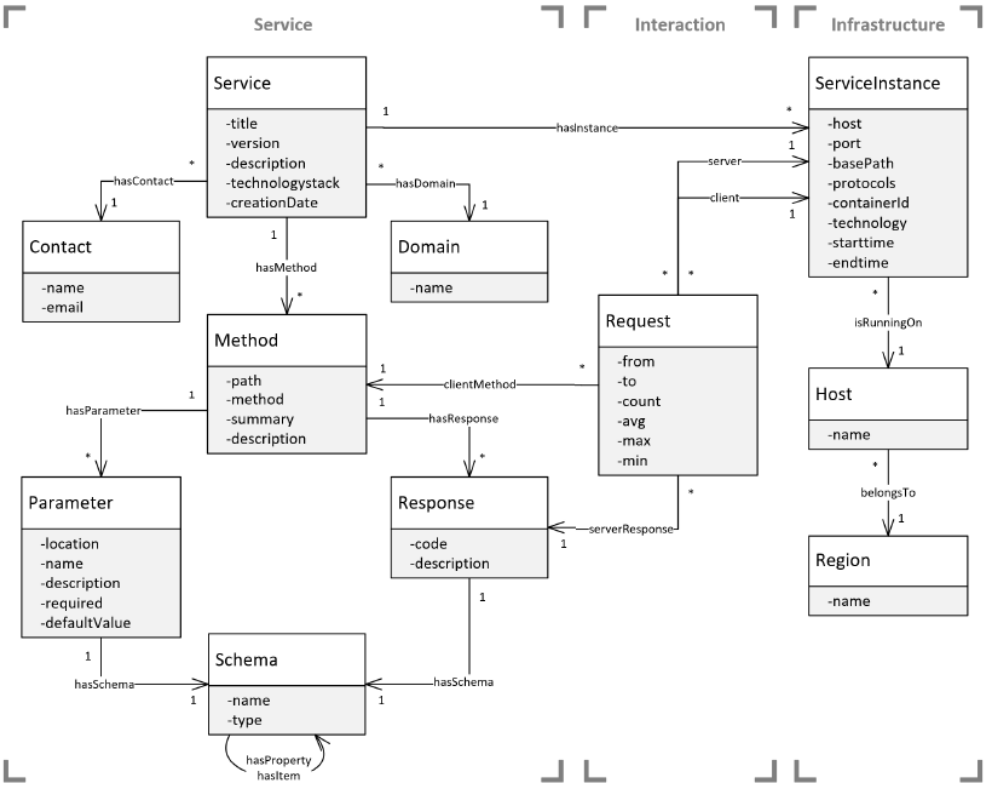

Figure 15. Data Model (Mayer & Weinreich, 2018)

C2 Implementations (Q3)

Could computing and open-source technologies are currently implemented in mission critical system with hard-real-time deterministic requirements in Public Sector research and DoD environments. These examples provide insight into what is potentially possible within a CMS implementation and potential benefits (see Question 3). (Bogner et al., 2019) provides an overview of state of the practice solution architectures followed by specific case study implementations.

During 17 interviews (i.e. P1-P17) with software professionals from 10 companies, (Bogner et al., 2019) analyzed 14 service-based systems summarized in Figure 16 (i.e. S1-S14). The interviews focused on applied technologies, microservices characteristics, and the perceived influence on software quality. The authors found that companies generally rely on well-established technologies for service implementation, communication, and deployment. Most systems, however, did not exhibit a high degree of technological diversity as commonly expected with microservices. The de facto standard for microservice communications was RESTful HTTP. Representative state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating web services that may be accessed using the web-based hypertext transfer protocol (HTTP). Even though it was not the primary protocol in each of the 14 cases, it existed in all of them, sometimes for minor interfaces. Participants named interoperability, technology independence, and loose coupling as advantages, even though most participants that used REST felt no need to justify this decision. Some participants saw direct synchronous RESTful communication between services as harmful (P5, P6, P15) and relied more on messaging to decouple services further, which P6 saw as follows: “We also have some REST-based communication between services, which is not 100% clean. In some cases, we

had to choose between performance or clear data ownership, so we compromised." Kafka was the preferred messaging solution followed by the Advanced Message Queuing Protocol (AMQP). In one case, the remote procedure call (RPC) developed by Google (gRPC) was chosen to replace REST, because its streaming nature was seen as more efficient and end-user friendly (P10). Reactive Microservices and event sourcing were used by some participants (P10, P15, P17). No new system relied on SOAP for communication. SOAP was sometimes simply kept to integrate with legacy systems.

TABLE II
SYSTEM CHARACTERISTICS

ID	Purpose	Inception	Hosting	# of People	# of Services	Communication	Languages	Artifacts
S1	Derivatives management system (banking)	Rewrite	public cloud (AWS)	7	9	REST, AMQP	Java, Kotlin	JAR
S2	Freeway toll management system	Rewrite & Extension	private cloud (OpenShift)	10 (only devs)	10	Oracle Advanced Queuing, REST	Java, C++ for algorithm	Docker
S3	Automotive problem management system	Rewrite & Extension	private cloud (OpenShift)	50	10	REST, Kafka	Java	Docker
S4	Public transport sales system	Rewrite & Extension	on-premise, public cloud (AWS)	~300	~100	REST, Kafka, AmazonMQ	Java, Node.js	Docker
S5	Business analytics & data integration system	Greenfield	private cloud, on-premise	7	6	REST, Kafka	Java, Scala	Docker
S6	Automotive configuration management system	Rewrite	private cloud	20	60	REST, Kafka, MQTT, JMS	Java, Node.js	Docker
S7	Retail online shop	COTS Replacement	on-premise, public cloud (AWS, Google)	~200	~250	REST (JSON or OData), SOAP (legacy)	Java, Node.js, Go, Kotlin	JAR/WAR
S8	IT service monitoring platform	Continuous Evolution	private cloud	15	9	REST, AMQP	Java, Node.js	Docker
S9	Hotel search engine	Continuous Evolution	on-premise, public cloud (Google)	~50	~10	gRPC, Kafka, REST, Unix domain socket	Java, PHP	Docker
S10	Hotel management suite	Rewrite & Extension	on-premise, public cloud (AWS)	50	20	REST, Google Pub/Sub	PHP, Java	Docker
S11	Public transport management suite (HR management part: S11a)	Continuous Evolution	on-premise	~175	10 products	REST, SOAP, file transfer, RPC/RMI	Java, C++	JAR/EAR
S12	Retail online shop	COTS Replacement	public cloud (AWS)	~85	~45	REST, Kafka	Java, Kotlin, Scala, Go	Docker
S13	Automotive end-user service mgmt. system	Rewrite & Extension	private cloud	30	7	REST, AMQP	Java	Docker
S14	Retail online shop	COTS Replacement	public cloud (AWS)	~350	~175	REST, Kafka, Amazon SQS	Java, Clojure, Node.js, Ruby, Scala, Swift	Docker, JAR/WAR, ZIP

Figure 16. System Characteristics (Bogner et al., 2019)

The European Organization for Nuclear Research (CERN) is reliant on a messaging technology that is common in microservice implementations. The CERN Health & Safety and

Environmental (HSE) protection develops and operates the Radiation and Environmental Unified Supervision (REMUS) system that provides supervision, control, and data acquisition (SCADA) of CERN accelerators, experiments, and their surrounding environment (Ledeul et al., 2019). At the time of the article, REMUS interfaced with 80 device types, contained 650,000 tags, managed 84,000 alarms, and handled a throughput of 3,700 changes per second. REMUS archives roughly 38 billion measurements per year. To comply with SCADA safety regulations, REMUS needed to meet requirements for security, reliability, scalability, performance, and loose coupling. REMUS opted for Apache Kafka for log processing due to its successful implementations with more than 13,000 companies including Netflix, Uber, Spotify, Cisco, Yahoo, Twitter, Square, and overall, a third of Fortune 500 companies. All outgoing measurements retrieved by the REMUS supervisory system are published in near-real time through Kafka, in a dedicated Kafka topic. REMUS contains 3,300 publication tags (e.g., topics), sending about 600 message per second to CERN Kafka brokers.

(Bruza & Reith, 2018; Bruza, 2018) discusses the integration of Multi-Domain Command and Control (MDC2) capabilities to conduct effective cyberwarfare within a USAF Air Operations Center (AOC). The Air Force Life Cycle Management Center (AFLCMC) and Defense Innovation Unit Experimental (DIUx) initiated an AOC Pathfinder program as a case study for developing software. MDC2 starts with data collection from multiple sources; this includes Intelligence, Surveillance, and Reconnaissance as well as other data sources such as weather forecasts, maintenance schedules, or other information regarding the status of blue forces (Bruza, 2018). MDC2 data must be processed and displayed effectively to enable rapid decision making and produce effects. The key is that this process should be completely domain agnostic. Data from

multiple domains should be collected and processed by experts in those domains, and then presented to decision makers who will select a course of action to produce the effects needed to accomplish their mission. The course of action can include capabilities and assets from multiple domains all working together to produce the needed effects in the battlespace. The MDC2 process is summarized in Figure 17. DevOps solutions to MDC2 requirements are summarized in Table 3.

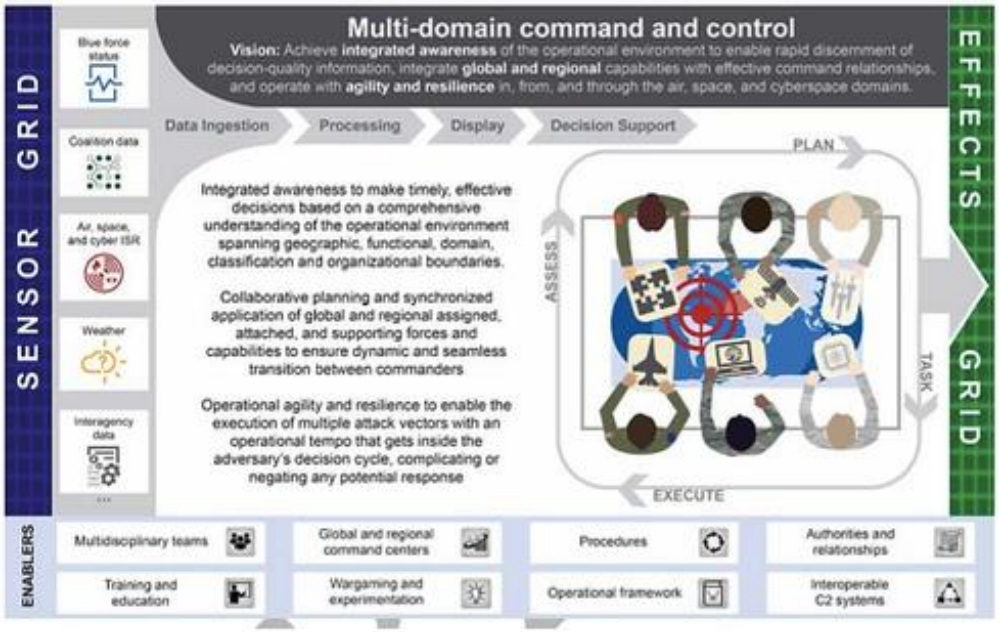

Figure 17. MDC2 Vision "Multi-Domain Command and Control Operating Concept," 2016 from (Bruza, 2018)

Table 3. DevOps Answers to MDC2 Requirements

DevOps Answers to MDC2 Requirements	
MDC2 Application Development Requirement	**DevOps Answer**
Rapid Software Release	Designed for reduced development timelines
Data Sharing	Additional data pipelines can be added as microservices and/or application programming interfaces
User Feedback Informs Development	Development is iterative based on user feedback for the entire lifecycle of the application
Frequent Changes and Updates	Designed to enable frequent development of new code

DevOps Answers to MDC2 Requirements	
Security and Reliability	DevOps toolsets ensure secure software design principles are followed

(Bruza, 2018) deems the AOC Pathfinder integration to be a success and offers the following research findings:

- AOC Pathfinder has extensibility to "increase the scope of their applications in the future and ensure that the proper intelligence is collected during the target development cycle to provide multi-domain options at the end of the cycle rather than developing a target with kinetic effects in mind."
- The AOC Pathfinder experienced success with applying DevOps to develop software in months or weeks rather than years that is extensible to MDC2. Additionally, a microservice architecture provides a means of making MDC2 software more flexible to changing needs. The author cites that "MDC2 is not qualitatively different from a single-domain C2 software (it is qualitatively different because there is [*sic*] more information and people to manage), so software development practices that work for C2 software can work for MDC2."
- Using pairing of government with industry for training new members, integrated security teams, the ATO-in-a-day concept, and the DevOps toolset provided by industry were discussed as main attributes of success. In addition, the AOC Pathfinder project embraced Silicon Valley's entrepreneurial culture to promote creative problem solving and avoid falling back into old patterns.

(Kho Lin et al., 2018) explores the use of Kubernetes technology for an Australian Defense Forces (ADF) ATHENA platform. ATHENA is a strategic simulation and analysis platform focused on manpower planning. ATHENA provides a framework for what-if analysis, e.g., what if we close a flight school, what if the number of instructors does not meet the needs and demands of the ADF in 5-year time, what should the intake of new students be in the next few years. As more organizations within the ADF used ATHENA, a robust horizontally scalable platform was needed. ATHENA leverages container-based technologies for auto-scaling. As the market leader, Docker was used as the core container-based solution and Kubernetes was used as the container orchestration. In Kubernetes, the concept of a Pod is used to encapsulate containers. A Kubernetes Pod object holds one or more containers and introduces an *IP-per-Pod* network model. Therefore, containers within a Pod share their network namespaces included their IP address. In Kubernetes, Pods are ephemeral. That is, a Kubernetes cluster can replicate Pods (destroy and re-create new ones) for dynamically scaling up and down, for self-healing purposes and/or for self-managing purposes. This is challenging for application developers to keep track of.

To experiment with the auto-scaling setup and benchmarking, (Kho Lin et al., 2018) configured the ATHENA Worker Horizontal Pod Autoscaler (HPA), to use 80% target CPU utilization with one CPU resource request for each Worker Pod instance. The HPA replication factor was set to a minimum of one Pod to a maximum of six Pods. The trend lines showed that the rate of increase in runtime decreases as more resources were add, this is a clear indication that auto-scaling was successful. However, during the experimental runs, it was noted that the auto-scaler doesn't react immediately to usage spikes. The authors concluded that an auto-

scaling system cannot meet the user performance demands by simply relying on CPU utlization and memory usage metrics. Most web and mobile applications require auto-scaling based upon Requests per Second to handle bursty traffic and stochastic user load. The authors intented to extend a Kubernetes API to provide more insight to the HPA controller to predict when auto-scaling is required.

Cloud Computing Architecture (Q4)

The foundation of a CMS architecture and its ability to meet hard-real-time requirements is the underlying communications architecture. The following papers address studies related to message broker and pub/sub technologies (see Question 4). (John & Liu, 2017) provides an overview of message broker technologies with a primary focus on Apache Kafka and AMQP. (Dobbelaere & Esmaili, 2017) adds to the Kafka versus RabbitMQ body of knowledge through a qualitative and quantitative assessment of the technologies from the perspective of publish and subscribe architectures. RabbitMQ is primarily known as an efficient and scalable implementation of AMQP.

Despite commonalities, Kafka and AMQP have different histories and design goals (Dobbelaere & Esmaili, 2017). Kafka was built at LinkedIn as its centralized event pipelining platform, replacing a disparate set of point-to-point integration systems (Goodhope et al., 2012). Kafka is designed to handle high throughput (billions of messages). In its design, particular attention has been paid to the efficient handling of multiple consumers of the same stream that read at different speeds (e.g., streaming vs. batch). Figure 18 from (Dobbelaere & Esmaili, 2017) shows a high-level architecture of Kafka. Producers send messages to a Kafka topic that holds a feed of all messages of that topic. Each topic is spread over a cluster of Kafka

brokers, with each broker hosting zero or more partitions of each topic. Each partition is an ordered write-ahead log of messages that are persisted to disk. All topics are available for reading by any number of consumers, and additional consumers have very low overhead. (John & Liu, 2017) describes the fundamental features behind Kafka are performance over reliability and it offers high throughput, low latency message queuing. The loss of a single record among a multitude is not a huge deal-breaker. The rationale behind this is, for log aggregated data, delivery guarantees are unnecessary. Kafka is used at CERN for their Radiation and Environmental Unified Supervision (REMUS) system that manages 600 messages per second to CERN Kafka brokers (Ledeul et al., 2019).

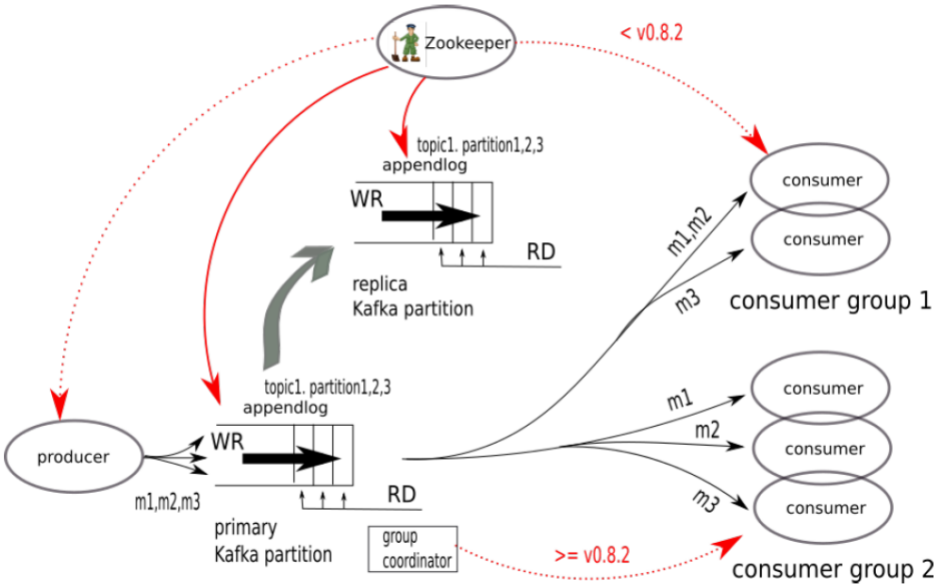

Figure 18. Kafka Architecture (Dobbelaere & Esmaili, 2017)

AMQP is an asynchronous message queuing protocol, aiming to create an open standard for passing messages between applications and systems regardless of internal design (John & Liu, 2017). It was initially designed for financial transaction processing systems, such as trading

and banking systems, which require high guarantees of reliability, scalability, and manageability. AMQP was born out of the need for interoperability of different asynchronous messaging middleware implementations (Dobbelaere & Esmaili, 2017). While various middleware standards existed for synchronous messaging (e.g., SOAP), the same did not hold true in the world of asynchronous messaging. What is now known about AMPQ originated in 2003 at JPMorgan Chase. Figure 19 presents a high-level architecture for RabbitMQ (AMPQ). The design of AMPQ has been driven by stringent performance, scalability, and reliability requirements from the finance community. AMPQ takes on a modular approach, dividing the message brokering task between exchanges and message queues. The implementation can be summarized as follows:

- An exchange is essentially a router that accepts incoming messages from applications and based on a set of rules or criteria, decides which queues to route the messages to.
- A message queue stores messages and sends them to message consumers. Message durability is up to the queue implementation.
- Joining together exchanges and message queues are bindings, which specify the rules and criteria by which exchanges route messages. Specifically, applications create bindings and associate them with message queues, thereby determining the messages that exchanges deliver to each queue.
- Channels can be used to isolate message streams from each other. In a multi-thread environment, individual threads are typically assigned their own channel.

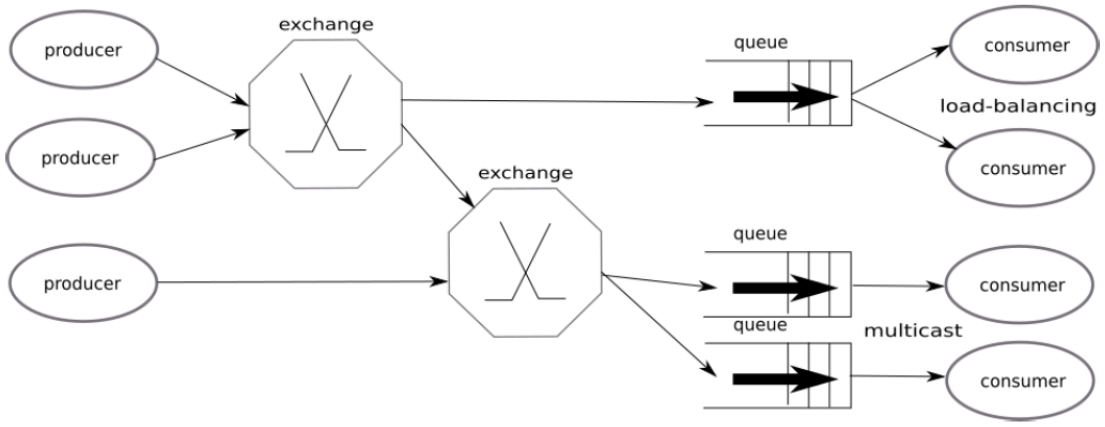

Figure 19. RabbitMQ (AMQP) Architecture (Dobbelaere & Esmaili, 2017)

(John & Liu, 2017) uses the RabbitMQ implementation of AMQP for analysis of broker technologies. A survey of message broker implementations is provided in Appendix D of (John & Liu, 2017). Two types of tests were run for both Kafka and AMQP technologies. The single producer/consumer test keeps the overall total workload constant (1 million messages, 50B each) and scales the queue deployment from 1 to 5 nodes. The multiple producer/consumer setup keeps the number of nodes constant and scales the number of producers/consumers connecting from each node. All benchmarks were run using a modified version of Flotilla, which is a message broker benchmarking tool written in Go. The results reveal that Kafka has a higher throughput while AMQP has a lower latency. AMQP uses a consumer push-model for data distribution; while Kafka uses the pull-model where consumers must fetch messages from brokers. The push-model results in better mean latency in AMQP. The authors conclude that throughput and reliability are key aspects that should be considered when making a choice of a message broker. If reliability for an application is not critical, Kafka is a better choice. If messages are important, such as financial transactions, the cost of losing any

of the messages is far higher than not achieving an optimal throughput, and the application is encouraged to use AMQP. Additionally, AMQP can encrypt messages out-of-the-box.

(Dobbelaere & Esmaili, 2017) uses empirical methods to conduct quantitative analysis to compare efficiency and performance of Kafka and RabbitMQ implementations. For both technologies, the authors used the test tools provided by the respective distributions. Latency measurements for ideal operating conditions are summarized in Figure 20. Since the tools reported different statistical summaries, the table presents a selected subset that is relevant and semantically comparable. The authors draw two conclusions from the results: (i) Both systems can deliver millisecond-level low-latency guarantees. The results for Kafka seem a little better; however, Kafka was tested at an ideal setting (zero cache miss) and in a more realistic setting RabbitMQ outperforms it. (ii) Replication does not drastically hamper the results. More specifically, in the case of RabbitMQ the results are almost identical. For Kafka, it only appears after the median value, with a 100% increase in the 99.9 percentile.

	mean	max
with and without replication	1–4 ms	2–17 ms

(a) RabbitMQ

	50 percentile	99.9 percentile
without replication	1 ms	15 ms
with replication	1 ms	30 ms

(b) Kafka

Figure 20. RabbitMQ vs. Kafka Latency Results (Dobbelaere & Esmaili, 2017)

(Wu et al., 2019) analyzes the reliability for a specific distributed messaging system. The authors cite that Kafka's use cases vary from tracking clicks in a website, network, and infrastructure monitoring, to electronic financial trading and customer service for online reservations. The requirements of Kafka's reliable data delivery differ among those use cases.

An application that collects streams of web page logs to count views per web page can tolerate some inaccurate processing. In this situation quick response from the application is prioritized over reliability. However, for the streams of debit and credit card payments, reliability is the top priority and there is no tolerance for errors in processing. Specifically, every stream of data should be processed exactly once without exception.

(Wu et al., 2019) developed a testbed and tool for Testing the Reliability of Apache Kafka (TRAK) to study the different delivery semantics in Kafka and compare their reliability under poor network quality. Additionally, faults were injected using an open-source network emulation tool called Pumba. The authors varied Kafka delivery semantics by changing Kafka settings: (1) at-most-once, (2) at-least-once, (3) exactly once. Two failure types were injected from TRAK: (1) broker and (2) client. A “network failure” type was injected using Pumba. To evaluate the reliability of data delivery, two metrics were defined, the loss rate and duplicate rate of messages. In tests without any fault injection, the network was observed to be fast (less 0.1 ms delay), without any packet lost, and no messages lost or duplicated. In tests under fault injection, the network delays range from 1ms to 300ms and the pack loss from 1% to 10%.

2.2 LIMITATIONS OF EXISTING STUDIES

According to the literature review and previous studies, none of the previous efforts provide a quantitative assessment of the ability of a microservice based architecture to meet the hard-real-time deterministic demands of a Combat Management System (CMS). Efforts

assess the performance of the components of an CMS architecture; however, none of efforts focus on the assessment of an E2E architecture that is representative of a CMS. Observed limitations are summarized in Table 4. These limitations and associated gaps in knowledge, sets the stage for focused research.

Table 4. Limitations of Existing Studies

Question	Limitations
Q1: Microservices Performance	Lacks node architecture that is representative of CMS components. Variance in container start up times is tolerated without assessment of design alternatives.
Q2: E2E Analysis	Environment used to generate architecture products and conduct analysis is not representative of CMS architecture and challenges of CMS interface management (e.g. APIs).
Q3: C2 Implementation	MDC2 example is similar to CMS challenges; however, the results are qualitative not quantitative. Focus is on change in DevOps culture.
Q4: Architecture	Provides detailed assessment of Kafka vs. RabbitMQ; however, a mix may be required within a CMS to meet unique CMS domain requirements (e.g. external interface to weapons/sensors vs internal processing). Analysis is required to facilitate experiment design.

3 METHODOLOGY

The purpose of this chapter is to describe the methodological approach, conceptual analysis framework, and approach for analysis.

3.1 Methodological Approach

This research seeks to use an Empiricist, Positivist, Deductive paradigm as defined by (Siangchokyoo & Sousa-Poza, 2012).

- Epistemological Position: Empiricist – Justification of Knowledge through observation
- Ontological Position: Positivist – Seek to find reality independent of the observer
- Mode of Reasoning: Deductive – Usage of confirmatory reasoning to obtain knowledge

A mission-based approach will be used to set the context for applied research. A hypothetical yet operationally relevant Strait Transit scenario has been established to provide context for definition of experimental parameters to be set while assessing the hypothesis. System models and data from a cloud computing environment will be used to collect data for quantitative analysis. To achieve this goal, this research will leverage an existing CMS Prototype built within a DevSecOps cloud computing infrastructure (e.g., Amazon Web Services) to simulate an operational scenario and generate operationally relevant CMS data under operational mission conditions for analysis. A CMS Prototype exists; however, statistical analysis has yet to occur. The use of a DevSecOps environment supports the positivist

ontological position through the use of automation. A primary precept of DevSecOps is to automate everything which creates an environment of repeatability.

This research will focus on a core hard real-time thread within the software core of the combat system (e.g., representative response to a weapons options request within the CMS) that will be built from commercial products and not all elements of a full combat system. Variables are summarized in the following paragraphs.

3.2 Conceptual Analysis Framework

Figure 21 presents a conceptual analysis framework for the study variables, hypothesis, and associated dependent variables (based upon discussions above). Positive predicted independent variable effects on dependent variables are indicated with a positive (+) hypothesis (e.g., +H1). Negative predicted independent variable effects on dependent are indicated with a negative (-) hypothesis (e.g. -H2). Framework details are discussed below.

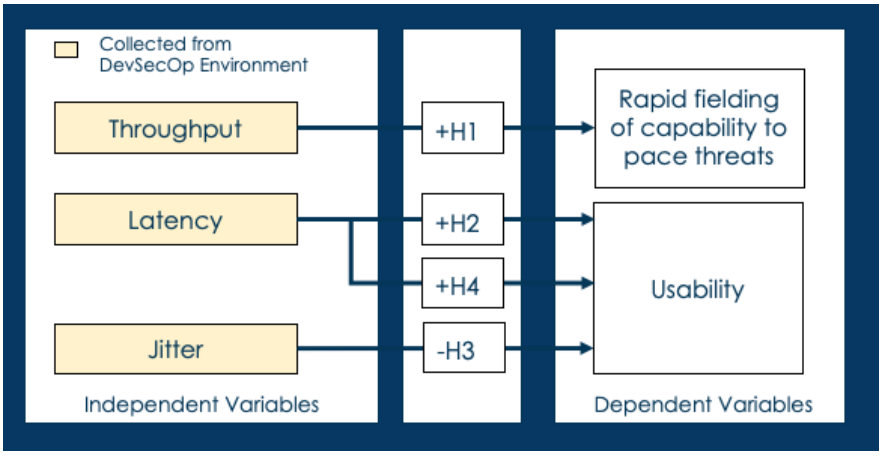

Figure 21. Conceptual Analysis Framework

<u>Independent Variables</u>

Latency, throughput, and jitter will be the primary independent variable used to assess system usability and ability to rapidly field capability through having a scalable system (i.e., dependent variables). The following definitions are provided from (Moreland Jr., 2013):

- Latency - This factor refers to the end-to-end processing time between the sending of information from one application to the receipt of that information by other applications. Processing latency directly impacts the reaction time of a system when trying to provide a timely response to an operational incident.
- Throughput - Throughput capacity is derived as the total number of messages processed for a given period of time. Throughput can be used as a performance index to evaluate a web service provider. The determination of the number of providers being serviced also defines how many users can be processed concurrently in a networked environment.
- Jitter - This factor measures the variability in latency measurements between successive messages. The objective is to produce a computing architecture with small latencies and a low, insignificant jitter. This result is advantageous for the quick delivery of messages with very limited to no data outliers. In the case of hard real-time systems, jitter is a crucial performance parameter due to the necessity of providing reliable performance under tremendous computing speeds requirements. Further detail for “deterministic control systems“ is provided in (Roa et al., 2011).

Measuring determinism means the capability to accurately characterize the worst-case time to exchange information end to end, no matter what other network traffic is occurring. The

“throughput”, “latency time” and “jitter time” of this response are the quantified measures of determinism. (Roa et al., 2011) defines these terms as follows:

- Throughput - In communication networks, such as Ethernet or packet radio, throughput or network throughput is the average rate of successful message delivery over a communication channel. This data may be delivered over a physical or logical link or pass through a certain network node. The throughput is usually measured in bits per second (bit/s or bps), and sometimes in data packets per second or data packets per time slot.
- Latency - a measure of the time delay experienced by a system. Latency in a packet-switched network is measured either one-way (the time from the source sending a packet to the destination receiving it), or roundtrip (the one-way latency from source to destination plus the one-way latency from the destination back to the source). Round-trip latency is more often quoted because it can be measured from a single point.
- Jitter - In the context of computer networks, the term jitter is often used as a measure of the variability over time of the packet latency across a network. A network with constant latency has no variation (or jitter). Packet jitter is expressed as an average of the deviation from the network mean latency. However, for this use, the term is imprecise. The standards-based term is packet delay variation (PDV). PDV is an important quality of service factor in assessment of network performance.

These definitions will be combined to provide definitions that are inclusive of mission capability, technical implementation, and resultant hard-real-time behavior.

Dependent Variables

Usability – (Krug, 2014) provides a methodology for assessment of usability. The author breaks down usability into the following attributes:

- Useful: Does it do something people need done?
- Learnable: Can people figure out how to use it?
- Memorable: Do they have to relearn it each time they use it?
- Effective: Does it get the job done?
- Efficient: Does it do it with a reasonable amount of time and effort?
- Desirable: Do people want it?
- Delightful: It using it enjoyable, or even fun?

All of these attributes can provide an assessment of an microservice implementation. For example, if microservice processing is latent or unpredictable (e.g. jitter), the resultant microservice design fails to be useful, desirable, efficient, and delightful. Useful, learnable, memorable, and desirable are related to the user interface (e.g. UI) design, but also related to the allocation of microservice functionality.

Ability to Rapidly Field Capability – The principles of the Agile Manifesto summarize the attributes required to rapidly field capability (Beck et al., 2001):

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Businesspeople and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development.
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

A microservice implementation designed around business (e.g., mission) capability enables achievement of these goals to achieve rapid fielding of capabilities. The allocation of functionally to microservices and associated throughput through interface definition will impact achievement of these goals.

Predictions and Hypothesis

Hypothesis statements were derived from predicted results based upon the research questions identified in Chapter 1 and literature review in Chapter 2.

Hypothesis: Modern DevSecOps architectures can be designed to meet hard-real-time latency (μ) requirements using modern computing environments and computing infrastructure:

H_0: $\mu \leq$ tbd ms with jitter within latency bounds

H_a: $\mu >$ tbd ms with jitter exceeding latency bounds

$\propto = 0.05$

Experiments will include single-node and multi-node configurations per the referenced literature (see Chapter 2).

Additional hypothesis statements were derived from predicted results based upon the previously identified research questions to put the null and alternate hypothesis results within a SoS and mission context. The independent variables to be measured are throughput, latency, and jitter while the associated dependent variables are rapid fielding of capability and usability. The Research Framework relationships to be used for analysis of the hypotheses are depicted in Figure 21. A similar format was used in (Moreland Jr., 2013). The independent variables are throughput, latency, and jitter that are measurable as the tactical environment varies to assess the dependent variables of “Rapid fielding of capability to pace threats” and “Usability.” Associated hypotheses are depicted in the center column.

From Q1 it is predicted (P1) that AI microservices architecture will have a positive impact on time to implement capability upgrades and development cost. Therefore, Hypothesis 1 (+H1) is captured accordingly: The scalability of microservices as new data sources added to

an architecture enables maintaining high **throughput** and predicted to have a positive impact on **rapid fielding of capability**.

From Q2 it is predicted (P2) that DevSecOps container orchestration technologies (e.g., Kubernetes) will have a positive impact on combat system availability and latency due to the ability to tune deployment configurations and load balance. The resulting Hypothesis 2 (+H2) reads: Microservices orchestration through DevSecOps technologies enables maintaining low **latency** service call responses and is predicted to have a positive impact on **usability**.

From Q3 it is predicted (P3) that web-based user interface technologies will have a negative impact on deterministic hard-real-time performance needed for positive control of organic weapons. This prediction results in the following Hypothesis 3 (-H3): Web-based interfaces (e.g., RESTful HTTP) will increase **jitter** which is predicted to have a negative impact on **usability** and the deterministic performance needed for positive control of organic weapons.

From Q4 it is predicted (P3) that a systems architecture model will have a positive impact on micro- services architecture performance prediction and prediction of associated mission thread impacts. This leads to the final hypothesis or Hypothesis 4 (+H4): A system architecture model can predict end-to-end **latency** within a mission thread to quantify SoS **usability**.

Figure 22 summarizes question, prediction, and hypothesis relationships through the use of a Mind Map.

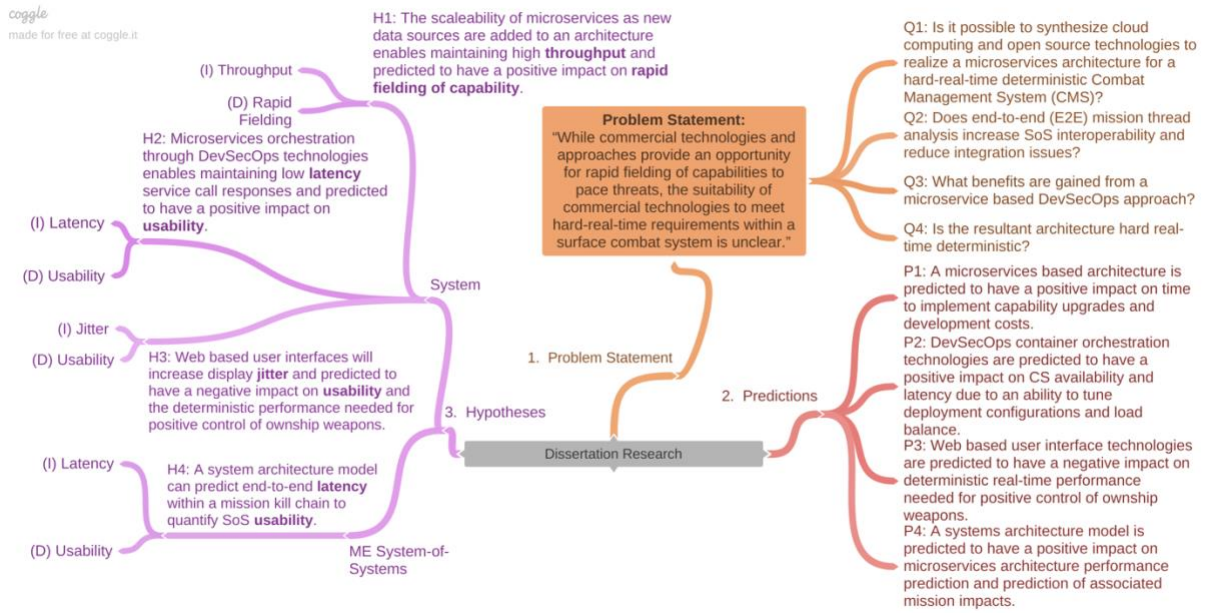

Figure 22. Research Mind Map

3.3 Mission Context

The following scenario has been defined to add context to the experiment design. The scenario is based upon the end-to-end mission thread presented in Section 1.2 (see Figure 7).

Political hostility between Country Red and Country Green has escalated over the past 6 months to a point where military conflict is imminent. The Country Green Navy Cruiser (CG), USS Dahlgren, has been tasked with a mission to escort a Country Green Command Ship (LCC) through a Strait between the two countries to establish a command post in the event of wartime activity. USS Dahlgren will lead a Surface Action Group (SAG) composed of a destroyer (DD) and an unmanned surface vehicle (USV). Additionally, USS Dahlgren is equipped with an unmanned aerial vehicle (UAV). During mission planning, USS Dahlgren acquires intelligence data that indicates that Country Red has made modifications to their surface combatant weapon systems that will impact the USS Dahlgren abilities to engage and develop fire control

solutions against Country Red's surface-to-surface missiles (SSM). USS Dahlgren contacts its Country Green shore support activity to identify and develop software upgrades within 72 hours to react to the emergent threat. Country Green develops potential solutions within their DevSecOps Software Factory to provide a rapid response. After User Centered Design (UCD), system analysis through simulation, CI/CD system integration testing, and DevSecOps automation enabled certification, Country Green can deliver an effective and suitable solution from their Software Factory prior to Country Green SAG deployment. The USS Dahlgren Task Group enters the Strait. Once the SAG enters the Strait, National Technical Means (NTM) detects a Country Red surface threat near the end of the Strait. A non-organic UAV in close proximity to the surface threat is tasked to provide additional targeting information (e.g. Triton). Joint assets may also be tasked to provide support. A targeting solution is provided to the CMS and over-the-horizon (OTH) weapons are employed. Figure 23 depicts the operational concepts in an operational view (OV-1) with the corresponding mission thread using the Find-Fix-Track-Target-Engage-Assess (F2T2EA) mission essential tasks taxonomy. For our research we shall focus on the deterministic responsiveness of the CMS/AI Services provided to support operator decisions.

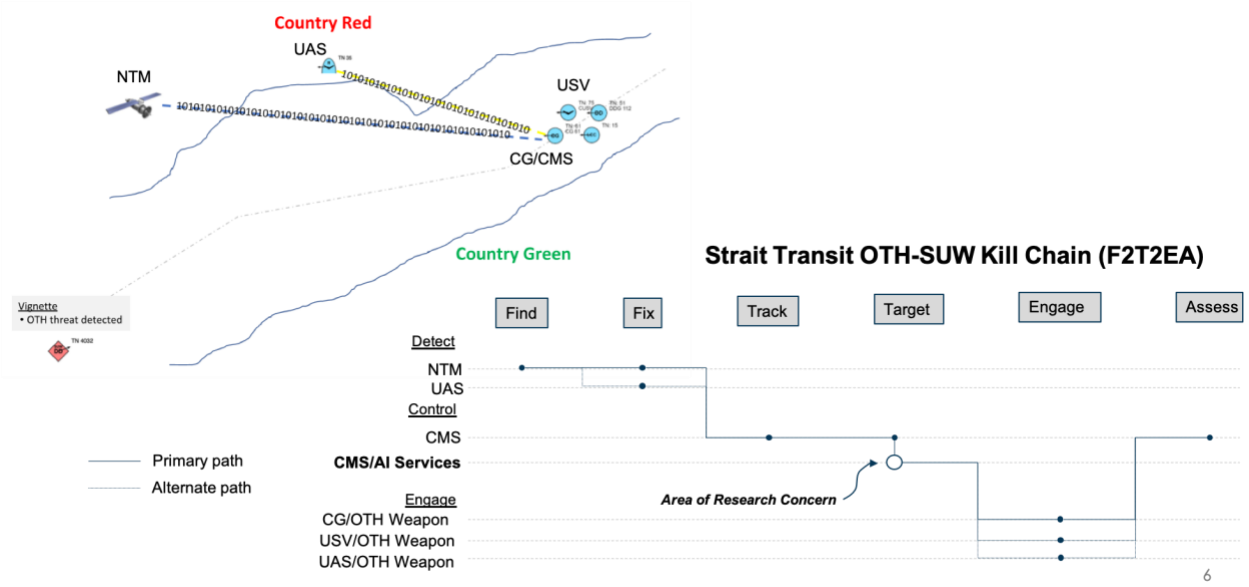

Figure 23. Mission Scenario OV-1 and Kill Chain

3.4 Experiment Design

Ishikawa's Fishbone Diagrams provide a practical way of analyzing the causes of a particular effect before identifying a solution. Fishbone Diagrams are also referred to as cause-and-effect diagrams. (Wong et al., 2016) use Fishbone Diagrams to analysis social and behavior issues. (Coccia, 2017) uses Fishbones to evaluate technologies. A Fishbone Diagram depicted in Figure 24 was used to analyze and prioritize potential causes of degraded hard-real-time performance within a microservice based architecture. The analysis is focused on the implementation of artificial intelligence (AI) services to provide decision aids to system operators. Three primary areas of interest are identified: computing infrastructure, AI processing complexity, and software architecture.

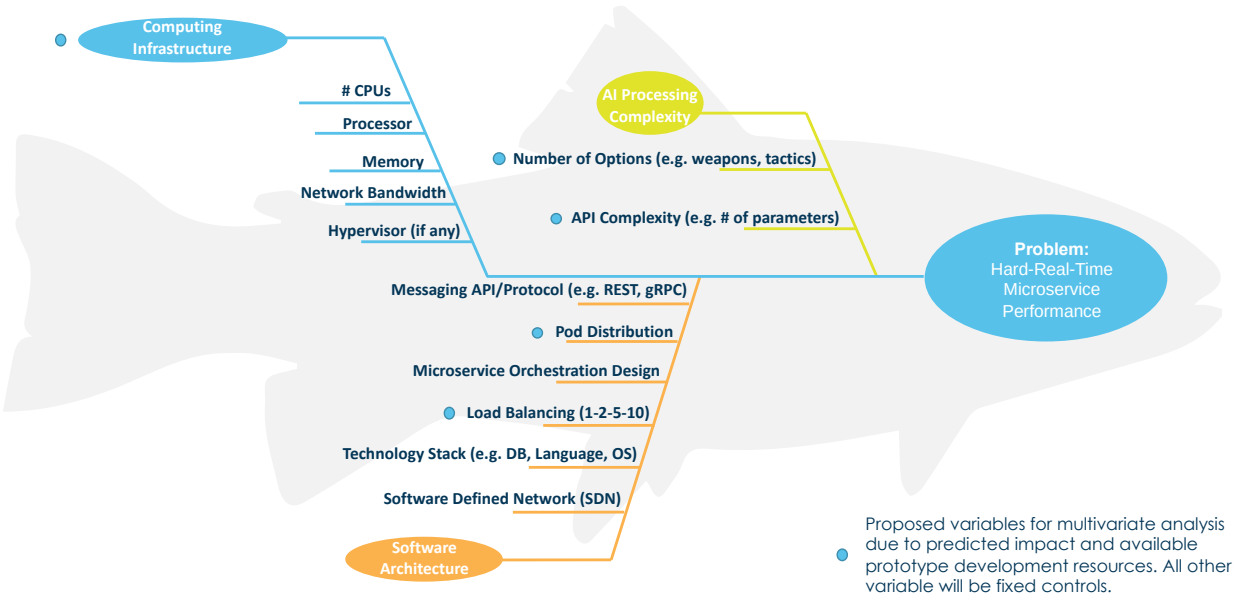

Figure 24. Microservice Performance Cause and Effect Diagram (Ishikawa)

Computing infrastructure is concerned with selection of the computing hardware to implement the microservices. Variations in processor performance, number of processors, memory, network, and virtualization approach (e.g., hypervisor) may have significant effects on hard-real-time microservice performance. These factors will vary by selection of the host environment but will be fixed within a host environment for data collection. For example, experiments may be conducted within a home computing environment and later replicated on cloud hardware (e.g., Amazon Web Services Infrastructure as a Services (IaaS)).

Software architecture is concerned with microservice implementation design and coordination among microservices to provide capability. The technology stack choices, implementation patterns, and component communication approach will have varying degrees of impact on performance.

AI processing complexity is concerned with microservice design. While software architecture was concerned with technology choices, this factor is based up software

implementation choices. Software metrics such as McCabe's Cyclomatic Complexity can be used as a methodology for measurement (McCabe, 1976).

This research will provide software design patterns that can be reused by implementors of future safety-critical system capabilities, e.g., Surface Navy combat systems. Additionally, it is envisioned that this research will inform requirements for future CMS software applications.

3.5 Software Implementation

A notional experimental software prototype context is depicted in Figure 25. The prototype system is envisioned to be a Decision Support System (DSS) that my provide hard-real-time decision support to system operators. The DSS provides artificial intelligence services to assist the user in decision making. The architecture is presented using the context, container, component, and code (C4) model style from (Brown, 2019).

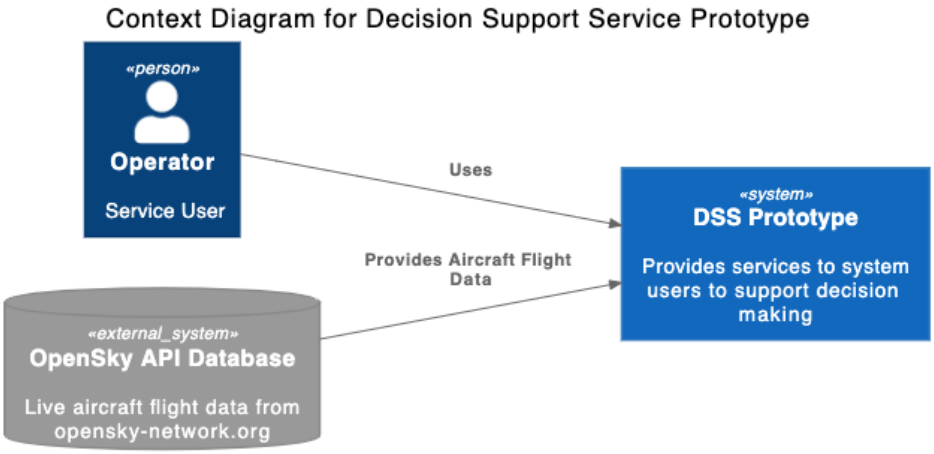

Figure 25. Experimental Prototype Software Context

Use case were developed to identify services required to realize the DSS Prototype. The use cases are depicted in Figure 26. The primary operator for our system is a Tactical Actions

Officer (TAO) that wants to use the DSS to enable informed weapons decision. The primary use cases allow the system operator to observe information about aircraft in the area through the "Review Tactical Information" use case, assess what weapons are capable against threat aircraft through the "Review Weapon Recommendations" use case, and assess the effectiveness of a single weapon against a target through the "Review Predicted Weapon Effectiveness" use case.

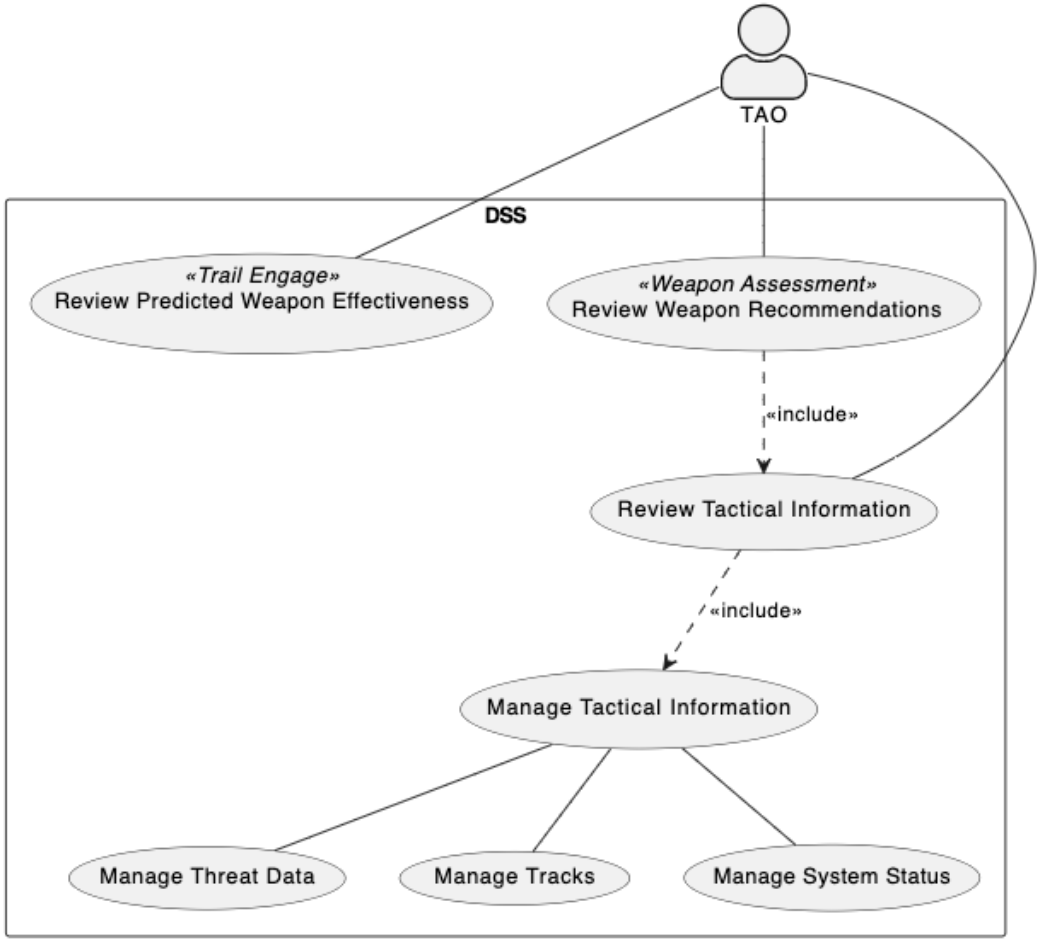

Figure 26. DSS Prototype Use Cases (OMG, 2015)

SOLID Principles such as the Single Responsibility principle discussed in Chapter 1 of this dissertation were used to define the componentization of microservices used to realize the DSS Prototype. A sequence diagram depicted in Figure 27 was used to explore interaction

alternatives based upon the use cases. Each interaction was designed to have varying levels of dependencies among the microservice based software configuration items (SCIs). The “Trial Engage” use case was designed to be the least complex with a single dependency between the User Interface and the Trial Engage application. However, the “Review Tactical Information” use case was designed to be the most complex with a dependency on an external flight data services provided by an OpenSky API (OpenSky, 2021).

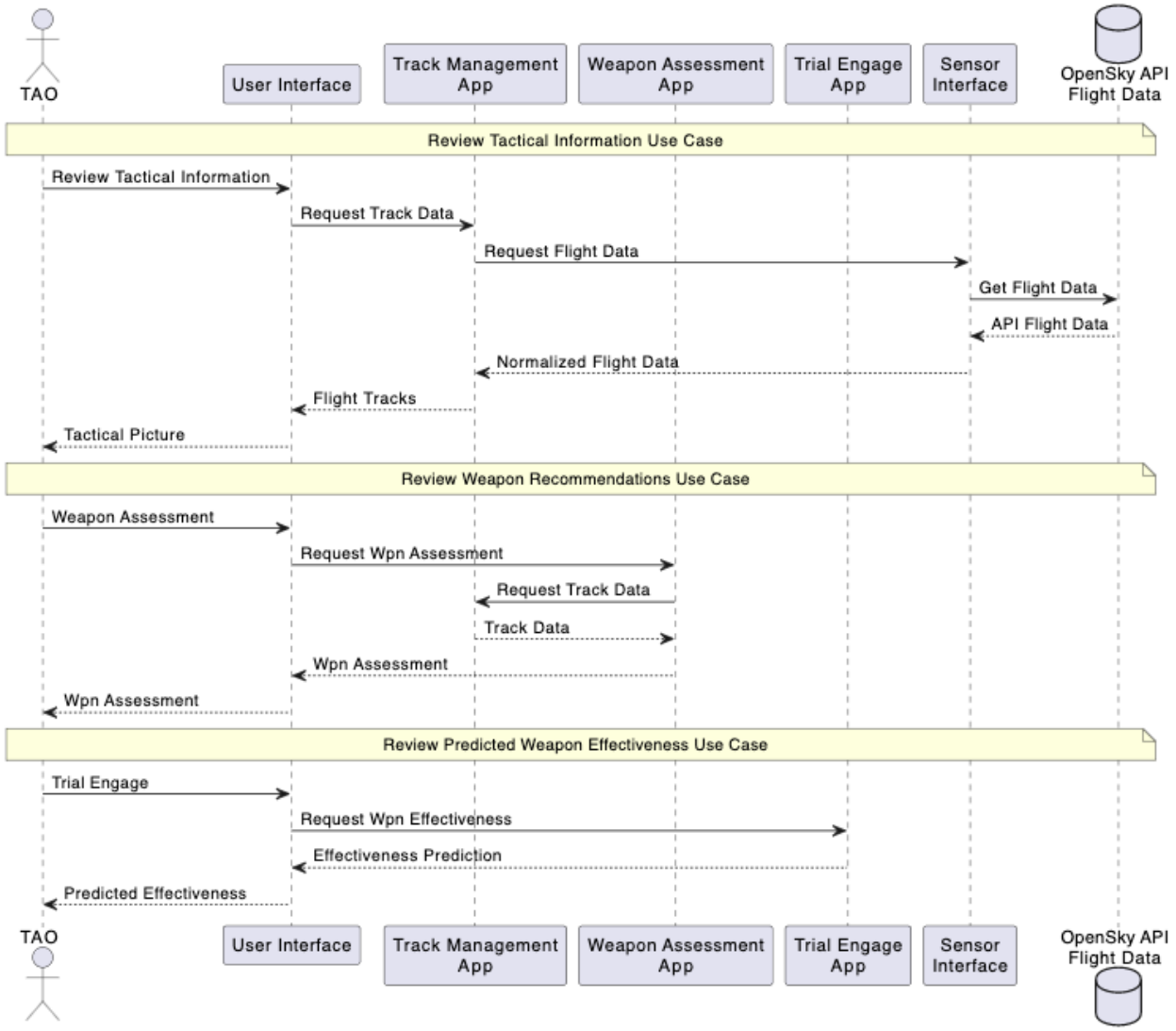

Figure 27. DSS Service Interactions (UML Sequence Diagram) (OMG, 2015)

It should be noted that the Track Management application was inserted create a layer of abstraction between flight data requests through the Sensor Interface application to the OpenSky API. Access to the OpenSky API for flight data adds unpredictable latency through the use of public networks to access the endpoint. However, the Track Management application can provide immediate deterministic responses to local requests for last known flight data without relying on reaching out to external non-deterministic networks.

The DSS is broken down into the software configuration items depicted in Figure 28 that satisfy an end use function that can be uniquely identified (ISO/IEC/IEEE, 2008). The component allocation is driven by the DSS Prototype use cases and needs for data collection and analysis. The primary services provided are Weapon Assessment and Trial Engage. The Weapon Assessment Service reviews target track kinematics and organic weapon systems capabilities to provide and assess weapon/target pairing for engagement of the specific target. The assessment is based upon known weapons' capabilities against the position of the target and kinematic capabilities. The Trial Engage Service assesses the target against weapon capabilities and known tactics to use against a target. Data for display of engagement profiles is provided, e.g., earliest time to launch, latest time to launch, flyout pattern. Both services require hard-real-time responses to enable operator weapon selection within time for weapons release and consummation of the engagement. If recommendations are latent, destruction of the targeted platform is imminent. The component diagram breaks the primary services down into microservices that are needed to provide DSS functionality, e.g., a sensor interface, track management. Monolithic analysis services are included as pre-packaged containers from the Docker Container Repository.

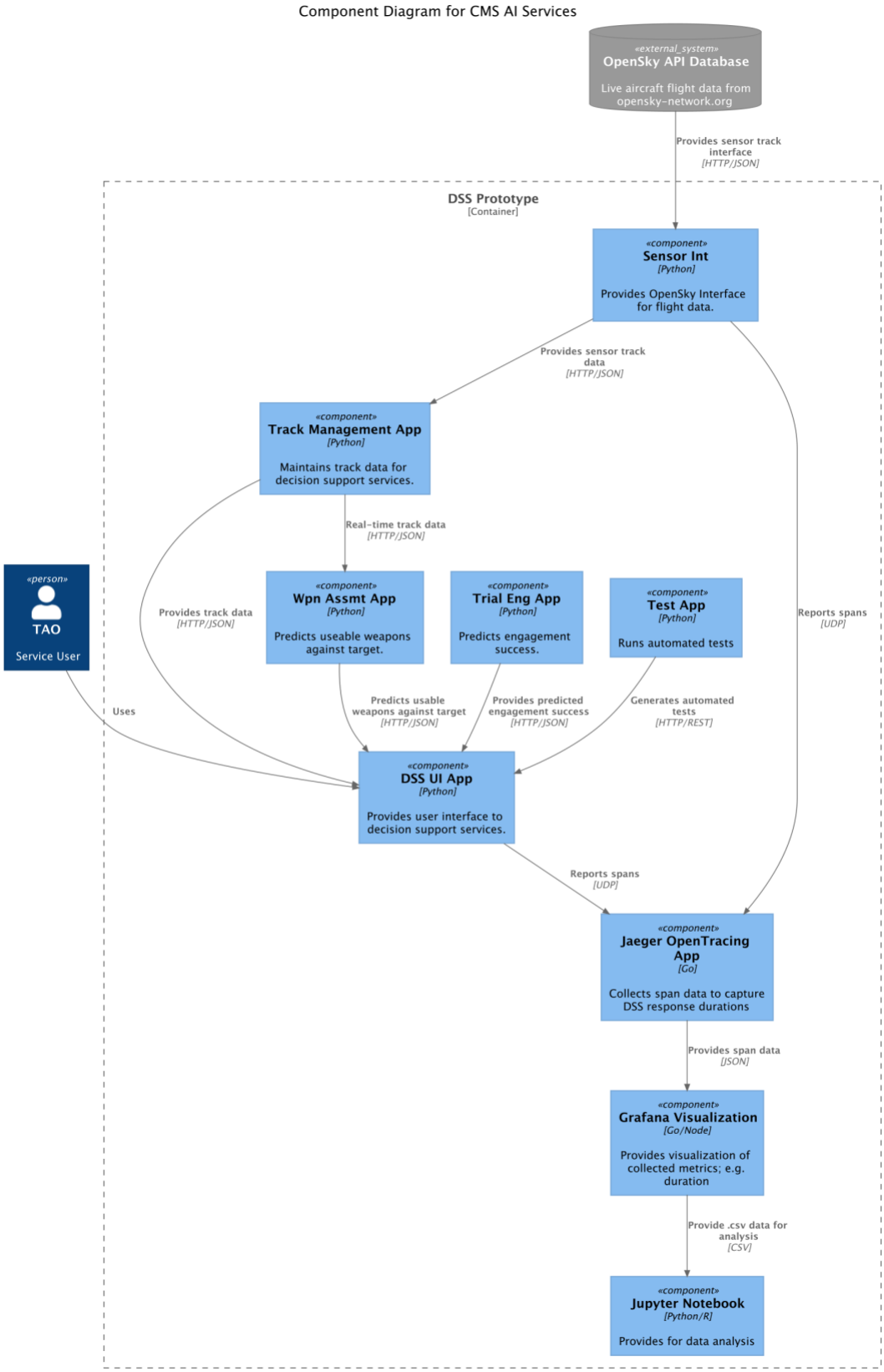

Figure 28. DSS Component Diagram

The physical deployment to realize the components is depicted in Figure 29. Physical container descriptions are provided below.

DSS Applications (Microservices)

- opensky-int: Provides the OpenSky API for flight data. The app provides data about aircraft within 60 NM of Richmond (RIC) or Dulles (IAD) airports.
- tm-server: Provides sensor track data (e.g. OpenSky) and system tracks to support DSS services. System tracks represent the system-wide common understanding of track object states used for decision support.
- wa-app: The Weapon Assessment Application determines which weapons are capable to successfully engage a target. The wa-app uses the tm-server api to get track data.
- te-app: The Trail Engage Application predicts the success probability of an engagement with a specific weapon target pairing. The predicted track kinematic data at engagement time is provided; therefore, the current track kinematics from the tm-server are not queried prior to providing a response.
- test-app: Provides an ability to initiate automated tests. the test-app uses the dss-ui to call dss-ui endpoint to replicate operator interactions with the DSS Prototype.
- dss-ui: Provides a simple graphical interface to launch DSS services.

Tools (Service Applications)

- telem-jaeger: The open source Jaeger container collects "span" data from the DSS applications. Spans collect duration data for service calls amongst containers; e.g. latency. This the fundamental data that is being analyzed here.

- grafana: The open source Grafana container connects to the telem-jaeger container to create visualization dashboards. Also, Grafana facilitates the export of data as a .csv file for analysis.
- notebook: The Jupyter Notebook container supports analysis of the data recorded by Jaeger and exported by Grafana. An embedded R software library is used for analysis.

In our experiment we will apply various technologies to assess the hard-real-time deterministic nature of the architecture. Specifically, we'll look at Docker containers as well as container orchestration using Kubernetes. An excursion using ODU Coastal Virginia (COVA) Commonwealth Cyber Initiative (CCI) resources is planned to gain access to high end process with Kubernetes orchestration and an Istio service mesh is planned. The Istio service mesh adds a layer of security (e.g., trustworthiness), but may impact deterministic performance. The Istio data plane within the mesh is used to define what services can talk to each other via the proxies that reside within the container pods. All traffic within the mesh is controlled and protected by the Istio technology. (Li et al., 2019) discusses service mesh challenges in detail as well as future research opportunities.

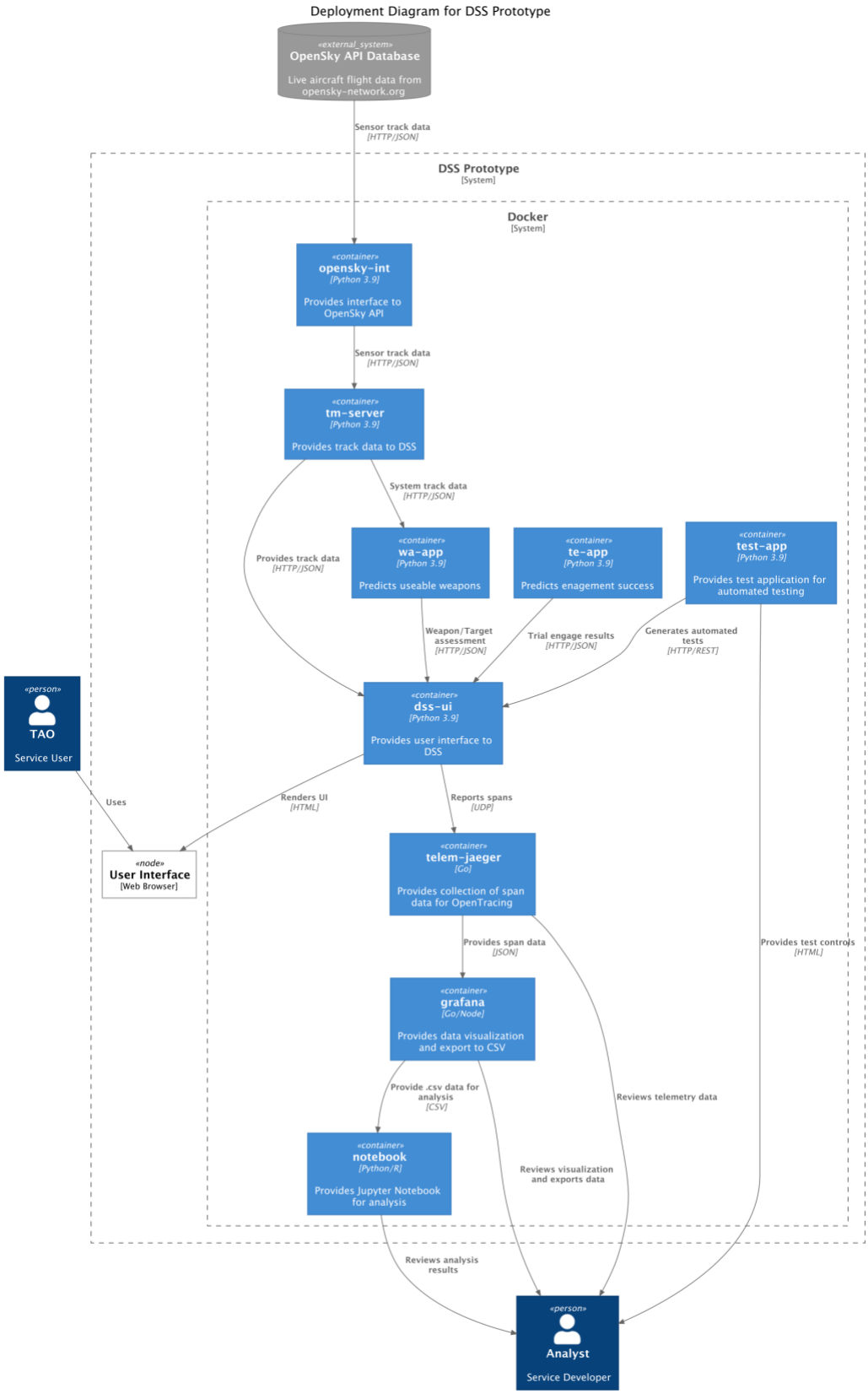

Figure 29. DSS Deployment Diagram

3.6 COMPUTING INFRASTRUCTURE

Figure 30 from (Humble & Farley, 2011) shows a diagram of the use of an artifact repository in a typical installation. It is a key resource which stores the binaries, reports, and metadata for each of your release candidates.

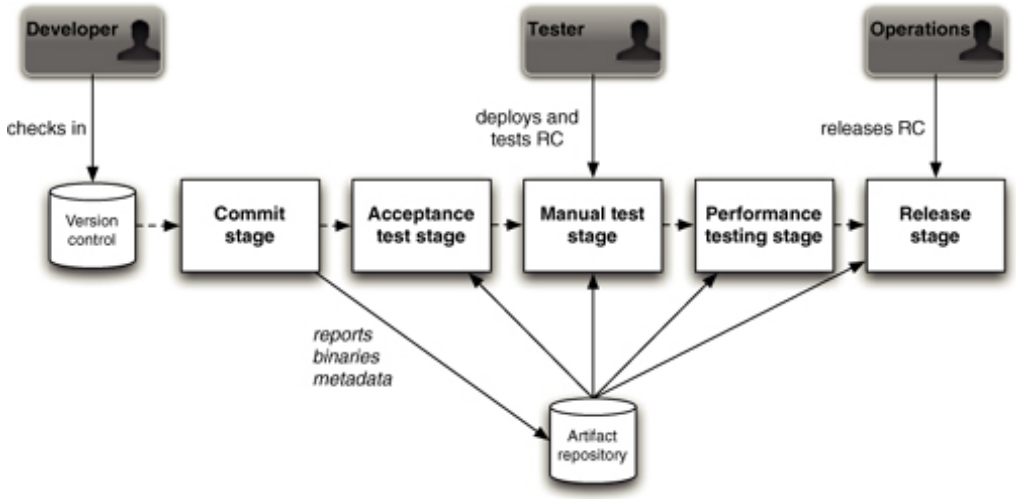

Figure 30. The Role of the Artifact Repository (Humble & Farley, 2011)

An artifact repository on GitHub is core to the development and deployment of the DSS Prototype. Figure 31 depicts the environment used for DSS Prototype development, integration, and test. Technical specification for each of the compute environments are provided below. Software is developed and initially tested on a MacBook Air laptop. Code is periodically committed to the GitHub repository. Once the build meets requirements, the code is cloned from GitHub onto the PC and Raspberry Pi 4 Linux environments for integration testing. Additionally, code is cloned onto an Amazon EC2 instance to demonstrate compatibility in a “cloud based” environment external to the “home lab.” The ODU Commonwealth Cyber Initiative (CCI) Research Environment provides an opportunity to test the DDS Prototype in an

environment that replicates a target real-time environment being used for commercial and DoD applications. DSS Prototype compute resources are summarized in Table 5.

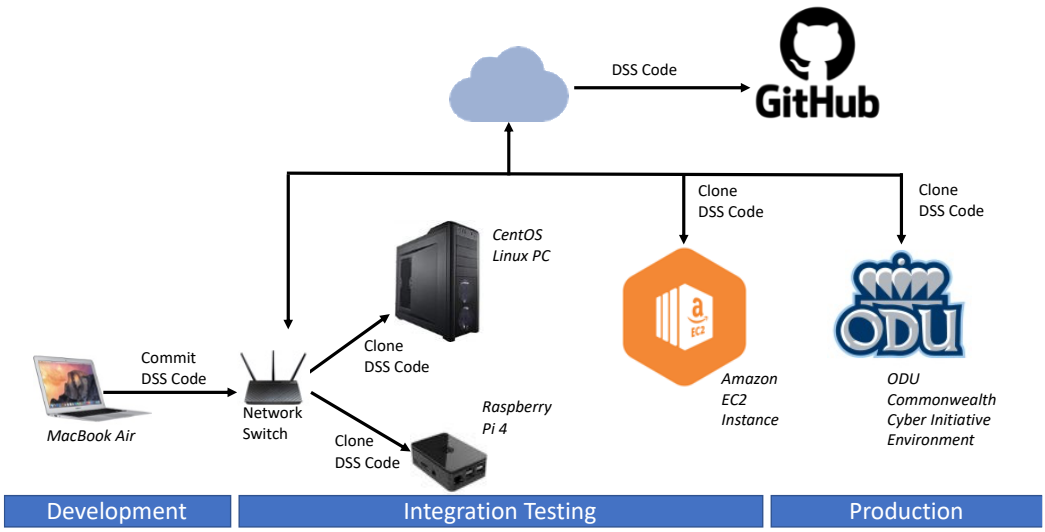

Figure 31. DDS Prototype Development, Integration, and Test Environment

Table 5. DSS Prototype Compute Resources

Env ID	Platform	Chipset	Processor	Memory	OS
0	MacBook Air (2017)	Intel Core i5	Dual-Core Intel Core 5 @ 1.8 GHz	8 GB 1600 MHz DDR3	MacOS 12.4 (Monterey)
1	Linux PC (2012)	Intel Core i7	Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz	16 GB 1600 MHz DDR3	CentOS Linux 8 (Core)
2	Raspberry Pi 4 (2020)	Broadcom BCM 2711	Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5 GHz	4 GB LDDR4-3200 SDRAM	Debian GNU/Linux 11 (bullseye)
3	Amazon Elastic Compute Cloud (EC2): t2.micro	Intel Xeon	Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz	1 GB	Debian GNU/Linux 10 (buster)
4	ODU Commonwealth Cyber Initiative (CCI)	Intel Xeon	Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz	128 GB	Red Hat Enterprise Linux 8.5 (Ootpa)

Home Lab

The “Home Lab” is a network of clustered computers in the researcher’s home. The lab was used for software development and build testing before deployment. The “Home Lab” includes an assortment of computer ranging from 2012 to 2020. The Raspberry Pi configuration features a Broadcom 2711 system on chip (SoC) (*BCM2711 ARM Peripherals*, 2022).

Amazon Web Services

The Amazon Web Service (AWS) Free Tier was leveraged to provide an ability to test beyond the “home lab.” The AWS Free Tier provides access to an Amazon Elastic Cloud Compute (EC2) t2.micro instance. The t2.micro on has 1 virtual CPU, but T2 instances are a low-cost, general purpose instance type that provides a baseline level of CPU performance with the ability to burst above the baseline when needed. T2 instances are one of the lowest-cost Amazon EC2 instance options and are ideal for a variety of general-purpose applications like micro-services, low-latency interactive applications, small and medium databases, virtual desktops, development, build and stage environments, code repositories, and product prototypes (AWS, 2022).

Debian Linux 10 was selected for the Amazon Machine Image (AMI). The Linux “cat /proc/cpuinfo” command was used to obtain relevant CPU details depicted in Figure 32.

```
admin@ip-172-31-93-240:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
…
cpu MHz         : 2400.001
…
cpu cores       : 1
```

```
…
bogomips        : 4800.00
…
```

Figure 32. Amazon Machine Image (AMI) CPU Details

ODU COVA CCI

The CCI environment provided for access to the Rancher version of Kubernetes that was designed for Government applications. Additionally, addition of an Istio Service Mesh is planned for the environment. The specification of the CCI environment is detailed below (Tucker, 2022). A diagram of the CCI research environment at ODU is depicted in Figure 33 (Pratt, 2022).

Compute Nodes:

- 20 x Dell PowerEdge R630
- E5-2683v4/32C/128GB RAM
- Total Cores: 640
- Total RAM: 2.5TB

Storage Nodes:

- 8 x Dell PowerEdge C6420
- 2x Xeon 6230/40C/48GB RAM/6 x 960GB SSD
- Total Storage: 46 TB each usable

GPU Nodes:

- x Dell PowerEdge C4140

- 2x Xeon 6230/20C/192GB RAM/4x NVIDIA V100
- Total GPU: 16 x NVIDIA V100 32GBVRAM GPUs with NVLink

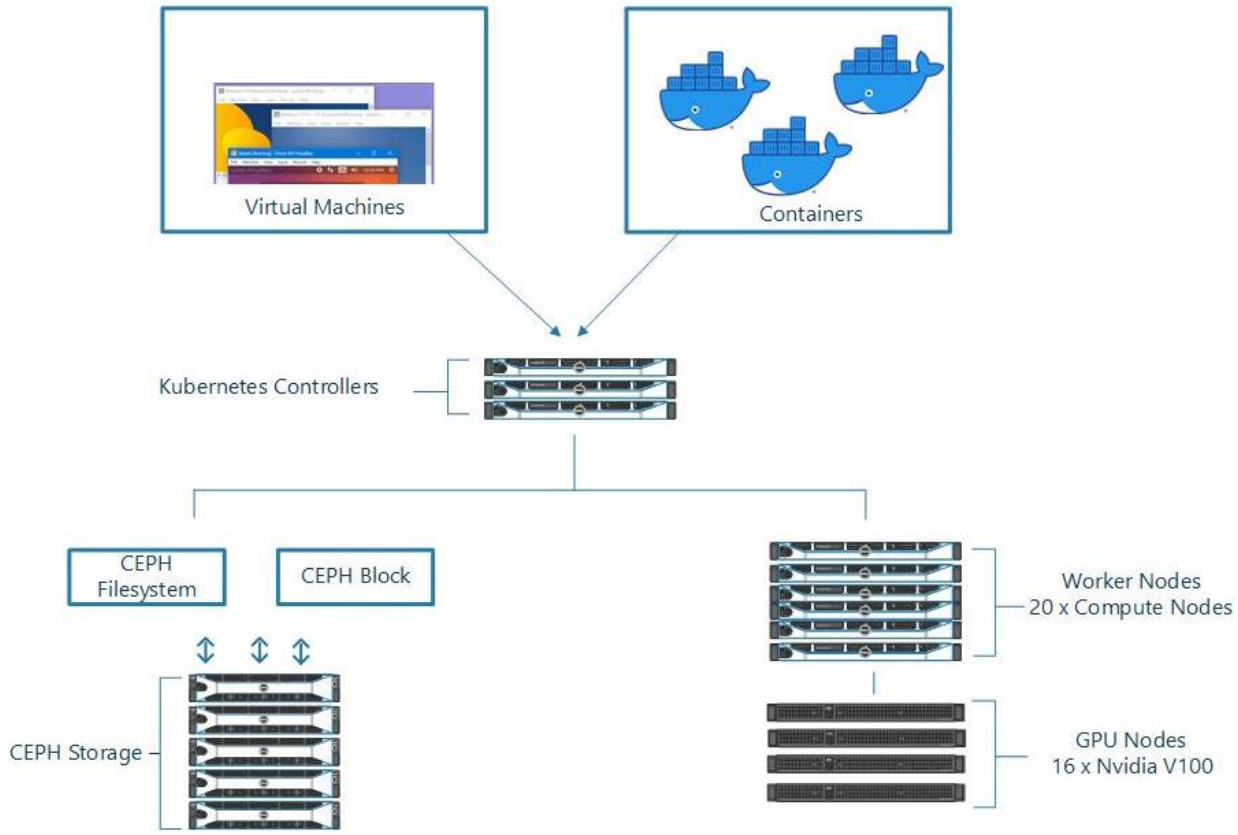

Figure 33. CCI Research Environment (Pratt, 2022)

3.7 Telemetry Collection

Telemetry collection using the OpenTelemetry standard is the primary means for collection of latency metrics from the Docker containers.

OpenTelemery

OpenTelemetry is a collection of tools, APIs, and SDKs. Use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help you analyze your

software's performance and behavior (OpenTelemetry, 2022). A cloud-based tool called Jaeger was used to collect telemetry data and export for analysis. A sample visualization of telemetry data from the ODU CCI environment is depicted in Figure 34.

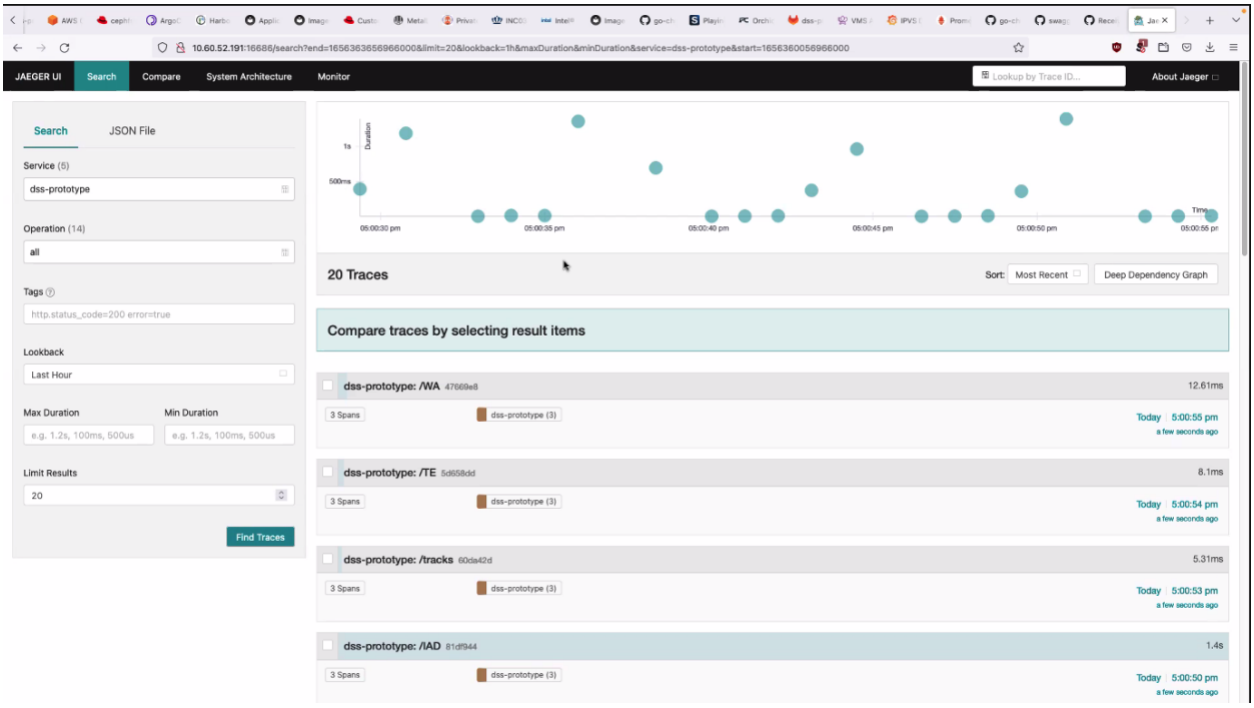

Figure 34. Jaeger Visualization

I/O Metrics

The DSS Test Application was later modified to use a Python psutil library to log input/output metrics (Rodola, 2022). The following metrics were collected and reported as OpenTelemetry attributes:

- cpu.load.avg
- num.cpu
- net.io.count

- start.io.count
- end.io.count

The io.count metric is a tuple including the following attributes:

- # bytes_sent: number of bytes sent
- # bytes_recv: number of bytes received
- # packets_sent: number of packets sent
- # packets_recv: number of packets received
- # errin: total number of errors while receiving
- # errout: total number of errors while sending
- # dropin: total number of incoming packets which were dropped
- # dropout: total number of outgoing packets which were dropped

Figure 35 was generated from the ODU CCI production environment. The I/O metrics are collected as “tags” as part of the OpenTelemetry standard. It should be noted that with every run the error and packet loss metrics were zero (e.g., errin, errout, dropin, dropout). This is due to the nature of the Docker environment where a single computer is internally running multiple containers without relying on external networks. It should also be noted for the external API calls, the error and packet loss metrics were also zero. The I/O count metric is counter that starts at zero and grows with each I/O interaction. Since the responses are “canned” the growth was always consistent for internal calls, but were variable for interactions with the flight data API due to variance in the number of active flights.

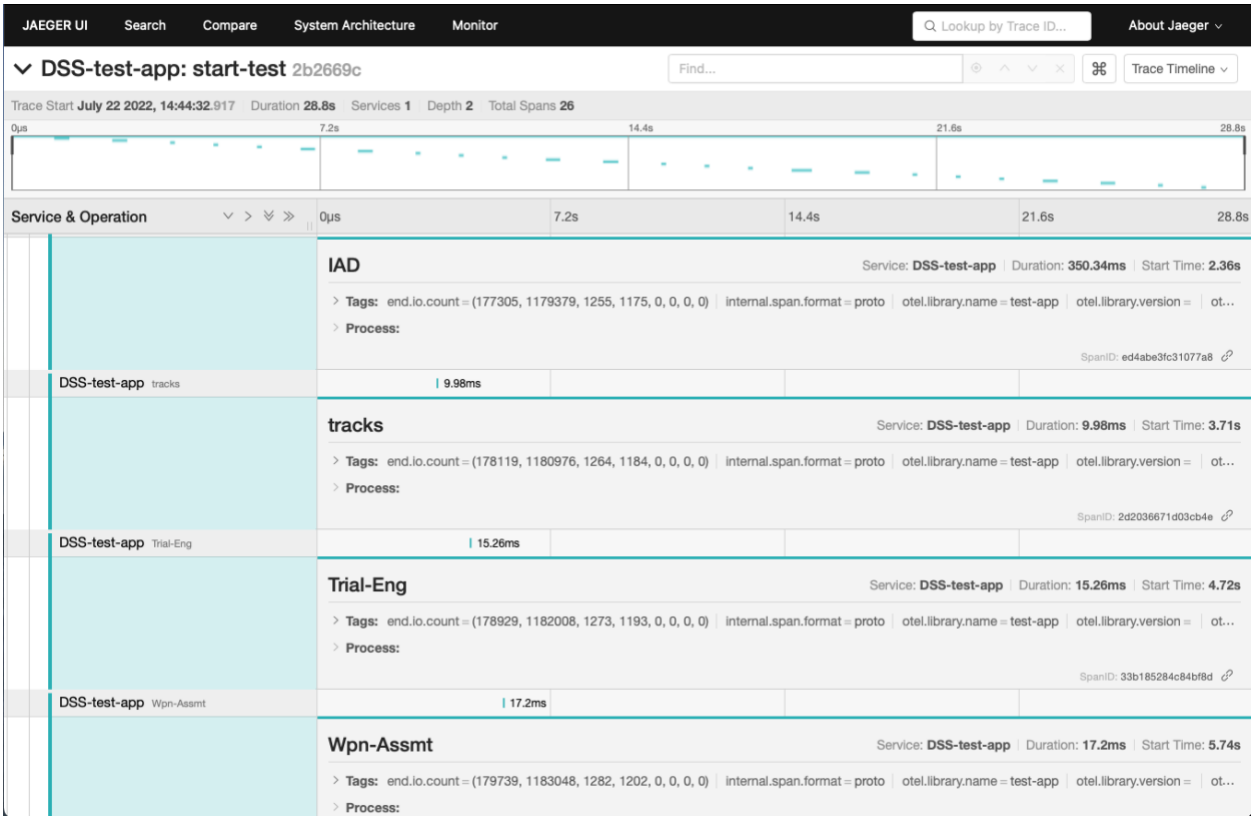

Figure 35. Collection of I/O Metrics within Jaeger

4 RESULTS

The purpose of this chapter is to report results from statistical analysis.

4.1 Load Data Files

Data files were loaded from the 5 different environments discussed in Section 3.6 and merged into a combined “spanData” data set Figure 36.

```
macData <- read.csv('DSS_SpanData-mac-2022-05-02 18_38_26_s10-5-1.csv', header = TRUE)
linpcData <- read.csv('DSS_SpanData-linuxpc-2022-06-06 17_38_29_s10-5-1.csv', header = TRUE)
rpi4Data <- read.csv('DSS_SpanData-rpi4-2022-06-06 17_52_59_s10-5-1.csv', header = TRUE)
awsEC2Data <- read.csv('DSS_SpanData-aws_ec2-2022-06-07 17_44_08_s10-5-1.csv', header = TRUE)
cci_Data <- read.csv('DSS_SpanData-odu_cci-2022-06-28 17_47_20_s10-5-1.csv', header = TRUE)
```

Figure 36. DSS Prototype Data Files

4.2 Convert Data into Useable Metrics

To make the data more usable and easier to understand we apply conversions from text to numeric data and add additional columns with supporting information. A **useCase** column is added to identify specific DSS request use cases, e.g., Get Dulles Airport Data. The data also indicates whether the request is managed internally or a connection to an external service is required to provide a response (i.e., https://opensky-network.org). A **numContainers** column is added to indicate the number of containers involved in providing a use case response (e.g. independent variable). An **ext** column is added to indicate whether an API external to the Docker environment is used; e.g., ext = TRUE for OpenSky API calls.

4.3 Exploratory Data Analysis

A summary of the resulting data is presented in Figure 37. Figure 38 shows that the **Mac implementation of Docker Containers** adds latency within the Docker environment. In non-linux based plaforms, a Docker desktop running a virtual machine is required to provided the Docker capability that is native to Linux platforms. The Mac is considered to be the DSS development environment and not representative of the integration and production environments.

```
Rows: 500
Columns: 9
$ Trace.ID   <chr> "9ee3577fb1b427bc4fc17fecc5154d7d", "f05ddc4dc13aff5c309801…
$ Trace.name <chr> "/TE", "/tracks", "/IAD", "/RIC", "/WA", "/TE", "/tracks", …
$ Start.time <chr> "2022-05-02 10:25:01.366", "2022-05-02 10:25:00.309", "2022…
$ Duration   <dbl> 36.0, 43.3, 464.0, 494.0, 139.0, 30.3, 30.0, 478.0, 546.0, …
$ platform   <chr> "2017-macbook", "2017-macbook", "2017-macbook", "2017-macbo…
$ env        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ useCase    <chr> "Trial Engage (Internal)", "Get Stored Local DSS Tracks (In…
$ useCaseNum <dbl> 2, 1, 4, 5, 3, 2, 1, 4, 5, 3, 2, 1, 4, 5, 3, 2, 1, 4, 5, 3,…
$ ext        <lgl> FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, …
```

Figure 37. Combined Data Summary

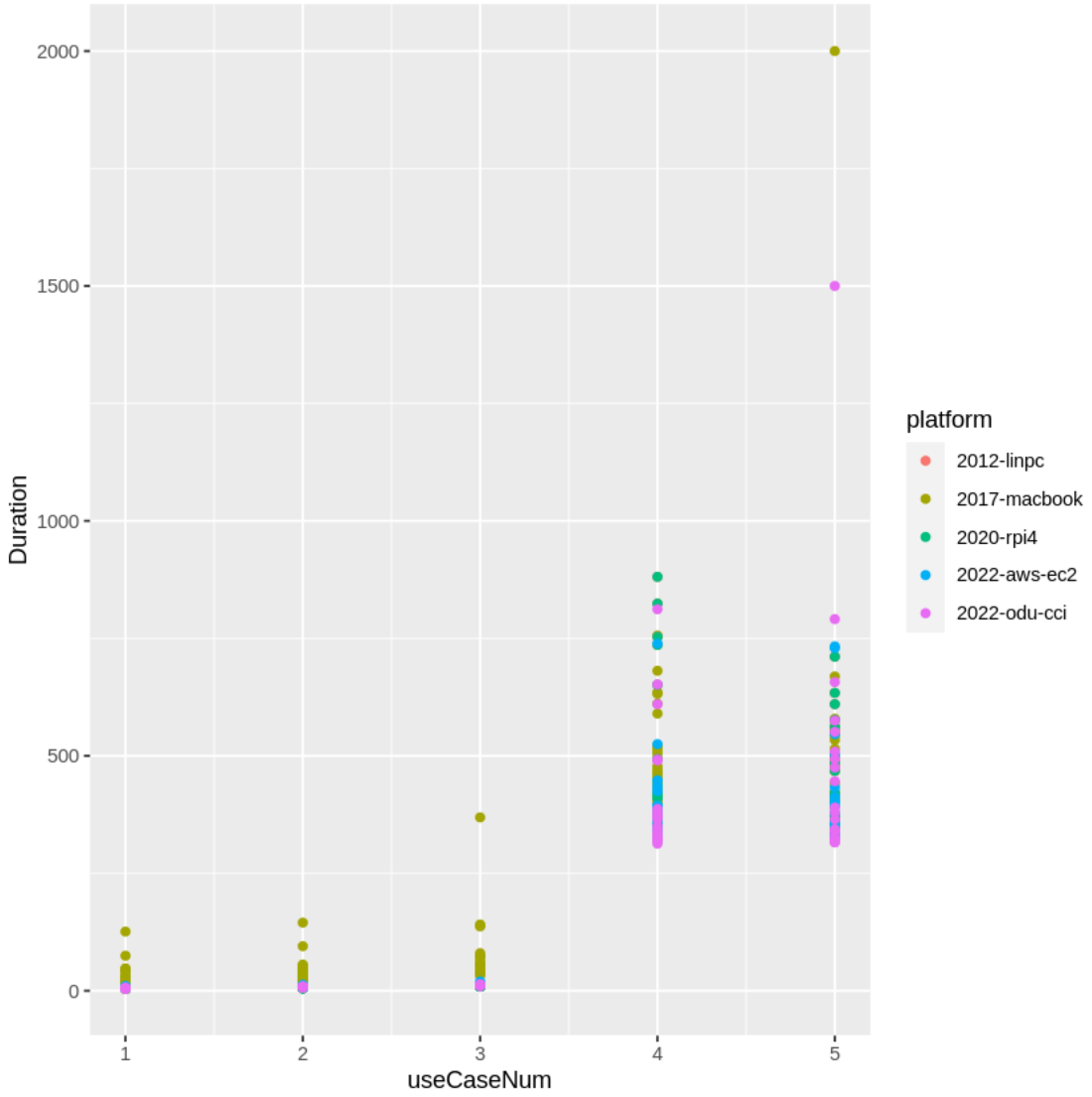

Figure 38. Duration vs. Use Cases and Computing Platform Source

In Figure 39, Figure 40, Figure 41, and Figure 42 we examine the data without the data from the MacBook platform. The plots seem to indicate the presence of 2 clusters. Each plot shows that internal and external duration data is heavily separated. We shall use cluster analysis to investigate.

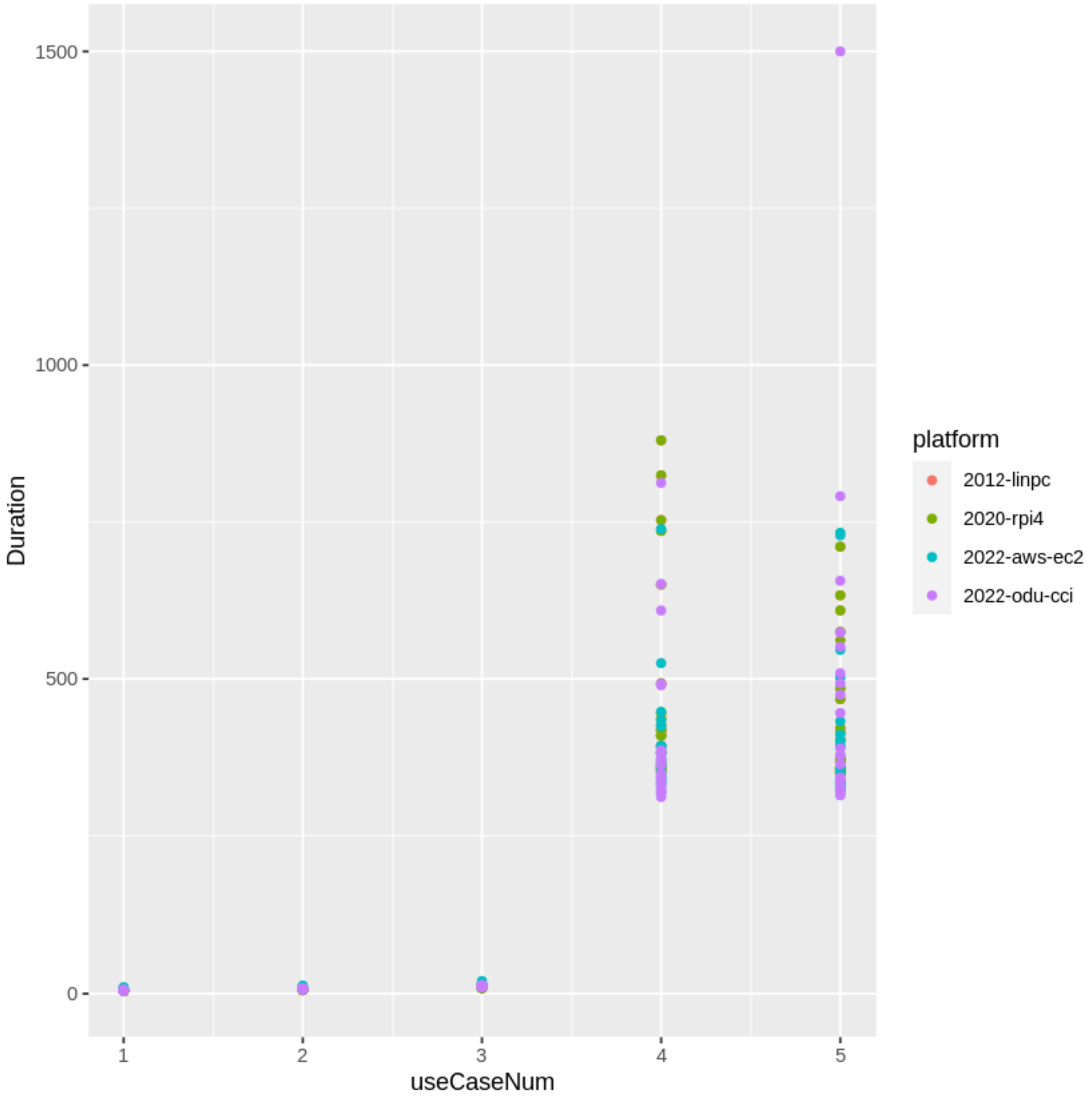

Figure 39. Duration vs. Use Cases with Mac Platform Removed

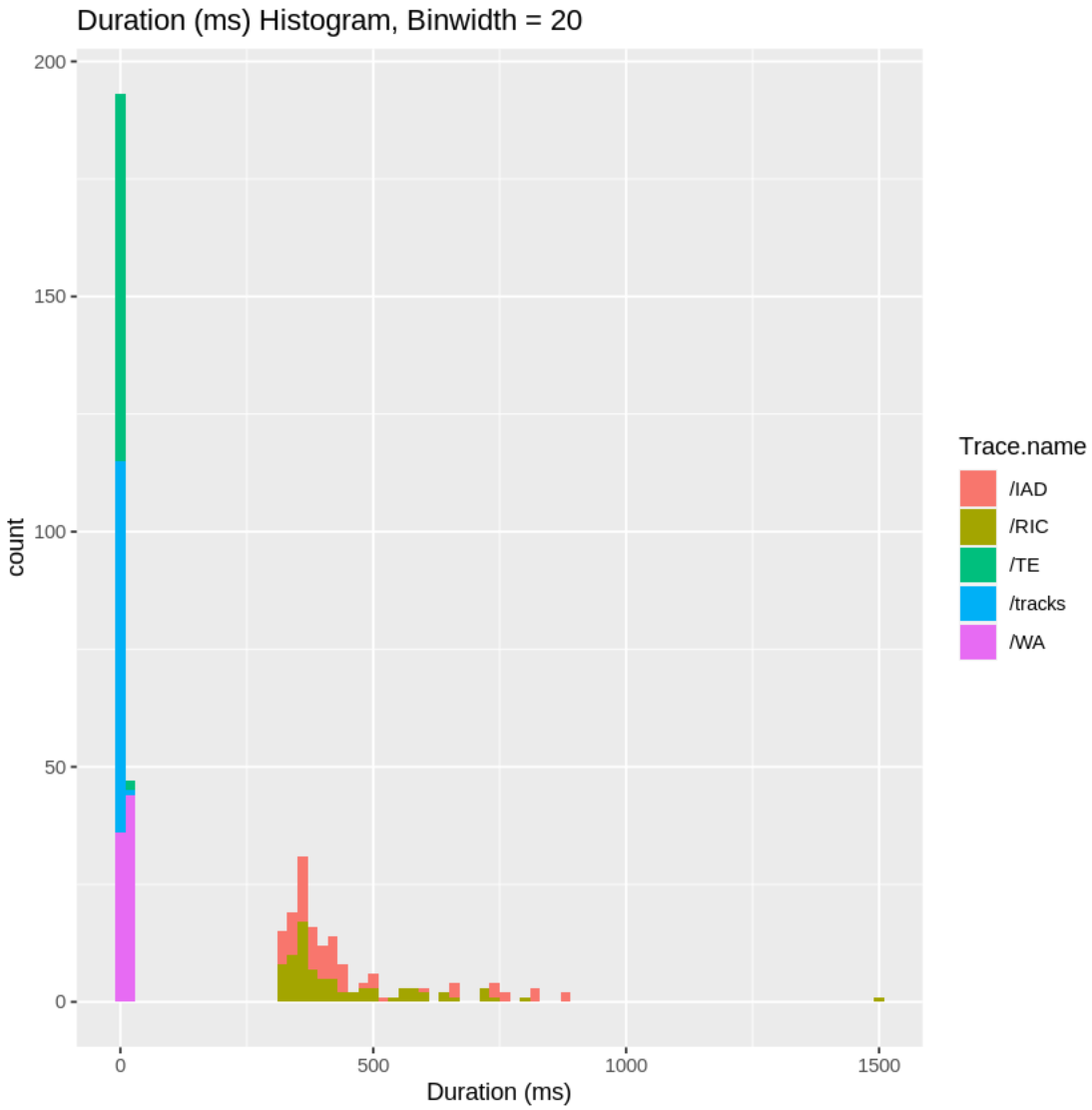

Figure 40. Duration Histogram with Use Case Indicated (i.e. Trace.name)

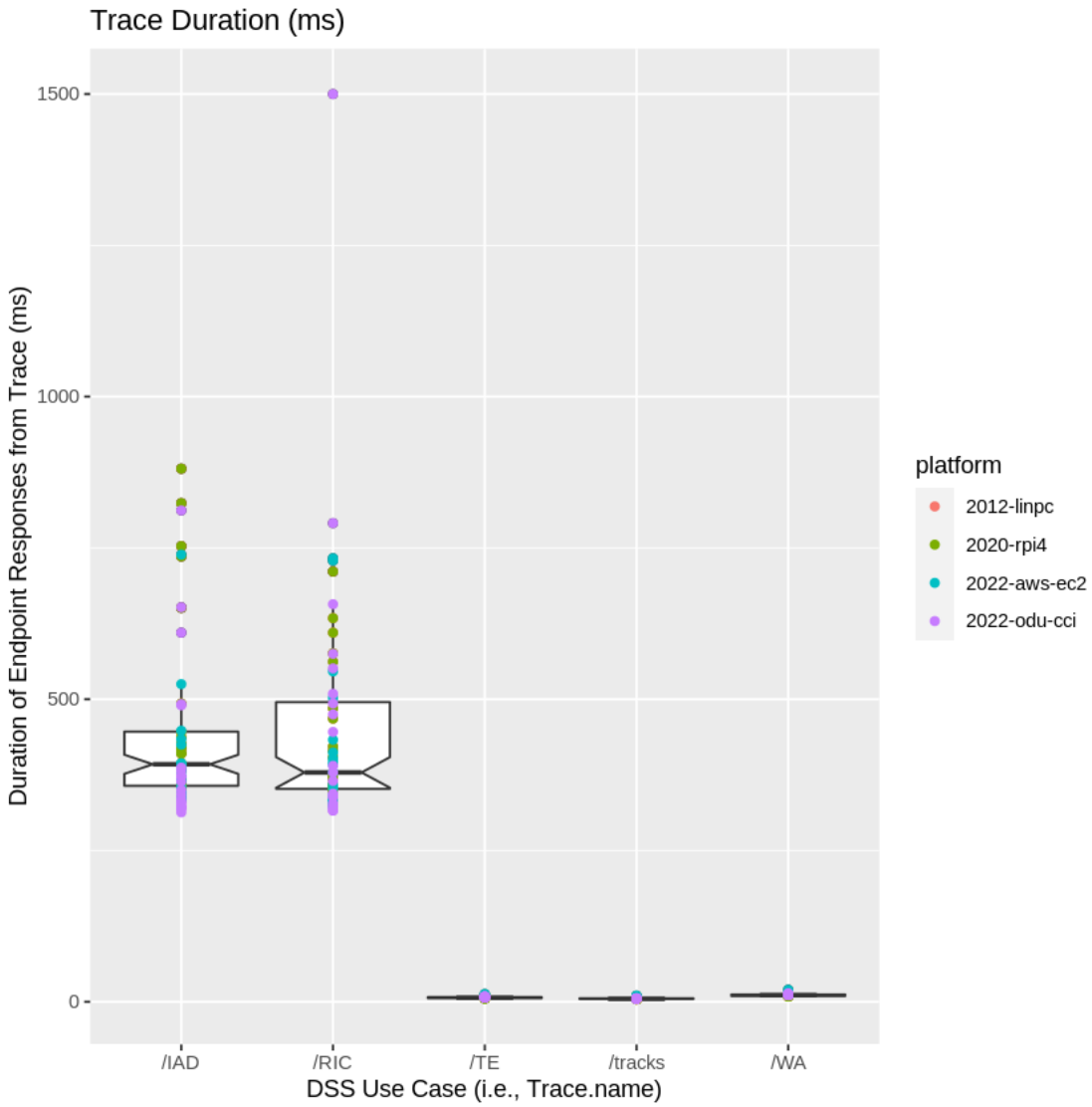

Figure 41. Duration vs. Use Case Boxplot

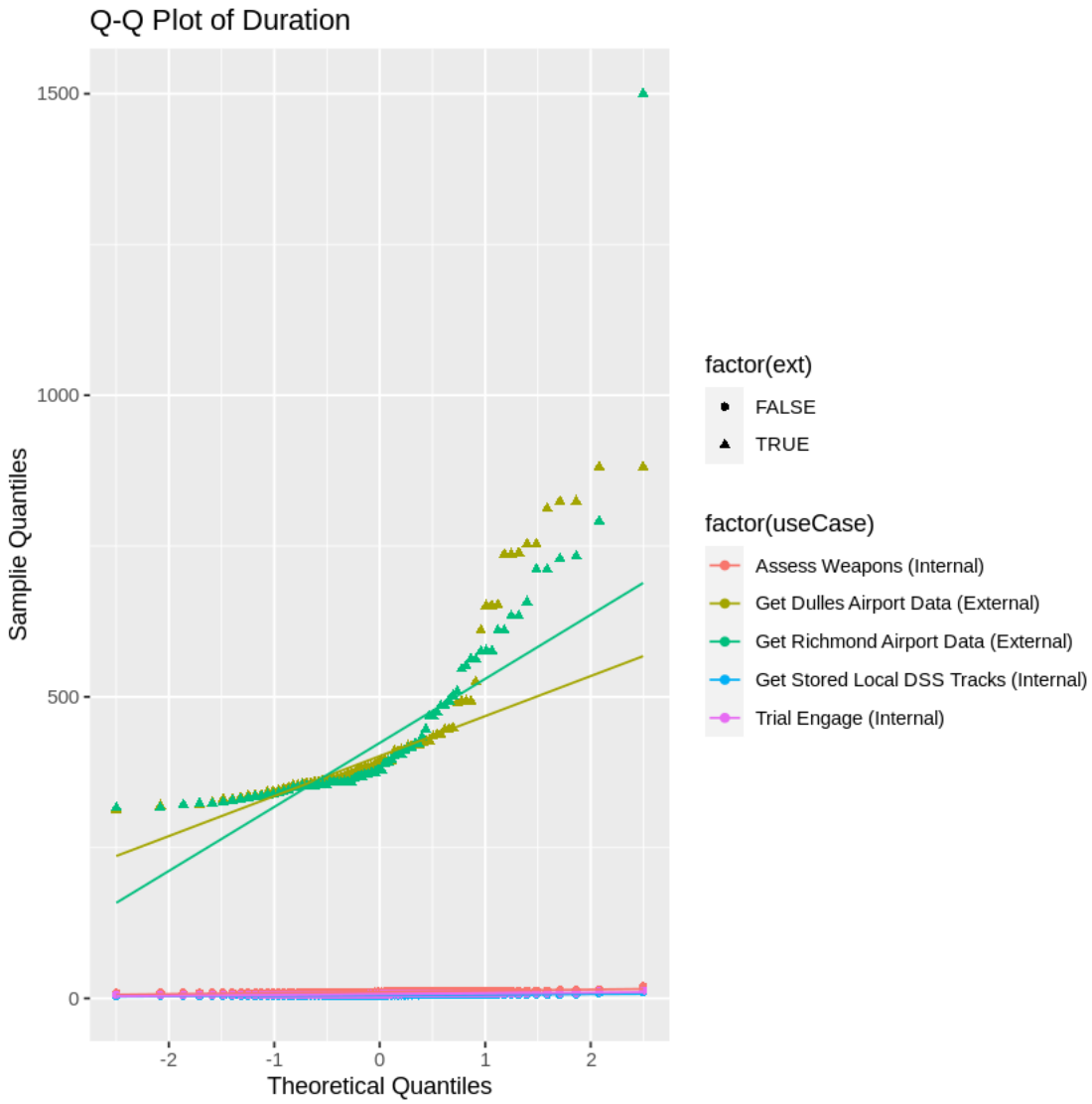

Figure 42. Q-Q Plot of Duration per Use Case

The data presented Figure 43 provides a sample of server routing used to access flight data from the OpenSky API to help explain the change in duration for external interface calls. Multiple non-deterministic network hops account for the increased duration latency. Figure 44 depicts the service endpoint in Gretzenbach, Switzerland.

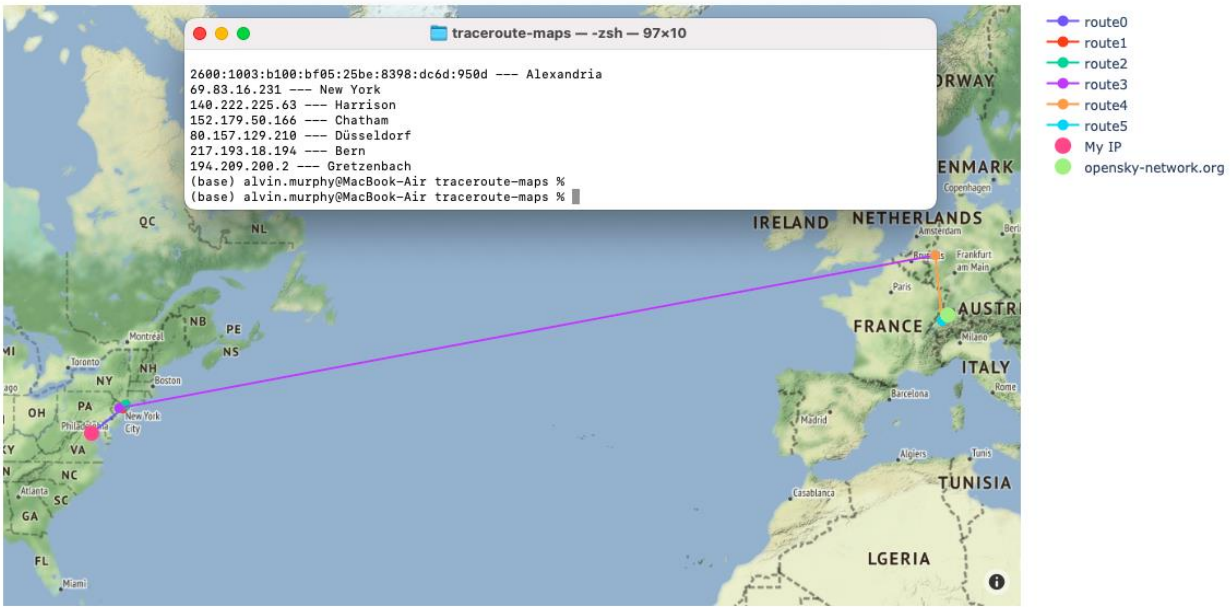

Figure 43. Trace-Route from Home Lab to OpenSky API

Figure 44. OpenSky Endpoint in Gretzenbach, Switzerland

Cluster Analysis

The R library “mclust” was used to verify the separation of internal and external models as indicated from the plots, i.e., use cases that use an external API to collect external flight data

from Richmond (RIC) and Dulles (IAD) airports. The library mclust is a contributed R package for model-based clustering, classification, and density estimation based on finite normal mixture modelling. It provides functions for parameter estimation via the EM algorithm for normal mixture models with a variety of covariance structures, and functions for simulation from these models. MclustBIC returns an object of class 'mclustBIC' containing the Bayesian Information Criterion (BIC) for the specified mixture models numbers of clusters. Auxiliary information is returned as attributes (Scrucca et al., 2016).

A density plot of the duration data is presented in Figure 45. As indicated above, two clusters separating internal and external services call seem to be present.

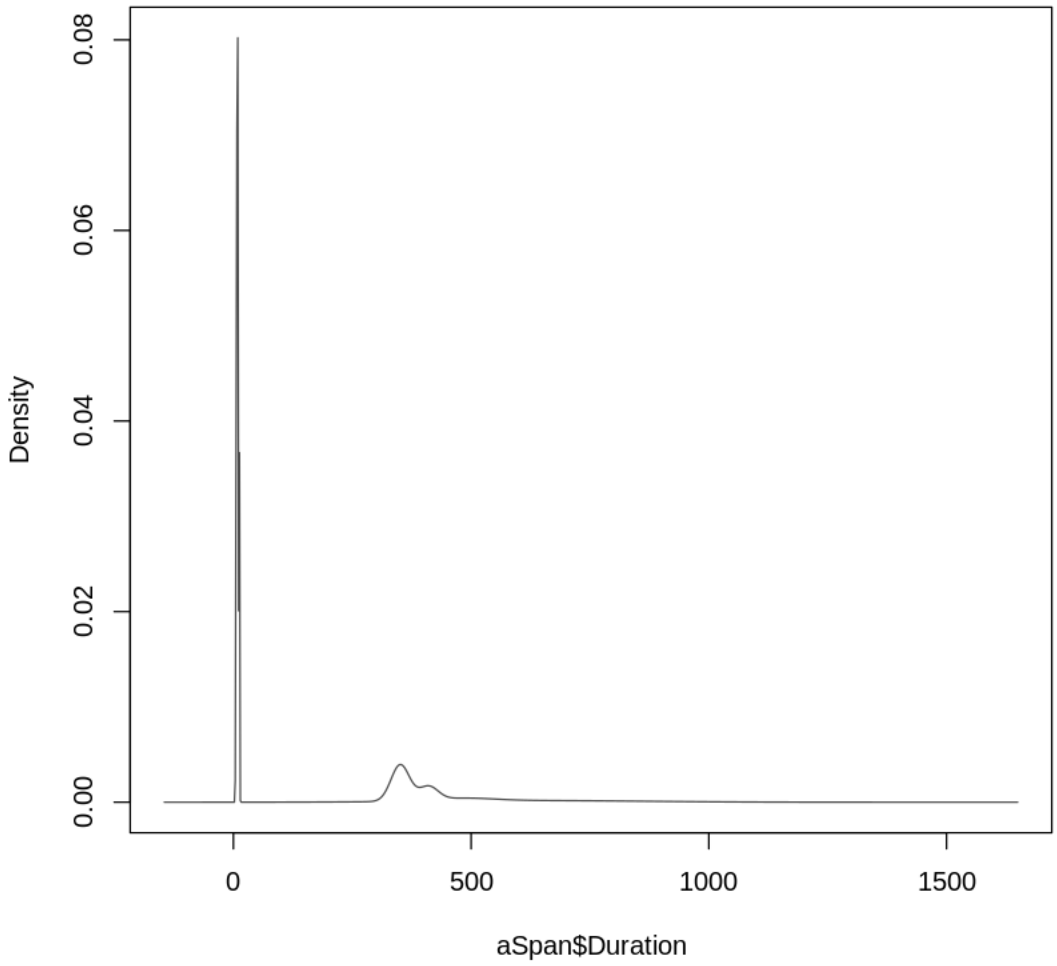

Figure 45. Density Plot of Duration Clusters

Figure 46 and Figure 47 summarize the best Bayesian Information Criterion (BIC) values from BIC analysis. The data indicates that 2 clusters exist. The two best matches are:

- VEV:varying volume,equal shape,varying orientation (ellipsoidal covariance)
- EEE:equal volume,equal shape,equal orientation (ellipsoidal covariance)

The clusters are depicted in Figure 48. We shall separate internal from external data.

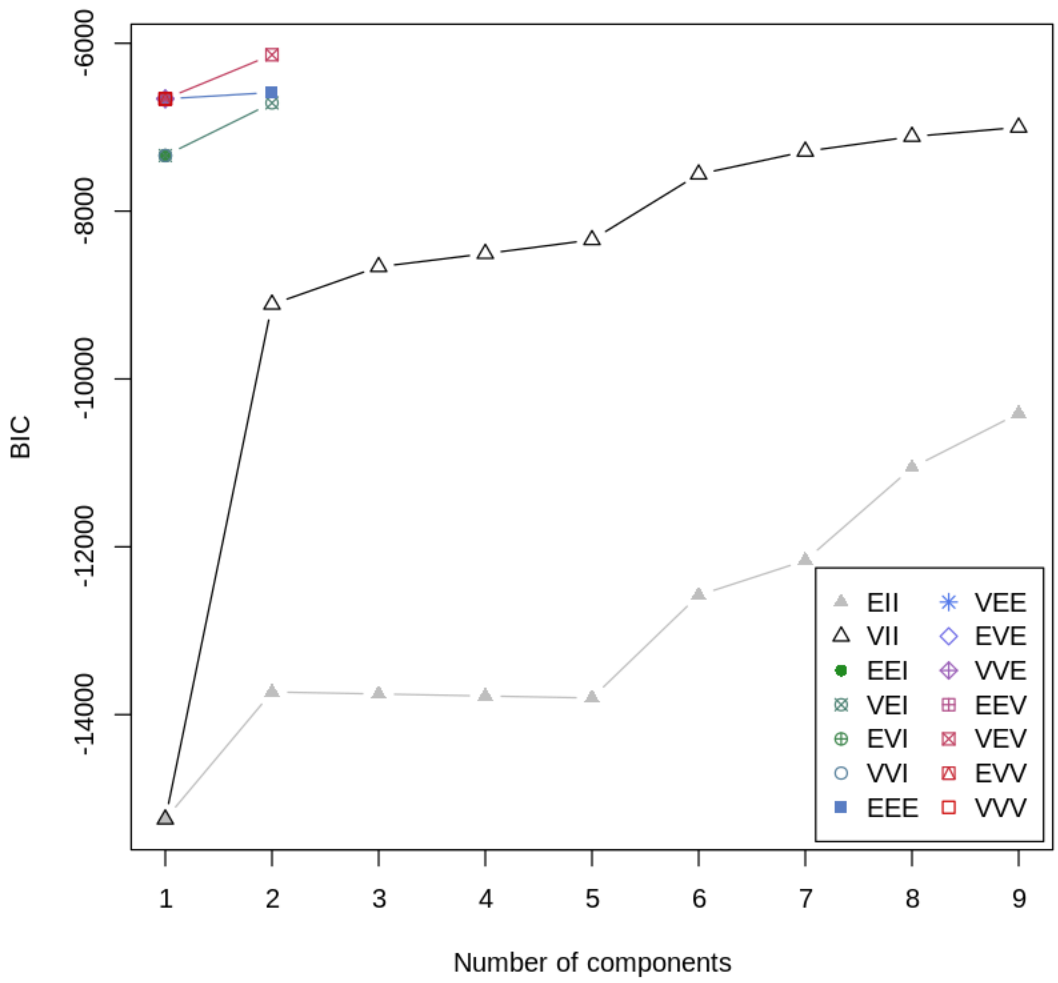

Figure 46. BIC Plot of Components

```
Best BIC values:
             VEV,2      EEE,2      EEE,1
BIC      -6136.963 -6586.3351 -6662.2804
BIC diff     0.000  -449.3724  -525.3177
```

Figure 47. Best BIC Values

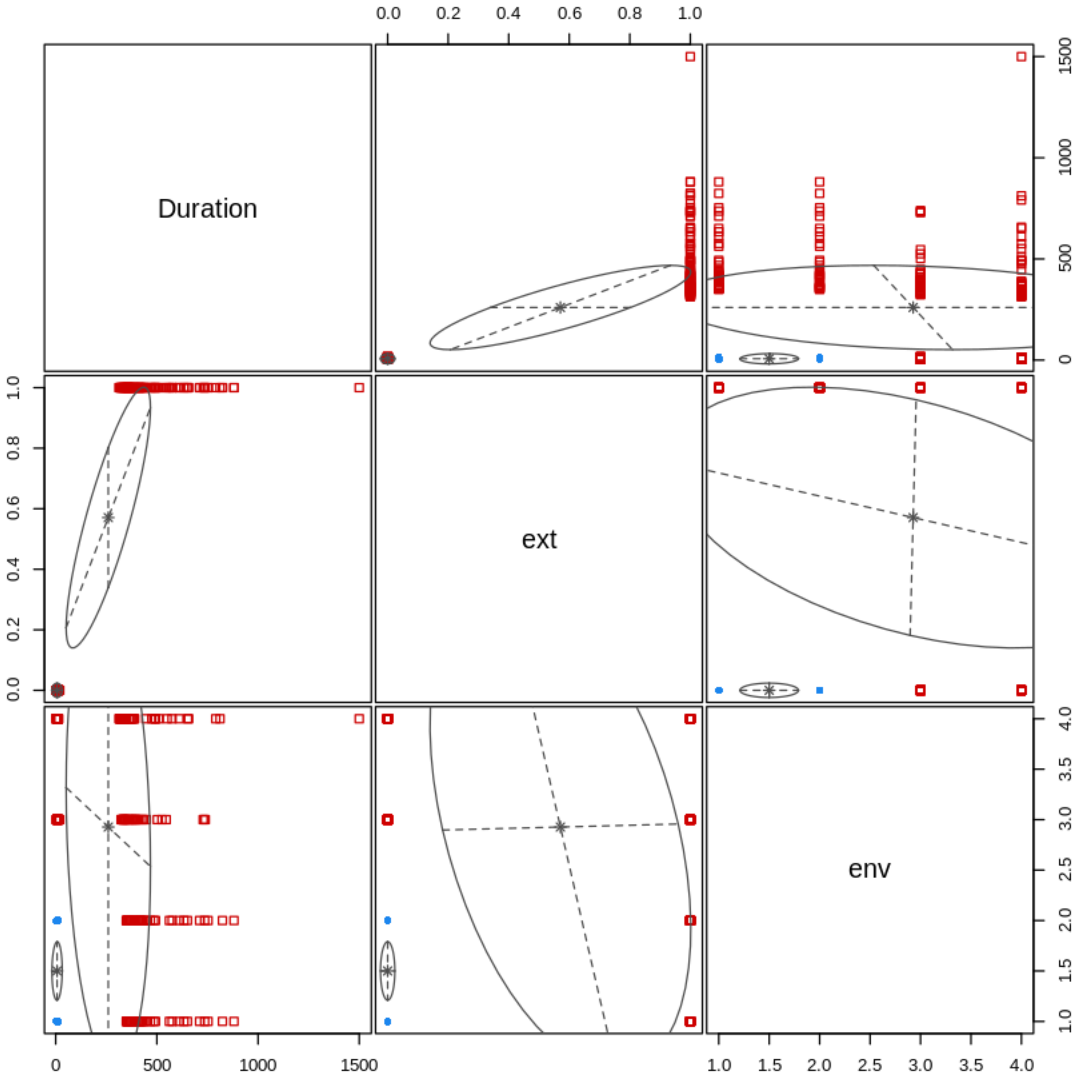

Figure 48. Classification Plot of BIC Component Relationships

Internal Data Analysis

The revised Boxplot for internal data is depicted in Figure 49. An internal duration histogram is presented in Figure 50. The histogram plot indicates that the data is not normally distributed and suggests an adjustment will be needed enable application of statistics.

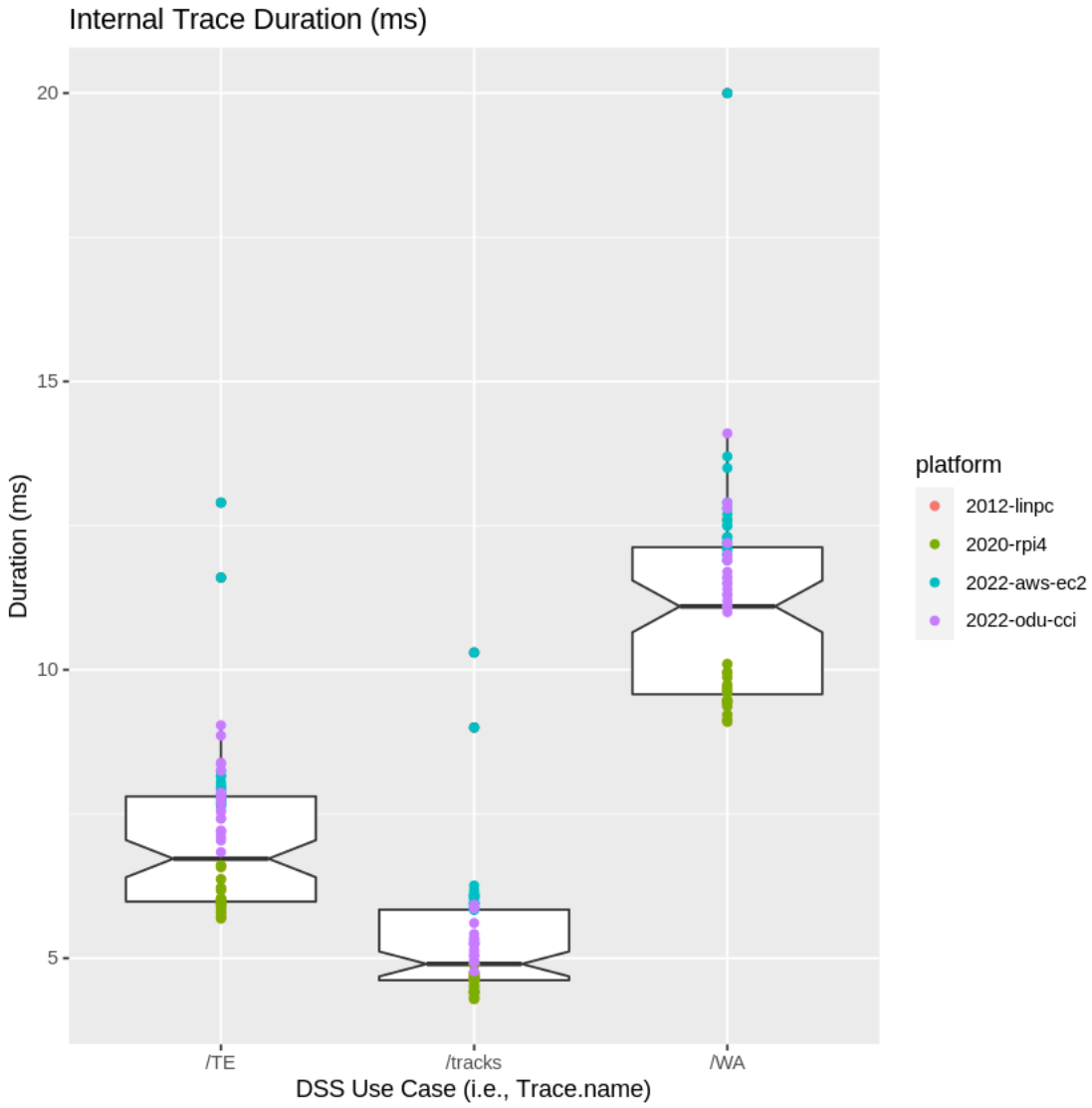

Figure 49. Internal Duration Boxplot

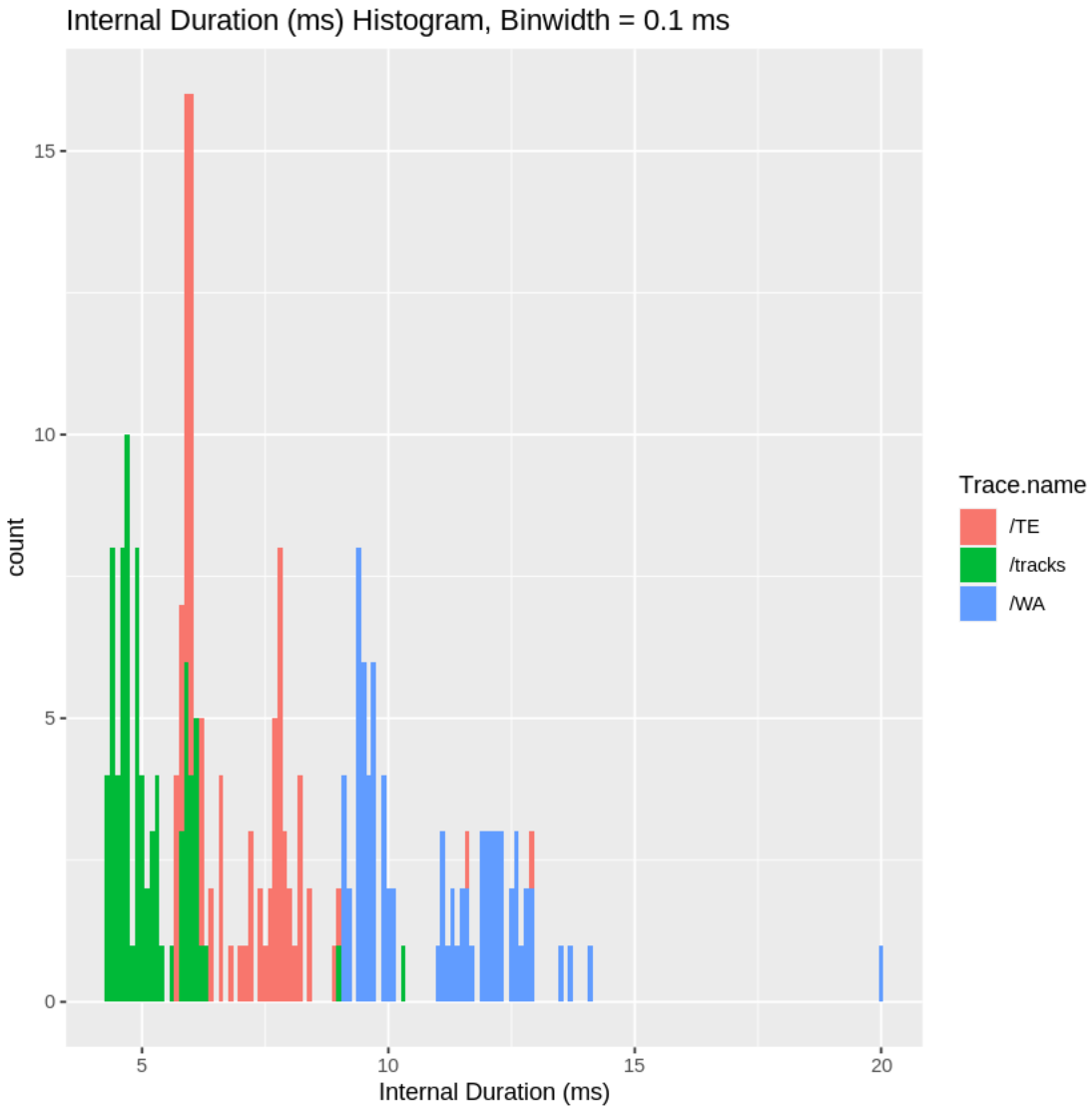

Figure 50. Internal Duration Histogram

External Data Analysis

The revised Boxplot for external data is depicted in Figure 51. An external duration histogram is presented in Figure 52. The histogram plot indicates that the data is not normally distributed and suggests an adjustment will be needed enable application of statistics.

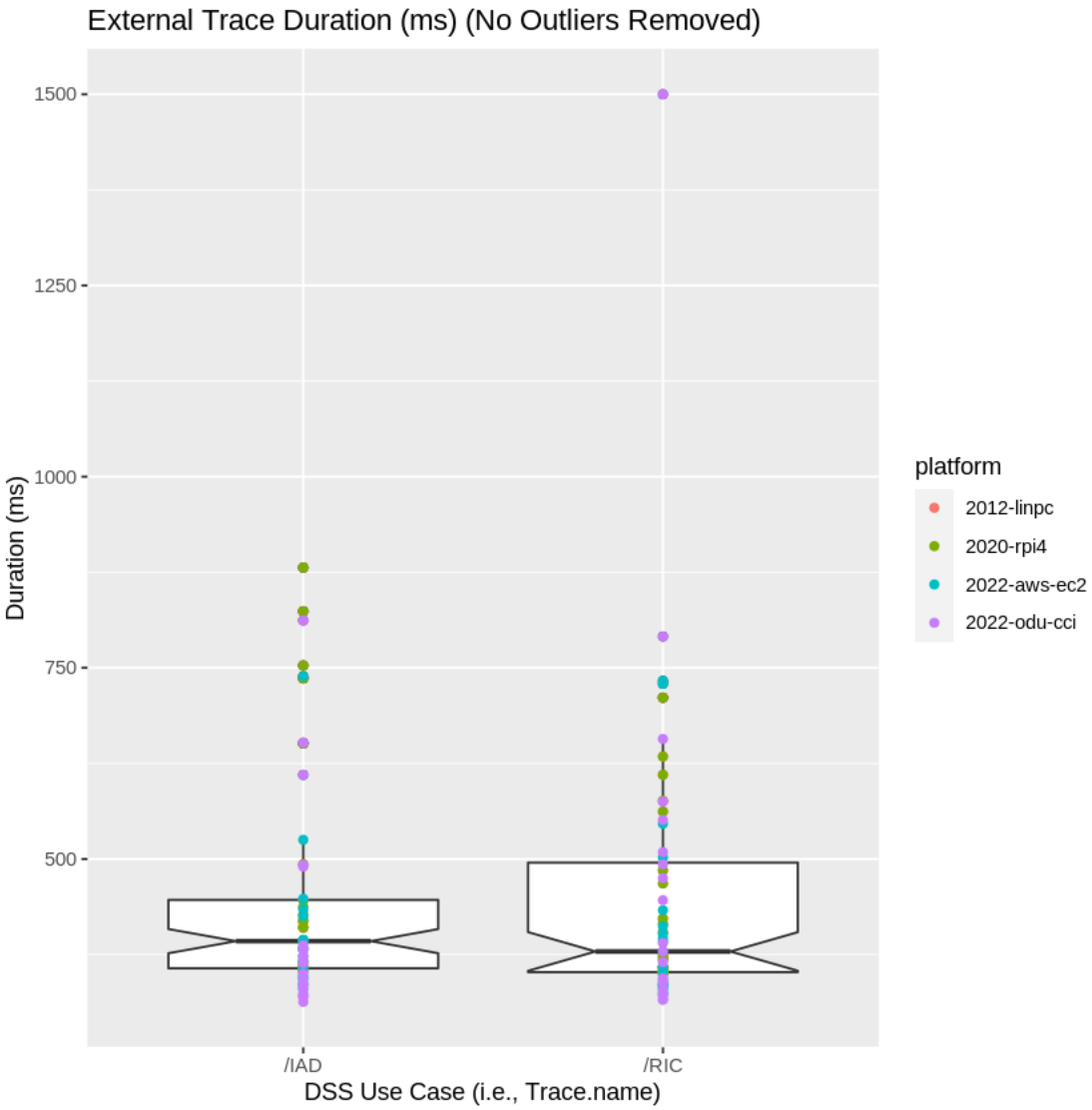

Figure 51. External Duration Boxplot

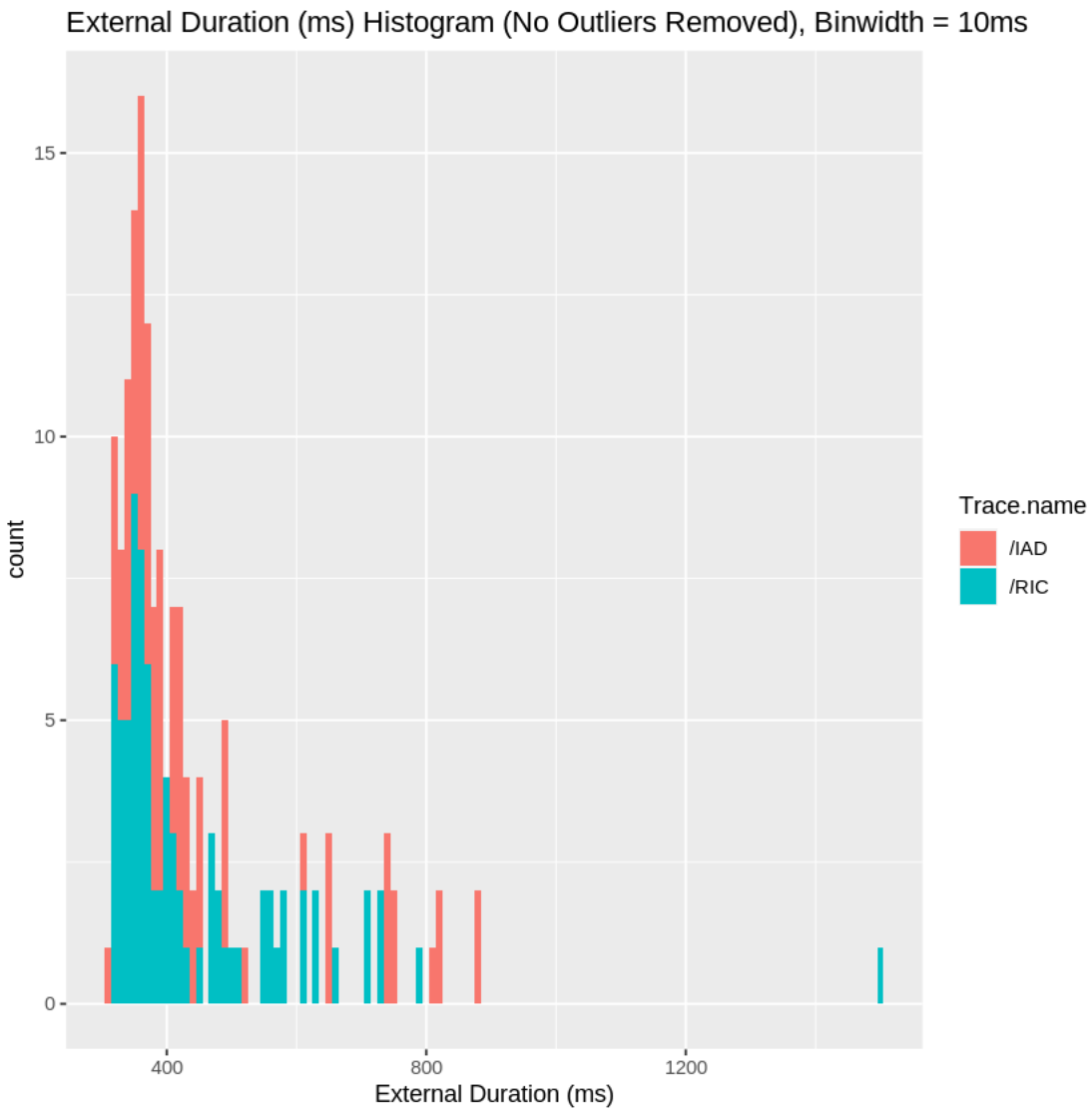

Figure 52. External Duration Histogram

Shapiro-Wilk Test for Normal Distribution

The Shapiro-Wilk test is used to determine if a dataset is normally distributed to enable application of the t-Test for statistical hypothesis testing. The null-hypothesis of the Shapiro-Wilk test is that the population is normally distributed. Thus, if the p-value is less than the chosen alpha level, then the null hypothesis is rejected and there is evidence that the data

tested are not normally distributed. On the other hand, if the p value is greater than the chosen alpha level, then the null hypothesis (that the data came from a normally distributed population) cannot be rejected (e.g., for an alpha level of .05, a data set with a p value of less than .05 rejects the null hypothesis that the data are from a normally distributed population).

The Shapiro-Wilk test results in Figure 53 indicate that the internal and external data samples are not normally distributed and need to be adjusted to apply hypothesis testing. The exploratory data analysis plots indicated significant gaps in the internal data with short durations.

```
Shapiro-Wilk normality test

data:  iSpan$Duration
W = 0.9075, p-value = 5.081e-11


    Shapiro-Wilk normality test

data:  eSpan$Duration
W = 0.71543, p-value = 3.053e-16
```

Figure 53. Shapiro-Wilk Normality Test

The exploratory data analysis and Shapiro-Wilk test results seem to indicate measuring of an API response with very little variation. Given that each use case is expected to have some processing delay, the R rnorm function is used to add a normally distributed processing delay to each of the use cases with a mean of 50 ms and a standard deviation of 10 ms. The resultant change in Shapiro-Wilk test results after application of the processing delay is presented in Figure 54. The data form the internal use cases now yields a p-value of 0.2265 which indicates

that the data is now normally distributed. However, the data from the external use cases still indicates that the distribution is not normal with a p-value far less than the alpha level of 0.05.

```
    Shapiro-Wilk normality test

data:  pd_iSpan$Duration
W = 0.9921, p-value = 0.2265


    Shapiro-Wilk normality test

data:  pd_eSpan$Duration
W = 0.72568, p-value = 6.034e-16
```

Figure 54. Shapiro-Wilk Testing with Processing Delay

Histograms of the original internal use case duration and internal use case duration with a normally distributed processing delay added are plotted with a binwidth of 1 ms in Figure 55 and Figure 56. The Original internal duration is skewed right. It can also be noted that processing delays for each of the internal use cases are somewhat visibly separated. However, the duration with processing delay indicates data that is normally distributed with normally distributed processing delays across the internal use cases.

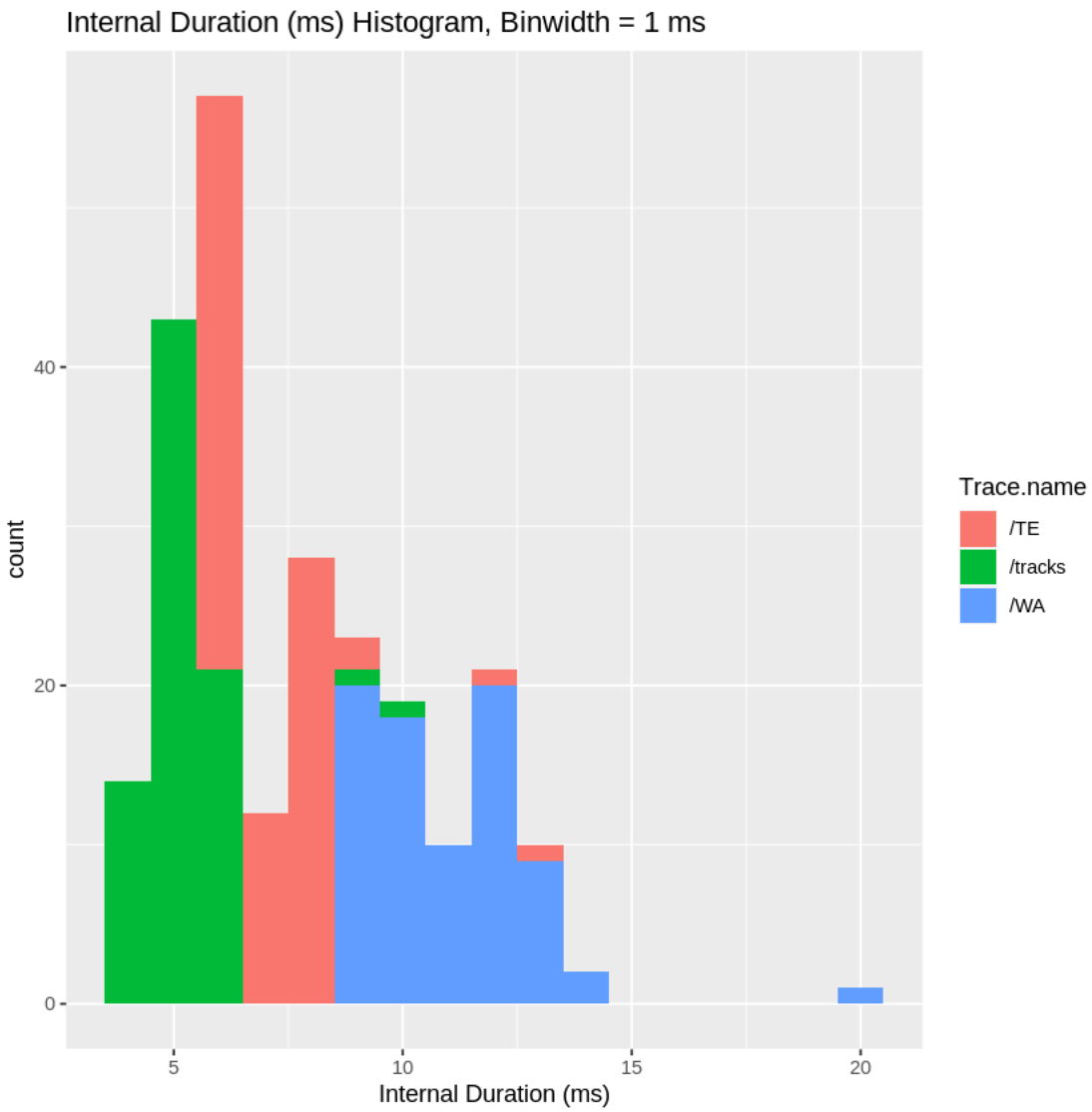

Figure 55. Original Internal Duration

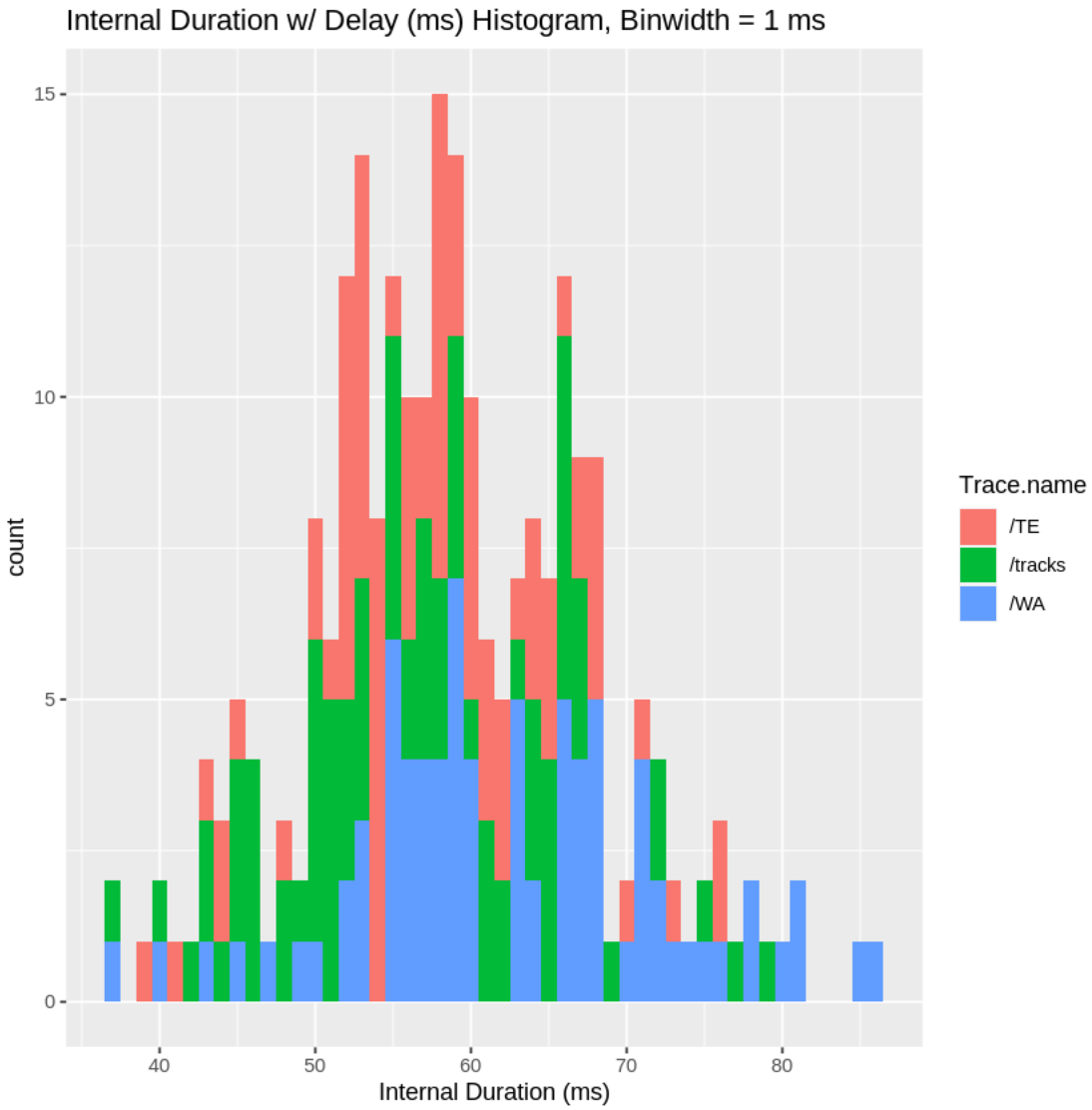

Figure 56. Modified Internal Duration with Processing Delay

Histograms of the original external use case duration and external use case duration with a normally distributed processing delay added are plotted with a binwidth of 10 ms in Figure 57 and Figure 58. Both plots of the external durations continue to be skewed right indicating that the data is not normally distributed.

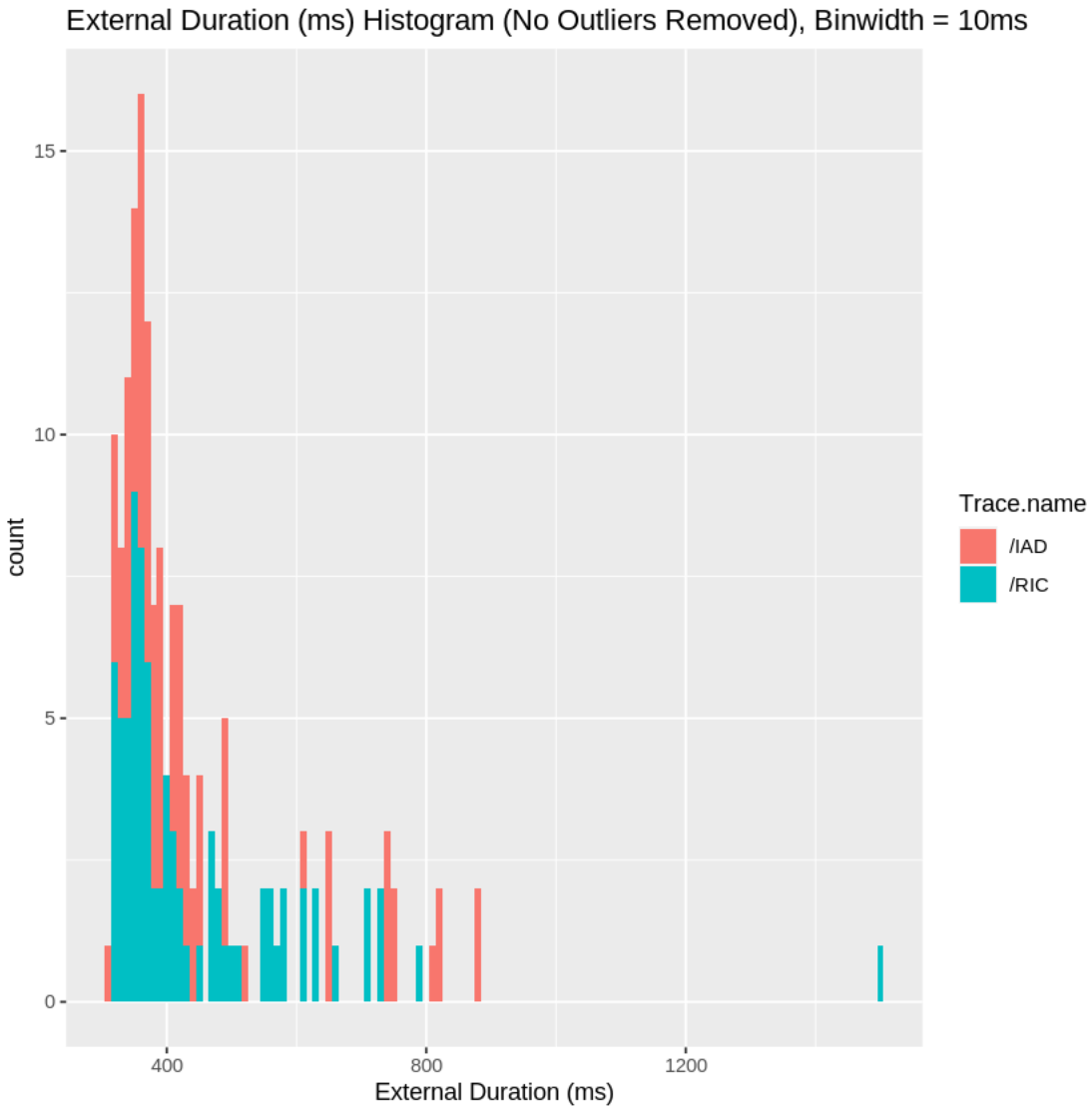

Figure 57. Original External Duration

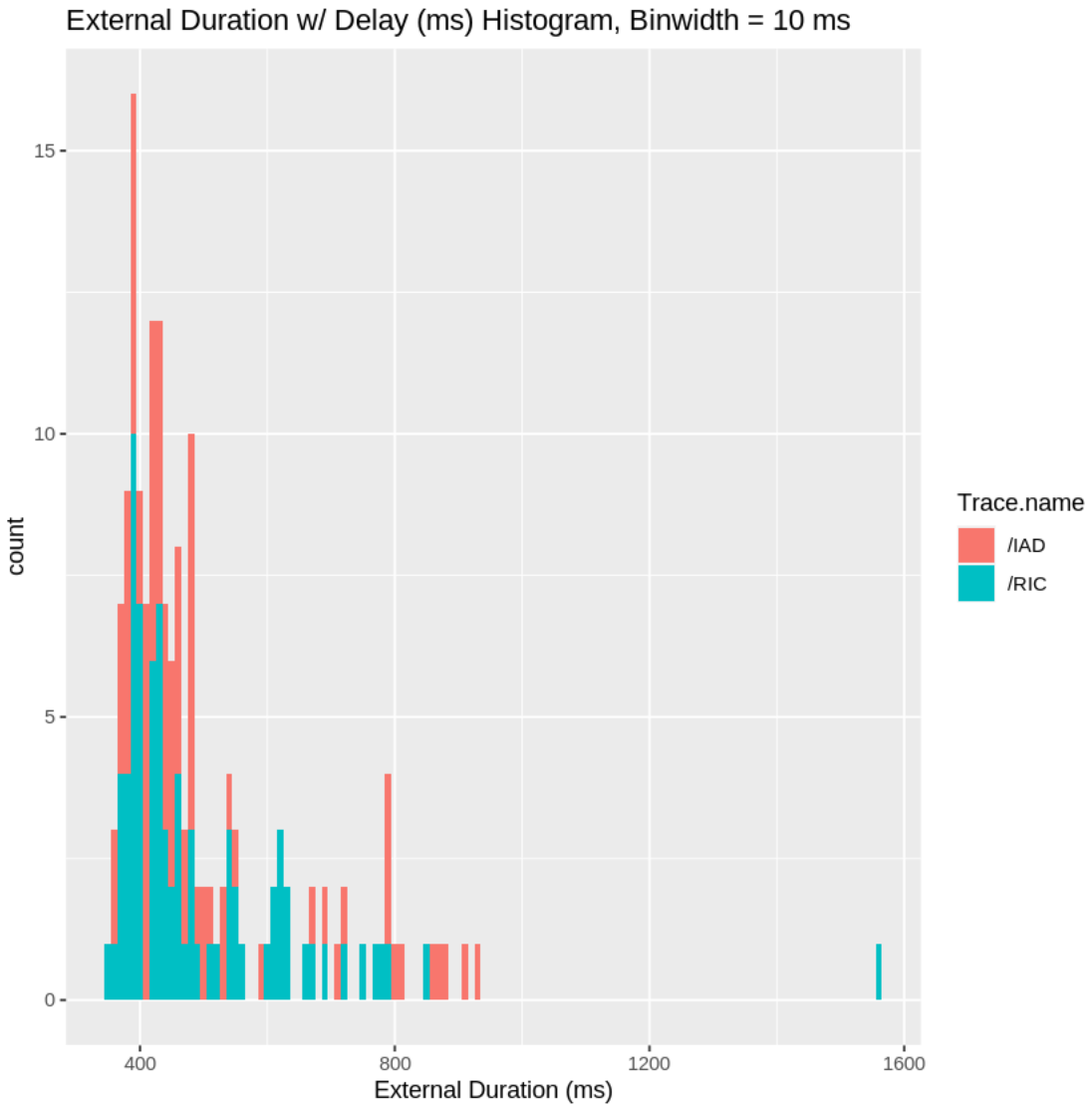

Figure 58. Modified External Duration with Processing Delay

The data indicates that a normal distribution is achieved with the internal data but not with the external data due to the extreme variation in response times through external servers and routers to return flight data for Richmond and Dulles airports. We shall apply a t-test for the internal data, but for the external data we shall apply a binomial test to see if the threshold of 500 ms can be maintained. This will require adding a threshold indication to the data sets;

i.e. an indication of TRUE if the duration is less than 500 ms and FALSE if not. Figure 59 presents the revised data with the threshold indicator (hthreshold) added.

```
Rows: 160
Columns: 10
$ Trace.ID   <chr> "0d8efde6f35af9599ae0ffc9cd68b6fb", "d6c36d3d53a329daf1f72e…
$ Trace.name <chr> "/RIC", "/RIC", "/RIC", "/RIC", "/RIC", "/RIC", "/RIC", "/R…
$ Start.time <chr> "2022-06-06 21:36:51.531", "2022-06-06 21:36:45.723", "2022…
$ Duration   <dbl> 476.2477, 416.5003, 671.6718, 404.2449, 427.0154, 391.0872,…
$ platform   <chr> "2012-linpc", "2012-linpc", "2012-linpc", "2012-linpc", "20…
$ env        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,…
$ useCase    <chr> "Get Richmond Airport Data (External)", "Get Richmond Airpo…
$ useCaseNum <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,…
$ ext        <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,…
$ hthreshold <lgl> TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, TRU…
```

Figure 59. Revised Data "Glimpse" with Threshold Indicator

4.4 Hypothesis Testing

We can now proceed with hypothesis testing. We will apply a t-Test to our normalized internal use case data. However, we will use a Binomial test to assess the probability of meeting the 500 ms requirements.

t-Test (Internal Use Case Data)

Given that we were able verify a **normal distribution** with the process delay applied to the internal use case data, we are able to use a Student's t-Test to test the hypothesis on the internal use case data. Our mean is 500 ms (e.g. $\mu = 500$ ms) and our null hypthesis is less than 500 ms. This is an example of what is called a one-tailed hypothesis; e.g. evidence against the null hypothesis comes from only one tail of the distribution (namely, duration above 500).

Sample t-test results in Figure 60 of the internal span data indicates a p-value of 1 so we fail to reject the null hypothesis that that the duration mean is less than 500 ms. The p-value converges to 1 because all of the internal duration results are far less than 500 ms.

```
    One Sample t-test

data:  x
t = -745.31, df = 239, p-value = 1
alternative hypothesis: true mean is greater than 500
95 percent confidence interval:
 57.9299     Inf
sample estimates:
mean of x
 58.90716
```

Figure 60. t-Test Results for Internal Use Case Data

Binomial Tests

We'll use Binomial Tests to test the probability of success for meeting the 500 ms duration. For Binomial Test we need to review the number of trials and number of successes.

4.4.1.1 All Use Case Data

Let's look at the combined data first. For all of the use case data, we had 400 use case runs resulting in 354 successes where the duration was less than 500 ms. The Binomial Test results in Figure 61 indicate that the probability of success for all data is 89%.

```
   Mode   FALSE    TRUE
logical      46     354


    Exact binomial test

data:  354 and 400
number of successes = 354, number of trials = 400, p-value = 1
alternative hypothesis: true probability of success is less than 0.5
95 percent confidence interval:
 0.0000000 0.9102965
```

```
sample estimates:
probability of success
                0.885
```

Figure 61. Binomial Test (All Use Case Data)

For the internal data we had 240 use case trials with 240 trials resulting of durations of far less than 500 ms; i.e. 100 probability of success. We used a t-Test to test our hypothesis. However, a Binomial Test will be used to test the probability of success for the external use cases.

4.4.1.2 External Use Case Data

For the external use cases we have 160 trials with 114 with a duration less than 500 ms. The results of the Binomial Test in Figure 62 indicate a 71% probability of success for the external use cases.

```
   ext          hthreshold
 Mode:logical   Mode :logical
 TRUE:160       FALSE:46
                TRUE :114

    Exact binomial test

data:  114 and 160
number of successes = 114, number of trials = 160, p-value = 1
alternative hypothesis: true probability of success is less than 0.5
95 percent confidence interval:
 0.0000000 0.7711356
sample estimates:
probability of success
                0.7125
```

Figure 62. Binomial Test (External Use Case Data)

5 DISCUSSION

5.1 OVERVIEW OF FINDINGS

Findings are summarized through the analysis of questions and hypothesis developed in Chapters 2 and 3 of this dissertation.

PRIMARY HYPOTHESIS

General Discussion of Normality

It was required to separate external data from internal to establish normality of the data samples. A processing delta with a gaussian distribution was applied to the data set to replicate the variation in processing time for each call to the services. The data from the internal use cases exhibit a normal distribution after application of a gaussian processing delay. The data from the external use cases could not be transformed into a normal distribution. However, a binomial test was used to assess the probability of maintaining the 500 ms threshold with external data routing uncertainties.

Hypothesis Results

Hypothesis testing using the Student's t-Test and Binomial Test indicates that latency constraints of 500 ms can be maintained internally and external. However, several external samples were greater than 500 ms. This is most likely due to the non-deterministic nature of internet (e.g. http) requests. Within the internal environment, data is directly routed between microservices within the Docker environment within a private network. The data shows that a container based microservice architecture can meet the requirement; however, care must be taken to manage processing per container that may increase container response times.

DSS Prototype Environment

The non-deterministic nature of the Docker environment on the MacBook laptop significantly affected the ability to assess deterministic behavior. Boxplots of data inclusive of what was sampled from the MacBook clearly depicted this issue. Linux platforms run a container as intended; however, non-linux platforms require the use of a Linux based Virtual Machine on top of the host OS to implement containers. While the MacBook met the needs for rapid software development, the use of a separate integration and test environment was clearly validated through the collected data.

HYPOTHESIS DERIVED FROM QUESTIONS

The question-based hypothesis results were not directly measured and tested; however, evaluation of the primary hypothesis provided evidence to assess the question-based hypothesis statements. The results are summarized in Table 6.

Table 6. Question Based Hypothesis Results

Hypothesis	Description	Assessment
H1	*The scalability of microservices as new data sources are added to an architecture enables maintaining high **throughput** and predicted to have a positive impact on rapid fielding of capability of capability.*	True
H2	*Microservices orchestration through DevSecOps technologies enables maintaining low **latency** services call responses and predicted to have a positive impact on **usability**.*	True
H3	*Web-based interfaces (e.g. RESTful HTTP) will increase **jitter** and predicted to have a negative impact on usability and the deterministic performance needed for positive control of ownship weapons.*	True
H4	*A system architecture model can predict end-to-end **latency** within a mission kill chain to quantify SoS usability.*	True

Hypothesis H1

The sequence diagram in Figure 27 provides an overview of microservice interactions used to realize each of the use cases identified through mission engineering. A summary of how microservices applications are reused to realize mission threads is presented in Table 7. A Sensor Interface Application is used to mitigate change when new data sources are added. The analysis revealed that low latency and high throughput can be maintained amongst the internal applications. However, external sources exhibited a non-deterministic behavior. The use of an external interface layer (e.g., Sensor Interface Application) can help to isolate the external behavior from internal and enable design flexibility and scalability to ensure that requirements are met (e.g. less than 500 ms duration) and enable rapid fielding of new capability. Hypothesis H1 is assessed as True.

Table 7. DSS Prototype Applications Mapped to Mission Use Cases

Mission Use Case	User Interface App	Track Management App	Weapon Assessment App	Trial Engagement App	Sensor Interface App
Review Tactical Information	X	X			X
Review Weapon Recommendations	X	X	X		
Review Predicted Weapon Effectiveness	X			X	

As seen with Hypothesis H1, the orchestration of microservices can enable control of latency to ensure that the system is useable. We saw through the analysis that our 500 ms threshold can be met. Additional we demonstrated an ability to vary processing delay and still meet requirement. Out software protype was based in the Python programming languages; however, system design technologies can also be varied (e.g., C++, Java) to enable design flexibility to meet requirements. Hypothesis H2 is assessed as True.

Hypothesis H3

Our analysis revealed significant variation in response time when dealing with external data sources. To mitigate this risk, we included a sensor interface layer in the design to abstract the external delays from the management of response times internal to the system. Hypothesis H3 is assessed as True for external interfaces; however, system engineers and software developers have a high degree of design flexibility to control internal latency related jitter.

Hypothesis H4

Plots of the internal duration latency mapped directly to the software architecture threads depicted in the sequence diagrams. Figure 63 demonstrates the predictability of latency to each use case based upon microservice application interactions. This changed when gaussian process delay was added to each of the threads; however, this same processing delay can be implemented in software architecture models to support simulation. Hypothesis H4 is assessed as True.

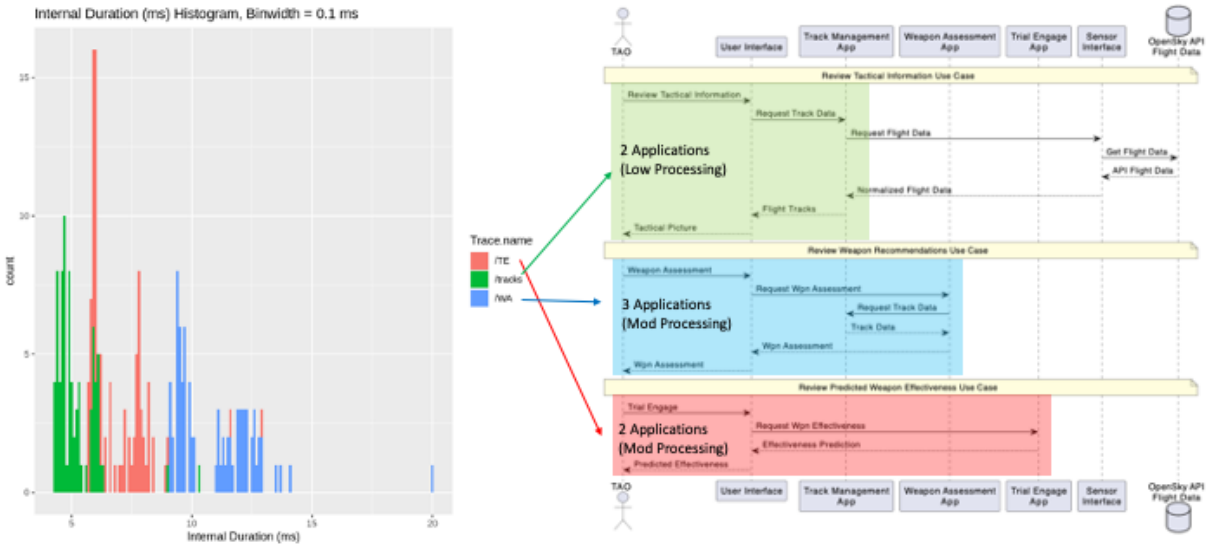

Figure 63. Mapping of Measured Durations to Internal Use Cases

5.2 RESEARCH IMPLICATIONS

This research has shown how mission and system engineering can be used to design a relevant experimental prototype to support capability analysis. Use cases were used to define mission objectives for a notional Decision Support System (DSS). Sequence diagrams were used to define associated mission threads for the identified use cases. These use cases were then mapped to reusable containerized application (e.g., software configuration items (SCIs)) to create a system of system to meet mission requirements. The use of containers demonstrated an ability to quickly re-configure (e.g., composability) to meet emergent mission requirements beyond what was possible with a monolithic system. We demonstrated that mission requirements can be met with a containerized system of systems approach.

5.3 RESEARCH LIMITATIONS

While this research explored requirements definition and system architecture design through the use of mission and system engineering practices, it does not cross into detailed software design that would be within the domain of computer science. Computer scientists would take the requirement and system architecture defined through mission and system engineering and been to identify software technologies and software implementation patterns to ensure that requirements continue to be met with the detailed software design. Messaging frameworks such as Kafka and AMQP and programming language such as C++, Java, Rust, and Node.js all have implications that may affect meeting requirements. Many of the SCIs may be decomposed into smaller services that become part of the SCI. Some of these software technology options were briefly discussed in Section 2.

6 CONCLUSIONS

6.1 PRIMARY CONTRIBUTIONS OF THIS STUDY

The research offers the following contributions to the mission and system engineering body of knowledge:

- Insight into current microservice and container performance within context of hard-real-time combat system constraints
- Insight into system design factors affecting hard-real-time performance
- Documented traceability approach from microservices to mission capabilities
- Insight into hard-real-time implementation patterns
- Insight into system engineering based microservice design documentation approach (e.g. SysML/UML)
- Insight into how to assess hard-real-time performance in mission critical systems (e.g., multi-variate analysis)
    - Current microservices designs have been industry based
    - Have not required “hard real-time” assessment due to nature of business (e.g., Netflix, Amazon)

6.2 WIDENING THE SCOPE

Profiling of known companies was conducted to identify similarities of this research to known commercial concerns. Table 8 summarizes challenges that would be relevant to combat systems engineering and this research.

Table 8. Research Applicability to Known Commercial Concerns

COMPANY	PERSONA CONCERN (https://www.cncf.io)	APPLICABILITY TO HARD-REAL-TIME SAFETY CRITICAL DOMAIN
Capital One	Challenges of **resilience** and velocity. **Millions of transactions per day**. Some apps deal with critical functions like fraud detection and credit decisioning; e.g. AI."Now, a team can come to us and we can have them up and running with a basic decisioning app in a fortnight, which before would have taken a whole quarter, if not longer." Deployments increased by several orders of magnitude.	**Directly applicable:** Resilience challenge is directly related to management of combat situations. Other concerns are also of interest but not within research scope.
Netflix	Challenges of **Latency**, Productivity, and Velocity. Netflix developed its own technology stack for interservice communication using HTTP/1.1. For several years, that stack supported the company's stellar growth. But by 2015, there were pain points: Clients for interacting with remote services were often wrapped with handwritten code.	**Directly applicable:** Latency concern is relatable. Productivity and velocity relate to other domain concerns but not within the research scope.
Pinterest	Challenges of Efficiency and Velocity. After eight years in existence, Pinterest had grown into 1,000 microservices and multiple layers of infrastructure and diverse set-up tools and platforms. The first phase involved moving services to Docker containers. Once these services went into production in early 2017, the team began looking at orchestration to help create efficiencies and manage them in a decentralized way. After an evaluation of various solutions, Pinterest went with Kubernetes.	**Not applicable to this research:** However, concerns are related to broader combat system domain concerns.
Spotify	Challenges of Efficiency and Velocity. After eight years in existence, Pinterest had grown into 1,000 microservices and multiple layers of infrastructure and diverse set-up tools and platforms. The first phase involved moving services to Docker containers. Once these services went into production in early 2017, the team began looking at orchestration to help create efficiencies and manage them in a decentralized way. After an evaluation of various solutions, Pinterest went with Kubernetes.	**Not applicable to this research:** However, concerns are related to broader combat system domain concerns.

6.3 SUGGESTIONS FOR FUTURE RESEARCH

The source code, documentation, and analysis scripts for the prototype can be found on GitHub at https://github.com/amurp003/dss-prototype to facilitate repeatability and continued research. The following suggestions for future research can be divided into 2 categories. Technical suggestions are proposed to add more fidelity and progress closer to the objective microservices discussed within (Murphy & Moreland, 2021). Cosmetic suggestions are proposed to increase the usability of the prototype applications.

TECHNICAL

- Service Mesh with mTLS
- Distributed clusters (e.g., sensor management, track management, command and control)
- Add representative gaussian delays to each microservice. Determine and add based upon historical data. This will make the prototype and analysis likely classified for some safety critical systems (e.g., weapon systems).

COSMETIC

- Plot flight data
- Add more flight center points beyond Dulles (IAD) and Richmond (RIC) airports. Consider manual latitude and longitude entry with a variable range to extend from the center point (e.g., 60 NM).
- Add flight motion on the user interface plot

- Add selection from live flights and associated kinematics in DSS calculated responses (e.g., Trial Engage, Weapon Assessment)

REFERENCES

Abbott, D. (2017). *Linux for Embedded and Real-time Applications* (4th Edition ed.). Newnes.

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015, 28-30 Sept. 2015). *Performance Evaluation of Microservices Architectures using Containers* 2015 International Symposium on Network Computing and Applications, Cambridge, MA.

AWS. (2022). *Amazon EC2 Instance Types*. Amazon Web Services. Retrieved June 29 from https://aws.amazon.com/ec2/instance-types/

BCM2711 ARM Peripherals. (2022). https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

Beck, K., Beddle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. https://agilemanifesto.org

Bogner, J., Fritzsch, J., Wagner, S., & Zimmermann, A. (2019). *Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality* 2019 IEEE International Conference on Software Architecture Companion (ICSA-C),

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014, 16 September 2014). *The Reactive Manifesto, v2.0*. Retrieved 25 December from https://www.reactivemanifesto.org

Brown, S. (2019). *Software Architecture for Developer, Volume 2*. https://leanpub.com/visualising-software-architecture

Bruza, M., & Reith, M. (2018). Teaming with Silicon Valley to Enable Multi-Domain Command and Control. International Conference on Cyber Warfare and Security,

Bruza, M. R. (2018). *An Analysis of Multi-Domain Command and Control and the Development of Software Solutions through DevOps Toolsets and Practices* Air Force Institute of Technology].

Coccia, M. (2017). The Fishbone diagram to identify, systemize and analyze the sources of general purpose technologies. *Journal of Social and Adminstrative Sciences*, *4*(4), 291-303. https://doi.org/10.1453/jsas.v4i4.1518

Dobbelaere, P., & Esmaili, K. S. (2017). *Industry paper: Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations*, Association for Computing Machinery, Inc.

DoD. (2019). *DoD Enterprise DevSecOps Reference Design, Version 1.0*. DoD CIO Retrieved from https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf?ver=2019-09-26-115824-583

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2014, 21 July 2014). *An Updated Performance Comparison of Virtual Machines and Linux Containers* 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA.

Fernåndez-Villamor, J. I., Iglesias, C. Á., & Garijo, M. (2010). *Microservices: LIghtweight Service Descriptions for REST Architectural Style* ICAART 2010, Valencia, Spain.

Firesmith, D. (2019, 5 August). *Mission Thread Analysis Using End-to-End Data Flows - Part 1*. Retrieved 5 May from https://insights.sei.cmu.edu/sei_blog/2019/08/mission-thread-analysis-using-end-to-end-data-flows---part-1.html

Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-Native Application. *IEEE Cloud Computing*(September/October 2017), 16-21.

Goodhope, K., Koshy, J., Kreps, J., Narkhede, N., Park, R., Rao, J., & Ye, Y. (2012). Building LinkedIn's Real-time Activity Data Pipeline. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *35*(2), 33-45.

Google. (2020, 20 April). *Pod*. Google. Retrieved 25 April from https://cloud.google.com/kubernetes-engine/docs/concepts/pod

Grant, C., & Osanloo, A. (2014). Understanding, Selecting, and Integrating a Theoretical Framework in Dissertation Research: Creating the Blueprint for Your "House". *Administrative Issues Journal Education Practice and Research*. https://doi.org/10.5929/2014.4.2.9

Humble, J., & Farley, D. (2011). *Continuous Delivery*. Addison-Wesley.

INCOSE. (2015). *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition*. Wiley.

ISO/IEC/IEEE. (2008). 12207-2008 - ISO/IEC/IEEE International Standard - Systems and software engineering -- Software life cycle processes. In: IEEE.

ISO/IEC/IEEE. (2015). ISO/IEC/IEEE 15288, Systems and software engineering — System life cycle processes. In: IEEE.

Janetakis, N. (2017, 2 July). *Virtual Machines vs Docker Containers* YouTube. https://www.youtube.com/watch?v=TvnZTi_gaNc

John, V., & Liu, X. (2017). A Survey of Distributed Message Broker Queues.

Kho Lin, S., Altaf, U., Jayaputera, G., Li, J., Marques, D., Meggyesy, D., Sarwar, S., Sharma, S., Voorsluys, W., Sinnott, R., Novak, A., Nguyen, V., & Pash, K. (2018). *Auto-Scaling a Defence Application across the Cloud Using Docker and Kubernetes* 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),

Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*.

Ledeul, A., Millan, G. S., Savulescu, A., & Styczen, B. (2019, 7-11 October). Data Streaming with Apache Kafka for CERN Supervision, Control, and Data Acquisition System for Radiation and Environmental Protection. International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS), New York, NY, USA.

Lewis, J., & Fowler, M. (2014, 1 March). Microservices. https://martinfowler.com/articles/microservices.html

Li, W., Lemieux, Y., Gao, J., Zhao, Z., & Han, Y. (2019). *Service Mesh: Challenges, State of the Art, and Future Research Opportunities* 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE),

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design, First Edition*. Prentice Hall.

Mayer, B., & Weinreich, R. (2018). *An Approach to Extract the Architecture of Microservice-Based Software Systems* 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE),

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308-320.

Moreland, J. D., Sarkani, S., & Mazzuchi, T. (2014). Service-Oriented Architecture (SOA) Instantiation within a Hard Real-Time, Deterministic Combat System Environment. *INCOSE Systems Engineering*, *17*(3), 264-277. https://doi.org/10.1002/sys.21268

Moreland Jr., J. D. (2013). *Service-Oriented Architecture (SOA) Instantiation Within a Hard Real-Time, Deterministic Combat System Environment* [Dissertation, The George Washington University]. ProQuest.

Murphy, A., & Moreland, J. (2021). Integrating AI Microservices into Hard-Real-Time SoS to Ensure Trustworthiness of Digital Enterprise Using Mission Engineering. *JIDPS*, *25*(1), 38-54. https://doi.org/10.3233/JID-210013

Nikdel, Z., Gao, B., & Neville, S. W. (2017). *DockerSim: Full-stack Simulation of Container-based Software-as-a-Service (SaaS) Cloud Deployments and Environments* 2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM),

OMG. (2015). *OMG Unified Modeling Language (OMG UML), Version 2.5*. https://www.omg.org/spec/UML/2.5/PDF

OpenSky. (2021). *The OpenSky Network API*. Retrieved July 1 from https://openskynetwork.github.io/opensky-api/

OpenTelemetry. (2022). *OpenTelemetry Homepage*. Retrieved 1 July from https://opentelemetry.io

OUSD(R&E). (2020). *Mission Engineering Guide*.

Piraghaj, S. F., Dastjerdi, A. V., Calheiros, R. N., & Buyya, R. (2017). ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers. In *Software - Practice and Experience* (Vol. 47, pp. 505-521): John Wiley and Sons Ltd.

Pratt, J. (2022). ODU CCI Research Environment. In A. Murphy (Ed.), (Discussions between J. Pratt and A. Murphy regarding the CCI capabilities ed.).

Roa, M., Cantrell, W., Cartes, D., & Nelson, M. (2011). *Requirements for deterministic control systems*.

Rodola, G. (2022). *psutil documentation*. Retrieved 28 Sep from https://psutil.readthedocs.io/

Scrucca, L., Fop, M., Murphy, T. B., & Raferty, A. E. (2016). mclust 5: Clustering, Classification and Density Estimation Using Gaussian Finite Mixture Models. *The R Journal*, *8*(1), 289-317.

Sotomayor, J. P., Allala, S. C., Alt, P., Phillips, J., King, T. M., & Clarke, P. J. (2019). *Comparison of Runtime Testing Tools for Microservices* 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC),

Tucker, A. (2022). Deployment of DSS Prototype on ODU CCI Research Environment. In A. Murphy (Ed.), (Discussion between A. Tucker and A. Murphy regarding the CCI deployment ed.).

Walsh, D., & Duffy, M. (2015). The Container Coloring Book. Who's Afraid of the Big Bad Wolf? In: Red Hat.

Wang, R. R. (2011, 1 Mar 2020). Opher Etzion on Four Types of Real-time. http://blog.softwareinsider.org/2011/06/20/mondays-musings-real-time-versus-right-time-and-the-dawn-of-engagement-apps/screen-shot-2011-06-20-at-6-38-45-am

Wei, T., Malhotra, M., Gao, B., Bednar, T., Jacoby, D., & Coady, Y. (2018). No Such thing as a "Free Lunch"? - Systematic Benchmarking of Containers. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),

Wiggins, A. (2017). *The Twelve-Factor App*. Retrieved 1 March from https://12factor.net

Wong, K. C., Woo, K. Z., & Woo, K. H. (2016). Ishikawa Diagram. In *Quality Improvement in Behavioral Health* (pp. 119-132). https://doi.org/10.1007/978-3-319-26209-3_9

Wu, H., Shang, Z., & Wolter, K. (2019). *TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka* 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),