NPS-AM-26-044

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Delivering Resilient Warfighting Capability at the Speed of Relevance

June 2025

Christopher M. McCall, CIV

Thesis Advisors: Jeffrey R. Dunlap, Lecturer
Dr. Robert F. Mortlock, Professor

Department of Defense Management

Naval Postgraduate School

The research presented in this report was supported by the Acquisition Research Program of the Department of Defense Management at the Naval Postgraduate School.

To request defense acquisition research, to become a research sponsor, or to print additional copies of reports, please contact the Acquisition Research Program (ARP) via email, arp@nps.edu or at 831-656-3793.

ABSTRACT

Advancements in technology are transforming how U.S. military systems, especially those in the Navy, are designed, developed, and maintained. In the 20th century, as the private sector led technological innovation with the rise of the internet and personal computing, the Navy increasingly adopted commercial technologies. Post–World War II military systems relied on purpose-built electronics and specialized software (SW) running on unique operating systems. With limited storage and processing power, these systems had to be lean and deterministic. For example, the total storage of a dozen military specification (MILSPEC) devices like the UYH-16 now fits on an $8 Universal Serial Bus (USB) drive. Over time, as memory and processing capabilities expanded, these monolithic SW programs grew in size, incorporating new functions but retaining outdated architectures. This created challenges in transitioning to modern technologies like microservices and advanced hardware. Modernization though costly and complex, is critical to maintaining readiness. Efforts like the unmanned surface vessel (USV), Aegis Virtualization, and Integrated Combat System (ICS) demonstrate progress in adapting more agile, scalable systems and accelerating deployment to the fleet. These initiatives reflect the Navy's commitment to leveraging technological advances effective and efficiently to stay operationally prepared.

THIS PAGE INTENTIONALLY LEFT BLANK

NPS-AM-26-044

ACQUISITION RESEARCH PROGRAM
SPONSORED REPORT SERIES

Delivering Resilient Warfighting Capability at the Speed of Relevance

June 2025

Christopher M. McCall, CIV

Thesis Advisors: Jeffrey R. Dunlap, Lecturer
Dr. Robert F. Mortlock, Professor

Department of Defense Management

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AI	artificial intelligence
API	application programming interface
ATP	Aegis time processors
AWS	Aegis Weapon System
BOM	bill of materials
C2	command and control
CaaS	communications as a service
CBB	capability building block
CDD	capability definition document
CI	computing infrastructure
CI/CD	continuous integration and continuous deployment
CMS	combat management system
COTS	commercial off the shelf
CPU	central processing unit
CRI	container runtime interface
CS	combat system
DDD	domain-driven design
DevOps	development operations
DNS	domain name server
DoD	Department of Defense
DON	Department of the Navy
EDAC	error detection and correction
FAR	Federal Acquisition Regulation
FARS	Federal Acquisition Regulation System
FMES	failure mode and effects analysis
GAO	Government Accountability Office
GDC	gyro data converters
GFE	government furnished equipment
GPS	Global Positioning System
GPS	gigabytes per second

HMI	human-machine interface
HTTP	Hypertext Transfer Protocol
HW	hardware
I/O	input/output
IaaS	infrastructure as a service
IaC	infrastructure as code
ICD	interface control document
ICS	integrated combat system
IP	internet protocol
IWS	Integrated Warfare System
JSON	JavaScript object notation
KPP	key performance parameters
LPTA	lowest price technically acceptable
LUSV	large unmanned surface vessel
MILSPEC	military specification
MOSA	modular open system approach
MOTS	military off the shelf
MPM	major program manager
MTTR	mean time to repair
NASA	National Aeronautics and Space Administration
NoSQL	not only Structured Query Language
NPS	network, processing, and storage
NTDS	Navy Tactical Data System
OE	operating environment
OKR	objective and key result
OS	operating system
OTA	other transaction authority
PaaS	platform as a service
RCM	reliability centered maintenance
REST	representational state transfer
RIS	reduced instruction set
SCS	ship control system

SDN	software defined networking
SE	systems engineering
SI	system integration
SME	subject matter expert
SOW	statement of work
SPS	software production specification
SQL	Structured Query Language
SRP	single responsibility principle
SSDS	ship self-defense system
SW	software
T&E	test and evaluation
TDP	technical data package
TWS	Tomahawk Weapon System
UAV	unmanned aerial vehicle
UI	user interface
UPS	uninterruptable power system
USN	U.S. Navy
USV	unmanned surface vessel
VLP	vertical launch processor
VLS	vertical launch system
VM	virtual machine
VPS	Virtual Pilot Ship
XML	Extensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The U.S. Navy's (USN) combat system (CS) development is hindered by lengthy test, evaluation, and certification processes, due in part to traditional monolithic software (SW) architectures. Monolithic computing integrates all components into a single block of code, simplifying deployment but creating scalability and maintenance challenges. Any modification requires full recompilation, increasing complexity and slowing development cycles (Lewis & Fowler, 2014).

A. SHIFT TO MODERN ARCHITECTURES

Technological advancements have transitioned USN systems from purpose-built military hardware (HW) to commercial-based architectures. Modern microservices provide a modular approach that improves flexibility and scalability (Newman, 2021). By decentralizing SW design, microservices structure applications into independent, loosely coupled services that communicate via APIs or message queues. Unlike monolithic systems, microservices enable independent development, deployment, and scaling, enhancing maintainability and resilience (Lewis & Fowler, 2014).

Each service operates autonomously, allowing targeted scaling and reducing resource inefficiencies. For example, a high-traffic recommendation engine can be scaled separately from a billing system. Additionally, microservices support diverse technology stacks, optimizing performance based on service-specific needs (Nadareishvili et al., 2016).

B. OVERCOMING LEGACY CHALLENGES

Modernizing SW architecture requires understanding existing systems and their original design choices. Monolithic architecture integrates user interfaces, business logic, and data management into a tightly coupled entity, making development and maintenance complex. Minor updates can inadvertently effect unrelated components, leading to increased risk of bugs and slower development cycles (Bass et al., 2021).

Transitioning to microservices mitigates these challenges by breaking down applications into independent services. This approach enhances flexibility, reduces dependencies, and improves fault isolation, ensuring SW evolves efficiently. However, microservices introduce complexities such as communication overhead, latency, and data consistency challenges across distributed systems. Ensuring seamless interactions while maintaining reliability and security requires sophisticated solutions.

C. ENABLING AGILE DEVELOPMENT WITH DEVOPS

Adopting microservices alongside continuous integration/continuous deployment (CI/CD) accelerates upgrades, mitigates risks, and enhances system capabilities. Advanced developmental operations (DevOps) practices—such as automated testing, infrastructure as code, and containerization—facilitate microservices deployment and monitoring (Newman, 2021). Tools like Docker and Kubernetes streamline operations, improving scalability and agility for large, evolving applications.

Despite operational complexities, DevOps fosters automation, collaboration, and continuous improvement. Key practices such as continuous integration, automated testing, and centralized monitoring contribute to faster time-to-market and enhanced SW quality. However, integrating DevOps with microservices also presents challenges, including increased tooling complexity, cultural adaptation, and cybersecurity concerns. Strategic planning and modern development workflows are essential for overcoming these obstacles.

D. THE NAVY'S TRANSITION TO MICROSERVICES

The USN's transition from purpose-built computing HW and SW to modern architecture has enabled the implementation of microservices-based processes. These advancements allow the rapid fielding of SW capabilities within weeks instead of months or years. When fully implemented, these technologies will significantly reduce development, testing, certification, and distribution costs while enhancing operational readiness and responsiveness to emerging threats.

The USN has made the transition from purpose build computing HW and SW to modern architectures which enable the implementation of relevant processes which field granular microservices-based SW in weeks vice months or years. These enabling technologies, when implemented fully, will reduce the cost of development, test, certification, and distribution of capability.

E. RAPID PROTOTYPING OF UNMANNED SURFACE VESSEL COMPUTING INFRASTRUCTURE

In the fall of 2020, NSWC Dahlgren's Computing Infrastructure Group was assigned to develop and deliver prototype equipment capable of supporting Aegis Weapon System and Tomahawk Weapon System elements for the large unmanned surface vessel (LUSV) program. The initial deployment required government furnished equipment (GFE) for LUSV's first platform by late 2023. PEO IWS 80, the Major Program Manager for the LUSV Integrated Combat System (ICS), emphasized the need for modernized computing infrastructure, featuring virtualized combat computer programs and advanced network, processing, and storage (NPS) HW. At the time, TI16, a Federal Acquisition Regulations System (FARS) procurement initiative, was underway to provide NPS to the USN's Surface Forces. However, the HW selected for TI16 was already nearing obsolescence. By the time LUSV fielded its initial GFE in 2023, the TI16-based infrastructure was over eight years old, raising concerns about its long-term viability and technological relevance.

In the modern defense landscape, the need for advanced computing and network capabilities is paramount to ensuring mission success. For the USV program the ability to procure cutting-edge technology quickly, flexibly, and efficiently was critical.

The USV program utilized other transactional authorities (OTAs) for prototypes to enable rapid, flexible, and cost-effective development of innovative solutions. OTAs enabled the government to deliver modern network, storage, and compute HW in MILSPEC cabinetry in 10 months from design to delivery at the first site. This approach immediately closed the technology gap between military and commercial capability and provided a path to field modernized virtualized and containerized microservices-based SW.

Utilizing OTAs for delivery of USV ICS CI HW and infrastructure as a service (IaaS) and Platform as a Service (PaaS) ensured that we could deliver relevant solutions in months vice six to eight years. While use of OTAs limits the developer to the solutions awarded in the OTA SOW, it enables the team to focus all its energy on the design of the assembled components and integration to deliver a workable solution. This OTA-based rapid HW prototyping and delivery is called the Foundry HW Factory, delivering both HW and IaaS to USN surface CSs.

While more traditional acquisition processes would include lengthy product selection processes, complete with often years long assessments, the USV OTA process allows a small development team to focus on "Make Work" and schedule. Knowing that the solutions being delivered were modern and backed by considerable commercial fielding and utilization, the team focused on the key development features, confident that the underlying HW/SW supported system requirements.

F. INTEGRATED COMBAT SYSTEM CONCEPT

The next key enabler to delivering combat capability rapidly is SW re-architecture. This effort is based on an ICS concept. Dr. Alvin Murphy, an engineer at the Naval Surface Warfare Center in Dahlgren, Virginia, in a paper for PEO IWS, detailed an ICS Combat Management System (CMS) Conceptual Reference Model that provides a comprehensive framework for modern naval CSs. The model ensures seamless integration of sensors, weapon systems, communication networks, and decision-making tools to enhance operational effectiveness. ICS is designed to improve situational awareness, optimize resource use, and enable interoperability across different platforms (Murphy, 2022).

A key component, the CMS, serves as the core processing unit, integrating sensor data, assessing threats, and coordinating weapon engagement. It employs AI and machine learning for real-time data fusion, predictive analytics, and decision support, allowing operators to swiftly respond to evolving threats. The system's modularity and scalability facilitate upgrades and adaptation to new technologies, ensuring long-term viability.

The model incorporates cybersecurity measures to protect against cyber threats while maintaining operational resilience. Additionally, the system-of-systems engineering approach enables joint operations across naval, air, and land forces, promoting coordinated defense strategies. Addressing challenges such as resource constraints, human factors, and life cycle management, the ICS-CMS framework establishes a future-proof, adaptable, and highly effective combat system for modern naval environments. Key principles of the ICS model are:

- **Agility:** Emphasizing flexibility to adapt to changing operational and technological environments.
- **Interoperability:** Ensuring seamless communication across platforms and allies.
- **Cost-Effectiveness:** Reducing life cycle costs through modular design and efficient acquisition processes.
- **Scalability:** Supporting diverse mission requirements through adaptable systems.

G. THE FORGE

A key enabler for modern defense SW development is the Forge SW Factory. The Forge is a Department of Defense (DoD) initiative focused on revolutionizing SW development and delivery for defense systems. Spearheaded by the USN, the Forge operates as a SW factory, leveraging modern development practices and technologies to produce high-quality, mission-critical applications with speed and efficiency. The Forge aims to address key challenges in traditional defense SW development, including lengthy delivery cycles, limited flexibility, and challenges in integrating emerging technologies. Its main objective is accelerated development and deployment:

- Using Agile and DevSecOps methodologies, The Forge emphasizes continuous integration, testing, and delivery to reduce the time required to deliver SW to operational platforms.
- Ensuring scalability and adaptability.
- Focusing on producing modular, scalable solutions that can be easily adapted to evolving mission requirements, ensuring long-term relevance and usability.
- Embedding cybersecurity in development.

- Integrating security into every phase of the SW life cycle (DevSecOps), ensuring robust defenses against cyber threats from the outset.
- Providing cross-platform interoperability.
- Building SW with a Modular Open Systems Approach (MOSA) to ensure compatibility across diverse systems and platforms within the USN and joint forces.

H. CONCLUSION

The USN's transition to modern SW architectures, particularly microservices, is critical for enhancing operational efficiency and adaptability. Microservices offer a modular and scalable approach that aligns with evolving technological requirements by enabling service independence, decentralized data management, and faster updates. These benefits enhance system resilience and adaptability, ensuring combat systems can rapidly evolve in response to mission demands. However, adopting microservices introduces challenges, including increased management complexity, security risks, and operational overhead.

To maximize the benefits of microservices, the USN must enhance its DevOps capabilities. Continuous integration, automated deployment, and rapid iteration improve scalability and SW quality. However, successful implementation requires overcoming cultural shifts, tooling complexities, and integration challenges.

Additionally, modern procurement methods like OTAs facilitate faster innovation. The LUSV program exemplifies the benefits of rapid prototyping and virtualized combat SW deployment, ensuring operational readiness against evolving threats.

The ICS and CMS further advance USN combat capabilities. These systems emphasize modularity, real-time data processing, AI-driven decision support, and enhanced cybersecurity. By integrating Agile and DevSecOps methodologies with rapid HW prototyping, the USN ensures its combat systems remain flexible, efficient, and future-ready.

I. REFERENCES

Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley.

Murphy, A. (2022). Integrated Combat System (ICS) Combat Management System (CMS) conceptual reference model (NSWCDD/TR-22/48). Naval Surface Warfare Center Dahlgren Division.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture*. O'Reilly Media.

Newman, S. (2021). *Building microservices: Designing fine-grained systems*. O'Reilly Media

Lewis, J., & Fowler, M. (2014, March 25). *Microservices: A definition of this new architectural term*. Martin Fowler. https://martinfowler.com/articles/microservices.html

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

U.S. Navy (USN) combat system (CS) development efforts require lengthy development, test and evaluation (T&E), and certification processes. The existing USN CS development efforts, T&E, and certification require years to deliver upgraded capability to warfighters. A key driver of this problem is the nature of USN software (SW) design, which delivers large blocks of computer code updated with added functionality over decades. These large monolithic computer programs are single, tightly integrated applications with interdependent components bundled together. While such programs may work well for smaller applications, they can create numerous issues as the system grows in size and complexity. These issues are explored in depth in this paper.

Current systems do not support granular iterative SW development and rapid certification. To pace the threat, the USN must adopt modern microservices implementations in naval CSs supporting continuous integration and continuous deliver/deployment (CI/CD) and rapid upgrade capability. But, while refactoring a monolithic application to a microservices architecture offers many benefits, it also comes with a variety of challenges. The process involves breaking a tightly coupled system into smaller, independently deployable services, which can be complex and risky if not done carefully. Naval systems can overcome transitional risk by using a strangler pattern methodology to incrementally refactor a monolithic system and gradually replacing its components with newer, modular services or systems. In the strangler pattern, a new system is "grown" around or alongside the old one, and over time, the new system takes over, ultimately allowing the old system to be phased out. This approach will allow military systems to transition incrementally, reducing risk and the probability of large-scale disruption while a more modern architecture is achieved, and CI/CD processes are implemented.

Embracing this approach will enable the USN to deliver enhanced combat capabilities faster, more efficiently, and with greater resilience in the face of evolving operational demands.

This capstone project provides a background on monolithic computing architecture, an introduction to microservices architecture, and a comparative analysis between both. It also presents a discussion on transitioning strategies, an introduction to real world applications/case studies, and an outlook on emerging technologies/architecture paradigms to which the USN is migrating.

II. BACKGROUND

Changes in technology development are reshaping the U.S. military and, in particular, the way USN systems are designed, developed, fielded, and maintained. In the late 20th century, as the private sector began to take the lead in technology development and the internet and personal computing became commonplace, a shift toward commercial technologies in the USN became essential.

As Moore's Law, a prediction that the number of transistors in a defined space would double every two years, became reality, advanced electronics and computing devices developed at a rapid pace. This foundational change in technology development transformed the way naval systems are designed and ushered the government from purpose-built military computers and large monolithic blocks of SW to commercial-based hardware (HW) and microservices-based architecture.

Purpose-built electronics and computers, designed to execute highly specialized SW products built upon specialized operating systems (OSs), were a necessary construct of the post–World War II landscape in military systems. Today, the total storage size of a dozen military specification (MILSPEC) devices, like the UHY-16, is equal to what would fit on an $8 USB drive. The limited storage size of legacy military systems meant SW products had to be lean, highly tailored, and deterministic to achieve their design ends. As technology capabilities increased and USN development processes tried to benefit from memory and processing leaps, these purpose-built monolithic SW programs, running on specialized OSs, kept growing in size as they were repackaged over and again.

By the 1990s, a connected, e-commerce driven world, aided by the pace of technology, private companies quickly became the leaders in large-scale computing and data centers, OS development, and high reliability systems serving banking, commerce, and advance engineering needs.

Today, it is estimated that a modern smartphone is millions of times more powerful than the Space Shuttle was. But the Space Shuttle then, like modern military systems now, required high levels of reliability. These levels of reliability have also been adopted by many commercial applications, from modern aircraft avionics to commercial spacecraft outpacing National Aeronautics and Space Administration (NASA). These high reliability systems are built from SW that is more modular, micro-segmented, and code-based. These SW modules are designed to scale on modern data centers/computing to meet the most demanding high reliability applications for industry and, the U.S. military.

Emerging threats worldwide, hastened by technological leaps, require the USN to institute greater speed to capability through CI/CD pipelines coupled with modern microservices oriented architectures, which are able to make rapid warfighting improvements to small pieces of code and deliver them in days rather than years.

The transition from the systems of the past to modern microservices and HW is costly and difficult. The shift must be accomplished effectively to ensure the safety of sailors and assets and efficiently to maintain current readiness levels, all while building the infrastructure to deliver warfighting improvements that outpace the threat posed by advisories of the U.S.

This transition requires institutional change that empowers smaller groups to rapidly deliver incremental change. Before this can happen, secure high-bandwidth communication paths capable of delivering SW and assessing system readiness (both pre- and post-update) across the globe must be in place. The USN's Project Overmatch aims to deliver this multidomain communications requirement. With both modern computing architectures and containerized microservices-based code in place, this secure communications bandwidth can be used to rapidly bring modernized SW capability to the fleet.

While progress is slow, it is still being made in programs like the unmanned surface vessel (USV), Aegis Virtualization, and others, and the USN is trending in the right direction.

Table 1. Modernization Elements and Key Factors

Aspect	Legacy Systems	Modern Systems	Key Transition Factors
HW	Purpose-built military electronics and computers	Commercial off the shelf HW	Rapid advancements in electronics and computing (e.g., Moore's Law)
SW Architecture	Monolithic, highly tailored, deterministic	Modular, microservices-based	Shift to scalable, reusable, and reliable SW modules
Development Speed	Years to deliver upgrades	Days to deploy incremental updates	Adoption of CI/CD pipelines
Storage and Processing	Limited (e.g., MILSPEC devices)	Vast (e.g., modern smartphones, data centers)	Advancements in commercial data storage and computational capacity
Reliability	Critical for military systems, built into specialized HW and SW	Adopted from high reliability commercial applications	Commercial industries driving high reliability SW development (e.g., avionics, commercial spacecraft)
Key Challenges	Specialized OSs, repackaged monolithic SW	Transition complexity, high costs, infrastructure development	Balancing safety and readiness during the transition
Emerging Solutions	Reliance on legacy systems and gradual upgrades	Modern architecture, containerized microservices, secure global communication	Programs like Project Overmatch addressing global SW delivery and readiness assessment
Example Programs	UHY-16	Aegis Virtualization	Demonstrates progress toward modern architecture and systems

A. INCREASING PACE OF TECHNOLOGICAL ADVANCEMENT

Moore's Law, first articulated by Gordon Moore in 1965, is a foundational observation in the field of semiconductor technology and electronics. Moore, a co-founder of Intel, posited that the number of transistors on an integrated circuit would double approximately every 2 years, leading to a corresponding increase in computational power and a decrease in cost per transistor (Moore, 1965). This prediction was based on the rapid advancements in microchip technology observed at the time and served as a benchmark for the pace of technological progress in the semiconductor industry.

The law has proven remarkably prescient, guiding the development of technology for several decades. For instance, the doubling of transistor counts roughly every 2 years has enabled exponential growth in processing power and memory capacity while reducing costs. This progression has facilitated the proliferation of powerful devices like smartphones, personal computers, and sophisticated data centers, reshaping modern life and industry.

However, this pace of change has become increasingly difficult due to physical and technical limitations. As transistors approach atomic scales, challenges such as heat dissipation, quantum effects, and material constraints emerge, complicating efforts to continue the historical pace of advancement (Shalf, 2020).

B. SHIFT IN TECHNOLOGY DEVELOPMENT

The transition from government-led to industry-led technology development marks a significant shift in the innovative landscape, driven by changes in funding, priorities, and the broader economic environment.

a. Government-Led Technology Development

- **Post-War Period:** After World War II, governments, especially in the U.S., played a central role in technology development. Much of this post–WWII technology focused on national defense.
- **Key Innovations:** This era saw the development of foundational technologies such as the internet, GPS, and large-scale computing.

b. Transition to Industry-Led Development

(1) Economic Shifts

In the late 20th century, as global economies grew more interconnected, the private sector began to take the lead in technological development. Advances in home computing, cellular phones, and consumer electronics provided economic incentives to invest in technological development.

(2) Rise of Tech Companies

In this period of technological advance, companies like IBM, Microsoft, and Apple began to dominate technology innovation. Market incentives and soaring profits incentivized these companies to invest heavily in research and advancing miniaturization of electronics components.

(3) Shift in Focus

While governments continued to invest in technology, especially in areas like defense and space, the focus of innovation shifted to consumer-driven markets. The private sector became the primary driver of advancements in computing, telecommunications, and biotechnology (Bresnahan & Trajtenberg, 1995).

c. Current Landscape

(1) Industry Leadership

Today, the technology industry continues to be led by private companies, which are driven by profit to invest heavily in technological innovation. This expanding push to miniaturize solid-state computing and develop consumer products will, for the foreseeable future, perpetuate this industry-led technology development model.

(2) Government's Role

While still engaged in certain areas of technological development, like basic science research and infrastructure, the government's role has shifted toward investment in critical technologies and policy and adapting commercial products to defense needs.

C. INTRODUCTION TO MONOLITHIC COMPUTING ARCHITECTURE

Monolithic computing architecture refers to systems in which the SW is designed as a single, unified block of code. In these architectures, all components—such as the user interface, business logic, and data management—are interconnected, compiled, and executed as one entity. This design was predominant in the early days of computing, especially in mainframe systems, because of its simplicity and performance benefits in a limited-resource environment (Lewis & Fowler, 2014).

This architecture is characterized by a tightly coupled structure. All functionalities are interdependent, which makes deploying applications easier in certain contexts, as everything is bundled into a single executable application. However, this can also be a drawback because any change to one part of the system typically requires recompiling and redeploying the entire application. As systems grow in complexity, monolithic architecture becomes increasingly difficult to maintain and scale. The tightly coupled nature can lead to slower development times and increased risk of bugs, as minor updates may impact unrelated components. Additionally, scaling requires duplicating the entire monolithic application, which can result in the inefficient use of resources (Bass et al., 2021).

Overall, while monolithic architecture offers simplicity in certain contexts, their scalability and maintainability challenges have made them less favored for large-scale applications in modern computing.

In contrast, modern alternatives, such as microservices architectures, have gained popularity due to their modular design, which decouples different functionalities into independent services that can be scaled and maintained individually (Newman, 2021). Nevertheless, monolithic systems are still prevalent in many legacy applications and can be suitable for smaller systems where the benefits of microservices do not outweigh their complexity.

D. INTRODUCTION TO MICROSERVICES COMPUTING ARCHITECTURE

Microservices computing architecture represents a modern, decentralized approach to SW design, where applications are structured as a collection of independent, loosely coupled services. Each service within a microservices architecture is responsible for a specific business functionality and operates as a standalone unit that communicates with other services through lightweight protocols, typically HTTP-based application programming interface (API) or message queues (Newman, 2021). This architecture has gained popularity due to its ability to enhance flexibility, scalability, and maintainability, particularly for large-scale applications.

Unlike traditional monolithic architectures, where all components are integrated in a single codebase, microservices break down an application into smaller, autonomous services. Each service can be developed, deployed, and scaled independently, without affecting the rest of the system. For instance, a team responsible for user authentication can make changes or updates without needing to redeploy the entire application. This independence also makes it easier to isolate and resolve issues, as faults are contained within individual services, rather than spreading across the entire system.

A major advantage of microservices is the ability to scale services independently. In a monolithic architecture, scaling typically involves duplicating the entire application to handle increased demand, leading to resource inefficiencies. However, with microservices, only the specific services that require additional resources are scaled. This selective scaling enables more efficient use of computing resources, as different services can be optimized according to their workloads (Lewis & Fowler, 2014). For example, a high-traffic service like a recommendation engine can be scaled separately from less resource-intensive services like a billing system.

Microservices architectures also allow for greater technological diversity. Since each service is independent, development teams can choose the most appropriate technology stack for the functionality they are building. This flexibility enables the use of different programming languages, databases, and frameworks across various services, optimizing the performance of each based on specific requirements (Nadareishvili et al., 2016). In contrast, monolithic architecture often imposes a single technology stack across the entire application, which can limit the ability to optimize for specific needs.

Despite these advantages, microservices introduce added complexity. As each service operates independently, building consistently efficient and reliable communication between services becomes more challenging. Distributed microservices systems, as they transit data between services face issues such as latency, fault tolerance, and consistency. Furthermore, ensuring data consistency can be difficult since each service may have its own database, requiring distributed transaction management techniques. Implementing proper security across multiple services also becomes more complex than in monolithic systems, as microservices architectures require additional mechanisms to secure communication between services.

Microservices architectures demand robust Development Operations (DevOps) practices to handle the complexities of deployment, monitoring, and troubleshooting in distributed environments. Continuous integration, automated testing, and containerization are commonly used to manage microservices effectively (Newman, 2021). Tools such as Docker and Kubernetes have become standard in deploying and managing microservices-based applications.

In summary, microservices architectures offer significant advantages in terms of flexibility, scalability, and maintainability, making them ideal for large, complex, and rapidly evolving applications. However, they also introduce new challenges in terms of service coordination, security, and operational complexity. Successful adoption of microservices requires careful planning, advanced DevOps practices, and appropriate tooling.

III. LITERATURE REVIEW

This literature review provides an assessment of reference material used in this paper to ensure that the cited material provides well-rounded and well-reasoned information and sufficient synthesis to support the goals of the capstone. The review also presents gaps in the reference material to highlight areas that require additional sources to round out the research needed for preparation of the paper.

In this capstone, it is critical to fully explore the foundations of monolithic and microservices architectures and the features, advantages, and disadvantages of each. Additionally, understanding best practices for transitioning a monolithic USN system to a microservices architecture is necessary to move the USV CS to a modern architecture.

Once SW modernization has occurred, it is essential to understand how the advantages of the new design are transferred to operational units to complete speed to threat distribution of warfighting improvements. Figure 1 provides a simplified outline of the required processes.

Figure 1. Software Modernization Process

The reference material cited in this paper is used to assess the main areas of foundational technologies and monolithic SW design and attributes, modernization approaches, microservices SW design, and deployment/delivery processes to complete the modernization effort.

A. FOUNDATIONAL TECHNOLOGIES

Gordon Moore (1965), a co-founder of Intel, provided data on the accelerating pace of technological advancement in integrated circuit design that offers context on the limitations of early naval systems driving monolithic SW designs.

Shalf (2020), discussed the future of computing beyond Moore's Law, and illustrated the difficulty in maintaining this pace of change as transistors approach atomic scales and challenges such as heat dissipation, quantum effects, and material constraints emerge, complicating efforts to continue the historical pace of advancement.

These two sources give context to the state of technology and its inherent limitations. Understanding monolithic applications and the challenges of modernization are foundational to this paper. Kalske et al. (2018) and Kuryazov et al. (2020) explore the challenges and methodologies associated with transitioning from monolithic to microservice architectures, emphasizing both theoretical frameworks and practical approaches.

Kalske et al. (2018) identify key challenges organizations face during the migration process, focusing on technical, organizational, and cultural barriers. Technically, breaking down monoliths requires reengineering existing systems to ensure loose coupling, scalability, and fault isolation. Organizational challenges include aligning development teams with domain-driven design (DDD) principles to foster a microservices mindset. Cultural shifts involve encouraging collaboration and adapting to continuous delivery and DevOps practices.

Kuryazov et al. (2020) complement these insights by proposing systematic methodologies for decomposing monolithic applications into microservices. They emphasize the importance of domain analysis and boundary identification to create service boundaries that align with business capabilities. The authors also discuss the use of automated tools and clustering algorithms to analyze existing codebases and facilitate the decomposition process. Additionally, they highlight best practices, including implementing APIs, defining communication patterns, and monitoring inter-service dependencies to maintain system integrity.

Both studies converge on the need for thorough planning and evaluation to mitigate risks associated with the transition. The authors of both studies stress the importance of understanding the trade-offs between operational efficiency and the increased complexity introduced by microservices. They also underscore the significance of stakeholder engagement, as the migration impacts not only technical teams but also broader business processes.

Kalske et al. (2018) provide a broad perspective, offering a conceptual framework to address the multifaceted challenges of transitioning, while Kuryazov et al. (2020) present a more hands-on approach, detailing specific techniques and tools for decomposition. Together, these studies provide a holistic view, illustrating that successful migration to microservices hinges on integrating technical innovation with organizational readiness and cultural adaptability.

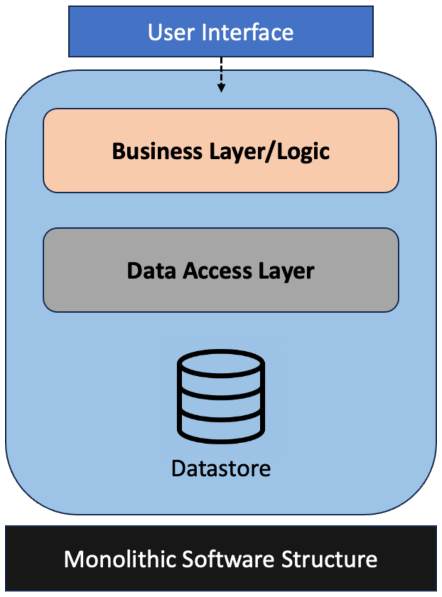


Figure 2. Monolithic Software Design

As naval systems seek to modernize, the USN increasingly looks to the private sector, which has become the primary driver of advancements in computing, telecommunications, and biotechnology (Bresnahan & Trajtenberg, 1995).

B. MODERNIZATION AND TRANSITION

To modernize SW architectures, an in depth understanding of where systems are, why they were constructed as they were, and what the advantages and disadvantages of monolithic design are important. It is also essential to understand how all components, such as the user interface, business logic, and data management, are interconnected, compiled, and executed as one entity (Lewis & Fowler, 2014). An understanding of how the tightly coupled nature can lead to slower development times and increased risk of bugs, as minor updates may impact unrelated components, is equally important (Bass et al., 2021).

To address the disadvantages of monolithic designs, modern alternatives, such as microservices architectures, have gained popularity due to their modular design, which decouples different functionalities into independent services that can be scaled and maintained individually (Newman, 2021). These independent services are depicted in Figure 3 as "Node 1" and "Node 2" independent SW services.

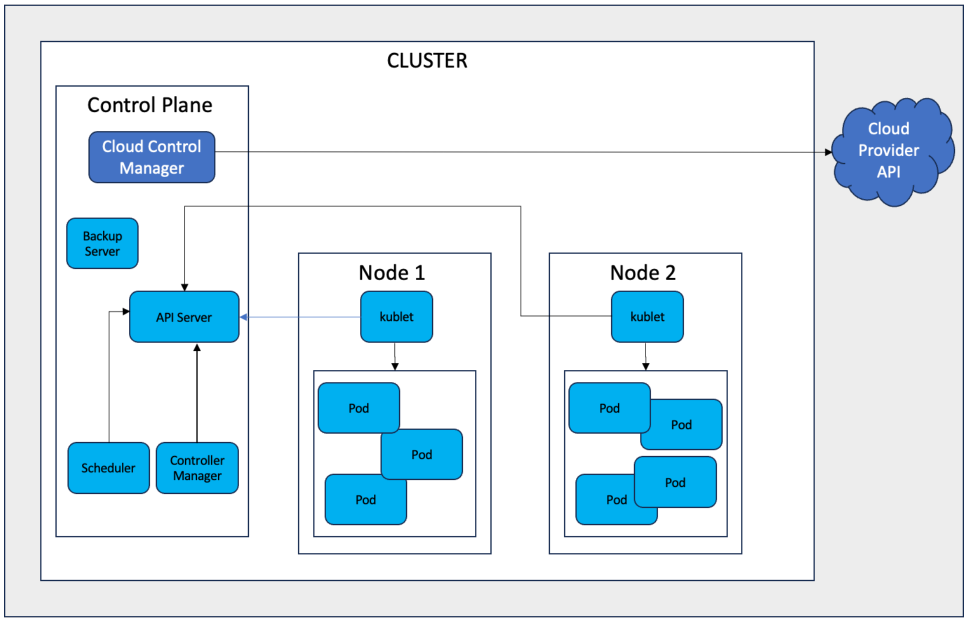


Figure 3. Microservices Software Architecture

Key attributes of microservices architectures are automated deployment and CI/CD. These practices facilitate the efficient management and release of services. Figure 4 illustrates the continuous cycle of the CI/CD Pipeline.

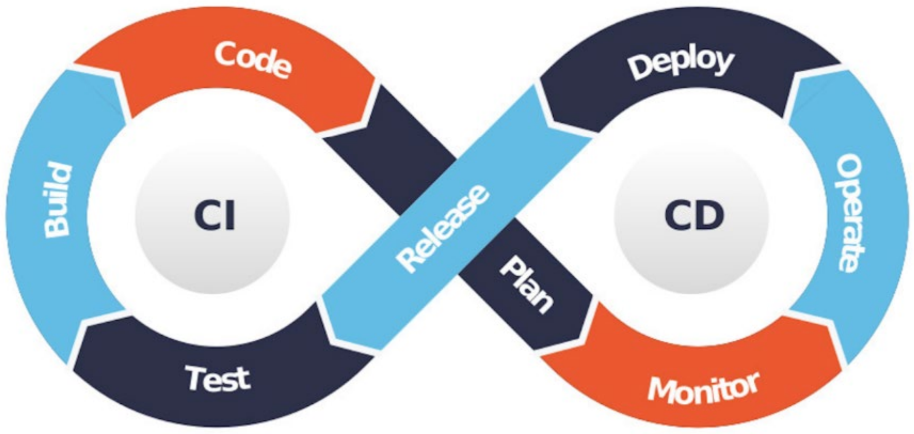


Figure 4. Continuous Integration and Deployment Pipeline

Humble and Farley (2010), Nadareishvili et al. (2016), and Burns et al. (2018) integrate key principles and practices for building scalable, reliable, and efficient SW systems, focusing on continuous delivery, microservices, and distributed system design.

Humble and Farley (2010) establish the foundation of continuous delivery, emphasizing the importance of automating build, test, and deployment pipelines to ensure reliable SW releases. They advocate for frequent, incremental updates to reduce risks and improve system stability. Continuous delivery necessitates robust testing frameworks, infrastructure as code, and automated deployment to enable agility while maintaining high-quality standards.

Nadareishvili et al. (2016) extend these principles to the realm of microservices architecture, highlighting how its decentralized nature aligns with continuous delivery practices. They argue that microservices promote modularity, allowing teams to work independently and release services without affecting the entire system. The authors also underscore the cultural shift required for successful adoption, advocating for practices like

DDD, API-first development, and DevOps. These cultural aspects ensure collaboration and alignment between technical and business goals.

Burns et al. (2018) provide a complementary perspective by focusing on the design patterns and paradigms for scalable, reliable distributed systems. Their work emphasizes containerization, orchestration, and service discovery as critical components of modern distributed systems. They argue that adopting patterns like sidecars, service meshes, and immutable infrastructure helps address challenges like fault tolerance, scalability, and system observability, which are inherent in microservices.

The three groups converge on the need for automation and standardization as the backbone of modern SW systems. Humble and Farley's (2010) emphasis on automation in continuous delivery is reinforced by Burns et al.'s (2018) insights into orchestration and container management, such as Kubernetes, which provide a robust foundation for deploying and managing distributed services. Meanwhile, Nadareishvili et al. (2016) bridge these practices by framing microservices as an architectural enabler that integrates well with continuous delivery pipelines and distributed system design principles.

Together, these sources highlight that the successful implementation of modern SW systems requires a holistic approach that integrates technical, organizational, and cultural factors. Continuous delivery fosters rapid iteration, while microservices enable modular scalability and innovation. Distributed system patterns provide the reliability and performance needed to scale these practices effectively. The synthesis underscores that the intersection of these domains empowers organizations to build resilient systems capable of meeting the demands of dynamic and competitive environments.

C. IDENTIFICATION OF GAPS AND THREADS

The sources collected for this paper provide a synthesized and broad view of both the underlying technologies and commercial practices leveraged to transition older monolithic SW to modern microservices architectures. Throughout these sources, examples are presented on how commercial enterprises are making this transition and the benefits gained from this often expensive and complex move to microservices.

It is much more difficult to find examples of successful transitioning to microservices and modular SW in military applications. As more naval systems are rearchitected, additional studies and post-mortem analysis will become available, but many systems, like the USV program, are in the early stages of this transition. Moving from monolithic blocks of SW, through virtualization efforts, to containerized microservices, and the associated strangler patterns to root out decades-old codebases, are occurring day by day.

This study leans heavily on private sector commercial efforts and the documented best practices to guide the capstone forward while the USV program SW is modernized.

D. SUMMARY AND CONCLUSION

Just as the world transitioned from government-led development of technology to industry-led development, the USN will depend on commercial tools and practices to modernize its SW architectures. The challenges of a less flexible system in government industry and unique applications will become apparent as more systems modernize.

It is incumbent on the USN to leverage the agility and best practices of commercial industry as it modernizes its way out of decades-long stovepipes and legacy practices. So, while there is much more data and reference material on commercial initiatives, it is fitting for this capstone, and military modernization, to bend to the greatest extent possible to these commercial practices.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. MONLITHIC COMPUTING ARCHITECTURE

Monolithic computing SW refers to a design architecture where a SW system is built as a single, indivisible unit. In this architecture, the entire application is typically compiled into a single executable or binary file. This contrasts with modular or microservices architectures, where the SW is divided into discrete, loosely coupled components. Figure 5 represents a simple view of a monolithic computing architecture where all components of the end use are integrated into a single SW module or executable.

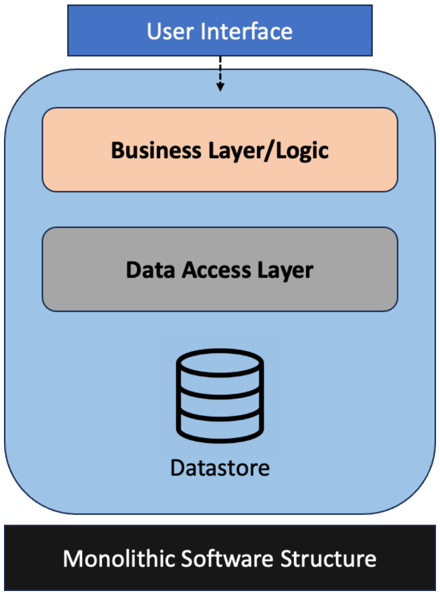


Figure 5. Simplified Monolithic Computing Architecture

A. KEY CHARACTERISTICS

1. Single Codebase

A single codebase refers to a SW development practice where the entire application's source code is maintained in one unified repository or structure. This means that all components—such as the user interface, backend logic, and data access layers—are housed in the same location, often forming the basis of monolithic architectures (Fowler & Lewis, 2014). In this setup, developers work within the same code repository, and all changes or updates are applied uniformly across the system. A single codebase can simplify version control and deployment processes, especially in smaller applications, as there is only one location to manage, compile, and deploy the SW (Dragoni et al., 2017).

One key benefit of a single codebase is consistency. Since all components reside in the same repository, it ensures that developers work with the same version of the code, reducing the risk of fragmentation and incompatibility. This also facilitates simpler builds and deployments, as there is only one application to manage. However, as the system scales, maintaining a single codebase can become problematic. Large, monolithic applications become cumbersome to update, as changes in one part of the codebase can affect unrelated areas, making the development process slower and more error-prone (Newman, 2021).

Furthermore, a single codebase limits the ability to adopt multiple technologies and languages within the same application. In contrast to microservices architectures, where different services can be employed to join SW products developed in different teams, a single codebase typically requires uniformity in programming languages, frameworks, and libraries which greatly reduces flexibility.

In summary, a single codebase is efficient for smaller applications, but as the application grows in complexity, this approach can introduce scalability and maintainability challenges. In a monolithic architecture the components of the application, like databases, graphic user interfaces, and data management layers, are integrated, tested, and deployed as a single codebase. This codebase is compiled into one executable or binary file.

2. Tight Coupling

Components within a monolithic system are tightly coupled. This means changing one part of the end application often drives changes, regression, to other parts. This tight coupling can lead to complexities in development and maintenance, as modifications can have widespread effects.

Tight coupling in monolithic architecture refers to the close interdependence between the various components of a system, such as the user interface, business logic, and data access layers. In a tightly coupled system, changes made to one part of the codebase often require adjustments in other parts, as the components are not designed to function independently. This creates a situation where all parts of the system are highly interconnected, and altering a single feature may require recompiling and redeploying the entire application (Newman, 2021).

Tight coupling is a defining characteristic of monolithic systems, especially in early computing architectures where resource limitations and design simplicity favored single unit systems (Fowler & Lewis, 2014). In this architecture, the components share the same memory space and are often designed to rely on internal calls to each other. While this can be beneficial for performance, as communication within the system is fast, it also introduces challenges, especially as the system grows in complexity.

One of the primary disadvantages of tight coupling is the difficulty in maintaining and scaling the system. Since all components are interconnected, even small changes to a single module may inadvertently affect other parts of the system. This interdependence can lead to longer development cycles and increased risk of introducing bugs or errors during updates (Dragoni et al., 2017). Additionally, as the system grows, it becomes harder to isolate issues and fix bugs without impacting other functionalities.

Tight coupling also limits the scalability of a monolithic system. In modern computing environments, applications often need to scale, or spawn additional instantiations of a service to handle increased user demand. With a monolithic system scaling usually causes a replication of the application, including components that may not need additional resources. Additionally, this replication causes ripple effects as these secondary instantiations drive the need to synchronize data between them and development of additional code. This leads to inefficient resource utilization and can result in performance bottlenecks.

Moreover, tight coupling restricts flexibility in adopting new technologies. In a tightly coupled system, all components typically need to use the same programming language, frameworks, and databases, which can prevent teams from leveraging more suitable technologies for specific tasks (Nadareishvili et al., 2016). This technological uniformity can limit the system's adaptability and make it harder to adopt modern practices.

In summary, tight coupling in monolithic architecture creates significant challenges in terms of maintenance, scalability, and flexibility. While this approach might be simpler for small systems, it becomes increasingly problematic as the system grows in complexity.

3. Single Deployment Unit

A single deployable unit in a monolithic architecture refers to an application where all components—such as the user interface, business logic, and data access layers—are packaged together and deployed as one cohesive entity. This means that the entire application is built, tested, and released as a single artifact, simplifying the deployment process since there is only one unit to manage (Newman, 2021). In such an architecture, the tight integration of components ensures consistency across the application, as all parts are developed and updated simultaneously.

This approach can be advantageous in terms of initial development speed and simplicity, especially for smaller applications or teams. With a single deployable unit, there is less complexity in version control and continuous integration pipelines because there are fewer moving parts (Richards, 2015). However, this model also has significant drawbacks. As the application grows, even small changes require the entire system to be redeployed, which can lead to longer downtime and increased risk of introducing bugs into unrelated areas (Fowler & Lewis, 2014).

Furthermore, the single deployable unit makes it challenging to scale specific parts of the application. As the entire system must be replicated to increase capacity, resources are often wasted on components that do not need scaling (Bass et al., 2021). Thus, while this model may be manageable for smaller applications, it can become a hindrance in large-scale systems.

In summary, while a single deployable unit in a monolithic architecture simplifies deployment and ensures component compatibility, it poses challenges in maintenance, scalability, and rapid iteration as applications become more complex.

4. Performance Considerations

Monolithic systems benefit from performance efficiencies because all components are compiled and executed together. This construct often leads to challenges in military applications as the application becomes too large or complex, affecting performance, scalability, and drives a magnitude of T&E requirements to assess regression of the monolithic application from the smallest change or SW update.

Performance considerations in a monolithic architecture are shaped by its tightly coupled, unified structure, where all components of an application—such as the user interface, business logic, and data access layers—operate within a single codebase. This tight coupling can offer performance benefits in certain contexts, particularly due to the reduced overhead in communication between components. Since all parts of the application share the same memory space and execute within the same process, internal calls between components are typically faster compared to distributed systems, where services may need to communicate over a network (Fowler & Lewis, 2014).

However, as a monolithic system grows in complexity, performance issues can arise. One significant challenge is the scalability of the system. In a monolithic architecture, scaling often requires replicating the entire application, even if only a specific part of the system, such as the business logic or data access, needs additional resources. This can lead to inefficient resource utilization, as components that do not require scaling are duplicated unnecessarily (Dragoni et al., 2017).

Another performance consideration is the risk of bottlenecks. A failure or slowdown in one part of the monolithic application can impact the entire system, as all components are closely intertwined. Furthermore, as the codebase grows larger, build and deployment times can increase, which can hinder continuous delivery efforts (Newman, 2021).

Monolithic architecture can also struggle with maintainability over time, which may indirectly affect performance. As more features and functionalities are added, the system becomes harder to optimize, making it more difficult to ensure that all parts of the application perform efficiently.

In summary, while monolithic architecture can offer performance benefits for smaller applications, their scalability and bottleneck issues make them less ideal for larger, more complex systems.

5. Development Simplicity

Development simplicity in a monolithic architecture is often cited as one of its key advantages, especially for small to medium-sized applications. Monolithic architecture consolidates all components—such as the user interface (UI), business logic, and data access layers—into a single, unified codebase. This centralized structure simplifies the development process because everything is in one place, making it easier for developers to understand the entire application (Fowler & Lewis, 2014). In contrast to more complex architectures like microservices, where various services operate independently, monolithic systems allow developers to build, test, and deploy the entire application as a single unit.

One of the primary benefits of this simplicity is the straightforwardness of the development environment. With all the code in a single repository, there are fewer dependencies to manage, and developers can work without worrying about the communication between distributed services. This centralization reduces the need for complex configuration and infrastructure, making it easier to set up a development environment, especially for small teams (Bass et al., 2021). Additionally, testing is simpler in monolithic architecture because tests can cover the entire system in one go, without needing to mock or simulate multiple services.

Another aspect of development simplicity in monolithic architectures is the ease of deploying the system. With a single deployable unit, developers can push updates to production with one build and deployment process, ensuring that all changes are applied consistently across the system (Newman, 2021). This consistency reduces the complexity of versioning, deployment scripts, and rollback strategies, which are often more intricate in distributed systems.

However, the simplicity of monolithic architecture tends to decrease as the application grows. As more features and modules are added, the codebase becomes more complex and harder to manage. Even though the development of smaller applications is simplified, larger monolithic systems often face difficulties with maintainability and scalability. When a developer changes one part of the system, it can have unintended consequences elsewhere, leading to longer testing and debugging cycles. As a result, the initial simplicity of monolithic architecture can give way to complexity over time (Dragoni et al., 2017).

Monolithic architecture offers significant development simplicity, particularly for smaller applications, due to their unified codebase, simplified deployment, and reduced dependencies. However, this simplicity can become a limitation as the system grows in size and complexity, leading to challenges in maintainability and scalability.

B. ADVANTAGES

1. Ease of Development

In a monolithic architecture there are significant advantages in ease of development. This design simplifies the development process by consolidating all components into one application. The main advantage of this approach is the reduced complexity of integrating multiple services or components, thereby increasing application performance and reducing latency.

One of the advantages of monolithic development is managing a single codebase, across developers, making it easier to manage dependencies and version control. Generally, developers "check out" portions of the unified code while it is in revision/update. Furthermore, a monolithic approach typically requires fewer coordination efforts between teams, as changes are made in a single repository rather than across distributed services (Fowler, 2018).

While these benefits can streamline the initial development phase, monolithic architectures become cumbersome as the application grows, often through years of update and inefficient practices of documenting updates/changes over years of growth, potentially leading to challenges in maintainability and deployment. Nonetheless, for many projects, especially smaller ones or those in their early stages, the monolithic approach offers a more straightforward and efficient development process.

2. Performance

Monolithic architecture provides several performance advantages due to its unified structure and lack of inter-service communication overhead. In a monolithic application, all components—such as the UI, business logic, and data access layers—are encapsulated within a single codebase and runtime environment (Fowler, 2018). This integration leads to streamlined execution processes and minimizes the latency associated with inter-service communication often encountered in microservices architectures.

Generally, monolithic architectures benefit from simplified data transfer and management within the application. Since all components are part of a single application, transactions can be managed more effectively within a single context. This centralized approach enables more efficient resource management and can lead to improved application performance.

Another performance advantage is the ease of optimization. Developers can focus on optimizing a single application rather than multiple services, allowing for more targeted performance improvements and efficient use of resources. Additionally, monolithic applications often leverage a single database, which can streamline data access patterns and reduce the performance overhead associated with managing multiple databases (Fowler, 2018).

The monolithic architecture's unified structure, in-memory communication, and simplified transaction management contribute to its performance advantages, particularly in terms of reduced latency and more efficient resource utilization.

3. Simplicity in Deployment

Monolithic architecture offers notable advantages in deployment simplicity due to its integrated and unified structure. In a monolithic system, the entire application is packaged into a single executable or deployable unit. This consolidated approach simplifies the deployment process, reducing the complexity often associated with managing multiple services and components Fowler, 2018).

One significant advantage of monolithic SW architecture is the ease of deployment. Since all the functional pieces or modules are contained within one codebase, there is no need to manage multiple SW parts and their dependencies or coordinate installations across different environments. Deployment of this single unit simplifies set up and maintenance.

Another benefit of monolithic applications is the reduced overhead in managing deployment documentation. In monolithic architecture, deploying updates or patches involves releasing a single artifact. This contrasts with microservices, where multiple services may require individual updates and deployments, increasing the risk of inconsistencies and deployment errors. The monolithic approach ensures that all components are updated simultaneously, maintaining consistency across the application Fowler, 2018).

Additionally, monitoring and logging application performance are simplified in monolithic systems. Since the entire application operates as one unit, monitoring tools and logging mechanisms can be more centrally and easily managed. This often provides a unified view of the application's performance and behavior.

In summary, the monolithic architecture's simplicity in deployment is characterized by its unified deployment unit, ease of configuration, reduced management overhead, and centralized monitoring. These factors contribute to efficiency.

C. DISADVANTAGES

1. Scalability Issues

Monolithic architecture, despite its initial simplicity and ease of development, faces significant scalability issues as applications grow. In a monolithic system, all components are integrated into a single codebase and executed within a single process or server Fowler, 2018). This design can lead to several challenges when scaling the application to meet increased demand.

A drawback to developing and deploying monolithic SW applications is the difficulty in scaling individual components. In a monolithic architecture, the entire application must be scaled together, even if only specific parts require additional resources. For example, if a particular component experiences high load the entire application must be scaled to handle this increased load. This often results in higher costs and complexity as the application grows.

Additionally, the single codebase approach can lead to performance bottlenecks. As the application's size and complexity increases, the codebase very often becomes unwieldy. Over years of updates, without modernization of the code structure, monolithic SW becomes difficult to optimize and manage. Performance issues in one part of the application can affect the entire system, leading to decreased overall efficiency and responsiveness.

Another challenge is the impact on deployment and maintenance. Scaling a monolithic application often requires deploying the entire application, and any updates or changes need to be tested and deployed as a whole, increasing the risk of introducing regression and unintended defects in the entire application.

In summary, while monolithic architecture offers advantages in terms of initial development and deployment simplicity, it faces significant scalability issues. The need to scale the entire application together, potential performance bottlenecks, complex deployment and maintenance processes, and coordination challenges all contribute to the difficulties in scaling monolithic systems effectively.

2. Maintenance Complexity

As the application grows, over the course of years and numerous update cycles maintaining and updating a monolithic codebase becomes increasingly complex. This complexity often leads to issues when modifying or adding new features.

Maintenance complexity in a monolithic architecture arises from its unified and interdependent structure, where all application components are integrated into a single codebase. As the application grows, managing and modifying this extensive codebase can become increasingly challenging (Fowler, 2018).

One significant issue is the risk of introducing bugs when changes are made. Since all components of the singular application are tightly coupled, changes in one part of the application can inadvertently affect other parts, leading to unintended SW regression and increased complexity of debugging efforts and tools. This interconnectedness can make isolating and fixing issues more difficult compared to modular architectures where components are more decoupled.

Another challenge is the increased complexity of deploying updates. Updating a monolithic application often requires redeploying the entire system, which can be time-consuming and prone to implementation errors. This SW design approach means that the smallest updates necessitate comprehensive testing to determine if the update achieved the desires outcome free of unintended regression. This lack of granularity drives up the scope, costs, and time required to validate performance and ensure the stability of the monolithic application.

A monolithic architecture, in large part, prevents the use of distributed development teams. As the application codebase grows, coordinating multiple developers working on different functions/features can become difficult. This leads to integration conflicts, slower development cycles, and validation test events which are significantly larger in scope.

3. Limited Flexibility

Monolithic architecture, while straightforward in its design, often faces significant flexibility limitations. In this architectural model, all components and functionalities are encapsulated within a single codebase, which can restrict the ability to adapt and evolve the application in response to changing requirements or technological advancements (Fowler, 2018).

One major limitation is the difficulty in implementing new features or modifying existing ones without affecting the entire application. With all the components of a monolithic application tightly coupled, any change in one part of the system may bring performance regression elsewhere. As years pass this tight coupling drives an increasing number of interdependencies which makes it increasingly more challenging to introduce new technologies or frameworks. Consequently, the pace of innovation can be slowed as developers must ensure that modifications do not compromise the stability of the entire system.

Additionally, the monolithic structure can hinder the adoption of modern development practices such as CI/CD. The deployment of monolithic applications usually involves releasing the entire application, even when the smallest updates are made. This prohibits rapid, incremental updates, and reduces the agility of the development and delivery process. This inhibitive paradigm greatly limits the ease in which new features or fixes can be quickly delivered and increases costs.

In summary, monolithic architecture's flexibility limitations arise from its tight coupling of components, challenges in adopting new technologies or practices, and inefficiencies in scaling. These factors can restrict an application's adaptability and hinder its ability to evolve in a dynamic technological landscape.

D. CONCLUSION

Monolithic computing SW is characterized by its unified structure where all application components—UI, business logic, and data access layers—are integrated into a single codebase (Fowler & Lewis, 2014). A monolithic architecture often simplifies initial development and deployment. Another benefit is a more easily managed and centralized version control. A single codebase ensures that all developers work with the same version of the application, reducing fragmentation and incompatibility issues (Bass et al., 2021). However, as applications grow, maintaining a monolithic codebase can become cumbersome. Changes in one area can impact others, making updates more error-prone and time-consuming (Newman, 2021).

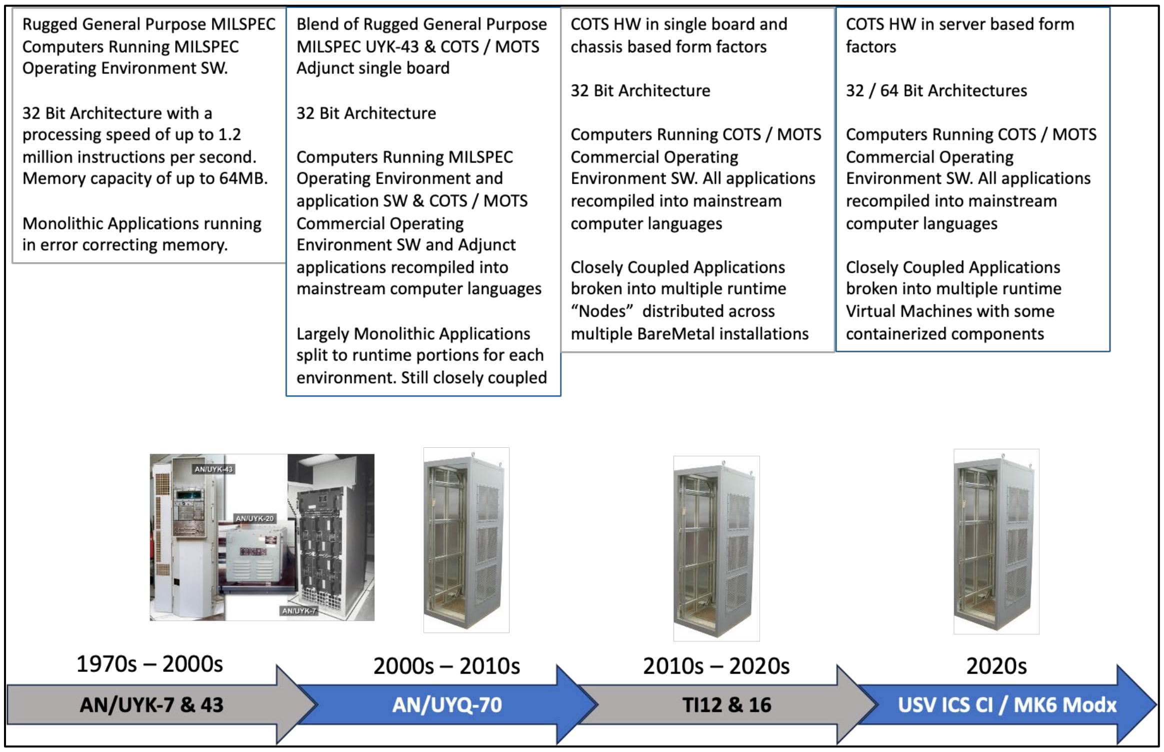
Tight coupling is another defining characteristic of monolithic systems. Components within these systems are interdependent and the smallest updates can necessitate changes across the entire application, complicating maintenance and scaling. Additionally, even minor updates can drive the need for extensive testing and redeployment, driving costs and slowing the pace of improvement.

Despite these challenges, monolithic architecture offers simplicity in deployment. Since the entire application is packaged as a single deployable unit, managing updates and configuration is more straightforward (Newman, 2021). As the application grows, after years of addition/update, the initial simplicity gained in monolithic architecture diminishes adding difficulties in managing large codebases. In summary, while monolithic SW provides development ease and deployment simplicity, it can struggle with scalability, maintenance complexity, and flexibility as applications evolve.

E. CASE STUDY, COMPUTNG INFRASTRUCTURE: A CORE ENABLER OF NAVAL COMBAT SYSTEMS SOFTWARE MODERNIZATION

This case study will explore how naval CS CI enabled decades of USN power projection and global stability throughout the Cold War era. And, while the purpose-built computers and computing systems utilized prior to transition to commercial technologies served the USN well, they also were an inhibitor of the implementation of modern SW architectures and Pace-of-Threat deployment of CS improvements.

In this case study, four CS computing architecture capability stages will be presented with their inherent capabilities, and the limitations they placed on SW modernization. Figure 6 provides an overview of each stage in CSs computing environments.


Rugged General Purpose MILSPEC Computers Running MILSPEC Operating Environment SW.

32 Bit Architecture with a processing speed of up to 1.2 million instructions per second. Memory capacity of up to 64MB.

Monolithic Applications running in error correcting memory.

Blend of Rugged General Purpose MILSPEC UYK-43 & COTS / MOTS Adjunct single board

32 Bit Architecture

Computers Running MILSPEC Operating Environment and application SW & COTS / MOTS Commercial Operating Environment SW and Adjunct applications recompiled into mainstream computer languages

Largely Monolithic Applications split to runtime portions for each environment. Still closely coupled

COTS HW in single board and chassis based form factors

32 Bit Architecture

Computers Running COTS / MOTS Commercial Operating Environment SW. All applications recompiled into mainstream computer languages

Closely Coupled Applications broken into multiple runtime "Nodes" distributed across multiple BareMetal installations

COTS HW in server based form factors

32 / 64 Bit Architectures

Computers Running COTS / MOTS Commercial Operating Environment SW. All applications recompiled into mainstream computer languages

Closely Coupled Applications broken into multiple runtime Virtual Machines with some containerized components

Figure 6. Stages of Combat System Computing Environments Evolution

1. Stage 1: AN/UYK-7 and AN/UYK-43

The AN/UYK-7 and AN/UYK-43 computers were a critical 32-bit computer system developed for the USN, primarily used for tactical data processing in systems such as the Navy Tactical Data System (NTDS) and Aegis CSs. The UYK-7 was introduced in the early 1970s and marked a significant advancement over its predecessors in terms of processing power, memory capacity, and input/output (I/O) capabilities.

The AN/UYK-43 was the USN's standard 32-bit computer system for general-purpose applications from the early 1980s through the 2000s, replacing the AN/UYK-7. The UYK-43 was highly reliable, flexible, and brought significant performance improvements over its predecessors. The AN/UYK-43 was widely used in shipboard and submarine systems for tactical data processing, command and control (C2) processing, sonar, radar control, and other mission-critical applications.

a. AN/UYK-7 Capabilities

(1) Processing Power

The AN/UYK-7 could process up to 725,000 instructions per second:

- **Architecture:** The AN/UYK-7 utilized a 32-bit architecture, and featured a 32-bit word length, allowing for greater data precision and faster processing of larger numbers compared to older systems.
- **Clock Speed:** The system operated at a clock speed of approximately 2.5 MHz, typical for military-grade systems of that era.
- **Instruction Set:** The instruction set included both fixed-point and floating-point arithmetic, making the system versatile for both general-purpose and real-time operations.

(2) Memory

The AN/UYK-7 initially came with 64K words (65,536 words) of magnetic core memory, which equals 256 KB in modern terms (based on the 32-bit word size). This could be expanded to 1 MB with additional memory modules.

(3) I/O and Networking

The UYK-7 had a flexible and scalable I/O system, supporting 16 to 32 independent I/O channels. These channels allowed the system to interface with numerous external devices, such as C2, radar systems, sonar, and missile control.

(4) OS and SW

The AN/UYK-7 ran NTDS SW, as well as the Aegis Tactical Operating Environment which were developed for real-time combat data processing. This SW allows the system to manage data from sensors, track objects, and guide weapons in real-time.

(5) Applications

The AN/UYK-7 was widely used in various naval applications, including the NTDS. It processed real-time combat data from sensors, including radar and sonar, and provided data to weapons systems for missile control and engagement. Aegis Combat System: Early Aegis Weapon System (AWS) baselines used the UYK-7 to process radar, C2, and weapons control data before Aegis transitioning to the AN/UYK-43

b. AN/UYK-43 Capabilities

(1) Processing Power

The AN/UYK-43 was based on a 32-bit architecture, similar to its predecessor, the AN/UYK-7, but with a more modern design that enhanced performance and scalability. Like the UYK-7 the UYK-43 computer utilized a 32-bit word length and supported both fixed-point and floating-point arithmetic. This architecture allowed for more efficient processing of complex computations (Harris, 1991).

(2) Memory

The AN/UYK-43 initially was configured with 2 MB of core memory, but later versions supported up to 64 MB of semiconductor memory. The move to semiconductor memory increased speed, reduced power consumption, and provided greater reliability compared to the AN/UYK-7. The memory used error detection and correction (EDAC) techniques to ensure data integrity, which was critical for military operations.

(3) Modularity

The AN/UYK-43 was designed with a high degree of modularity. Its design allowed for the use of multiple central processing units (CPUs) and the addition of specialized co-processors (input/output processors [IOPs]), and multiple memory module configurations.

(4) I/O and Networking

The AN/UYK-43 had a highly flexible I/O subsystem, supporting a variety of external peripherals and interfaces. The system supported up to 64 I/O channels, which allowed it to communicate simultaneously with multiple external devices.

(5) Processing Modules and Multiprocessing

The AN/UYK-43 supported multiprocessing, enabling multiple processors to work simultaneously. This feature allowed for parallel processing of data providing much higher performance than previous USN computers.

(6) OS and SW

The UYK-43 typically ran NTDS SW as well as the Aegis Tactical Executive System supporting AWS applications.

(7) Applications

Combat Direction Systems: The UYK-43 was widely deployed in combat direction systems on aircraft carriers, cruisers, and destroyers.

c. Limiting Factors

The UYK computers were purpose build MILSPEC computers designed to field on USN vessels. As such they needed to be limited in size and power consumption. These design constraints are limited to performance capabilities for processing and memory/storage. Within these limitations specialized operating environment (OE) SW was needed.

2. Stage 2: AN/UYQ-70

The AN/UYQ-70 (commonly referred to as the UYQ-70) is an advanced, modular display and processing system developed for the USN to replace older systems like the AN/UYK-43 and AN/UYK-44. It became the standard USN combat console and processing system during the late 1990s and early 2000s. The Q-70 program, and all those that have followed, sought to harness commercially relevant HW and package/deliver it to the fleet more rapidly than previous MILSPEC developments.

a. Capabilities

The UYQ-70 was designed to be highly modular and scalable, making it adaptable for different platforms and missions. The system could be configured in various ways, such as single-processor or multi-processor configurations, to meet specific requirements.

It was built using commercial off the shelf (COTS) HW components. The use of COTS technology also meant the system could benefit from rapid advancements in commercial computing technologies.

UYQ-70 processing, within the AWS, was used to run adjunct computer programs along UYK-43 computers transitioning portions of the application to a more modern and flexible computer language and commercial/modified commercial OSs like HP-UX and HP-RT.

b. Processing Power

The UYQ-70 featured multi-processor capability, which allowed it to handle multiple tasks simultaneously. It could support a variety of different processors packaged on single board computers or symmetric multi-processor arrays. The system's ability to integrate modern processing units meant it offered significant improvements in speed and computational power over older systems like the AN/UYK-43.

c. I/O and Networking

The UYQ-70 had robust networking and data communication capabilities. It could interface with various external sensors and systems, including C2, radar, sonar, and electronic warfare systems. It supported a wide range of I/O interfaces, including Ethernet, fiber optic data transmission, and serial connections, allowing for flexibility in integrating with onboard systems.

d. OS and SW

The UYQ-70 typically ran Unix-based OSs, which allowed for reliable and secure multitasking in real-time environments. The system was designed to support real-time processing, which was crucial for tasks like missile guidance, threat tracking, and electronic warfare.

e. Modularity

One of the key features of the UYQ-70 was its upgradability. Because it used COTS components, the system could easily be upgraded with new HW and SW as commercial technology advanced.

f. Applications

The UYQ-70 is used in a wide range of naval applications, including:

- **Aegis CS:** The UYQ-70 is an integral part of the Aegis system, providing real-time radar and weapons control data to operators.
- **Submarine CSs:** It is also used in submarine combat control systems, where it processes sonar data, weapons control, and navigation information.
- **C2 Systems:** Shore-based installations and mobile command centers use the UYQ-70 for mission planning, situational awareness, and tactical data coordination.

The Q-70 era was an important step for modernization of USN systems. In this period portions of the CS applications were recompiled into commercial-based languages, commercial HW and OSs executed these applications.

g. *Limiting Factors*

(1) HW

A limiting factor in this period was the continued reliance on MILSPEC UYK computers. While COTS/MOTS HW provided vastly superior performance from processing, memory, and storage these applications still ran as bare metal machines where an entire single board computer was dedicated to a single application. The use of these COTS products opened the door for use of commercial operating system SW.

(2) OS and Environment SW

The continued use of UYK computers relied on the same specialized Reduced Instruction Set (RIS) OE SW. The addition of commercial UNIX-based OS, while an important step, continued the limits of closely coupled monolithic applications running "tied" to bare metal locations providing only N+1 resiliency.

3. Stage 3: TI12 and TI16

TI12 and TI16 built upon successes of the Q70 program establishing a notional 4-year interval between revamped and modernized COTS HW for naval applications. One major advancement in this period was the elimination of MILSPEC computers and OE/OS.

a. *Capabilities*

Like UYQ-70, Tis were designed to be highly modular and scalable, making it adaptable for different platforms and missions. Efforts were made to deliver capability which supported many applications and delivered performance characteristics which met the most stringent requirements to make it adaptable to end users. TIs were built using all COTS HW components. The use of COTS technology continued the benefits of rapid advancements in commercial computing technologies.

b. *Processing Power*

TIs featured multi-processor capability, packaged in chassis like IBM Blade Centers and ATCA form factors. This allowed it to handle multiple tasks simultaneously. It could support a variety of different processors packaged on single board computers within these processing centers. The system's ability to integrate modern processing units meant it offered significant improvements of the previous generation of COTS computing infrastructure

c. I/O and Networking

TIs also leveraged industry trends in networking bringing improved network switching and network speeds.

d. OS and SW

TIs transitioned from UNIX-based to Linux-based OS like Red Hat Enterprise Linux (RHEL) and RedHawk adaptations for real-time processing.

e. Modularity

TIs continued the modular approach delivering a series of cabinets designed to provide NPS functions. TIs, built on COTS components, were upgradable with new HW and SW as commercial technology advanced.

f. Applications

TIs HW continued in many of the same applications of previous generations of HW across the surface USN.

g. Limiting Factors

(1) HW

While TI equipment was "off the shelf" capable of advancing from bare metal implementations for well over a decade the SW continued to be deployed as bare metal. In the last couple of years, these systems have been making efforts to virtualize, thereby taking advantage of all the resources delivered.

(2) OS and Environment SW

The move to RHEL based OS, offering the ability to jump to more modern SW deployments continued the limits of closely coupled monolithic applications running "tied" to bare metal locations providing only N+1 resiliency.

4. Stage 4: Unmanned Surface Vessel Integrated Combat System Computing Infrastructure and MK6 ModX

USV ICS CI, based upon HW and SW products under development for the Enterprise ICS and IWS X solutions, is the first fully modern ICS CI fielded in a tactical AWS. This rapidly developed and fielded ICS brought virtualized and containerized applications and an IaaS environment to prototype vessels for the USV program, opening the door to complete modernization of naval CS and their SW.

a. Capabilities

Built on modern commercial technologies these transformational systems fully support the most modern practices for SW development and delivery. The MK6 ModX based system features Software Defined Networking (SDN) which delivers virtual switching and routing functions on ultra-modern Cisco switching, modern storage arrays and hyper-converged infrastructure, and IaaS and PaaS SW enabling virtualized and containerized microservices oriented applications.

b. Processing Power

The newest generation of COTS computing infrastructure, these systems break from previous Computing and CI in that they no longer build systems which are "fixed" to variants of HW for a given period or TI interval. The processing in these systems is a selection of the most technically relevant HW available at a given time. Current processing is built on 32 core Intel Zeon processors.

c. I/O and Networking

Components, like processing, are fielded from the newest of the breed available. Switch to switch interfaces currently operate at 100GPS.

d. OS and SW

MK6 ModX based systems employ ultra-modern SW products in the IaaS and PaaS layers which abstract tactical applications away from the underlying infrastructure. It is this industry relevant SW environment that enables virtualization, microservices, and continuous integration and deployment pipelines.

e. Modularity

MK6 ModX based systems employ HW modularity defining 12 unique 8U (a measure of rack-based appliances) modules. These modules form the building block of an ICS CI where four modules can be integrated into a common cabinet to deliver the needed capability.

f. Applications

MK6 ModX is the enterprise ICS CI supplier for all applications across the USN.

g. Limiting Factors

(1) HW

MK6 ModX based systems only limitation is the common module-based arrangement. The rapid integration of the newest available HW removes HW limitations from CS deployments.

(2) OS and Environment SW

This modern IaaS/PaaS environment removes all foreseen barriers to CS modernization of tactical applications. The foundations in CI/CD usher in the potential to close the gap between military and commercial SW deployment and hasten the speed to threat capability for the U.S. military.

5. Summary

As presented, the USN has made the transition from purpose build computing and SW, based on the technology of the day, to modern architectures that enable the implementation of relevant processes which field granular microservices-based SW in weeks versus months/years. These enabling technologies, when fully implemented, will reduce the cost of development, testing, certification, and distribution of capability. Table 2 summarizes the progression of USN CS core capabilities and their inherent limitations.

Table 2. Capability Progression as an Enabler of Navy Combat System Modernization

Stage	System	Key Features	Applications	Limitations
1	**AN/UYK-7 and AN/UYK-43**	32-bit architecture, fixed-point, and floating-point arithmetic	NTDS	Limited processing power and memory
		Processing speeds of 0.725 MIPS (UYK-7) and 1.2 MIPS (UYK-43)	Aegis CS	Specialized OE tied to bare metal HW
		Memory of 256 KB (UYK-7) expandable to 1 MB, up to 64 MB (UYK-43)	Shipboard and submarine tactical data processing	Monolithic applications hindered modernization efforts
		Flexible I/O channels, 16–64		
		MILSPEC design		
2	**AN/UYQ-70**	Modular, scalable design	Aegis CS	Continued reliance on MILSPEC UYK systems
		Use of COTS components	Submarine combat control	Bare metal deployments with limited virtualization
		Multi-processor capabilities	C2 systems	Monolithic applications, restricted flexibility and resiliency
		Unix-based OSs		
		Upgradable HW and SW		
3	**TI12 and TI16**	Fully COTS-based design	Continuation of legacy applications	Bare metal deployments persisted
		High modularity with chassis-based processors	Processing improvements across USN systems	Closely coupled applications still limited virtualization
		Linux-based OS (RHEL)		N+1 resiliency remained dominant
		Improved networking and storage systems		
		Cabinet-style modularity for processing and storage functions		

Stage	System	Key Features	Applications	Limitations
4	**USV ICS CI and MK6 ModX**	Fully virtualized and containerized microservices	Enterprise ICS CI for all surface USN applications	Minimal HW limitations due to modular designs
		SDN	Rapid modernization of combat systems	Abstracted SW layers remove most barriers to modernization
		Hyper-converged infrastructure		
		CI/CD		
		Modern Intel Xeon processors (32 cores)		

V. MICROSERVICES ARCHITECTURE

The following section presents an overview, key principles, advantages, disadvantages, and implementation considerations of a microservices-based computing architecture.

A. INTRODUCTION

1. Overview

Microservices architecture is a design approach in which an application is composed of multiple small, independent services that communicate over a network. Each service is designed to perform a specific business function and operates as a separate entity with its own codebase, data storage, and deployment life cycle (Fowler & Lewis, 2014). This modular structure contrasts with monolithic architectures in which all functionalities are tightly integrated into a single application.

In a microservices architecture, services are loosely coupled, meaning that each service can be developed, deployed, and scaled independently. This independence allows for greater flexibility in technology choices, as different services can use different programming languages, frameworks, and databases suited to their specific needs (Nadareishvili et al., 2016). The services interact through well-defined Application Program Interfaces (APIs), typically using lightweight protocols such as HTTP or messaging queues to facilitate communication between disparate components (Bass et al., 2021).

One of the key benefits of microservices is scalability. Services can be scaled independently based on demand, allowing for more efficient use of resources and improved performance (Dragoni et al., 2017). Additionally, microservices enhance resilience; if one service fails, it does not necessarily impact the entire application, which improves overall system reliability.

However, microservices also introduce complexities, such as managing inter-service communication and data consistency. The distributed nature of microservices can lead to challenges in maintaining consistent data and coordinating service interactions, requiring sophisticated orchestration and monitoring tools.

2. Key Principals

Microservices architecture represents a significant shift from traditional monolithic SW design. This architectural style involves breaking down an application into smaller, self-contained services that communicate through well-defined APIs. The goal is to enhance flexibility, scalability, and resilience. This section explores the key principles of microservices architecture, highlighting their implications and benefits.

a. Service Independence

One of the core principles of microservices architecture is service independence. In a microservices-based system, each service operates as a standalone unit responsible for a specific business function or capability (Fowler & Lewis, 2014). This independence allows for several important benefits.

(1) Autonomous Development and Deployment

Each microservice can be developed, tested, and deployed independently. This autonomy facilitates CI/CD practices, allowing for frequent and reliable releases. Changes to one service do not necessitate changes or redeployment of other services, thereby minimizing disruptions.

(2) Technology Diversity

Services can be built using different programming languages, frameworks, or data stores best suited to their specific requirements (Nadareishvili et al., 2016). This technological diversity enables teams to leverage the best tools for each service without being constrained by a uniform technology stack.

(3) Fault Isolation

Failures in one service do not directly impact others. This isolation enhances the overall system's resilience and reliability, as the impact of a failure is contained within the service experiencing the issue (Dragoni et al., 2017).

b. Loose Coupling

Loose coupling is another fundamental principle of microservices architecture. It refers to the design of services such that they interact with each other through well-defined APIs rather than being tightly integrated. Loose coupling offers several advantages.

(1) Reduced Dependencies

Services are designed to be minimally dependent on one another, which reduces the risk of changes in one service impacting others. This design facilitates more straightforward modifications and upgrades (Fowler & Lewis, 2014).

(2) Interoperability

Services communicate through standardized protocols such as HTTP, REST, or messaging queues. This standardization ensures that services can interact seamlessly despite being built using different technologies or platforms (Bass et al., 2021).

(3) Flexibility and Agility

The decoupling of services supports a more agile development process, as teams can work on different services concurrently without worrying about integration issues.

c. Single Responsibility Principle

The Single Responsibility Principle (SRP) is a design guideline stating that each microservice should have one primary responsibility or business capability (Fowler & Lewis, 2014). SRP contributes to several aspects of microservices architecture.

(1) Focused Functionality

Each service is designed to handle a specific business function, leading to a clear separation of functions. This focus simplifies development, testing, and maintenance.

(2) Manageability

By adhering to SRP, services remain smaller and more manageable. This manageability makes it easier to understand, test, and deploy each service individually (Dragoni et al., 2017).

(3) Scalability

Services designed around SRP can be scaled independently based on their specific workloads, improving resource utilization and performance.

d. Decentralized Data Management

In microservices architecture, each service typically manages its own data store. This decentralized approach contrasts with the monolithic model, in which a single central database is often used (Dragoni et al., 2017). Decentralized data management offers several benefits.

(1) Data Ownership

Each service has complete control over its own data, including the schema and storage technology. This autonomy allows services to be optimized for their specific data needs and requirements (Nadareishvili et al., 2016).

(2) Reduced Data Coupling

By avoiding a shared database, microservices reduce the risk of data conflicts and integration issues. Each service is responsible for its own data consistency and integrity (Bass et al., 2021).

(3) Flexibility

Services can use different types of databases (e.g., SQL, NoSQL) based on their requirements. This flexibility allows for the use of the most appropriate data storage solution for each service (Fowler & Lewis, 2014).

e. Automated Deployment and Continuous Integration

Automated deployment and continuous integration are essential principles in microservices architecture. These practices facilitate the efficient management and release of services.

(1) Continuous Integration

Continuous integration involves frequently integrating code changes from multiple developers into a shared repository. Automated build and test processes help identify integration issues early and ensure that code changes do not introduce new defects.

(2) Automated Deployment

Automated deployment pipelines enable the continuous delivery of services to production environments. This automation reduces manual errors, accelerates release cycles, and improves deployment reliability (Fowler & Lewis, 2014).

(3) Testing

Automated testing is crucial for maintaining service quality. Tests can include unit tests, integration tests, and end-to-end tests, all of which are integrated into the CI/CD pipeline to ensure that services function correctly (Dragoni et al., 2017).

f. Resilience and Fault Isolation

Resilience and fault isolation are critical principles in microservices architecture. These principles enhance the system's ability to handle failures and recover from them effectively.

(1) Fault Isolation

By isolating failures within individual services, microservices architecture prevents issues from propagating throughout the system. This isolation helps maintain overall system stability and reliability (Dragoni et al., 2017).

(2) Fault Tolerance

Microservices often employ patterns such as circuit breakers, retries, and timeouts to handle failures gracefully. These patterns help manage service disruptions and maintain service availability (Nadareishvili et al., 2016).

(3) Resilient Design

Services are designed to be resilient and capable of recovering from failures. Techniques such as redundancy, failover, and backup strategies are employed to enhance resilience and minimize the impact of failures.

g. *Scalability*

Scalability is a key advantage of microservices architecture. Services can be scaled independently based on their specific needs, which allows for more efficient resource utilization (Dragoni et al., 2017).

(1) Horizontal Scaling

Microservices can be scaled horizontally by adding more instances of a service to handle an increased load. This approach improves performance and capacity without requiring changes to the service itself (Bass et al., 2021).

(2) Resource Allocation

Independent scaling allows for targeted resource allocation. Resources can be allocated to specific services experiencing high demand, optimizing overall system performance (Fowler & Lewis, 2014).

(3) Dynamic Scaling

Microservices architecture supports dynamic scaling, in which services can be scaled up or down based on real-time demand. This flexibility ensures that resources are used efficiently and cost-effectively.

h. DDD

DDD is a methodology often used in conjunction with microservices architecture to model services around business domains (Fowler & Lewis, 2014).

(1) Domain Modeling

DDD encourages the modeling of services based on specific business domains or capabilities. This approach aligns services with business needs and promotes a clear separation of responsibilities (Nadareishvili et al., 2016).

(2) Bounded Contexts

DDD introduces the concept of bounded contexts, which define clear boundaries around a particular domain or subdomain. Each microservice operates within its own bounded context, ensuring a clear focus and reducing ambiguity (Dragoni et al., 2017).

(3) Collaborative Design

DDD fosters collaboration between business stakeholders and development teams. By aligning services with business requirements, DDD ensures that the system accurately reflects the organization's needs and goals.

i. Service Discovery

Service discovery is essential for managing and locating services dynamically in a microservices architecture.

(1) Dynamic Registration

Services register themselves with a service registry upon startup. The registry maintains a list of available services and their locations, enabling other services to discover and interact with them (Nadareishvili et al., 2016).

(2) Service Lookup

Services can query the service registry to locate other services they need to interact with. This dynamic discovery mechanism allows for flexible service interactions and adaptation to changes in the system (Dragoni et al., 2017).

(3) Load Balancing

Service discovery mechanisms often include load balancing capabilities to distribute requests across multiple instances of a service. This distribution improves performance and ensures high availability.

j. API Gateway

An API Gateway acts as a single-entry point for clients accessing a microservices-based application (Bass et al., 2021). It provides several important functions:

(1) Request Routing

The API Gateway routes incoming requests to the appropriate microservices-based on the request path or other criteria. This centralizes request management and simplifies client interactions (Fowler & Lewis, 2014).

(2) Cross-Cutting Concerns

The API Gateway handles cross-cutting concerns such as authentication, authorization, logging, and rate limiting. By centralizing these concerns, the API Gateway reduces the complexity of individual services.

(3) Aggregation

The API Gateway can aggregate responses from multiple services into a single response, simplifying the client-side logic and improving performance.

3. Conclusion

Microservices architecture offers a modern approach to SW design, emphasizing service independence, loose coupling, and decentralized data management. By adhering to key principles such as the SRP, automated deployment, and resilience, organizations can build flexible, scalable, and reliable systems. Microservices architecture aligns closely with business needs through DDD, supports dynamic service discovery, and centralizes cross-cutting concerns through the API Gateway. These principles collectively contribute to the success and efficiency of microservices-based applications. Table 3 summarizes the features of a microservices architecture.

Table 3. Features of a Microservices Architecture

Aspect	Details
Definition	Microservices architecture is a modular design approach where applications are composed of small, independent services. Each service performs a specific function and operates as an autonomous entity.
Key Features	**Independence:** Services have separate codebases, data storage, and deployment life cycles.
	Communication: Services interact through APIs using lightweight protocols (e.g., HTTP, messaging queues).
	Scalability: Each service can be independently scaled to optimize resource use.
1. Service Independence	**Autonomous Development and Deployment:** Services can be developed, tested, and deployed independently.
	Technology Diversity: Services can use different technologies as needed.
	Fault Isolation: Failures in one service do not impact others.
2. Loose Coupling	Services interact through well-defined APIs, minimizing dependencies.
	Ensures interoperability across diverse technologies.
3. Single Responsibility	Each service focuses on a specific business function.
	Enhances manageability and scalability.

Aspect	Details
4. Decentralized Data	Each service manages its own data store, avoiding shared databases.
	Allows data ownership and technology flexibility.
5. CI/CD Automation	Facilitates continuous integration, automated deployment, and testing.
	Reduces errors and accelerates releases.
6. Resilience and Fault Isolation	Faults are isolated within individual services to enhance stability.
	Employs patterns like circuit breakers and retries for fault tolerance.
7. Scalability	Horizontal and dynamic scaling enables efficient resource utilization.
8. DDD	Services are modeled around specific business domains and bounded contexts.
	Aligns architecture with organizational goals.
9. Service Discovery	Enables dynamic registration and lookup of services through a registry.
	Facilitates load balancing and high availability.
10. API Gateway	Acts as a central entry point for routing requests.
	Handles cross-cutting concerns like authentication and logging.

B. KUBERNETES CLUSTER COMPONENTS

Figure 7 Presents a simplified diagram of a microservices-based architecture.

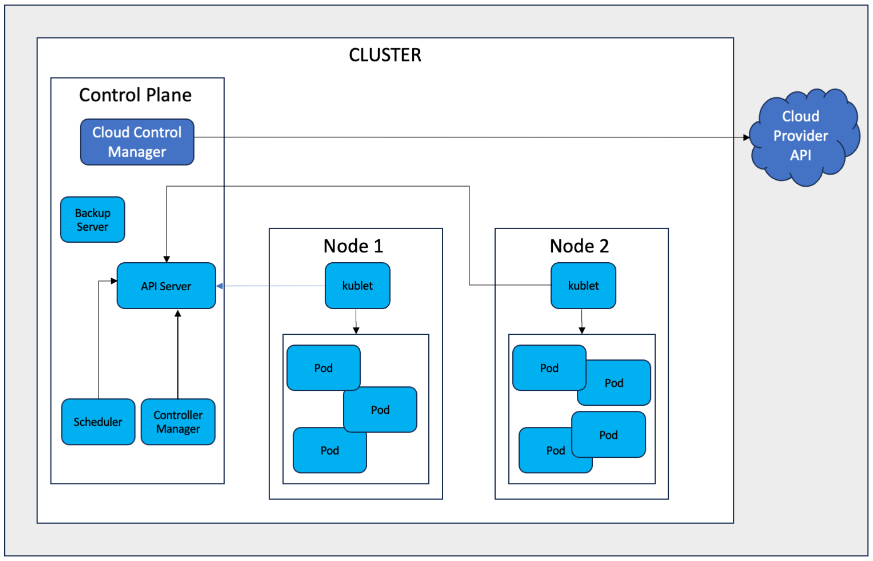


Figure 7. Simplified Microservices and Kubernetes Diagram

1. Pod

A Kubernetes Pod is the smallest and most fundamental unit of deployment in Kubernetes, an open-source container orchestration platform. A Pod represents a single instance of a running process in a Kubernetes cluster and can host one or more containers. Containers within the same Pod share the same network namespace, IP address, and storage, which allows them to communicate with each other via localhost and share data more easily. Pods are essential in Kubernetes because they abstract much of the complexity of managing containers directly, allowing users to focus on deploying applications rather than worrying about the underlying infrastructure.

a. Components

Each Pod in Kubernetes consists of one or more containers, typically Docker containers, which share resources. These shared resources include the following:

- **Networking:** Pods are assigned unique IP addresses in the cluster, and all containers within a Pod share the same network interface. This means they can communicate internally using localhost, even though they are distinct containers.
- **Storage:** Pods can define one or more volumes (persistent storage), and these volumes are shared across the containers within the Pod. This is useful for scenarios in which containers need to persist data or share files (Burns et al., 2018).

Pods can host multiple containers, but they are typically used to group containers that are tightly coupled and need to run together. For example, a web server container might be grouped with a logging or monitoring sidecar container. In Kubernetes, multi-container Pods follow the principle of *shared fate*, if one container in the Pod dies, Kubernetes treats it as if the entire Pod has failed and may reschedule the Pod accordingly.

b. Life Cycle

Pods in Kubernetes are ephemeral in nature. They are not designed to be persistent over long periods of time; instead, they are created, used, and eventually terminated or replaced. Kubernetes automatically handles the scheduling and rescheduling of Pods across nodes in a cluster. This self-healing capability allows Kubernetes to ensure that desired application states are always maintained (Hightower et al., 2017). Pods can have different states such as Pending, Running, Succeeded, Failed, or Unknown, reflecting their life cycle and health.

c. Scaling

A key feature of Kubernetes is its ability to scale applications based on demand. However, Kubernetes does not scale Pods directly. Instead, it uses higher-level abstractions like Deployments and ReplicaSets to manage scaling. Deployments define the desired state of Pods and allow Kubernetes to create or terminate Pods to match that state. This decoupling allows Kubernetes to maintain high availability and handle the dynamic nature of containerized applications (Hightower et al., 2017).

d. Networking and Service Discovery

Pods in Kubernetes are assigned ephemeral IP addresses. Since Pods can be destroyed and recreated dynamically, their IPs are not stable. To solve this issue, Kubernetes provides services, which act as stable network front ends to Pods. Services enable consistent access to Pods, regardless of their IP changes. Kubernetes also includes a built-in domain name server (DNS) service that allows Pods to discover other services via DNS names, making service discovery seamless in large clusters (Burns et al., 2018).

e. Summary

In summary, Kubernetes Pods provide the foundational building blocks for running containerized applications in a cluster. They group one or more containers into a single logical unit, offering shared network and storage resources. Pods are ephemeral by design, and Kubernetes uses abstractions like Deployments to manage their life cycle and scaling. With Pods, Kubernetes abstracts much of the complexity involved in container orchestration, allowing for automated deployment, scaling, and self-healing of applications.

2. Node

A Kubernetes Node is a worker machine in a Kubernetes cluster, responsible for running the workloads defined by the Kubernetes control plane. Nodes can be physical servers or virtual machines (VMs), and they host the necessary services and runtime to manage and execute Pods, the smallest deployable units in Kubernetes. Each node is controlled by the Kubernetes control plane, which assigns Pods to nodes and monitors their performance. Nodes are fundamental components of Kubernetes clusters, enabling distributed deployment and scaling of containerized applications.

a. Componenets

Every Kubernetes node runs several essential services that are necessary for it to participate in the cluster.

(1) Kubelet

The kubelet is an agent that runs on each node and ensures that the containers inside the assigned Pods are running. It communicates with the Kubernetes control plane and receives instructions on which Pods to run, manages Pod life cycles, and reports the health of the node (Hightower et al., 2017).

(2) Container Runtime

The container runtime (such as Docker, containerd, or CRI-O) is responsible for pulling container images from a registry, running the containers, and managing their life cycles. Kubernetes supports multiple container runtimes through the Container Runtime Interface (CRI) (Burns et al., 2018).

(3) Kube-Proxy

Kube-proxy is a network proxy that runs on each node and ensures that networking for Pods is properly configured. It manages network rules that allow Pods to communicate with each other, as well as external traffic, ensuring that services are reachable both within and outside the cluster.

(4) Node API Server

In addition to kubelet and kube-proxy, the node may also run the node API server, which exposes data about the state of the node to the Kubernetes control plane (Burns et al., 2018).

b. Types

Kubernetes clusters generally have two types of nodes:

- **Worker Nodes:** These are the standard nodes where application workloads (Pods) are run. Each worker node runs the necessary services (kubelet, container runtime, and kube-proxy) and is responsible for executing and managing the containers that are part of the Pods scheduled on it (Hightower et al., 2017).
- **Master Nodes (Control Plane):** These nodes host the control plane components, such as the Kubernetes API server, scheduler, and controller manager. While the control plane nodes do not typically run user workloads, they are critical for managing the cluster by making scheduling decisions and maintaining the desired state of the cluster (Burns et al., 2018).

c. Life Cycle

The life cycle of a Kubernetes node includes several key states (Hightower et al., 2017):

- **Ready:** A node is in the "Ready" state when it is healthy and available to run Pods. The control plane regularly checks the health of nodes via heartbeats from the kubelet.
- **Not Ready:** If a node is unhealthy or unresponsive, it enters the "Not Ready" state. The control plane will avoid scheduling new Pods on this node and may reschedule existing Pods on other healthy nodes, ensuring high availability.

- **Cordoning and Draining:** Nodes can be temporarily removed from service through a process called cordoning, which prevents new Pods from being scheduled on the node. Draining moves the existing Pods off the node, typically in preparation for maintenance or scaling down infrastructure.

d. Scaling

Kubernetes supports cluster autoscaling, in which the number of nodes in the cluster automatically increases or decreases based on resource demand. For example, if the current nodes are fully utilized, Kubernetes can provision additional nodes to handle the increased load. Conversely, underutilized nodes can be decommissioned to optimize resource usage (Hightower et al., 2017).

Node pools are a way to manage groups of nodes with similar configurations, such as the same instance type or geographic location. Node pools are particularly useful in cloud environments, where different types of workloads may require different machine configurations. Kubernetes can schedule Pods to specific node pools based on resource requirements or other constraints.

e. Summary

In summary, a Kubernetes node is a critical component of the cluster, providing the environment to run containerized workloads. Nodes host essential services such as kubelet and kube-proxy, which enable the node to communicate with the control plane and manage the life cycle of Pods. With features like autoscaling, node pools, and seamless orchestration, Kubernetes nodes offer a robust and scalable infrastructure for running modern applications.

3. Control Plane

The Kubernetes control plane is the central component responsible for managing and orchestrating the operations of a Kubernetes cluster. It serves as the brain of the cluster, overseeing the life cycles of Pods, maintaining the desired state of the system, and providing the necessary APIs for user interaction. The control plane operates on a set of master nodes that host several key components, including the API server, etcd, scheduler, controller manager, and cloud controller manager. Together, these components manage the cluster state, scheduling, and coordination of resources within the cluster.

a. Components

(1) API Server

The Kubernetes API server is the entry point to the Kubernetes control plane and serves as the central management hub. It exposes the Kubernetes API, which provides a RESTful interface for users and services to interact with the cluster. Every interaction with the cluster, such as deploying applications or scaling resources, passes through the API server. It authenticates requests, validates them, and processes them by interacting with other components of the control plane (Hightower et al., 2017). The API server is designed to be highly scalable and can be replicated across multiple nodes for redundancy.

(2) Etcd

Etcd is a distributed key-value store used by Kubernetes to store all cluster data. It maintains the cluster's state and is the source of truth for all configurations, including details about Pods, services, and network policies. etcd's consistency guarantees ensure that the state of the cluster is accurately recorded and can be recovered in the event of a failure (Burns et al., 2018). Because of its critical role, etcd must be backed up regularly, and it is usually run on highly available infrastructure.

(3) Scheduler

The Kubernetes scheduler is responsible for assigning Pods to nodes in the cluster. When a new Pod is created, the scheduler evaluates the current state of the cluster and assigns the Pod to an appropriate node based on resource availability and predefined constraints. Factors such as CPU, memory, storage, and affinity rules are considered during this decision-making process. The scheduler ensures that workloads are distributed efficiently across nodes to prevent overloading and to maintain optimal performance (Hightower et al., 2017).

(4) Controller Manager

The Kubernetes controller manager runs a collection of controllers, each responsible for monitoring the state of the cluster and making changes to ensure the desired state is achieved. Some of the primary controllers include:

(5) Node Controller

The node controller monitors the health of nodes and ensures that failed nodes are removed from service.

(6) Replication Controller

The replication controller ensures that the specified number of replicas of a Pod is always running.

(7) Endpoint Controller

The endpoint controller manages the association between services and Pods. The controller manager watches the cluster through the API server and continuously reconciles the actual state of resources with the desired state, as defined by users (Burns et al., 2018).

(8) Cloud Controller Manager

The could controller manager integrates Kubernetes with underlying cloud infrastructure. It allows Kubernetes to interact with cloud providers' APIs to manage resources such as load balancers, VMs, and storage. The cloud controller manager enables Kubernetes to abstract infrastructure details, making it easier to manage clusters across different cloud environments (Hightower et al., 2017).

b. Functions

(1) Cluster State Management

The control plane ensures that the actual state of the system matches the desired state specified by users through configuration files (manifests). If discrepancies arise, the control plane takes corrective action, such as restarting failed Pods or rescheduling them on different nodes.

(2) Scaling and Resource Management

The control plane manages the scaling of applications by monitoring resource usage and deploying additional Pods as needed. Autoscaling can be configured based on CPU utilization or custom metrics.

(3) Security and Access Control

The API server enforces access control policies, including role-based access control to manage user permissions and secure communication between components.

c. Summary

In summary, the Kubernetes control plane is the operational core of a Kubernetes cluster. It manages the scheduling, orchestration, and state of the cluster, ensuring that applications are running as intended. Through components like the API server, scheduler, etcd, and various controllers, the control plane enables Kubernetes to be a highly scalable, resilient, and dynamic platform for managing containerized applications.

4. Cluster

A Kubernetes cluster is a set of machines (physical or virtual) that work together to run containerized applications, managed and orchestrated by Kubernetes. It is the foundational infrastructure for Kubernetes, designed to automate the deployment, scaling, and management of applications across multiple nodes. A Kubernetes cluster is composed of two major components: the control plane, which oversees the cluster's operations and ensures the desired state of applications and the worker nodes, which run the actual containerized workloads.

a. Components

(1) Control Plane

The control plane is the centralized management unit of the cluster. It is responsible for maintaining the cluster's desired state, making scheduling decisions, and responding to failures. The control plane runs on one or more master nodes and consists of several key components:

(2) API Server

The Kubernetes API server acts as the gateway to the cluster. It exposes Kubernetes' RESTful API, which is used by administrators, users, and internal components to interact with the cluster. All requests to create, modify, or retrieve resources pass through the API server (Hightower et al., 2017).

(3) Etcd

A distributed key-value store, etcd stores the state of the entire Kubernetes cluster, including configurations and secrets. It serves as the source of truth for the cluster, ensuring consistency and reliability of data (Burns et al., 2018).

(4) Scheduler

The Kubernetes scheduler assigns newly created Pods to nodes based on resource availability, ensuring efficient use of cluster resources (Hightower et al., 2017).

(5) Controller Manager

The controller manager runs various controllers that monitor the state of the cluster and reconcile it with the desired state, such as ensuring the correct number of Pod replicas are running or monitoring the health of nodes (Burns et al., 2018).

(6) Cloud Controller Manager

This component integrates Kubernetes with cloud providers' APIs, allowing Kubernetes to manage cloud resources like load balancers and storage in a cloud-native environment (Hightower et al., 2017).

(7) Worker Nodes

Worker nodes are the machines that run the actual workloads in a Kubernetes cluster. Each node hosts Pods, which are collections of one or more containers. The key components of a worker node include:

(8) Kubelet

The kubelet is the primary agent on a worker node, ensuring that the containers specified in a Pod are running and healthy. It communicates with the control plane, receives instructions, and manages the life cycle of Pods on the node (Hightower et al., 2017).

(9) Container Runtime

The container runtime, such as Docker or container, is responsible for pulling container images from registries, starting containers, and managing their life cycles (Burns et al., 2018).

(10) Kube-Proxy

Kube-proxy ensures that network traffic reaches the appropriate Pods. It manages the networking rules that allow communication between services and Pods, both internally within the cluster and with external clients.

b. Networking

A key feature of Kubernetes is its networking model, which enables seamless communication between components. Each Pod in the cluster is assigned a unique IP address, and Kubernetes ensures that Pods can communicate with each other and with services inside and outside the cluster, regardless of which node they are running on. Networking is abstracted through services that provide stable endpoints, even as Pods are created, destroyed, or rescheduled (Burns et al., 2018).

c. Scaling

Kubernetes clusters are designed to be highly scalable. Horizontal Pod Autoscaling allows the system to automatically adjust the number of Pods based on demand. Cluster Autoscaling can add or remove worker nodes as necessary, depending on the resource requirements of the workloads (Hightower et al., 2017).

Kubernetes also has built-in self-healing capabilities. If a Pod or node fails, the control plane reschedules the workload on another node, ensuring high availability.

Kubernetes monitors the health of nodes and containers, automatically restarting or replacing unhealthy components to maintain the desired state of the cluster.

d. Summary

In summary, a Kubernetes cluster is a powerful and scalable platform for running containerized applications, designed to automate the complex tasks of deployment, scaling, and management. With their control planes managing the overall state and their worker nodes executing the workloads, Kubernetes clusters enable organizations to achieve high availability, efficient resource usage, and automated recovery from failures. Table 4 summarizes Kubernetes cluster components and their key features.

Table 4. Kubernetes Components Summary

Concept	Description	Key Features	Example Uses
Pod	Smallest deployable unit in Kubernetes, representing one or more containers running together.	Shared network and storage Containers communicate via localhost Ephemeral and self-healing	Hosting tightly coupled containers like a web server and sidecar for logging.
Node	A machine (physical or virtual) in the cluster that runs Pods and communicates with the control plane.	Includes kubelet, container runtime, kube-proxy Can be worker nodes or master nodes (control plane)	Hosting workloads (worker node) or running control plane components (master node).
Control Plane	Central management unit that orchestrates cluster operations and maintains desired state.	API server for cluster interaction Etcd for storing cluster state Scheduler for workload placement Controllers for state management	Assigning Pods to nodes, ensuring replicas, and coordinating resource usage.
Cluster	A set of nodes working together, managed by the control plane.	Unified networking model Scalable and self-healing Service discovery through DNS	Running containerized apps across distributed systems with load balancing and autoscaling.

C. ADVANTAGES

Microservices architecture represents a paradigm shift from monolithic design to a more flexible and scalable approach. By breaking down applications into smaller, self-contained services, organizations can reap significant benefits. This section explores the key advantages of microservices architecture, including enhanced scalability, flexibility, resilience, and maintainability.

1. Scalability

One of the most prominent advantages of microservices architecture is its ability to scale efficiently. Unlike monolithic systems, where scaling typically involves scaling the entire application, microservices allow for granular scaling of individual services (Dragoni et al., 2017).

a. Granular Scaling

Microservices can be scaled independently based on their specific needs. For instance, if a particular service experiences high demand, it can be scaled up without affecting other services (Nadareishvili et al., 2016). This capability ensures optimal resource utilization and improves performance during peak loads.

b. Resource Optimization

By scaling only the necessary services, organizations can optimize their resource allocation. This targeted approach helps in managing operational costs more effectively, as resources are not wasted on scaling components that do not require additional capacity (Fowler & Lewis, 2014).

2. Flexibility and Agility

Microservices architecture promotes flexibility and agility in development and deployment processes.

a. Independent Development

Each microservice is developed and deployed independently, allowing different teams to work on various services concurrently. This independence reduces dependencies and bottlenecks associated with monolithic systems. Teams can use different technologies and development frameworks that best fit their service requirements (Nadareishvili et al., 2016).

b. Faster Time-to-Market

With microservices, new features and updates can be introduced more rapidly. Since services are decoupled, changes to one service do not necessitate changes to or redeployment of other services (Dragoni et al., 2017). This agility accelerates the development cycle and helps organizations respond quickly to market demands and customer feedback.

3. Resilience and Fault Isolation

Microservices architecture enhances the resilience of applications by isolating faults and minimizing their impact.

a. Fault Isolation

In a microservices architecture, failures are contained within individual services. This isolation prevents failures in one service from cascading to others, thereby preserving the overall system's stability (Fowler & Lewis, 2014). For example, if a payment processing service fails, it does not affect other services like user management or the product catalog.

b. Resilient Design

Microservices often incorporate resilience patterns such as circuit breakers, retries, and fallbacks to handle service disruptions gracefully (Dragoni et al., 2017). These patterns enable services to recover from failures and continue functioning, enhancing the overall reliability of the system.

4. Improved Maintainability

Microservices architecture simplifies the maintenance and management of applications.

a. Modular Structure

Each microservice encapsulates a specific functionality or business capability, leading to a modular application structure (Nadareishvili et al., 2016). This modularity makes it easier to understand, test, and manage individual components, as changes are localized within specific services.

b. Reduced Complexity

By breaking down a large monolithic application into smaller services, the complexity of managing and updating the system is reduced. Developers can focus on one service at a time, which simplifies debugging, testing, and deployment.

c. CI/CD

Microservices support CI/CD practices. Automated pipelines enable frequent and reliable deployments of individual services, reducing the risk of introducing errors and improving the overall quality of the application.

5. Technology Diversity

Microservices architecture allows for the use of diverse technologies and tools, which can enhance the overall system's performance and capabilities.

a. Technology Choices

Different microservices can be built using different programming languages, frameworks, and databases, depending on their specific needs. This flexibility allows teams to select the most appropriate technology stack for each service, optimizing performance and efficiency (Fowler & Lewis, 2014).

b. Innovation and Experimentation

With microservices, teams can experiment with new technologies and approaches without impacting the entire system. This innovation fosters a culture of experimentation and continuous improvement (Nadareishvili et al., 2016).

6. Enhanced Security

Microservices architecture can enhance the security of applications by isolating services and managing access control more effectively.

a. Service Isolation

Each microservice operates independently, which means that security vulnerabilities are confined to individual services rather than the entire application. This isolation helps in minimizing the potential impact of security breaches (Dragoni et al., 2017).

b. Granular Access Control

Microservices allow for fine-grained access control. Security policies and authentication mechanisms can be implemented at the service level, ensuring that only authorized users and services can access sensitive data.

7. Better Alignment with Business Domains

Microservices architecture supports DDD, which aligns the architecture with business needs.

a. Domain Modeling

Microservices are often organized around business domains or capabilities, which helps in creating services that are closely aligned with organizational goals and processes (Fowler & Lewis, 2014). This alignment ensures that the architecture reflects the business structure and facilitates more effective decision-making.

b. Bounded Contexts

DDD introduces the concept of bounded contexts, where each microservice operates within its own defined boundaries. This approach reduces ambiguity and ensures that each service has a clear focus and responsibility (Nadareishvili et al., 2016).

8. Enhanced Developer Productivity

Microservices architecture can improve developer productivity by enabling more efficient development practices.

a. Parallel Development

With microservices, multiple teams can work on different services simultaneously, reducing development time and increasing productivity. Teams are not blocked by dependencies on other parts of the application, allowing for faster progress and quicker delivery of features.

b. Focused Expertise

Developers can specialize in specific services or technologies, leading to greater expertise and efficiency. This specialization enhances the quality of the service and speeds up development processes (Dragoni et al., 2017).

9. Optimized Performance

Microservices can enhance performance through various optimizations.

a. Service-Specific Optimization

Each microservice can be optimized based on its specific requirements and performance characteristics. For example, a service handling high-throughput data processing can be tuned for performance, while another service focused on user interactions can be optimized for responsiveness (Fowler & Lewis, 2014).

b. *Efficient Resource Utilization*

Microservices enable efficient resource utilization by scaling services independently and optimizing their performance. This approach reduces resource wastage and improves overall system efficiency.

10. Conclusion

Microservices architecture offers a range of advantages that contribute to the flexibility, scalability, and resilience of applications. By allowing for granular scaling, independent development, and fault isolation, microservices enhance the ability to manage complex systems effectively. The modular structure and support for continuous integration and deployment improve maintainability and developer productivity. Additionally, the use of diverse technologies, enhanced security, and alignment with business domains further highlights the benefits of microservices architecture. As organizations continue to seek ways to improve their SW systems, microservices provide a robust and adaptable framework for achieving these goals.

D. DISADVANTAGES

While microservices architecture offers several advantages, it also comes with its own set of challenges and disadvantages. These can impact development, deployment, and operational aspects of applications. This section explores the key disadvantages of microservices architecture, including complexity, communication overhead, data management issues, testing challenges, deployment difficulties, and increased operational overhead.

1. Increased Complexity

Compared to monolithic systems, microservices architecture introduces significant complexity due to the nature of managing multiple services.

a. *System Complexity*

Decomposing an application into numerous microservices can lead to a complex system of interconnected services. Each service must be designed, developed, deployed, and maintained separately, which increases the overall complexity of the system (Fowler & Lewis, 2014). Managing dependencies and interactions between services can become intricate, requiring careful orchestration and monitoring (Dragoni et al., 2017).

b. Operational Overhead

The complexity of managing multiple services extends to operational tasks such as deployment, monitoring, and troubleshooting. Ensuring that all services are properly configured, scaled, and maintained requires robust infrastructure and tools (Nadareishvili et al., 2016). This operational overhead can be challenging for organizations without the necessary expertise or resources.

2. Communication Overhead

Microservices architecture relies on inter-service communication, which can introduce several types of overhead.

a. Network Latency

Services in a microservices architecture communicate over a network, typically using HTTP/REST, gRPC, or messaging protocols. This network communication can introduce latency compared to in-process calls within a monolithic application. Increased latency can impact the performance of the system, particularly in scenarios in which services need to frequently exchange data.

b. Data Serialization

Communication between microservices often involves data serialization and deserialization, adding extra processing time and overhead. The choice of serialization formats (e.g., JavaScript object notation [JSON], extensible markup language [XML]) can influence performance and interoperability between services.

3. Data Management Challenges

Microservices architecture can complicate data management due to the distributed nature of data storage and access.

a. Data Consistency

Ensuring data consistency across multiple services can be challenging. Unlike monolithic systems in which a single database manages data consistency, microservices often require distributed data management strategies (Dragoni et al., 2017). Techniques such as eventual consistency and distributed transactions can help address consistency issues but may introduce additional complexity.

b. Database Fragmentation

In a microservices architecture, each service typically has its own database or data store. This fragmentation can lead to difficulties in managing data schema changes and ensuring data integrity across services (Nadareishvili et al., 2016). Synchronizing and aggregating data from multiple sources can be complex and require additional tooling.

4. Testing Difficulties

Testing microservices applications presents unique challenges compared to testing traditional monolithic systems.

a. Integration Testing

Testing interactions between multiple microservices can be more complex than testing a monolithic application. Integration tests need to account for the interactions between services and ensure that data flows correctly through the system (Fowler & Lewis, 2014). Managing test environments and ensuring that all services are correctly mocked or deployed for testing can be challenging.

b. End-to-End Testing

Conducting end-to-end tests in a microservices architecture requires setting up and managing a complex environment with multiple services. Ensuring that all services are available and functioning correctly during testing can be difficult, particularly in distributed environments.

5. Deployment and Release Management

Deploying and managing releases in a microservices architecture can be more complicated than doing so in monolithic systems.

a. Service Coordination

Deploying multiple services requires coordinating their release and ensuring compatibility between versions. This coordination can be challenging, particularly when services have interdependencies. Managing rollouts, rollbacks, and versioning of services requires robust deployment strategies and tools.

b. Deployment Automation

Automating the deployment of multiple services requires sophisticated CI/CD pipelines and orchestration tools. Setting up and maintaining these pipelines can be complex and may require significant investment in infrastructure and tooling (Dragoni et al., 2017).

6. Increased Operational Overhead

Microservices architecture can lead to increased operational overhead due to the need for managing and monitoring multiple services.

a. Monitoring and Logging

Monitoring and logging across multiple services can be more complex than doing so in monolithic systems. Each service needs to be instrumented for logging and monitoring, and aggregating and analyzing logs from various services requires comprehensive tools and practices (Nadareishvili et al., 2016). Ensuring end-to-end visibility and diagnosing issues across services can be challenging.

b. Resource Management

Microservices often require more resources than monolithic applications do because of microservices' need for separate instances of each service. Managing the deployment, scaling, and resource utilization of multiple services requires careful planning and infrastructure management.

7. Security Concerns

Microservices architecture introduces additional security considerations when compared to monolithic systems.

a. Attack Surface

With multiple services communicating over a network, the attack surface increases. Each service needs to be secured individually, and the communication channels between services must be protected (Dragoni et al., 2017). Ensuring that all services adhere to security best practices and are protected from vulnerabilities can be challenging.

b. Authentication and Authorization

Managing authentication and authorization across multiple services requires a robust strategy. Ensuring that users and services have appropriate access rights and that security policies are consistently enforced can be complex (Fowler & Lewis, 2014).

8. Service Discovery and Management

Service discovery and management are essential components of microservices architecture but can pose challenges.

a. Dynamic Discovery

In a dynamic microservices environment, services can be added, removed, or scaled dynamically. Implementing effective service discovery mechanisms to ensure that services can locate and communicate with each other is crucial. This dynamic nature adds complexity to the architecture and requires reliable discovery tools.

b. Configuration Management

Managing configurations for multiple services can be complex. Each service may have its own configuration settings, and ensuring consistency and correctness across services requires robust configuration management practices (Nadareishvili et al., 2016).

E. SUMMARY OF ADVANTAGES AND DISADVANTAGES

Table 5 summarizes the advantages and disadvantages of a microservices architecture discussed in this section.

Table 5. Advantages and Disadvantages of Microservices Architecture

Aspect	Details
Advantages	
Scalability	**Granular Scaling**: Individual services can be scaled independently based on their needs, optimizing resource utilization and performance (Dragoni et al., 2017).
	Resource Optimization: Targeted scaling reduces operational costs and prevents over-provisioning (Fowler & Lewis, 2014).
Flexibility and Agility	**Independent Development**: Teams can work on different services concurrently using varied technologies (Nadareishvili et al., 2016).
	Faster Time-to-Market: Decoupled services allow rapid feature updates without affecting other parts of the application (Dragoni et al., 2017).
Resilience and Fault Isolation	**Fault Isolation**: Failures in one service do not cascade to others, enhancing system stability (Fowler & Lewis, 2014).
	Resilient Design: Patterns like circuit breakers and retries improve recovery and reliability (Dragoni et al., 2017).
Improved Maintainability	**Modular Structure**: Clear separation of functionalities makes understanding and managing services easier (Nadareishvili et al., 2016).
	Continuous Integration and Deployment: Automated pipelines facilitate frequent, reliable updates.
Technology Diversity	**Technology Choices**: Different services can use the most suitable technologies, improving performance (Fowler & Lewis, 2014).
	Innovation and Experimentation: Teams can test new tools or methods without affecting the entire system.
Enhanced Security	**Service Isolation**: Limits the impact of vulnerabilities to specific services (Dragoni et al., 2017).
	Granular Access Control: Fine-grained security policies ensure tighter access management.
Alignment with Business Domains	**Domain Modeling**: Services align with business processes, enabling better decision-making (Fowler & Lewis, 2014).
	Bounded Contexts: Clear boundaries reduce ambiguity and streamline service responsibilities (Nadareishvili et al., 2016).

Aspect	Details
Enhanced Developer Productivity	**Parallel Development**: Teams can work on separate services simultaneously, speeding up development.
	Focused Expertise: Developers can specialize in specific services, enhancing efficiency (Dragoni et al., 2017).
Optimized Performance	**Service-Specific Optimization**: Services can be fine-tuned for their unique requirements.
	Efficient Resource Utilization: Independent scaling improves overall system efficiency.
Disadvantages	
Increased Complexity	**System Complexity**: Managing multiple interconnected services increases architectural complexity (Fowler & Lewis, 2014).
	Operational Overhead: Deployment, monitoring, and troubleshooting multiple services require robust tools and expertise (Nadareishvili et al., 2016).
Communication Overhead	**Network Latency**: Inter-service communication introduces delays.
	Data Serialization: Serialization/deserialization adds processing overhead.
Data Management Challenges	**Data Consistency**: Achieving consistency across distributed data stores is complex (Dragoni et al., 2017).
	Database Fragmentation: Managing schemas and aggregating data from multiple databases adds overhead (Nadareishvili et al., 2016).
Testing Difficulties	**Integration Testing**: Verifying interactions between services is challenging (Fowler & Lewis, 2014).
	End-to-End Testing: Complex environments complicate comprehensive testing.
Deployment and Release Management	**Service Coordination**: Ensuring compatibility during deployments requires careful planning.
	Deployment Automation: Setting up CI/CD pipelines for multiple services is resource-intensive (Dragoni et al., 2017).
Increased Operational Overhead	**Monitoring and Logging**: Aggregating logs from multiple services is complex (Nadareishvili et al., 2016).
	Resource Management: Managing resources for many services can lead to inefficiencies.
Security Concerns	**Attack Surface**: Increased communication channels raise security risks (Dragoni et al., 2017).
	Authentication and Authorization: Managing security across services requires robust solutions (Fowler & Lewis, 2014).
Service Discovery and Management	**Dynamic Discovery**: Ensuring that services can dynamically locate each other is critical.
	Configuration Management: Consistent and accurate configurations across services are essential (Nadareishvili et al., 2016).

1. Conclusion

Microservices architecture, while offering significant advantages, also presents several disadvantages that organizations need to consider. The increased complexity of managing multiple services, communication overhead, and data management challenges can impact development, deployment, and operational efficiency. Testing difficulties, deployment and release management complexities, and increased operational overhead further contribute to the challenges associated with microservices. Additionally, security concerns, service discovery and management issues, and cultural and organizational impacts must be addressed to successfully implement and maintain a microservices architecture. Understanding these disadvantages is crucial for organizations to make informed decisions about adopting microservices and develop strategies to mitigate potential issues.

F. DEVELOPMENT OPERATIONS

DevOps, a blend of development and operations practices, aims to improve collaboration, efficiency, and continuous delivery in SW development. When applied to microservices architecture, DevOps practices can significantly enhance the deployment, monitoring, and management of microservices-based systems. This section explores how DevOps integrates with microservices architecture, detailing the key practices, benefits, and challenges associated with this integration.

1. Introduction

DevOps is a cultural and technical movement that emphasizes collaboration between development and operations teams to streamline the SW development life cycle. It focuses on automating processes, improving communication, and enabling continuous delivery and integration. The primary goals of DevOps include reducing deployment times, increasing deployment frequency, and improving the overall quality of SW releases.

a. DevOps Practices in Microservices Architecture

Microservices architecture, characterized by decomposing applications into small, independently deployable services, benefits significantly from DevOps practices. Key practices include the following:

(1) Continuous Integration

Continuous integration involves regularly integrating code changes into a shared repository and running automated tests to detect integration issues early (Fowler, 2006). In a microservices environment, continuous integration pipelines are set up for each service, ensuring that changes are tested and integrated continuously. This practice helps maintain the quality and stability of each microservice while enabling rapid development and deployment.

(2) Continuous Development

Continuous development extends continuous integration by automating the deployment of code changes to production environments (Humble & Farley, 2010). For microservices, continuous development pipelines are designed to deploy individual services independently, allowing teams to release updates without affecting the entire system. This flexibility supports frequent releases and faster time-to-market.

(3) Infrastructure as Code

Infrastructure as code (IaC) involves managing infrastructure through code, enabling automated provisioning and configuration of resources (Morris, 2016). In microservices architecture, IaC tools like Terraform and Ansible are used to automate the setup and management of environments for each microservice. This approach ensures consistency across environments and simplifies scaling and maintenance.

(4) Automated Testing

Automated testing is crucial for ensuring the quality of microservices. It includes unit testing, integration testing, and end-to-end testing. Microservices architecture requires comprehensive testing strategies to verify the functionality and interactions of individual services. DevOps practices emphasize the automation of these tests to detect issues early and ensure reliable deployments.

(5) Monitoring and Logging

Effective monitoring and logging are essential for managing microservices. DevOps practices involve implementing centralized logging and monitoring solutions to gain visibility into the performance and health of each microservice (Morris, 2016). Tools like Prometheus and Elasticsearch, Logstash, Kibana Stack help collect, analyze, and visualize data from multiple services, facilitating troubleshooting and performance optimization.

(6) Collaboration and Communication

DevOps fosters collaboration between development and operations teams, which is crucial for managing microservices architecture. Teams work together to define service requirements, deployment strategies, and incident response plans. This collaboration ensures that microservices are developed, deployed, and maintained with a shared understanding of goals and responsibilities (Humble & Farley, 2010).

2. Benefits

The integration of DevOps practices with microservices architecture offers several benefits.

(1) Faster Time-to-Market

DevOps practices enable faster development and deployment cycles, allowing teams to release new features and updates more quickly. In a microservices architecture, independent deployment of services means that changes can be delivered without waiting for the entire system to be updated. This speed enhances competitiveness and responsiveness to market demands.

(2) Improved Quality and Reliability

Automated testing and continuous integration help identify and address issues early in the development process, reducing the risk of defects and outages in production (Beck, 2003). Monitoring and logging practices provide real-time insights into service performance, allowing teams to detect and resolve problems proactively.

(3) Enhanced Scalability and Flexibility

IaC and automated deployment practices support dynamic scaling and management of microservices environments. Teams can easily provision resources, scale services based on demand, and adapt to changing requirements (Morris, 2016). This flexibility enables organizations to handle varying workloads and optimize resource utilization.

(4) Increased Collaboration and Efficiency

DevOps practices promote collaboration between development and operations teams, breaking down silos and improving communication. This collaborative approach leads to more efficient workflows, faster problem resolution, and a shared understanding of goals and processes (Humble & Farley, 2010).

3. Challenges and Considerations

While DevOps offers significant benefits, integrating DevOps with microservices architecture also presents challenges.

(1) Complexity of Managing Multiple Services

Microservices architecture involves managing numerous services, each with its own CI/CD pipeline, infrastructure, and dependencies. Coordinating and maintaining these services can be complex and require robust DevOps practices and tools. Ensuring consistent configurations and managing inter-service interactions are key challenges.

(2) Need for Advanced Tooling and Automation

Implementing DevOps practices for microservices requires advanced tooling and automation capabilities. Setting up and maintaining CI/CD pipelines and monitoring systems and IaC frameworks can be resource-intensive and may require specialized expertise. Organizations need to invest in appropriate tools and training to effectively leverage DevOps practices.

(3) Cultural and Organizational Changes

Adopting DevOps practices involves cultural and organizational changes, including shifts in team dynamics and workflows. Organizations need to foster a culture of collaboration, continuous improvement, and shared responsibility (Humble & Farley, 2010). Managing these changes and aligning teams with DevOps principles can be challenging.

(4) Security and Compliance

Ensuring the security and compliance of microservices in a DevOps environment requires careful consideration. Automated deployments and frequent changes can introduce security risks if not properly managed. Implementing security best practices, such as automated security testing and secure coding practices, is essential (Morris, 2016).

4. Conclusion

DevOps practices significantly enhance the management and delivery of microservices architecture by promoting automation, collaboration, and continuous improvement. Key practices such as CI/CD, IaC, automated testing, and centralized monitoring contribute to faster time-to-market, improved quality, and enhanced scalability. However, integrating DevOps with microservices architecture also presents challenges related to complexity, tooling, cultural changes, and security. Organizations must address these challenges to fully leverage the benefits of DevOps and achieve a successful microservices implementation.

VI. UNMANNED SURFACE VESSEL INTEGRATED COMBAT SYSTEM: RAPID PROTOTYPING OF A MODERNIZED COMBAT SYSTEM

In the fall of 2020, NSWC Dahlgren, Computing Infrastructure Group, was tasked with developing and delivering prototype equipment capable of running AWS elements and required functions to support the large unmanned surface vessel (LUSV) program. The initial fielding of this system required GFE for the initial LUSV platform in late 2023.

Early direction from PEO IWS 80, who is the Major Program Manager (MPM) for delivery of the LUSV / USV Integrated Combat System (ICS) focused on delivery of computing infrastructure, and an ICS that delivered virtualized combat computer programs and technologically relevant and modern network, processing, and storage (NPS) HW (HW).

At the time technology Insertion 16 (TI16), a FARS based acquisition effort was underway to supply USN surface forces with NPS. Although the HW selected for TI16 was already near obsolescence it was not t in full rate production. In fact, by the time LUSV fielded initial GFE in late 2023 the NPS products which made up this TI16 based infrastructure would have been over 8 years old.

A. DEPARTMENT OF DEFENSE FEDERAL ACQUISITION REGULATION–BASED ACQUISITIONS

Previous iterations of NPS and CI for surface USN programs, such as TI12, and TI16, resulted in HW that was plagued by obsolescence issues and delivery of HW that was generations old before ever reaching in-service platforms. Much of this is a byproduct of utilizing a lengthy FAR-based acquisition system.

The DoD operates under the Federal Acquisition Regulation (FAR) framework to procure goods and services. FAR-based acquisitions serve as the regulatory backbone for ensuring that DoD procurement activities comply with federal laws, maintain accountability, and achieve value for taxpayers. While the FAR framework has distinct advantages, it also comes with limitations that may hinder efficiency and adaptability in certain scenarios.

B. OVERVIEW OF FEDERAL ACQUISITION REGULATION–BASED ACQUISITIONS

The DoD utilizes the FAR as the primary framework for managing acquisitions. This system governs the procurement of goods, services, and construction to ensure fairness, accountability, and transparency in the expenditure of public funds. However, while the FAR provides a robust structure, it also imposes significant limitations that can hinder efficiency, innovation, and flexibility. This document explores the core aspects of DoD FAR-based acquisitions, highlighting their strengths and inherent challenges.

a. Key Features

(1) Transparency and Accountability

The FAR mandates stringent oversight, ensuring that procurement processes are conducted transparently and that public funds are spent responsibly. This includes detailed documentation, audits, and reviews to minimize the risk of fraud and abuse (GAO, 2019). Standardized processes and regulations enhance public trust by providing clear rules for contractors and government agencies alike.

(2) Standardization

The FAR establishes uniform procedures across federal agencies, reducing variability and simplifying compliance for contractors. This standardization is designed to facilitate smoother interactions between agencies and suppliers.

(3) Competition and Cost Efficiency

FAR promotes open and fair competition, enabling the government to receive better value through competitive bidding processes. This competition often results in lower costs and higher-quality outcomes (Fowler, 2014). • Policies such as the "lowest price technically acceptable" (LPTA) ensure cost-effective acquisitions, though sometimes at the expense of innovation.

(4) Support for Small Businesses

FAR includes provisions to support small and disadvantaged businesses, ensuring they have opportunities to participate in government contracts.

(5) Risk Mitigation

FAR incorporates clauses to manage risks, such as performance bonds and termination rights, safeguarding government interests in case of contractor non-performance.

b. Limitations

Despite its strengths, the FAR framework presents several challenges that affect the DoD's ability to meet its evolving needs.

(1) Complexity and Bureaucracy

The FAR's extensive regulations and documentation requirements often lead to administrative burdens, particularly for small businesses and non-traditional contractors (Nadareishvili et al., 2016). Navigating these complexities requires significant expertise and resources, which can delay procurement processes.

(2) Limited Flexibility

FAR-based acquisitions are often rigid, making it challenging to adapt to changing requirements or emerging technologies. This rigidity is particularly problematic for fast-paced industries like technology and cybersecurity (Dragoni et al., 2017). The focus on compliance sometimes overshadows the need for innovation and rapid deployment.

(3) Lengthy Procurement Timelines

The structured nature of FAR processes can result in prolonged procurement cycles, delaying the delivery of critical goods and services. This is especially problematic in defense scenarios where timely acquisition is essential for operational readiness.

(4) Innovation Challenges

FAR's preference for proven technologies and cost efficiency often discourages contractors from proposing cutting-edge solutions. This approach can stifle innovation (Fowler, 2014). While alternative mechanisms like OTAs exist, their adoption within traditional FAR systems has been limited.

(5) Inter-Service Communication Overhead

FAR-based acquisitions rely heavily on inter-agency and inter-service coordination, which can introduce communication delays and inefficiencies (Dragoni et al., 2017). Networked communication between services, often necessary for complex acquisitions, increases the risk of errors and inconsistencies.

(6) Data and Cybersecurity Challenges

Managing cybersecurity across multiple contracts and vendors poses significant challenges. While the FAR includes security provisions, emerging threats necessitate additional frameworks.

(7) Operational and Monitoring Overhead

FAR-based systems require robust tools and practices for monitoring and auditing, which can be resource-intensive (Fowler, 2014). Aggregating and analyzing data from multiple contracts often necessitates advanced infrastructure, increasing costs.

c. Addressing the Limitations

(1) Adoption of Alternative Mechanisms

The DoD has increasingly turned to OTAs for flexibility in research and development projects, allowing faster acquisition of innovative solutions (GAO, 2019). Greater integration of these mechanisms within the FAR framework could balance the need for accountability with the demand for agility.

(2) Streamlining Processes

Simplifying compliance requirements, particularly for small businesses, can encourage broader participation and reduce administrative burdens. Leveraging technology to automate documentation and reporting processes can enhance efficiency.

(3) Enhanced Focus on Innovation

Introducing more flexible evaluation criteria that prioritize long-term value and innovation over cost alone can incentivize contractors to propose advanced solutions. Increasing investment in contractor education about FAR alternatives like OTAs can foster a culture of innovation. Table 6 provides a summary of FARS and the limitations.

Table 6. Summary of Department of Defense Federal Acquisition Regulation–Based Acquisitions and Limitations

Aspect	Details
Key Features	
Transparency and Accountability	FAR mandates strict oversight, documentation, audits, and reviews to ensure public funds are responsibly spent (GAO, 2019).
Standardization	Uniform procedures reduce variability and simplify compliance for contractors across federal agencies.
Competition and Cost Efficiency	Encourages open and fair competition, leveraging policies like LPTA for cost-effective acquisitions (Fowler, 2014).
Support for Small Businesses	Includes provisions to ensure small and disadvantaged businesses can access government contracts.
Risk Mitigation	Incorporates clauses for performance bonds and termination rights to safeguard government interests.
Limitations	
Complexity and Bureaucracy	Extensive regulations and documentation create administrative burdens, especially for small businesses (Nadareishvili et al., 2016).
Limited Flexibility	Rigid processes hinder adaptation to evolving requirements and emerging technologies (Dragoni et al., 2017).
Lengthy Procurement Timelines	Structured procedures result in delays, impacting operational readiness in critical scenarios.
Innovation Challenges	Focus on proven technologies discourages cutting-edge solutions; limited adoption of mechanisms like OTAs (Fowler, 2014).

Aspect	Details
Inter-Service Communication Overhead	Coordination between agencies introduces delays and risks of errors in complex acquisitions (Dragoni et al., 2017).
Data and Cybersecurity Challenges	Managing security across vendors poses risks; FAR includes provisions, but emerging threats require additional frameworks.
Operational and Monitoring Overhead	Monitoring multiple contracts increases resource demands and costs, requiring advanced infrastructure (Fowler, 2014).
Addressing Limitations	
Adoption of Alternative Mechanisms	Greater integration of OTAs for flexibility in research, development, and innovative procurement (GAO, 2019).
Streamlining Processes	Simplifying compliance for small businesses and leveraging automation to reduce administrative burdens.
Enhanced Focus on Innovation	Introducing flexible evaluation criteria that prioritize long-term value and innovation.
Contractor Education	Providing education on FAR alternatives like OTAs to foster innovative and agile procurement practices.

C. RAPID PROTOTYPING

In early 2021USV ICS CI prototyping efforts were accelerated to support initial capability testing onboard a pair of commercial vessels adapted to deliver initial autonomy and vessel control, and USC ICS capability and refine these critical program enablers in the years prior to delivery of the LUSV platform. This prototyping schedule adjustment required the delivery of ICS CI GFE HW to The Overlord Unmanned Surface Vessel (OUSV) # 4 in October 2021. To meet this condensed design, development, and delivery timeline the USC ICS CI team leveraged and updated existing OTA, established to rapidly produce CI and IaaS SW suite for the Virtual Pilot Ship (VPS) prototyping effort on the USS Monterey.

1. Use of Other Transactional Authority to Enable Speed to Acquisition

Other transactional authorities (OTAs) are a significant aspect of the United States government's acquisition and procurement system, particularly within the DoD. These authorities enable the government to enter into agreements that are more flexible and less restrictive than traditional contracts. OTAs have gained prominence in recent years due to their ability to promote innovation, streamline the procurement process, and facilitate the acquisition of emerging technologies.

In traditional government procurement, agencies often rely on FAR-based contracts. These contracts are designed to provide a structured, uniform approach to procurement but can be rigid and slow. In contrast, OTAs offer a more flexible alternative, enabling government agencies to engage with private industry, academic institutions, and nonprofit organizations in ways that are less encumbered by traditional federal contracting rules.

OTAs are used primarily in situations where traditional contracts would not be as effective, such as in areas where rapid technological advancements are required, or where the government wants to collaborate more effectively with innovative companies or non-traditional defense contractors. OTAs can be used for research, development, demonstration, and prototype work.

OTAs are codified in the U.S. Code, specifically under Title 10 and Title 41, which provide the legal basis for their use by federal agencies, particularly the DoD. These authorities are not part of the FAR, which governs most government procurement. There are three main types of OTAs:

- **Prototype Projects:** These are used primarily by the DoD to engage with non-traditional defense contractors to rapidly develop prototype systems and technologies.
- **Research Projects:** This category supports research and development efforts, allowing agencies to work with a wide range of partners, including universities, nonprofit organizations, and industry leaders.
- **Production:** These are less common and are used when the government wishes to transition from a prototype to full production, typically after successful demonstration and evaluation.

The primary legal authority for OTAs comes from:

- **10 U.S.C. 2371b:** This statute authorizes the DoD to use OTAs for prototypes, aiming to streamline the development process and encourage innovation from non-traditional contractors.
- **41 U.S.C. 1901:** This provides authority for broader OTA use, beyond defense applications, to include other federal agencies.

OTAs can also be used in situations where agencies need to engage in collaborative agreements, such as when working with industry consortia, universities, or small businesses.

a. Types

OTAs are flexible instruments that can take different forms depending on the nature of the project. While they all share some common features, such as their ability to allow the government to work with a broad range of entities, there are specific distinctions between the types of OTAs used for different purposes.

(1) Prototype Projects

OTAs for prototype projects are perhaps the most well-known and widely used type of OTA. These authorities allow the government to engage with contractors to develop prototypes of new technologies or systems. The use of OTAs for prototyping is particularly significant in the defense sector, where the pace of technological advancement is fast, and traditional procurement processes may not be well-suited to fostering innovation.

- **Purpose:** The primary purpose of OTAs for prototypes is to enable rapid, flexible, and cost-effective development of innovative solutions. These OTAs allow the government to work with non-traditional defense contractors who may not have experience with government contracts, but who possess the technical expertise to develop novel systems or technologies.
- **Advantages:** OTAs for prototypes allow the government to avoid the lengthy and often cumbersome processes associated with traditional contracts. They provide greater flexibility in terms of cost-sharing,

intellectual property rights, and other terms of the agreement. This flexibility is crucial when working with emerging technologies or companies that may have unique requirements or constraints.

- **Collaboration with Non-Traditional Contractors:** A significant feature of prototype OTAs is their ability to facilitate collaboration with non-traditional defense contractors. The DoD has recognized that non-traditional contractors, such as small businesses, startups, and companies from outside the defense industry, often bring innovative solutions to the table. However, traditional procurement processes can discourage these entities from engaging with the government due to their complexity and rigid requirements.
- **Example:** An example of an OTA for prototype development is the DoD's engagement with small technology companies to develop new SW, drones, or advanced sensors that can be used in combat scenarios.

(2) Research Projects

Research-focused OTAs provide a framework for government agencies to engage with a wide range of entities, including universities, nonprofit research institutions, and private industry, to support scientific research and technological development.

- **Purpose:** These OTAs are primarily used to fund research projects that are designed to advance knowledge in specific fields, such as aerospace engineering, cybersecurity, or biotechnology. The government can leverage the expertise of academic and private sector institutions to explore new technologies, scientific theories, and experimental designs.
- **Flexibility in Terms:** OTAs for research projects can offer more flexibility than traditional grants or contracts. The terms of these OTAs are negotiated based on the specific research objectives and can be tailored to meet the needs of both the government and the research institution. The

agreements are less prescriptive than traditional contracts, allowing for greater innovation and experimentation.

- **Collaboration with Academia:** One of the key advantages of OTAs for research projects is that they enable the government to tap into the research capabilities of universities and academic institutions. These institutions may not typically engage in government contracts, but an OTA allows them to collaborate on research that aligns with their academic goals while meeting government needs.
- **Example:** The government may use an OTA to fund a research project at a university focused on developing new artificial intelligence techniques for cybersecurity.

(3) Production

OTAs for production are used less frequently but are important when a government agency wants to transition from prototype development to full-scale production.

- **Purpose:** These OTAs allow for the production of goods or services that are initially developed under a prototype OTA. After successful demonstration and evaluation, the government can use an OTA to scale up production, particularly when traditional procurement processes would be too slow or rigid to meet the demand.
- **Collaboration with Industry:** OTAs for production can be used to engage with traditional defense contractors, as well as non-traditional manufacturers who are capable of scaling production quickly. These agreements provide flexibility in terms of cost-sharing, intellectual property, and other factors that might otherwise hinder production under a traditional contract.
- **Example:** If a prototype for a new unmanned aerial vehicle (UAV) is successfully demonstrated, the DoD may enter into an OTA for production to scale up manufacturing and deploy the UAV to military units.

b. Advantages

OTAs provide several key advantages, particularly in fostering innovation, accelerating the procurement process, and enabling the government to work with a broader range of organizations. These benefits have made OTAs an increasingly popular choice for federal agencies, especially the DoD.

(1) Flexibility

One of the main advantages of OTAs is their flexibility. Unlike traditional contracts, which are subject to the FAR, OTAs offer the ability to negotiate terms and conditions that are tailored to the needs of the project. This includes greater flexibility in how funds are allocated, how intellectual property is managed, and the specific deliverables that are required.

(2) Encouraging Innovation

OTAs are particularly beneficial in encouraging innovation. The traditional government procurement process is often slow and bureaucratic, which can stifle innovation and discourage companies from proposing cutting-edge solutions. In contrast, the flexibility provided by OTAs allows the government to engage with startups, small businesses, and non-defense companies that might have the technical expertise to develop groundbreaking technologies but may lack the experience or resources to navigate the traditional procurement system.

The ability to engage with a broader range of contractors also means that the government can access the latest technological advances, such as artificial intelligence, cybersecurity, and quantum computing, much faster than with traditional contracts.

(3) Reduced Administrative Burden

OTAs can reduce the administrative burden on both the government and contractors. Traditional contracts often require extensive documentation, compliance checks, and reporting, all of which can slow down the procurement process. With OTAs, the administrative requirements are less burdensome, which means that agreements can be negotiated and executed more quickly. This reduces the time it takes for the government to access new technologies and capabilities, which is critical in rapidly evolving fields like defense and cybersecurity.

(4) Promoting Collaboration

OTAs facilitate collaboration between the government and a wide range of entities, including non-traditional defense contractors, small businesses, universities, and research institutions. These collaborations can lead to innovative solutions that might not have emerged from more traditional government-contractor relationships. By working with diverse partners, the government can ensure that it is leveraging the full range of expertise and resources available in the private sector and academia.

c. Disadvantages

Despite their many advantages, OTAs also present challenges and limitations. These can include concerns related to competition, cost control, and ensuring that the government's interests are protected.

(1) Lack of Competition

One of the primary concerns with OTAs is that they may not always promote competition. Since OTAs are negotiated agreements rather than open bidding processes, there is a risk that a small number of contractors could be repeatedly awarded OT agreements without sufficient competition. While OTAs can be used in cases where competitive bidding is impractical or unnecessary, there is a need for safeguards to ensure that competition is maintained in cases where it is feasible.

(2) Cost Control

While OTAs offer flexibility, they can also pose challenges in terms of cost control. Traditional contracts typically have well-defined pricing structures and requirements for cost control, whereas OTAs are negotiated agreements that may not have the same level of cost transparency. This can lead to challenges in monitoring and controlling costs, especially for larger or more complex projects.

(3) Lack of Oversight and Accountability

Because OTAs are not subject to FAR, they can sometimes lack the level of oversight and accountability that is built into traditional procurement processes. This can lead to concerns about the effectiveness and efficiency of OTA-based projects, particularly when dealing with large sums of taxpayer money. It is essential to have mechanisms in place to ensure that OT agreements are properly managed, and that the government is getting value for expenditures..

(4) Complexity in Implementation

While OTAs provide flexibility, they also require skilled contracting officers and project managers who are familiar with the unique characteristics of OTA agreements. The lack of familiarity with OTAs in some government agencies can make it difficult to effectively implement these authorities, particularly in situations where rapid procurement is needed.

d. Conclusion

OTAs represent a critical tool in modernizing government procurement processes, particularly within the DoD. By offering greater flexibility, encouraging innovation, and streamlining the procurement process, OTAs enable the government to work more effectively with industry to delivery technologically relevant HW and SW to pace emerging threats and mature new defense concepts like the USV ICS CI. Table 7 summarizes the key aspect of an OTA.

Table 7. Summary of Other Transaction Authorities in the Department of Defense Procurement System

Aspect	Details
Introduction	OTAs provide a flexible alternative to FAR-based contracts, allowing rapid innovation and streamlined procurement processes.
Purpose	Facilitate collaboration with private industry, academia, and non-profits for research, prototyping, and production.
Legal Framework	Codified under Title 10 and Title 41 of the U.S. Code, with key statutes like 10 U.S.C. 2371b for prototypes and 41 U.S.C. 1901 for broader applications.

Aspect	Details
Types	
Prototype Projects	Focuses on developing innovative prototypes. Enables rapid development, cost-sharing flexibility, and collaboration with non-traditional contractors.
Research Projects	Supports research and development with flexible terms. Often engages universities and research institutions for advancing scientific knowledge.
Production	Used to transition from prototype to full production, especially when traditional procurement is too slow or rigid.
Advantages	
Flexibility	Tailored agreements allow negotiated terms, including cost, intellectual property, and deliverables.
Encouraging Innovation	Engages startups, small businesses, and non-defense contractors to bring cutting-edge technologies to government projects.
Reduced Administrative Burden	Fewer documentation and compliance requirements compared to FAR contracts, expediting agreements.
Promoting Collaboration	Facilitates partnerships with diverse entities, leveraging expertise from industry, academia, and research institutions.
Rapid Procurement	Accelerates access to emerging technologies, critical for fast-evolving fields like defense and cybersecurity.
Challenges and Limitations	
Lack of Competition	Negotiated agreements may reduce competitive bidding, increasing the risk of awarding contracts to a narrow pool of contractors.
Cost Control	Limited transparency in negotiated pricing structures can complicate monitoring and controlling costs.
Lack of Oversight	Absence of FAR regulations may reduce accountability, creating risks for inefficiencies and waste.
Implementation Complexity	Requires skilled contracting officers familiar with OTA agreements, which can be a challenge in certain agencies.

2. Use of Other Transactional Authority in Unmanned Surface Vessel Integrated Combat System Computing Infrastructure Prototyping

In early 2021 an addendum statement of work (SOW) was written and ratified to design, develop, deliver, and integrate into USV prototyping vessels a USV ICS CI based on the latest available NPS HW and IaaS SW products. Needing to deliver the USV ICS CI HW/IaaS suite to a developmental lab at the NWSC-Dahlgren in advance of the October 2021 In Yard Need Date for ICS GFE this rapid prototyping OTA required delivery of these systems 6 months after design.

3. Summary and Key Aspects of the Unmanned Surface Vessel Integrated Combat System Other Transactional Authority Statement of Work

Current USN combat management systems are highly complex, outdated, and tightly linked to specific HW, creating costly and cumbersome upgrades. Existing paradigms lead to inefficient HW use, redundancy, and logistical challenges. An IaaS solution is required to address these issues, enabling scalable, modular, and efficient computing infrastructure. This solution must support dynamic allocation of compute and storage resources, function in isolated environments, and meet specific USN CS requirements.

The contractor must adopt an agile development methodology, enabling iterative learning and adaptation to meet future CS requirements. Collaboration with USN personnel is essential for integration, validation, and system refinement.

a. Develop an IaaS Prototype Framework

- Scalable and upgradable infrastructure that decouples HW from SW.
- Support remote management and initialization in low-bandwidth, disconnected environments.
- Consolidate current cabinet configurations into a maximum of three air-cooled cabinets for NPS with a UPS system.

b. Support CSs

- Enable operation of AWS and Tomahawk Weapon System (TWS).
- Address challenges of unmanned CS computing infrastructure.

c. Technical Focus Areas

- Modular and open-architecture design to maximize use of COTS HW.
- Ensure infrastructure is secure and supports multi-tenant service environments.

- Provide UPS configurations for operational sustainability during power loss.

d. General and Technical Requirements Prototype Development

- Develop CI cabinets for the LUSV and land-based operations.
- Design UPS systems to support operations during power loss for various durations (e.g., 15 minutes to 48 hours).
- Incorporate advanced remote initialization and management features.

e. Studies and Prototyping

- Investigate technologies for remote operation, intrusion detection, and anti-tamper measures.
- Explore modular HW designs for efficient CI refresh cycles.

f. Deliverables

- USV ICS CI / IaaS Suites.
- Interface Control Document, Software Production Specification (SPS), Bill of Materials (BOM), and final technical reports.
- Training materials for users and administrators.

g. Post-Delivery Support

- Technical support for troubleshooting, maintenance, and validation of delivered systems.

4. Small Empowered Teams to Enable Speed to Design, Development, Delivery, and Ship Integration

To meet the challenging timeline for the USV ICS CI prototyping and fielding effort, a small team approach was adopted. A government team consisting of 4 CS and CI experts with experience with previous CI development initiatives, extensive shipboard and ship integration, Technical Data Package (TDP) development, mechanical, reliability, and

USN CS maintenance backgrounds was formed. This small team partnered with a handful of industry experts within the OTA organization to meet program objectives. The

a. Benefits and Drawbacks of a Small Team Approach to Technology Project Development

The approach to technology project development has evolved significantly over the years. While large teams were historically the norm for handling complex projects, the small team approach has gained popularity due to its unique advantages. However, it is not without its challenges. This section explores the benefits and drawbacks of a small team approach to technology project development, providing a comprehensive analysis of its impact on project success, innovation, and efficiency.

Benefits of a Small Team Approach

b. Enhanced Communication

One of the most significant advantages of small teams is the ease of communication. With fewer members, it is simpler to align objectives, share updates, and resolve misunderstandings. Team members can quickly exchange ideas, leading to more effective collaboration and faster decision-making. This streamlined communication is particularly beneficial in technology projects, where rapid iterations and adaptability are often required.

c. Agility and Flexibility

Small teams are inherently more agile than larger ones. They can adapt to changes in project requirements, pivot strategies, and implement new ideas with minimal bureaucratic delays. This flexibility is especially valuable in technology projects that operate in fast-paced or uncertain environments, such as startups or emerging technologies.

d. Greater Accountability

With fewer individuals, each team member has a clearer understanding of their responsibilities and a greater sense of ownership over their tasks. This heightened accountability can lead to higher-quality work, as team members are directly invested in the success of the project. Moreover, it reduces the likelihood of tasks being neglected or overlooked.

e. Cost-Effectiveness

Smaller teams typically require fewer resources, making them a cost-effective option for organizations. Reduced staffing means lower salaries, fewer tools and SW licenses, and smaller physical workspaces. For technology projects with limited budgets, this cost efficiency can be a deciding factor in choosing the small team approach.

f. Improved Innovation and Creativity

Small teams foster closer relationships and a more intimate work environment, which can encourage open discussions and creative problem-solving. Team members are more likely to contribute ideas, challenge assumptions, and experiment with innovative solutions. This dynamic is crucial for technology projects that rely on breakthrough innovations.

g. Faster Decision-Making

Decision-making processes in small teams are typically less bureaucratic than in larger teams. With fewer layers of approval and less need for extensive consultation, small teams can make decisions more quickly. This speed can be critical in technology projects, where time-to-market often determines success.

5. Drawbacks of a Small Team Approach

a. Limited Expertise and Resources

A smaller team means fewer people to bring diverse skills and perspectives to the table. Technology projects often require specialized knowledge in areas like SW development, cybersecurity, and data analysis. A small team may lack the breadth of expertise needed to address complex challenges, potentially leading to slower progress or lower-quality outcomes.

b. Higher Workload for Team Members

In small teams, each member must handle a larger share of the workload. This can lead to burnout, decreased morale, and reduced productivity over time. High workloads can also make it challenging for team members to dedicate time to skill development or long-term strategic planning.

c. Risk of Knowledge Silos

With fewer individuals, knowledge about specific aspects of the project may become concentrated in just one or two people. If these individuals leave the team or are unavailable, critical knowledge gaps can arise, jeopardizing the project's progress and continuity.

d. Vulnerability to Disruptions

Small teams are more vulnerable to disruptions caused by unexpected events, such as illness, personal emergencies, or turnover. Losing even a single team member can have a significant impact on the team's ability to meet deadlines and maintain productivity. The small team assembled for USV ICS CI were enabled to focus solely on this development.

e. Challenges in Scaling

While small teams excel in handling focused, short-term projects, they may struggle to scale their efforts as project demands grow. Larger, more complex technology projects may require additional personnel, infrastructure, and coordination, which small teams may find difficult to manage effectively.

f. Dependence on Individual Performance

In a small team, the performance of each member has a disproportionately large impact on the overall success of the project. If one member underperforms or fails to deliver, it can significantly hinder the team's progress. This dependence can create added pressure and stress for individual members.

Table 8. Assessing Small Team Approached to Project Success

Benefits and Drawbacks of a Small Team Approach		
Category	**Benefits**	**Drawbacks**
Communication	Streamlined, fosters collaboration, quicker decision-making	Risk of over-reliance on few channels, potential for echo chambers
Agility and Flexibility	Quickly adapts to changes and new ideas	May lack resources to handle rapid or large-scale shifts
Accountability	Clear ownership and high responsibility	Pressure on individuals, potential for overwork
Cost-Effectiveness	Fewer resources needed	High risk if critical skills or tools are missing
Innovation and Creativity	Close-knit team dynamics foster ideation	Limited external perspectives may hinder diverse or groundbreaking innovations
Resilience	Quick recovery from small-scale issues	Vulnerable to major disruptions or key member losses

6. Understanding the Requirement

In order to move fast and deliver the relevant capability required, it is crucial and foundational to understand the requirements of the system being developed. Effective requirements definition is essential for ensuring that a system aligns with operational needs and delivers desired capabilities. In military systems, requirements serve as the foundation for system design, development, and testing, and provide a clear roadmap for engineers and developers, guiding the system architecture and design. By accurately defining system requirements, the USN can ensure that it is developing a system that meets operational goals, such as mission readiness, interoperability, and survivability.

Furthermore, a clear set of well-documented requirements is instrumental in minimizing project risks. Without proper requirements, projects often experience scope creep (where the scope of the project expands beyond the original plan) and cost overruns and delays as changes to the design are needed as a fuller understanding of the requirements are gained. These risks are particularly significant in the development of defense systems, like the USV ICS CI, where budget constraints and tight timelines were present. The ability to track and manage requirements throughout the system development life cycle ensures that these risks are mitigated. Clear requirements also foster better communication among all stakeholders involved in the system's development.

a. Key Components of Requirements Definition

The process of requirements definition in systems development involves several key components, including functional and non-functional requirements, technical specifications, stakeholder engagement, and comprehensive documentation.

(1) Functional Requirements

These specify what the system must do. For instance, in the case of a naval CS, functional requirements would outline the system's ability to detect, track, and engage enemy targets (Defense Acquisition University, 2010). Functional requirements define the core capabilities and tasks the system must accomplish to support operational missions.

(2) Non-functional Requirements

These focus on how the system performs its functions. They may address performance attributes such as system reliability, response time, availability, and security (Ward & McCune, 2018). Non-functional requirements ensure that the system not only performs its tasks but does so in a manner that meets the USN's standards for efficiency, safety, and security.

(3) Technical Specifications

These describe the detailed technical aspects of the system, including HW, SW, interfaces, and performance thresholds (Department of the Navy, 2020). Technical specifications are necessary to ensure that the system integrates with existing infrastructure and meets specific operational constraints, such as environmental conditions or power consumption limitations.

(4) Stakeholder Involvement

Effective requirements elicitation involves gathering input from all relevant stakeholders, including military personnel, contractors, end users, and subject matter experts. Stakeholder engagement is essential for capturing the full spectrum of system needs, from operational capabilities to technical specifications.

(5) Documentation and Traceability

All requirements must be carefully documented and managed throughout the system development life cycle. Traceability refers to the ability to track each requirement from its origin to its implementation and testing (Defense Acquisition University, 2010). This ensures that no requirement is overlooked and that any changes to the system can be properly analyzed and evaluated.

b. Challenges in Requirements Definition

While the importance of requirements definition is clear, the process is not without its challenges. Some of the common difficulties in defining requirements for military systems include:

(1) Ambiguity and Lack of Clarity

Requirements that are vague or poorly defined can lead to confusion during system development. Ambiguous language can result in misinterpretation of system objectives, causing delays and rework (Ward & McCune, 2018). In the context of the USN, where operational needs can be complex and dynamic, precise language is essential.

(2) Conflicting Stakeholder Interests

Different stakeholders often have different priorities. For example, military personnel may emphasize operational capability, while engineers might focus on system performance and technical feasibility.

(3) Changing Requirements

As system development progresses, new requirements may emerge, or existing ones may need to be revised due to changes in technology, mission needs, or budget constraints. Managing changes to requirements can be challenging, especially in large, complex defense projects (Defense Acquisition University, 2010).

(4) System Complexity

Modern naval systems are highly complex, integrating numerous technologies, platforms, and operational domains. This complexity makes it difficult to capture all system requirements accurately. Additionally, systems must be designed to withstand evolving threats, which requires a high level of flexibility in the requirements definition process (Department of the Navy, 2020).

c. Methodology of Requirements Definition

During the early prototype phase of the USV ICS CI development, the requirements were still emerging. By thoroughly analyzing the requirements of previous AWS and ICS CI systems, the team was able to derive a comprehensive set of requirements that allowed the development to progress. The approach adopted for the USV ICS CI involved ensuring that it met all functional and performance requirements of the AWS, while modernizing the HW and form factor of core components. Additionally, the team incorporated a virtualization platform to support the modernization of application SW. This combination of approaches provided the team with a practical and achievable set of requirements, enabling them to meet the compressed development timelines.

7. Best Practices for Navy System Prototyping

The DoD Prototyping Guidebook serves as a comprehensive resource for defense acquisition professionals, providing guidance on the effective use of prototyping within the DoD's acquisition framework. Prototyping is a critical tool in reducing technical risk, refining requirements, validating designs, and accelerating the delivery of capabilities to the warfighter. This section summarizes the material and guidance contained in this guidebook.

a. Purpose and Scope

The guidebook emphasizes the strategic value of prototyping in fostering innovation and ensuring technological superiority. It outlines best practices for planning, executing, and transitioning prototypes, aiming to enhance decision-making and program outcomes across various acquisition pathways.

b. *Key Concepts*

Prototyping involves creating a preliminary version of a system or component to evaluate feasibility, explore design options, and identify potential issues before full-scale development.

c. *Types of Prototypes*

- **Technology Demonstrators:** Showcase new technologies to assess their maturity and integration potential.
- **Operational Prototypes:** Developed to evaluate system performance in operational environments.
- **Risk Reduction Prototypes:** Address specific technical or integration risks to inform development decisions.

d. *Prototyping Objectives*

- **Risk Mitigation:** Identify and address technical uncertainties early in the acquisition process.
- **Requirements Refinement:** Inform and validate user requirements through iterative development.
- **Technology Maturation:** Advance the readiness levels of critical technologies.
- **Concept Exploration:** Assess alternative solutions to meet mission needs.

e. *Prototyping Process*

(1) Planning

- **Define Objectives:** Clearly articulate the goals and success criteria for the prototype.
- **Stakeholder Engagement:** Involve end users, technical experts, and decision-makers to ensure alignment.

- **Resource Allocation:** Secure necessary funding, personnel, and facilities.

(2) Execution

- **Design and Development:** Employ agile methodologies to iteratively build and refine the prototype.
- **Testing and Evaluation:** Conduct assessments to measure performance against objectives.
- **Documentation:** Maintain detailed records of design decisions, test results, and lessons learned.

(3) Transition

- **Assessment:** Evaluate the prototype's success in meeting objectives and its potential for further development.
- **Decision-Making:** Determine the appropriate path forward, whether transitioning to a program of record, additional prototyping, or termination.
- **Integration:** Plan for the incorporation of successful prototypes into existing systems or new acquisition programs.

f. Best Practices

- **Early and Continuous User Involvement:** Engage end users throughout the prototyping process to ensure the solution meets operational needs.
- **Iterative Development:** Adopt flexible development approaches that allow for rapid iteration and incorporation of feedback.
- **Risk Management:** Continuously identify, assess, and mitigate risks to prevent issues from escalating.
- **Collaboration:** Foster partnerships with industry, academia, and other government agencies to leverage diverse expertise and resources.

- **Metrics and Evaluation:** Establish clear metrics to assess prototype performance and inform decision-making.

g. *Challenges and Considerations*

- **Resource Constraints:** Balancing limited resources while pursuing multiple prototyping efforts.
- **Transition Planning:** Ensuring successful prototypes are effectively integrated into acquisition programs.
- **Intellectual Property:** Navigating IP rights to facilitate collaboration and future development.
- **Regulatory Compliance:** Adhering to acquisition regulations while maintaining flexibility in prototyping efforts.

h. *Conclusion*

The DoD Prototyping Guidebook underscores the importance of prototyping as a strategic tool in defense acquisition. In the development of USV ICS CI prototype the team considered these factors, and the OTA was structured to support these objectives.

Table 9. Department of Defense Guidance Summary on Prototype Systems

Category	Key Points
Purpose and Scope	Provides guidance on effective use of prototyping to foster innovation, reduce risks, and ensure technological superiority in defense acquisition.
Prototyping Definition	Creation of preliminary system versions to evaluate feasibility, explore design options, and address potential issues.
Types of Prototypes	**Technology Demonstrators**: Assess new technologies.
	Operational Prototypes: Evaluate system performance in real environments.
	Risk Reduction Prototypes: Mitigate technical or integration risks.
Objectives	Mitigate technical risks.
	Refine user requirements.
	Mature critical technologies.
	Explore alternative solutions.
Prototyping Process	**Planning**: Define objectives, engage stakeholders, allocate resources.
	Execution: Design iteratively, test, document outcomes.

Category	Key Points
	Transition: Evaluate success, decide on next steps, and integrate into programs.
Best Practices	Engage users early and continuously.
	Use iterative development approaches.
	Identify and mitigate risks continuously.
	Collaborate with industry, academia, and agencies.
	Define clear metrics for evaluation.
Challenges	Balancing limited resources.
	Effective transition planning.
	Managing intellectual property.
	Ensuring regulatory compliance.
Conclusion	Prototyping is a strategic tool for innovation and risk reduction. Following the guidebook enhances decision-making and accelerates capability delivery.

8. Deliver an Excess Capability to Support Development and Redundancy for Reliability

The USV ICS CI development marked the first time an AWS based CS was deployed in a virtualized environment. The development team, to ensure that CI capability could support CS SW development the decision was made to deliver excess capacity.

Long-term plans for prototype iteration leading to final system designed included assessment of end item SW capacity requirements, coupled with redundant capacity to support maintenance free operating time, to “right size” the system prior to production. The nature of the USV ICS CI as an unmanned platform was also considered when designing the system.

The “Reliability and Maintainability Engineering Guidebook” addresses the concept of designing reliability through redundancy as a key principle for ensuring that systems are capable of maintaining their functionality and performance despite potential failures of individual components. Redundancy is widely employed in systems engineering to increase the reliability and availability of systems, especially in critical applications such as aerospace, defense, telecommunications, and industrial processes. This section summarizes what the guidebook says about designing reliability through redundancy.

a. The Role of Redundancy in Reliability Engineering

Redundancy involves incorporating extra components or systems into the design to take over the function of failed components, thus preventing the entire system from failing when one part malfunctions. It is one of the primary strategies to enhance system reliability and ensure continuous operation.

In reliability engineering, redundancy is often used in mission-critical systems where system failure can result in catastrophic consequences. The guidebook emphasizes that the goal of redundancy is to improve the system's mean time between failures (MTBF) and availability.

b. Types of Redundancy

The guidebook outlines different types of redundancy, each tailored to address specific needs of reliability and maintainability in a system:

(1) Active Redundancy

In active redundancy, multiple components or systems perform the same function simultaneously. If one component fails, the others continue to function without disruption.

- **Example:** In an aircraft, active redundancy might involve having two engines where both engines operate simultaneously, and if one fails, the other takes over the entire load. Both engines are continuously in operation, sharing the load, which maximizes operational reliability.
- **Advantages:** The system continues to operate normally even when a failure occurs, resulting in high system availability.
- **Disadvantages:** Active redundancy can increase system weight, complexity, and energy consumption, as additional components must be constantly operating.

(2) Standby Redundancy

Standby redundancy involves having backup components or systems that are idle during normal operations but are automatically activated when the primary component fails.

- **Example:** A backup battery in an uninterruptible power supply (UPS) is an example of standby redundancy. The backup battery is not active during normal operations, but when the primary power source fails, the battery activates to maintain operation.
- **Advantages:** Standby redundancy is less costly and more energy-efficient than active redundancy, as backup systems are only used when needed.
- **Disadvantages:** There is often a delay in response time when a failure occurs, and the standby systems may not always activate properly, especially if they have not been maintained or tested regularly.

(3) Cold Redundancy

Cold redundancy refers to backup components that are only activated in the event of a failure, but unlike standby redundancy, they are not immediately available. These components need some time to be brought online after a failure occurs.

- **Example:** A spare part stored in a warehouse that can be deployed after a component fails is an example of cold redundancy.
- **Advantages:** It is the most cost-effective redundancy strategy, as it does not require additional active components or systems.
- **Disadvantages:** There is a significant delay in bringing the backup system online, which can impact system performance and reliability in a real-time failure situation.

c. Design Considerations for Implementing Redundancy

The guidebook provides several important considerations when designing redundancy into a system.

(1) Redundancy Allocation

The process of deciding which parts of the system to make redundant and to what extent is critical. The guidebook stresses that redundancy should be applied strategically to the most critical components or subsystems that, if failed, would cause the system to stop functioning. Overuse of redundancy can lead to unnecessary complexity and increased costs.

- **Failure Mode and Effects Analysis:** Failure mode and effects analysis is often used to identify critical components that should be designed with redundancy. These components are those whose failure would cause significant degradation or total failure of the system.
- **Reliability Centered Maintenance (RCM):** When designing redundancy, the guidebook emphasizes incorporating RCM practices to ensure that redundant components are appropriately tested, maintained, and monitored. The redundant systems must be regularly exercised or tested to ensure they will function correctly when needed.
- **Probability of Failure:** When designing redundancy, engineers should account for the probability of failure of the components and systems in question. Redundant components should not increase the probability of failure due to issues such as common mode failures. For example, if redundant components share a common power supply, a failure in the power supply could lead to a failure in both components simultaneously, defeating the purpose of redundancy.
- **Cost-Effectiveness:** The guidebook highlights the importance of balancing the cost of redundancy with the level of reliability required for the system. Redundancy increases both initial design costs and operational costs (e.g., maintenance). Therefore, a trade-off analysis should be conducted to determine the optimal level of redundancy based on the system's reliability requirements, operational environment, and risk tolerance.

- **Design Complexity:** Adding redundant components to a system increases the complexity of the design. The more complex a system becomes, the harder it is to manage, troubleshoot, and maintain. Redundant systems must be carefully designed to integrate smoothly with the rest of the system while minimizing additional complexity. The guidebook stresses that engineers must ensure that the added redundancy does not inadvertently reduce the overall reliability of the system due to increased system complexity.
- **Redundancy and System Availability:** The concept of availability is critical when designing reliability through redundancy. Availability is a measure of the system's readiness for operation, and redundancy is often employed to ensure high availability. The guidebook explains that redundant components can significantly improve the Mean Time to Repair (MTTR) because backup systems can be quickly brought online when a failure occurs.
- **MTBF:** Redundancy increases the mean time between failures by ensuring that the failure of one component does not result in total system failure.
- **MTTR:** Redundant systems can be swapped in quickly, reducing downtime and minimizing the time needed to restore the system to full functionality.
- **System Availability (A):** This is typically calculated using the formula:

$$\mathrm{A} = \mathrm{MTBFMTBF} + \mathrm{MTTRA} = \mathrm{MTBF} + \mathrm{MTTRMTBF}.$$

Redundancy plays a major role in improving system availability by decreasing the impact of component failures on the overall system.

(2) Redundancy in Critical Systems

The guidebook places particular emphasis on the use of redundancy in critical systems, where failure could result in loss of life, severe damage to property, or large-scale system failures. For these systems, high reliability design through redundancy is essential.

(3) Common Mode Failures and Redundancy Design

A critical aspect of designing reliability through redundancy is avoiding common mode failures. The guidebook discusses how redundancy can be compromised if redundant systems are vulnerable to the same failure modes. For example, if two power supply systems are designed with the same components (e.g., the same battery type), they may both fail at the same time under certain conditions. To prevent common mode failures, redundancy must be designed with diversity. This could mean:

- Using different manufacturers for redundant components.
- Using different power sources for each redundant system.
- Ensuring that redundant components are not housed in the same environment where a single event could cause failure in both systems.

d. Conclusion

The "Reliability and Maintainability Engineering Guidebook" emphasizes that redundancy is a fundamental design principle for ensuring system reliability, particularly in critical applications. The use of redundancy helps to mitigate risks associated with component failure, enhance system availability, and reduce the impact of downtime.

Redundancy is not a one-size-fits-all solution; its implementation must be tailored to the specific needs of the system, the operational environment, and the risk tolerance of the organization. Therefore, designing reliability through redundancy is an essential, yet nuanced aspect of creating high-performance, fault-tolerant systems.

In prototyping the USV system the decision to deliver systems that meet the worst case capacity for both development and redundancy enabled confidence that the system would meet the requirements and enable lessons learned to drive the "Right Sizing" of the system before production.

9. Use of the Newest Commercial Hardware and Infrastructure/ Platform as a Service Software

In the modern defense landscape, the need for advanced computing and network capabilities is paramount to ensuring mission success. For the USV program the ability to procure cutting-edge technology quickly, flexibly, and efficiently was critical.

A central tenant of the USV OTA, and the following IWS X ICS, is continual iteration with next in breed HW. This approach is an innovative method for procuring modern computing and network HW, as well as IaaS.

The award of the OTA to an industry team assesses the viability of the HW/SW solutions presented to meet current requirements. In the awarded contract vehicle for USV ICS CI systems continual assessment of market solutions is a core activity which ensures that a system will continue to iterate and integrate the newest components.

10. Make it Work: Integrate

Utilizing the OTA for delivery of USV ICS CI HW and IaaS ensured that we could deliver relevant solutions in six months vice 6–8 years. While use of the OTA limits the developer to the solutions awarded in the OTA SOW it enables the team to focus all energy on the design of the assembled components and integration to deliver a workable solution.

While more traditional acquisition processes would include lengthy product selection processes, complete with often years long assessments, the USV OTA process allowed the small development team to focus on "Make Work" and schedule. Knowing that the solutions being delivered were modern and backed by considerable commercial fielding and utilization, the team focused on the key development features, confident that the underlying HW/SW supported system requirements.

These key system requirements were:

- USV computing infrastructure leverages OTA efforts to deliver modern capability in NPS equipment suites.

- USV ICS CI provides IaaS to tactical virtualized CS's applications.
- 3 Processing Cabinets and an UPS System.
- USV Prototype 1 systems, delivered in 8U modularity, will refresh incrementally as Prototype 2 and ICS CI development efforts bring system iteration leading to modular capability refresh ad infinitum.
- Prototype 1 CI is built on VPS Inc. I HW and IaaS products.
- PCIe based NTDS, Aegis Time Processors (ATP), Gyro Data Converters (GDC), Vertical Launch Processor (VLP).
- Prototype 1 CI supports AWS, SCS, CaaS, and TWS applications in a VM or container environment.
- Prototype 1 efforts to address:
- System security and wiping of SW upon intrusion.
- Positive firing chain control with non-resident actuation (given that solution resides in the VLP vice VLS).
- Remote maintenance and HW management strategy and functionality.

USV ICS CI systems were delivered meeting the GFE need dates to OUSV 4. Features of the OTA procurement included delivery of the cables needed to connect the USV ICS CI as well as site integration support from the industry OTA team. The decision was made to use members of the Government and OTA team to perform the integration of these systems at both the shore sites and onboard OUSV 4.

Figure 8. Unmanned Surface Vessel Integrated Combat System Computing Infrastructure Installed on Overlord Unmanned Surface Vessel 4

11. Assess and Iterate

The USV ICS CI successfully delivered three prototype 1 systems within 6 months of OTA SOW ratification. These systems, fielded on OUSV 4 and shore development sites, provided a wealth of data to support development of follow-on systems.

One significant benefit of the decision to deliver systems that meet the worst case capacity for development and redundancy was the ability in the development environment to utilize all of the system capacity, utilizing the SDN capabilities of the IaaS SW Plane, to run multiple instantiations (tenancies) within the CI. This enabled developers to run "many" USV platforms simultaneously supporting development of the system to allow identification and definition of USV key performance parameters (KPP). Specifically, exercising these KPPs, enabled by initial system design, are providing insight into operator workstation demands and the maximum number of USVs that a controlling unit can manage.

These KPPs, and the ability to provide systems in prototype that support an understanding of the systems and systems' behaviors and automation needed to meet them, is critical to unmanned programs.

KPPs are critical elements of a capability solution required to meet operational goals and mission effectiveness. In the context of the USN, KPPs are specific performance attributes that a system must meet to fulfill its role within naval operations. These parameters ensure that new systems and platforms align with the strategic goals and operational requirements of the USN.

a. Key Aspects of USN KPPs

KPPs are measurable, testable, and specific capabilities or characteristics that are essential for the system to perform its intended mission. Failure to meet a KPP typically renders the system unacceptable for deployment.

(1) Alignment with Operational Requirements

KPPs are derived from top-level requirements documents, such as the Initial Capabilities Document (ICD) and the Capabilities Development Document (CDD). These documents outline what the USN needs to achieve in terms of warfighting, readiness, and sustainment. KPPs in the USN are often categorized to ensure comprehensive system performance. Some common categories include:

- **Force Protection:** Ensuring the survivability of naval platforms and personnel.
- **Sustainment:** Addressing the life cycle management of systems, including logistics, maintenance, and reliability.
- **Net-Ready:** Ensuring interoperability and secure communications within the USN's networks and with joint/coalition forces.
- **Energy Efficiency:** Ensuring that platforms optimize fuel and energy use to support sustained operations.

- **Weapons System Effectiveness:** Ensuring the system can meet offensive and defensive mission requirements.

The USN adheres to several mandatory KPPs set by the Joint Staff to ensure interoperability and integration within joint military operations. These include:

- **Survivability:** Systems must withstand and function in contested environments.
- **Net-Ready:** Systems must integrate seamlessly with joint networks and ensure effective data exchange.
- **Force Protection:** Systems must provide adequate protection for personnel and equipment.

(2) Mission Success

KPPs are directly tied to the USN's ability to perform its missions, such as sea control, power projection, and maritime security.

(3) Acquisition and Testing

KPPs are integral to the DoD's acquisition process. They are used to evaluate system performance during developmental and operational testing.

(4) Risk Mitigation

By defining KPPs early in the development process, the USN can identify and mitigate risks, ensuring that systems meet mission-critical requirements.

(5) Accountability

KPPs provide a clear benchmark for evaluating contractor performance and holding vendors accountable for delivering systems that meet the USN's operational needs.

12. Development and Deployment of Unmanned Surface Vessel Integrated Combat System Computing Infrastructure Prototype 2

The USV program, using lessons earned in the initial USV ICS CI development, produced a second iteration to support USV program objectives and the T&E site needs. One key feature of the new OTA-based ICS paradigm is the immediate iteration to the newest in breed HW. Diverging from past CI development efforts where lifetime buys of HW would be made to maintain a consistent baseline, USV ICS CI fielded and integrated new HW, proving the transparency and abstraction of the HW from CS applications.

A key benefit of this continual iteration was the ability to reduce the HW footprint, delivering the same capacity with fewer physical servers, due to technological increases in storage and CPU core densities.

13. Conclusion

In 2020, NSWC Dahlgren Computing Infrastructure Group was tasked with developing prototype equipment for USV ICS CI fielding of a virtualized AWS to support the LUSV program. The effort aimed to deliver modernized computing infrastructure and NPS HW, emphasizing virtualized combat SW and advanced technologies. Early guidance from PEO IWS 80 highlighted the need for rapid, technologically relevant solutions for maritime operations.

14. Challenges in Legacy Procurement Approaches

Developing the LUSV ICS faced significant hurdles due to limitations in the FAR-based acquisition process, which governs DoD procurement. Legacy systems like Technology Insertions 12 and 16 often suffered from HW obsolescence, with components outdated before deployment. For example, TI16 infrastructure scheduled for LUSV deployment in 2023 was already eight years old, hindering the USN's ability to field cutting-edge solutions.

15. Addressing Limitations

The DoD has turned to OTAs for greater flexibility in research and development, enabling rapid prototyping and innovation. Streamlining compliance processes and fostering contractor education further enhances participation and efficiency. Prioritizing long-term value over cost efficiency incentivizes advanced solutions.

16. Lessons from Large Unmanned Surface Vessel Integrated Combat System Development

Key takeaways include the necessity of agile procurement strategies, a small team approach, and integrating modernization efforts to address obsolescence. Exploring alternatives like OTAs ensures timely delivery of cutting-edge systems, supporting the USN's operational relevance and readiness.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. INTEGRATED COMBAT SYSTEM: GATEWAY TO MODERN SPEED TO CAPABILITY

The ICS was a concept with technical basis established by a white paper written by Dr. Alvin Murphy. This whitepaper established the foundation for initial estimates based on historical timelines and general budget requests for ICS HW and SW work.

This chapter summarizes the ICS-CMS Conceptual Reference Model, as articulated by Dr. Murphy (Murphy, 2022). This summary covers key aspects like system components, subsystems, integration, and advanced technologies used within the system. The model is a framework that ensures the development of cohesive, flexible, and highly effective CSs for modern naval environments.

1. Introduction

a. Purpose

An ICS is designed to enable seamless integration and interoperability between various combat components such as sensors, weapon systems, communication channels, and decision-making systems. It enhances situational awareness, response times, and the overall combat effectiveness of naval forces by ensuring that all subsystems within a ship or a fleet are interconnected and working in sync. The ICS approach ensures that all elements of the CS work together, eliminating redundancies, optimizing resource usage, and improving operational outcomes.

b. Core Features

- **Modularity:** ICS systems are modular in nature, meaning that new technologies or systems can be integrated into existing platforms without significant overhaul. This is crucial for ensuring that combat systems can evolve over time.
- **Interoperability:** The system ensures that various subsystems communicate effectively, allowing multiple platforms or fleets to operate together, regardless of technological differences.

- **Adaptability:** As new threats, technologies, and operational needs emerge, the ICS can be updated or adapted to meet these requirements.

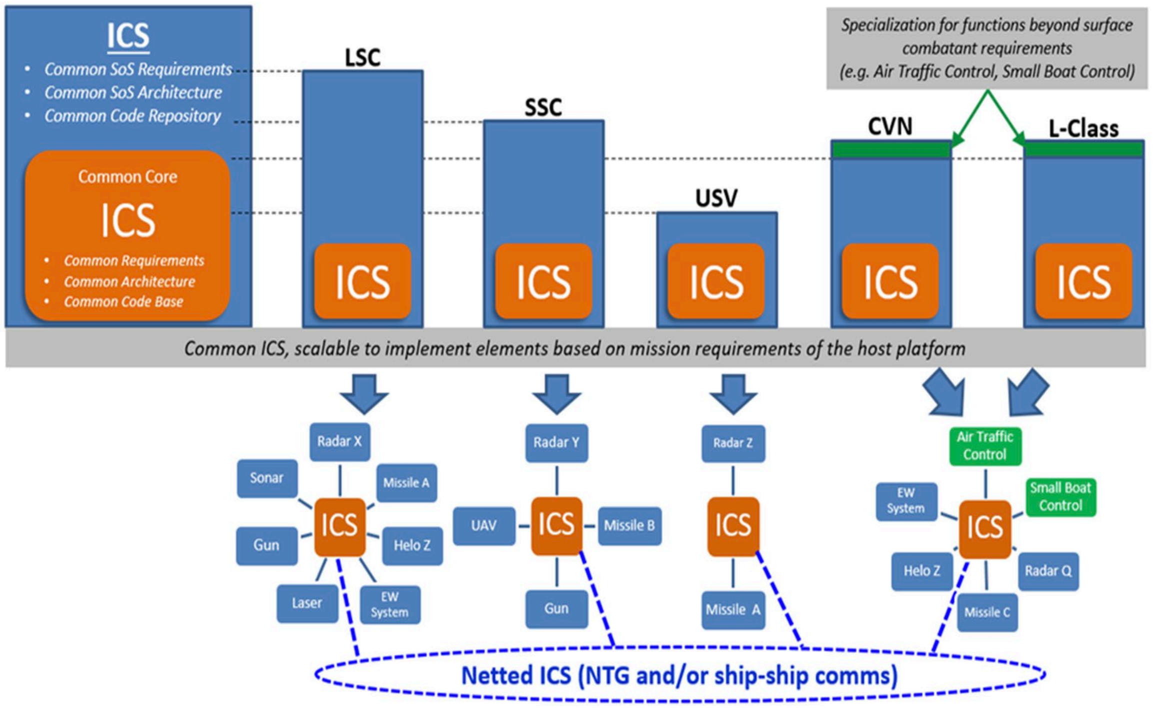


Figure 9. Integrated Combat System Common Core Concept. Source: Murphy (2022).

2. Combat Management System Overview

a. Role of CMS in ICS

The CMS is the heart of the ICS. It is responsible for the central processing of data, coordination of weapon systems, and providing operators with a unified operational picture. CMS integrates various combat components, including sensors (radar, sonar, etc.), decision support systems, and weapons control systems, into a single, cohesive entity. The CMS is the interface through which operators interact with the system, making it essential for successful combat operations.

b. Core Functions

- **Sensor Data Processing and Fusion:** The CMS collects raw data from a wide range of sensors (radar, infrared, sonar, etc.) and processes it to provide a coherent and actionable operational picture. This data fusion process ensures that operators have access to real-time information for decision-making.
- **Threat Detection and Assessment:** The CMS continuously evaluates the sensor data to detect and classify potential threats. By using advanced algorithms and AI-based techniques, the system can prioritize threats based on urgency, severity, and likelihood.
- **Weapon System Control:** The CMS manages the engagement of weapons, coordinating the targeting, firing, and reloading processes. It ensures that the most appropriate weapon system is selected for a given target.
- **Decision Support:** The CMS supports decision-making by providing tools and algorithms to assess various options based on the current combat situation. It ensures that operators can make well-informed decisions under pressure.

3. Key Components of the Conceptual Reference Model

This section outlines the essential components that make up the Conceptual Reference Model for an ICS-based CMS. These elements work together to provide the necessary functionality for modern combat systems.

a. Sensor Integration

- **Multi-sensor Systems**: The model advocates for integrating multiple types of sensors to provide redundant, accurate, and comprehensive situational awareness. This includes radar, sonar, infrared sensors, electro-optical sensors, and other environmental sensing equipment.

- **Sensor Fusion**: The fusion of data from these diverse sensors allows the CMS to build a comprehensive, multi-dimensional picture of the environment. Redundancy ensures that even if one sensor fails, others can compensate.
- **Real-time Data**: The integration of real-time data from all sensors ensures that the system responds dynamically to changing combat conditions, making it vital for modern warfare scenarios.

b. Data Fusion and Processing

- **Real-Time Processing**: The CMS processes raw data from sensors in real-time. Algorithms and computational models are used to identify, track, and assess potential threats and targets.
- **Artificial Intelligence (AI) and Machine Learning**: AI and machine learning models are used to enhance the accuracy of data fusion. These technologies enable the CMS to continuously learn and adapt based on changing operational contexts and evolving threats.
- **Threat Prioritization**: Data fusion algorithms help the system prioritize threats, focusing on the most imminent or dangerous targets. This is crucial for managing multiple engagements simultaneously.

c. Decision Support and C2

- **C2 Subsystems**: The C2 subsystem ensures communication between different units and facilitates decision-making processes. It allows commanders to issue orders, share information, and coordinate actions across various units within a fleet.
- **Real-time Operational Picture**: A unified operational picture, derived from fused sensor data, is displayed to operators and commanders, assisting them in making informed decisions quickly.

- **Automated Decision-Making**: In complex scenarios, the CMS can provide automated recommendations for actions. These recommendations are based on pre-programmed decision trees, predictive models, and real-time analysis.

d. Weapon Systems Management

- **Weapon Selection**: Based on the nature of the threat and the available resources, the CMS selects the most appropriate weapon system (e.g., missiles, torpedoes, guns) for engagement.
- **Targeting and Engagement**: The CMS coordinates targeting processes, ensuring that weapons are accurately aimed and fired at the right moment.
- **Multi-Layered Defense**: In certain scenarios, multiple weapon systems may be used in tandem (e.g., a ship's close-in weapon system for defense against incoming missiles, while a longer-range system targets air or surface threats).

e. Human-Machine Interface

- **Operator Interface**: The human-machine interface (HMI) is the interface through which operators interact with the system. It provides visual displays, alarms, and controls that allow operators to monitor and control the system.
- **Usability**: Ensuring that the HMI is intuitive is crucial for effective system operation. Operators should be able to process critical information and make decisions rapidly, even under stress.
- **Situational Awareness**: The HMI provides operators with a clear picture of the environment, including real-time status updates and mission-critical information, ensuring situational awareness during combat.

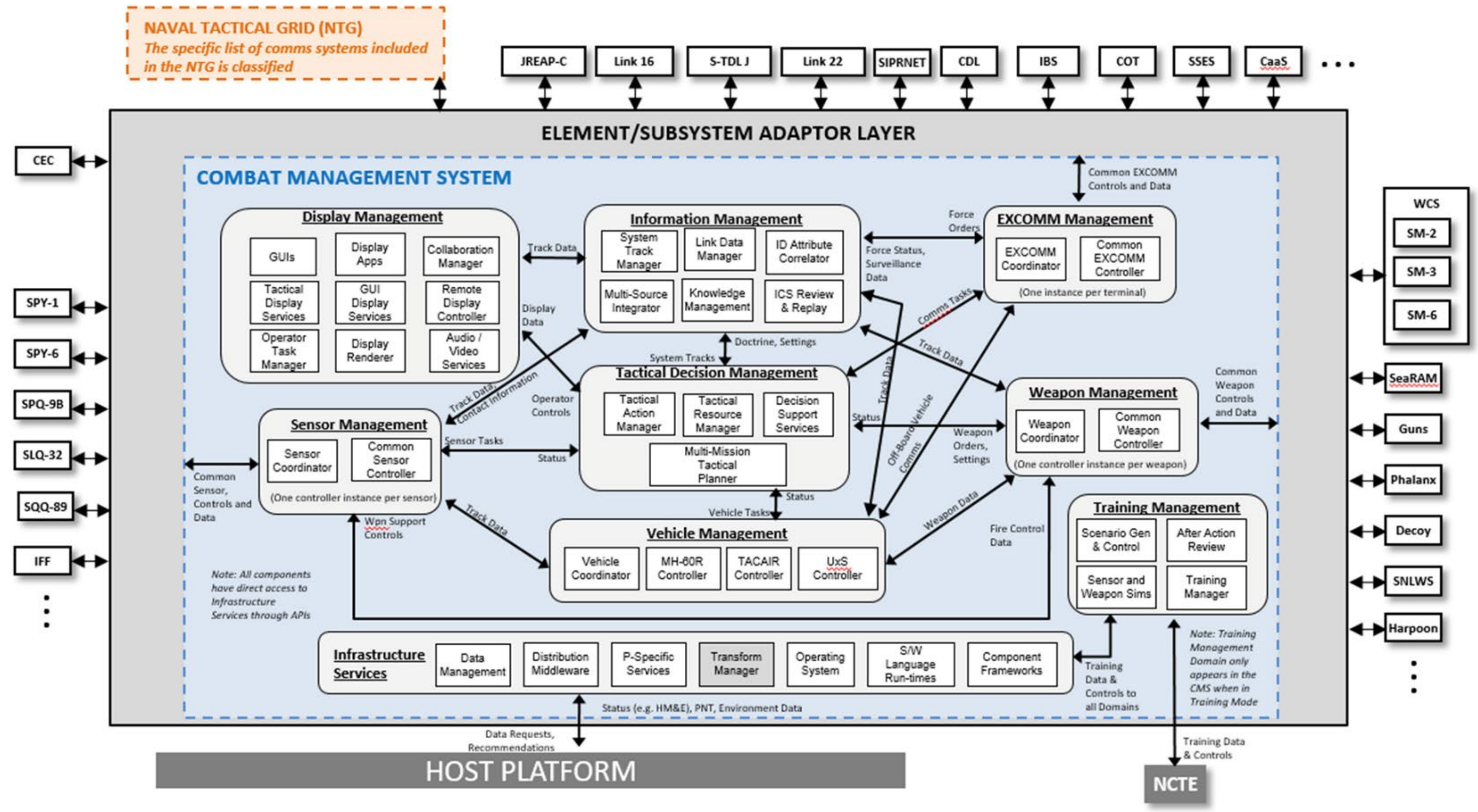


Figure 10. Murphy Integrated Combat System Concept. Source: Murphy (2022).

f. Modularity and Scalability

- **Modular Design**: The system is built to be modular, meaning individual subsystems (sensors, weapons, communication systems) can be replaced, upgraded, or expanded as new technologies become available.
- **Scalability**: The modular nature of the ICS allows it to scale across different platforms, from individual ships to entire fleets. This scalability ensures the system can be adapted for different mission sizes and complexities.
- **Future-Proofing**: The model is designed with future technologies in mind, ensuring that future upgrades can be seamlessly integrated without the need for major system redesigns.

4. Integration of Advanced Technologies

a. AI Microservices

- **AI in Decision Support**: AI microservices are integrated into the CMS to enhance decision support capabilities. AI algorithms can assess combat scenarios, predict enemy actions, and provide recommendations for countermeasures.
- **Autonomous Decision-making**: In some cases, the system can make autonomous decisions based on predefined parameters, helping to reduce operator load and increase reaction speed.
- **Predictive Analytics**: AI can analyze historical data and operational patterns to predict future outcomes, giving commanders a tactical advantage.

b. Cybersecurity

- **Threat Detection**: As combat systems become more interconnected, cybersecurity becomes a critical consideration. The model includes strong cybersecurity measures to protect the ICS from cyber-attacks.
- **Data Integrity**: Ensuring the integrity of the sensor data and communication channels is vital to prevent manipulation or corruption of critical information.
- **Resilience**: The system is designed to maintain operational capability even in the event of a cyber-attack, ensuring that backup protocols and redundancies are in place.

5. System-of-Systems Engineering Approach

a. System Integration

- **Holistic Design**: The ICS-CMS model follows a system-of-systems approach, recognizing the interdependencies between subsystems. A failure in one subsystem could potentially compromise the entire system, so all components must work in harmony.
- **Interoperability**: The system is designed to ensure that various subsystems, regardless of their manufacturer or technological basis, can communicate and operate together seamlessly.
- **Joint Operations**: By integrating systems across different platforms and services (e.g., Navy, Army, Air Force), the ICS model supports joint operations, improving overall coordination and effectiveness.

b. Complexity Management

- **Decomposition of Systems**: To manage the complexity of modern combat environments, the system is decomposed into smaller, manageable subsystems. This approach ensures that the overall system remains adaptable and maintainable.

- **Integration Challenges**: One of the challenges faced in system integration is ensuring that all components are compatible with one another and can be effectively managed by a central CMS.

6. Challenges and Considerations

a. Resource Constraints

The integration of multiple systems and advanced technologies requires significant resources in terms of time, funding, and technical expertise. Effective resource allocation is crucial for successful implementation.

b. Human Factors

- **Operator Training**: Ensuring that operators are adequately trained to use the CMS efficiently is crucial for the system's success.
- **User Experience**: The system's HMI design must be optimized for user experience, minimizing cognitive overload and reducing the chance of operator error.

c. Life Cycle Management

Managing the life cycle of the system, from initial design to decommissioning, requires careful planning and support. Maintenance, upgrades, and replacements must be considered at all stages of the system's life.

7. Conclusion

The ICS-CMS Conceptual Reference Model by Alvin Murphy presents a robust and adaptable framework for modern combat management. By focusing on integration, modularity, real-time data processing, and advanced technologies like AI, the model ensures that naval combat systems can meet the challenges of modern warfare. Through effective system design, integration of multiple subsystems, and adaptability to future needs, the ICS model provides a comprehensive approach to enhancing combat capabilities across naval platforms.

This conceptual reference model highlights the critical importance of system integration, decision support, and scalability, while also addressing challenges such as resource constraints, cybersecurity, and the complexity of modern combat environments. Ultimately, this framework enables the creation of highly effective, adaptable, and future-proof combat systems that can operate across a range of mission scenarios.

The ICS-CMS Conceptual Reference Model, as articulated by Dr. Alvin Murphy, provides a comprehensive framework for understanding and developing modern naval combat systems. This model emphasizes the integration of various subsystems to create a cohesive and efficient combat management environment.

Table 10. Summarizing the Integrated Combat System Concept

Section	Key Points
1. Introduction	ICS integrates various combat components such as sensors, weapons, and communication systems.
	Provides seamless interoperability for operational effectiveness.
	Enhances situational awareness and combat effectiveness.
Core Features	**Modularity**: Allows for easy integration of new technologies.
	Interoperability: Ensures smooth communication across platforms.
	Adaptability: Can evolve with emerging threats and technologies.
2. CMS	The CMS is the central hub for processing data, coordinating weapons, and aiding decision-making.
	Integrates sensors, weapons, and decision support systems.
	Provides operators with a unified operational picture.
Core Functions of CMS	**Sensor Data Processing**: Collects and processes raw data from various sensors.
	Threat Detection and Assessment: Uses AI to classify and prioritize threats.
	Weapon Systems Control: Manages engagement of weapon systems.
	Decision Support: Assists operators with decision-making under pressure.
3. Key Components of the Model	**Sensor Integration**: Integrates multiple sensors for accurate, redundant situational awareness.
	Data Fusion and Processing: Uses advanced algorithms for real-time analysis.
	Weapon Systems Management: Manages weapon selection and targeting.

Section	Key Points
	HMI: Provides operators with intuitive controls for effective system interaction.
	Modularity and Scalability: Ensures system can be upgraded and adapted for different platforms.
4. Integration of Advanced Technologies	**AI Microservices**: Enhances decision support and automates decision-making processes.
	Cybersecurity: Protects against cyber threats and ensures data integrity.
5. System-of-Systems Approach	Emphasizes integration of subsystems for overall operational effectiveness.
	Supports joint operations between various military branches.
	Focus on interoperability and scalability across platforms.
Challenges and Considerations	**Resource Constraints**: Need for significant resources (funding, time, expertise).
	Human Factors: Ensures effective operator training and intuitive interface design.
	Life cycle Management: Focus on maintenance, upgrades, and system longevity.
6. Conclusion	The ICS-CMS model enables a cohesive, adaptable, and scalable approach to modern naval combat.
	Focuses on integrating advanced technologies, AI, and real-time data processing.
	Ensures future-proofing through modular design and system integration.

B. TECHNICAL STRATEGY

To further the ICS vision concepts were put into appropriate forms to generate N96 interest and drive to the vision. The team established a rough breakdown with HW and SW as the Foundry and Forge, respectively. These teams moved quickly to implement build and development environments using an Agile CS methodology of establishing Objectives and Key Results (OKRs) for Planning Intervals (PIs) (3-4 months). While the OKRs helped to guide the initially quick steps the general vision needed more definition. The team wrote the Now, Next, Later, technical strategy.

This strategy established two entities as key enablers of the ICS concept, The Forge and the Foundry. Articulated throughout multiple USN guidance documents. The strategy outlined in Now, Next, Later (NAVSEA 2022) presents a phased approach to modernizing and integrating CSs for naval operations. It emphasizes evolving towards a unified ICS to enhance warfighting capabilities, maintain technological superiority, and adapt to emerging threats.

1. Now: Immediate Actions

The Now phase focuses on addressing urgent needs and setting the foundation for future advancements:

- **Incremental Modernization:** Updates to current CSs to enhance operational readiness, using existing infrastructure and technology.
- **Platform-Specific Capabilities:** Maintaining platform independence while optimizing CSs for individual ship classes.
- **Interoperability:** Ensuring seamless integration between legacy and new systems to enable joint operations and data sharing across platforms.

2. Next: Mid-Term Goals

The "Next" phase builds on the foundation laid in the Now phase by transitioning to a more integrated and flexible CS architecture:

- **Modular Open Systems Approach (MOSA):** Implementing modular and open architectures to increase adaptability, reduce costs, and facilitate technology upgrades.
- **Virtualization and Containerization:** Adopting SW virtualization to improve resource efficiency and support rapid deployment of capabilities.
- **Improved Cybersecurity:** Enhancing resilience against emerging cyber threats through robust security frameworks.

3. Later: Long-Term Vision

The "Later" phase focuses on achieving the fully realized ICS:

- **Integrated Fleet-Wide Capabilities:** Moving from platform-centric systems to a cohesive, networked fleet where all platforms share CS data in real-time.
- **Autonomy and AI:** Leveraging artificial intelligence and autonomous systems to enhance decision-making, reduce operator workload, and improve mission outcomes.
- **Continuous Modernization:** Establishing an agile, iterative development process to ensure the ICS evolves in response to future threats and technological advancements.

To achieve the strategic goals of the fully modularized CMS four key ICS phases have been identified in the ICS Strategic Vision [Office of the Chief of Naval Operations (OPNAV), 2022] to gauge the maturity of ICS definition, development, fielding, and support across these three epochs. These key phases are:

1. **ICS Foundation:** A portion of a CMS subdomain with refactored SW functions in the form of services (i.e., some legacy functionality strangled out) organized by common ICS architecture with cyber resilient subdomain enclaves (e.g., CMS, element), running atop an IaaS and PaaS infrastructure to bring new/improved warfighting capability.
2. **ICS Enabled**: Portions of multiple CMS subdomains with refactored SW functions in the form of services, organized by common ICS architecture with a cyber resilient enclave (e.g., full CS enclave), running atop an IaaS and PaaS infrastructure to bring multiple new/improved operational capabilities (Note: this is where the "ICS Tipping Point" is expected to be achieved.)
3. **ICS Node:** Multiple CMS subdomains with fully refactored SW functions in the form of services, organized by common ICS architecture with cyber resilient enclaves (e.g., platform and netted level), running atop an IaaS and PaaS infrastructure to bring new multiple new/improved operational capabilities.

4. **ICS Fully Realized:** All CMS subdomains have refactored SW functions in the form of services for all CMS subdomains, organized by common ICS architecture with (platform- and netted level) cyber resiliency, running atop an IaaS and PaaS infrastructure to bring multiple new/ improved operational capabilities.

a. Key Principles

- **Agility:** The strategy emphasizes flexibility to adapt to changing operational and technological environments.
- **Interoperability:** Ensuring seamless communication across platforms and allies.
- **Cost-Effectiveness:** Reducing life cycle costs through modular design and efficient acquisition processes.
- **Scalability:** Supporting diverse mission requirements through adaptable systems.

b. Key Enablers

The Forge is a DoD initiative focused on revolutionizing SW development and delivery for defense systems. Spearheaded by the USN, the Forge operates as a SW factory, leveraging modern development practices and technologies to produce high-quality, mission-critical applications with speed and efficiency.

c. Core Objectives

The Forge Software Factory aims to address key challenges in traditional defense SW development, including lengthy delivery cycles, limited flexibility, and challenges in integrating emerging technologies.

(1) Accelerated Development and Deployment

Using Agile and DevSecOps methodologies, The Forge emphasizes continuous integration, testing, and delivery to reduce the time required to deliver SW to operational platforms.

(2) Scalability and Adaptability

The Forge focuses on producing modular, scalable solutions that can be easily adapted to evolving mission requirements, ensuring long-term relevance and usability.

(3) Cybersecurity Embedded in Development

Security is integrated into every phase of the SW life cycle (DevSecOps), ensuring robust defenses against cyber threats from the outset.

(4) Cross-Platform Interoperability

Building SW with a MOSA to ensure compatibility across diverse systems and platforms within the USN and joint forces.

d. Development Methodology

The Forge Software Factory aims to address key challenges in traditional defense SW development, by use of modern methods.

(1) Agile Practices

The Forge adopts Agile principles, breaking projects into smaller, iterative sprints that enable faster delivery of incremental capabilities and better responsiveness to changing requirements.

(2) DevSecOps Integration

By embedding security and operations within the development process, The Forge ensures that SW is secure, operationally reliable, and deployable in real world environments.

(3) Cloud-Native Technologies

The Forge relies on cloud infrastructure to facilitate rapid scaling, resource optimization, and easier collaboration across teams.

(4) Automation Tools

Leveraging automation for testing, deployment, and monitoring reduces human error, accelerates timelines, and ensures consistency.

4. Key Achievements

a. Reduced Development Cycles

The Forge has significantly shortened the time needed to develop and deploy applications, moving from years under traditional models to months or even weeks.

(1) Enhanced Cyber Resilience

With security integrated into development, Forge-produced applications meet stringent DoD cybersecurity standards.

(2) Operational Software Delivery

The Forge has delivered tools for critical mission areas such as maritime operations, autonomous systems, and data analytics, showcasing its capability to support cutting-edge defense technology.

b. Strategic Importance

(1) Rapid Response to Emerging Threats

The Forge enables the USN to quickly adapt to shifting operational environments by rapidly developing and deploying new capabilities.

(2) Cost Efficiency

By modernizing development practices, The Forge reduces the financial burden associated with traditional waterfall-style SW development.

(3) Collaborative Ecosystem

The Forge promotes collaboration between government, industry, and academia, fostering innovation and leveraging expertise across sectors.

C. THE FOUNDRY: U.S. NAVY HARDWARE FACTORY

The Foundry is a USN initiative designed to streamline the design, production, and delivery of cutting-edge HW systems for naval operations. Similar to the Forge Software Factory, which focuses on SW, the Foundry aims to modernize and accelerate the USN's approach to HW development, ensuring warfighters have access to the most advanced and reliable systems to meet emerging challenges.

1. Core Objectives

The Foundry addresses inefficiencies and limitations in traditional HW procurement and development by focusing on the key objectives outlined here.

a. Rapid Prototyping and Deployment

- Emphasizes the quick design, testing, and fielding of HW prototypes to meet urgent operational needs (Department of the Navy [DON], 2021).
- Reduces the development-to-deployment timeline using modular and additive manufacturing techniques (DON, 2021).

b. Standardization and Modularity

- Implements a MOSA to ensure HW components are interoperable across platforms and can be easily upgraded or replaced (Office of the Under Secretary of Defense for Acquisition and Sustainment [OUSD(A&S)], 2020).

c. Integration with Software Development

- Synchronizes HW development with SW capabilities from initiatives like The Forge to ensure seamless performance in complex systems (DON, 2021).

d. *Scalability and Sustainability*

- Focuses on designing HW scalable for various platforms, including surface ships, submarines, and unmanned systems.
- Uses sustainable materials and processes to align with environmental and long-term operational goals (DON, 2021).

e. *Cost Efficiency*

- Reduces costs through advanced manufacturing techniques, streamlined supply chains, and efficient resource allocation (DON, 2021).

2. Key Features

a. *Advanced Manufacturing*

- Employs 3D printing and additive manufacturing to produce complex components quickly and cost-effectively (Defense Innovation Unit, 2020).
- Enables on demand production, reducing inventory needs and minimizing logistical challenges.

b. *Agility in Design and Production*

- Utilizes Agile methodologies for iterative HW development, allowing rapid adaptation to changing requirements (DON, 2021).
- Shortens the design cycle through digital engineering and simulation tools (OUSD(A&S), 2020).

c. *Cybersecurity for Hardware Systems*

- Ensures HW components are resistant to tampering and cyber threats, integrating security measures into every stage of the design and production process (DON, 2021).

d. *Collaborative Development*

- Partners with industry and academia to leverage the latest research and innovations in HW technology (Defense Innovation Unit, 2020).

e. *Data-Driven Decision-Making*

- Uses advanced analytics and digital twin technology to optimize designs, predict maintenance needs, and improve life cycle management (OUSD(A&S), 2020).

3. Key Achievements

a. *Prototyping Success*

- Delivered prototypes for USVs, including ruggedized computing HW and sensor systems, within compressed timelines (DON, 2021).

b. *Fleet-Wide Modernization*

- Contributed to upgrading aging combat systems by replacing obsolete HW with modern, modular components (OUSD(A&S), 2020).

c. *Enhanced Readiness*

- Improved operational readiness by producing HW systems that integrate seamlessly with existing platforms and SW systems (DON, 2021).

d. *Supply Chain Resilience*

- Built more resilient supply chains by adopting domestic manufacturing capabilities and reducing reliance on foreign suppliers (Defense Innovation Unit, 2020).

4. Strategic Importance

The Foundry is crucial to ensuring the USN can maintain technological superiority and operational readiness in an era of rapid technological advancement. Key benefits are outlined here.

a. *Adaptability to Emerging Threats*

- By enabling rapid prototyping and deployment, the Foundry allows the USN to respond quickly to new challenges in the maritime domain (DON, 2021).

b. *Innovation Ecosystem*

- Encourages innovation through collaboration with industry and academic institutions, driving the development of next-generation HW.

c. *Life Cycle Optimization*

- Extends the lifespan of critical systems by ensuring HW can be easily upgraded and maintained over time (OUSD(A&S), 2020).

d. *Force Multiplier*

- Enhances the USN's overall combat effectiveness by providing high-quality, interoperable HW systems that meet the demands of modern warfare (DON, 2021).

Table 11. The Forge Software Factory versus The Foundry Hardware Factory

Aspect	The Forge	The Foundry
Focus	Revolutionizing SW development and delivery for defense systems.	Streamlining the design, production, and delivery of advanced HW systems.
Core Objectives	Accelerated development using Agile and DevSecOps.	Rapid prototyping and deployment.
	Modular, scalable solutions.	Standardization through MOSA.
	Cross-platform interoperability.	Integration with SW systems.
Development Methodology	Agile practices for iterative progress.	Agile methodologies for HW design.
	DevSecOps integrating security throughout development.	Advanced manufacturing (e.g., 3D printing).
	Cloud-native technologies.	Data-driven decision-making using digital twin technology.

Aspect	The Forge	The Foundry
	Automation for testing and monitoring.	
Key Features	Continuous integration, testing, and deployment.	Modular, upgradable designs.
	Robust cybersecurity embedded in the life cycle.	Cybersecurity measures in HW systems.
	Automation-driven efficiency.	Collaboration with industry and academia.
Achievements	Reduced development cycles from years to months or weeks.	Delivered HW prototypes for USVs within compressed timelines.
	Enhanced cyber resilience.	Upgraded aging combat systems with modern components.
	Delivered SW for autonomous systems, maritime ops, and analytics.	Improved supply chain resilience.
Strategic Importance	Rapid response to emerging threats.	Enhanced adaptability to emerging threats.
	Cost-efficient development practices.	Life cycle optimization with scalable, sustainable designs.
	Collaboration ecosystem for innovation.	Force multiplier in combat effectiveness.

The Forge and the Foundry are key pillars in the USN's modernization strategy, driving digital transformation and HW innovation. The Forge exemplifies Agile, DevSecOps, and cloud-based practices to deliver efficient, secure, and adaptable SW solutions, serving as a model for military-wide application. Complementing this, The Foundry focuses on developing cutting-edge, flexible, and secure HW, ensuring the USN maintains technological superiority in an evolving global security landscape. Together, they align with initiatives like the ICS to redefine how defense capabilities are developed and sustained in the 21st century.

D. ARCHITECTURE

With architectures, workflow process, and development environments established the USN team has set-off to refine initial schedules and cost estimates based on realized Forge SW development progress and the newly awarded systems engineering (SE) and systems integration (SI) contract. The SE/SI contract is a key enabler to Forge and SW refactoring activities

1. Architecture Strategy

The PEO IWS (2024) ICS, CS Re-Architecture Strategy and Approach strategy document outlines goals, cornerstones, and an approach to build a common CMS on a modern services-based architecture. This document states, as the ICS Re-architecture Goal, "By FY28, our goal is to completely phase out heritage CSs by 100% re-architecting to a unified ICS" (PEO IWS 2024). ICS architecture while continuously deploying hybrid ICS configurations across IaaS-equipped Surface Platforms" (PEO IWS 2024).

The core result of ICS architecture and re-architecture efforts is a CMS which provides combat direction capability through the management and deconfliction of sensor and weapon tasks within the combat system. The CMS provides for the integration of multi-warfare area displays, track pictures, and decisions. Additionally, the CMS provides an integrated assessment of platform readiness to complete assigned missions (PEO IWS 2024).

The ICS-CMS is inclusive of infrastructure, display, planning, C2, training, and element interface SW. This CMS provides force-level functions to services necessary to coordinate information and effects across the force. Guiding the ICS-CMS architecture are five cornerstones. Table 7–3 summarizes the architecture cornerstones.

Table 12. Cornerstones of Combat Management System Modernization

Cornerstone	Key Focus	Objectives and Features
1. Software Agility	Delivering capabilities rapidly at the "speed of relevance."	Reduce resource, build, test, and delivery timelines for upgrades.
		Modular, loosely coupled SW functions for easy updates.
		Enable over-the-air updates.
2. Tactical Automation	Enhanced decision-making using AI, ML, and visualization.	Common data understanding through ontologies and common data models.
		Develop ICS-based reference architecture for consistent automation.
		Update interfaces for AI use.
		Upgrade once, proliferate widely.

Cornerstone	Key Focus	Objectives and Features
3. Single CMS "Core"	Unified CMS integration across all ship classes.	Standardize user interfaces to reduce training needs.
		Leverage UX/UI best practices for usability.
4. Seamless C2ISR&T Integration	Integrating C2ISR&T data.	Multi-level security for seamless data sharing.
		Reference architectures for C4I and CS integration.
		Enable actionable tactical insights.
5. Combat System-of-Systems	Connecting multiple ships' sensors and effectors into a unified Combat SoS.	Real-time situational awareness for decision-making.
		Access and control data across all ICS nodes.
		IoT-style architecture for distributed operations and cybersecurity.

The re-architecture approach follows a Refactor, Rehost, and Re-Write strategy. In the PEO IWS ICS document these activities are defined as:

- **Refactor:** Modernize existing code in place to improve how the code works, without changing what the code fundamentally does. This includes applying the strangler pattern and implementing Facades, reorganizing the code and other code improvements, wrapping that code in VMs, establishing external APIs as necessary, and carry-in the VMs into the new architecture. Migration directly to containers is also possible but depends on the degree of modernization of the code, and may require additional effort beyond what would be required to host in VMs (PEO IWS ICS 2024).

- **Rehost:** Wrap existing code in VMs, establishing external APIs as necessary, and carry-in the VMs into the new architecture. Rehosting code associated with a function (or functions) does not change the code itself, it focuses on wrapping it so that it can be extracted and carried-in to the new architecture (PEO IWS ICS 2024).

- **Greenfield Re-Write:** A re-write of the code associated with a function based on the intent of the original function. This can also be accomplished by pulling in a 3rd party solution that satisfies the function. The results in either case are new applications hosted in containers compliant with the ICS architecture (PEO IWS ICS 2024).

- **Greenfield New:** New function(s) written from scratch required as part of the ICS architecture that do not exist in heritage CS architectures today. These may be driven by technology advancements or ICS requirements beyond those of heritage systems. This can also be accomplished by pulling in a 3rd party solution that satisfies the function. Similar to greenfield re-write, the results in either case are new applications hosted in containers compliant with the ICS architecture.

Figure 11provides an overview of the ICS Re-architecture (PEO IWS ICS 2024).

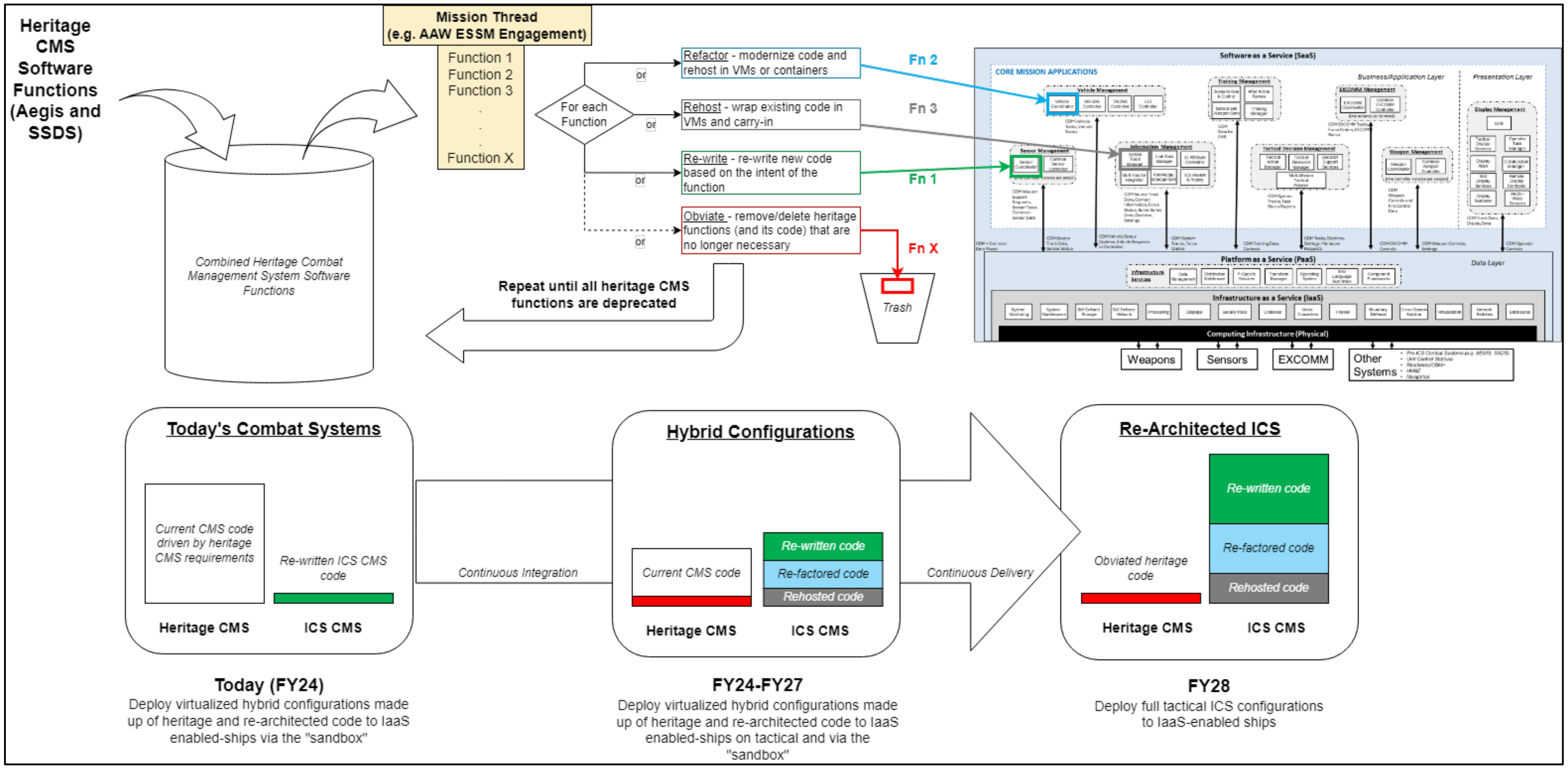

Heritage CMS Software Functions (Aegis and SSDS)
Mission Thread (e.g. AAW ESSM Engagement)
Function 1
Function 2
Function 3
Function X
For each Function
or
Refactor - modernize code and rehost in VMs or containers
Rehost - wrap existing code in VMs and carry-in
Re-write - re-write new code based on the intent of the function
Obviate - remove/delete heritage functions (and its code) that are no longer necessary
Fn 2
Fn 3
Fn 1
Fn X
Trash
Combined Heritage Combat Management System Software Functions
Repeat until all heritage CMS functions are deprecated
Software as a Service (SaaS)
CORE MISSION APPLICATIONS
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)
Computing Infrastructure (Physical)
Weapons
Sensors
EXCOMM
Other Systems
Today's Combat Systems
Current CMS code driven by heritage CMS requirements
Re-written ICS CMS code
Heritage CMS
ICS CMS
Continuous Integration
Hybrid Configurations
Current CMS code
Re-written code
Re-factored code
Rehosted code
Continuous Delivery
Re-Architected ICS
Obviated heritage code
Today (FY24)
Deploy virtualized hybrid configurations made up of heritage and re-architected code to IaaS enabled-ships via the "sandbox"
FY24-FY27
Deploy virtualized hybrid configurations made up of heritage and re-architected code to IaaS enabled-ships on tactical and via the "sandbox"
FY28
Deploy full tactical ICS configurations to IaaS-enabled ships

Figure 11. Integrated Combat System Heritage Code Re-Architecture Approach. Source Program Executive Office, Integrated Warfare Systems–Integrated Combat System (2024).

2. Services-Based Architecture

With ICS processes in place and a high level architecture established, the USN team developed a next level breakdown (ICS service-based architecture) of the system which was led by an external consultant in conjunction with ICS SMEs to help eliminate legacy CS bias.

The resultant notional ICS service-based architecture is expected to grow and change as solution white papers are developed, transforming legacy architectures into a modern containerized set of services for the CMS. Figure 12 provides this notional outline of the reallocation of legacy Aegis and SSDS functions into a services-based architecture.

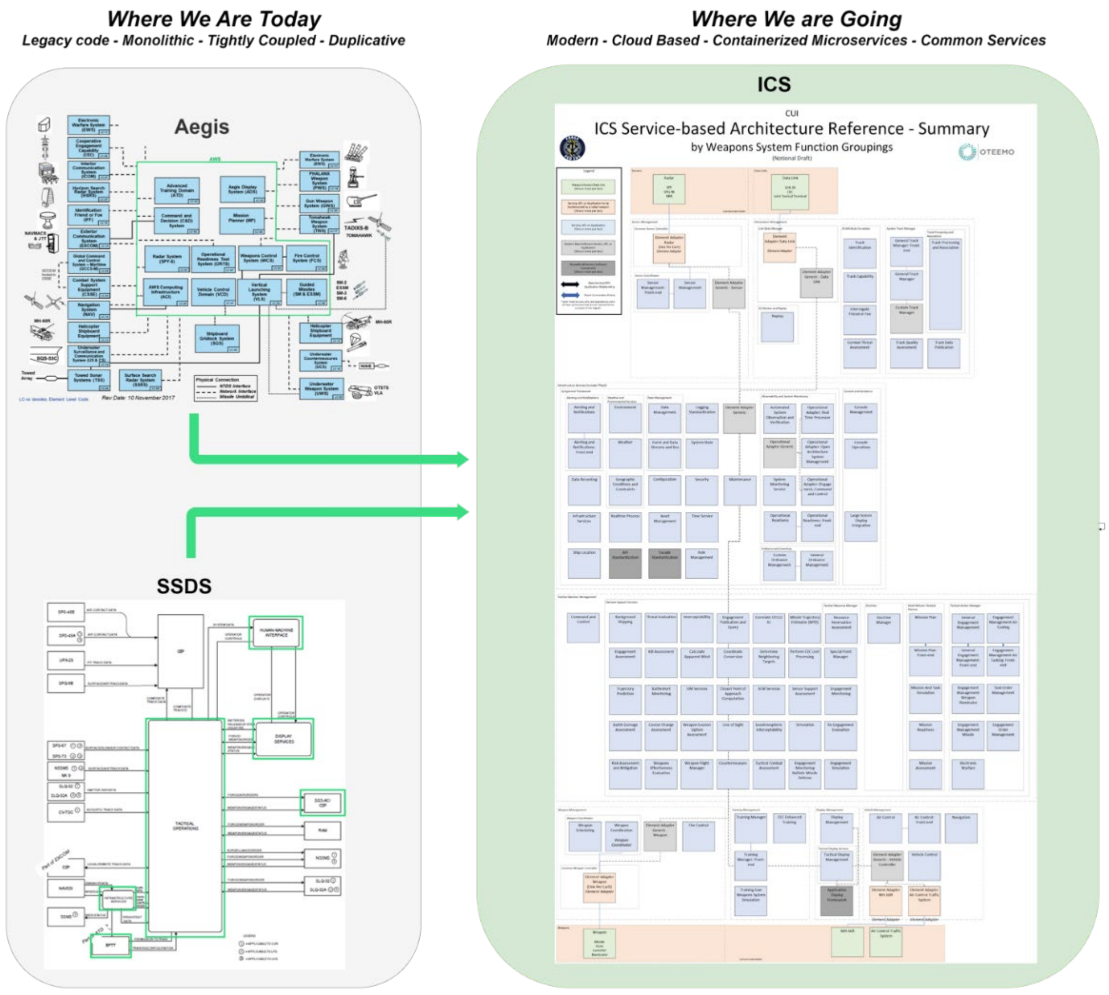

Where We Are Today
Legacy code - Monolithic - Tightly Coupled - Duplicative
Where We are Going
Modern - Cloud Based - Containerized Microservices - Common Services
Aegis
SSDS
ICS
CUI
ICS Service-based Architecture Reference - Summary
by Weapons System Function Groupings
(Notional Draft)
OTEEMO

Figure 12. Mapping Legacy Combat System Code to a Service-Based Architecture

To better assess the size of the effort to refactor two codebases into a common ICS the USN team leveraged the detailed Software Requirements of Aegis and SSDS. At the Software Requirement Specification, or B5, level the team identified Capability Building Blocks (CBBs). CBBs have a history with CS development dating to ~2012 with Aegis BL 9.2 development under the Advanced Capability Build 2016 (ACB-16) program. CBBs are further used on existing BL9 development, BL10, FFG, and other development efforts.

By breaking the refactoring effort into CBBs the team can "tackle" the services-based refactoring by addressing groups of services needed to perform an end capability.

The main benefit of this ICS / CMS re-architecture is the ability to rapidly deliver capability to the war fighter. Modern SW development demands flexibility, scalability, and rapid delivery, driving the DoD to adopt microservices architectures combined with CI/CD pipelines to achieve these goals. This combination enables SW teams to meet ever-evolving threats while maintaining high reliability, scalability, and developer productivity. Without developing a modernized delivery process the significant benefits of microservices would be lost.

E. MODERNIZING SOFTWARE DISTRIBUTION: DELIVERY PIPELINE

Increased capability to threat SW delivery is essential to meet threats to national security through maritime superiority. To provide this capability CI/CD of serviced based SW is needed. As important as CS SW capability is the underlying security of the CS components and the IaaS/PaaS SW.

As the technology landscape evolves, organizations are increasingly adopting microservices architecture and CI/CD pipelines to improve SW delivery efficiency, scalability, and adaptability. Together, these approaches address many of the limitations of traditional SW development methodologies, empowering businesses to innovate rapidly, scale seamlessly, and maintain high-quality applications.

CI/CD pipelines are critical components of modern SW development. CI refers to the practice of frequently integrating code changes into a shared repository, while CD automates the delivery of applications to production after the build process. Together, CI/CD pipelines enable faster, more reliable SW delivery.

1. Faster Time to the War Fighter

CI/CD pipelines streamline the development, testing, and deployment process, significantly reducing the time it takes to deliver SW updates.

- **Automated Testing:** With CI/CD, testing becomes an automated part of the process, ensuring that code changes are validated early and often. This reduces manual testing efforts and accelerates the release cycle.
- **Frequent Releases:** By automating the build, test, and deployment processes, CI/CD enables teams to deploy updates to production on a regular basis, reducing the time between development and customer-facing changes (Humble & Farley, 2010).

2. Improved Code Quality

CI/CD pipelines help maintain high standards of code quality by integrating automated testing and continuous feedback into the development process.

- **Early Detection of Bugs:** Automated testing within the CI pipeline ensures that bugs are detected early in the development cycle, preventing them from propagating to production. This early detection reduces the time and cost associated with fixing bugs later in the process.
- **Consistent Quality Assurance:** CI/CD pipelines enforce a consistent testing process, ensuring that each code change meets predefined quality standards before being deployed (Humble & Farley, 2010).

3. Reduced Risk of Deployment Failures

CI/CD pipelines reduce the risks associated with SW deployment by automating the process and ensuring that each release is thoroughly tested.

- **Continuous Feedback:** Developers receive immediate feedback when issues are detected, allowing them to address problems before they reach production (Humble & Farley, 2010).
- **Automated Rollback:** CI/CD pipelines can include automated rollback mechanisms, so if a deployment fails, the system can revert to the last known stable version without manual intervention (Humble & Farley, 2010).

4. Enhanced Collaboration and Transparency

CI/CD pipelines foster collaboration among developers, testers, and operations teams, promoting a shared responsibility for the success of the application.

- **Collaborative Development:** CI/CD pipelines facilitate communication between teams by providing a single point of integration and visibility for the development process (Humble & Farley, 2010). This transparency helps ensure that all stakeholders are aligned on the status of the application.
- **Continuous Improvement:** With regular deployments, teams can continuously assess and improve the development process, ensuring that workflows remain efficient and aligned with the evolving needs of the business (Humble & Farley, 2010).

5. Reliability and Consistency

By automating the build, test, and deployment processes, CI/CD pipelines reduce the variability associated with manual deployments.

- **Repeatable Deployments:** CI/CD ensures that the same process is followed for every deployment, eliminating human error and ensuring that every deployment is consistent with the previous one (Humble & Farley, 2010).

- **Reliability in Production:** With automated testing and continuous integration, CI/CD pipelines increase the reliability of production releases.

6. Greater Flexibility and Agility

CI/CD pipelines support agile development practices by enabling rapid iterations and frequent releases of new features and bug fixes.

- **Faster Feedback Loops:** With automated testing and continuous integration, teams can quickly assess new code and rapidly respond to feedback from stakeholders (Humble & Farley, 2010).
- **Iterative Development:** CI/CD pipelines support the agile model, coupled with micro services-based architectures it is easier to implement frequent, smaller releases rather than infrequent, large ones. This enables USN developers and the Forge to react quickly to emergent threats and make changes more efficiently.

The combination of microservices architecture and CI/CD pipelines offers significant benefits for modern SW development. Microservices enable scalability, modularity, and flexibility, while CI/CD pipelines streamline the development, testing, and deployment processes. Together, these practices reduce development cycles, improve code quality, and enhance collaboration, ensuring that USN organizations can deliver high-quality SW at scale, respond quickly to threats, and maintain operational agility.

F. CONCLUSION

The ICS-CMS Conceptual Reference Model by Alvin Murphy outlines a framework for modernizing naval combat systems by focusing on integration, modularity, real-time data processing, and advanced technologies like AI. The model aims to ensure naval platforms can adapt to modern warfare challenges by creating efficient, scalable, and adaptable combat systems. It addresses key issues such as system integration, decision support, cybersecurity, and resource constraints, enabling the USN to operate effectively in complex environments.

The ICS framework integrates various subsystems to improve decision-making through real-time data processing, AI, and automation. These technologies help reduce operator workload and speed up decision-making. The development of ICS is structured around the Now, Next, Later vision, which outlines a phased modernization process. The Now phase addresses urgent needs by enhancing legacy systems and ensuring interoperability. The "Next" phase emphasizes flexible, modular architectures and improved cybersecurity. The "Later" phase envisions a networked fleet with AI and autonomous systems, supported by continuous modernization.

The ICS development follows four key phases: ICS Foundation, ICS Enabled, ICS Node, and ICS Fully Realized. Each phase progressively integrates modular systems across platforms and enhances cybersecurity, preparing the USN for future operational needs. The ICS emphasizes agility, interoperability, cost-effectiveness, and scalability to meet diverse mission requirements.

Key enablers of ICS include the Forge Software Factory and the Foundry Hardware Factory. The Forge uses Agile and DevSecOps practices to accelerate SW development and ensure security, while the Foundry focuses on rapidly prototyping and deploying hardware using modular designs and advanced manufacturing techniques. Together, these initiatives allow the USN to respond quickly to emerging threats and ensure technological superiority.

In conclusion, the ICS framework provides a comprehensive modernization strategy that integrates SW, hardware, and advanced technologies. It ensures the USN's CSs remain flexible, resilient, and capable of meeting the evolving challenges of modern warfare. Through initiatives like the Forge and the Foundry, the USN can maintain its operational readiness and adapt to future needs.

This paper endeavored to lay a foundation from purpose-built DoD computing and weapon systems and the SW structures these early capabilities supported, to the path towards adoption of practices and components enabling the USN to "Deliver Resilient Warfighting Capability at the Speed of Relevance."

The following summarizes the key findings of the architectures and process improvements which are guiding efforts and modernizing the USN Surface Force.

1. Monolithic Architecture Overview

Monolithic SW integrates all components—UI, business logic, and data access—into a single codebase, simplifying initial development, deployment, and version control (Fowler & Lewis, 2014; Bass et al., 2021). However, as the application grows, maintaining the codebase becomes challenging, with updates in one area affecting others, leading to time-consuming and error-prone changes (Newman, 2021).

a. Challenges with Monolithic Systems

Monolithic systems are tightly coupled, meaning minor updates require widespread changes, complicating scaling and maintenance. This increases testing and redeployment costs, slowing progress. As the system grows, the initial simplicity diminishes, making the codebase harder to manage

b. Deployment Simplicity

While monolithic systems offer simple deployment as a single unit, this simplicity becomes less effective over time, as managing the system becomes increasingly difficult as the application evolves.

2. Transition to Microservices and the Navy's Approach

The USN has transitioned from purpose-built computing systems to modern architectures, enabling the fielding of granular microservices-based SW in weeks rather than months or years. This shift significantly reduces the cost of development, testing, certification, and distribution of capabilities. The USN's core capabilities and their limitations are outlined in Table 4–1, showcasing this technological progression.

a. Microservices Architecture Overview

Microservices architecture focuses on service independence, loose coupling, and decentralized data management. By adopting principles like the SRP, automated deployment, and resilience, organizations can create scalable and flexible systems. Microservices support business alignment through DDD and centralized concerns via the API Gateway, contributing to efficient and reliable applications.

b. Challenges with Microservices

Despite its advantages, microservices architecture comes with increased complexity in managing multiple services, communication overhead, and data management. Testing, deployment, and operational challenges can slow efficiency. Security concerns, service discovery, and organizational impacts also need addressing to ensure successful implementation.

c. DevOps Integration with Microservices

DevOps practices enhance microservices by promoting automation, collaboration, and continuous improvement. Practices like continuous integration, deployment, and automated testing enable faster market delivery and improved quality. However, integrating DevOps with microservices introduces complexity, tooling challenges, and security concerns that organizations must overcome.

3. Large Unmanned Surface Vessel Integrated Combat System Development and Challenges

The USN's development of the LUSV ICS encountered significant challenges due to legacy procurement approaches. Systems like Technology Insertions 12 and 16 suffered from hardware obsolescence, which hindered the deployment of cutting-edge solutions. To address this, the DoD turned to OTAs for more flexibility in R&D, enabling rapid prototyping and innovation.

The challenges of legacy procurement led to a focus on agile procurement strategies and integrating modernization efforts to combat obsolescence. OTAs, contractor education, and prioritizing long-term value over cost efficiency are key strategies for overcoming procurement barriers and ensuring timely system delivery.

4. Integrated Combat System Conceptual Framework for Modernization

The ICS framework emphasizes integration, modularity, real-time data processing, and AI to modernize naval combat systems. The framework supports efficient decision-making, improved cybersecurity, and the integration of new technologies. The phased approach—Now, Next, Later—focuses on urgent needs, flexible architectures, and autonomous systems, preparing the USN for future challenges.

5. Integrated Combat System Development Phases and Key Enablers

ICS development follows four phases: ICS Foundation, ICS Enabled, ICS Node, and ICS Fully Realized, progressively integrating modular systems and enhancing cybersecurity. Key enablers like the Forge Software Factory and the Foundry Hardware Factory accelerate SW development and hardware prototyping using agile practices and modular designs. These initiatives ensure the USN can respond quickly to emerging threats.

6. Integrated Combat System Modernization Strategy

The ICS framework integrates SW, hardware, and advanced technologies, ensuring that the USN's combat systems remain flexible, resilient, and capable of adapting to modern warfare challenges. Through initiatives like the Forge and the Foundry, the USN can maintain operational readiness and continuously modernize its capabilities to meet future needs.

G. IMPLICATIONS FOR FUTURE NAVY

The transition from monolithic computing SW to modern architectures, particularly microservices, has significant implications for the future of the USN. The monolithic approach, characterized by a unified codebase that simplifies initial development and deployment, faces challenges as systems grow. The difficulty in managing large codebases, the risk of error-prone updates, and the complexity of scaling hinder long-term effectiveness. As applications evolve, the simplicity of monolithic architecture diminishes, making them less adaptable to modern operational needs.

In contrast, microservices architecture provides a more flexible, scalable, and modular approach that better aligns with the USN's evolving technological requirements. By emphasizing service independence, loose coupling, and decentralized data management, microservices allow for quicker updates, enhanced resilience, and more efficient systems that can evolve rapidly in response to operational demands. However, the adoption of microservices comes with its own challenges, including the complexity of managing multiple services, testing difficulties, increased operational overhead, and security concerns.

To fully leverage the advantages of microservices, the USN must continue to integrate and refine modern practices like DevOps, which enable continuous integration, automated deployment, and rapid iteration. This allows for faster time-to-market, improved quality, and better scalability. However, DevOps adoption must overcome challenges such as complexity, tooling requirements, and cultural shifts within the organization.

Furthermore, the USN's transition from legacy procurement methods to more agile models, like OTAs, allows for faster innovation and the development of cutting-edge solutions, as demonstrated in the LUSV program. The ability to rapidly prototype and deploy systems, such as virtualized combat SW, helps the USN maintain operational relevance and readiness in the face of evolving threats.

Modernizing USN combat systems through initiatives like the ICS-CMS which emphasizes modularity, real-time data processing, and advanced technologies such as AI. By improving decision-making, reducing operator workload, and enhancing cybersecurity, ICS will ensure that the USN's combat systems remain flexible and resilient. The use of Agile and DevSecOps in the development of ICS, along with rapid hardware prototyping, supports a more adaptive and efficient approach to meeting future operational needs.

H. FINAL THOUGHTS

The shift from monolithic to microservices-based architectures, coupled with agile procurement practices and modernization efforts, positions the USN to maintain technological superiority and adaptability in the rapidly changing landscape of modern warfare. Through initiatives like the Forge Software Factory and the Foundry Hardware Factory, the USN can continue to innovate and respond effectively to emerging challenges.

But these efforts are not without challenges. The scope of the effort to re-architect CS SW and build the fully realized ICS is both very large and complex. And, while rapid prototype efforts like USV ICS CI demonstrate the value of streamlined procurement vehicles like the OTA, industry teams will gain greater control of technical solutions and costs.

It is incumbent on USN leadership and the government to exert and maintain the right balance of oversight and control and maintain the technical acumen to oversee these developments. The right balance is one where small and highly capable teams can maintain critical oversight while not significantly impeding the pace of development to Deliver Resilient Warfighting Capability at the Speed of Relevance!

LIST OF REFERENCES

Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley.

Beck, K. (2003). *Extreme programming explained: Embrace change*. Addison-Wesley.

Bresnahan, T. F., & Trajtenberg, M. (1995). General purpose technologies: 'Engines of growth'? *Journal of Econometrics*, *65*(1), 83–108.

Burns, B., Beda, J., & Hightower, K. (2018). *Designing distributed systems: Patterns and paradigms for scalable, reliable services*. O'Reilly Media.

Defense Acquisition University. (2010). *Requirements management: A primer for the defense acquisition professional*. DAU Press.

Department of the Navy. (2020). *Naval systems engineering*. U.S. Government Publishing Office.

Department of the Navy. (2021). *Modernizing naval capabilities through innovation*.

Dragoni, N., Dustdar, S., Larsen, S. T., & Mazzara, M. (2021). Microservices: Migration of a mission critical system. *IEEE Transactions on Cloud Computing*, *14*(5).

Dragoni, N., Giazzi, F., Larsen, S. T., Mazzara, M., Montesi, F. … Safina, L. (2017). *Microservices: Yesterday, today, and tomorrow*. Present and Ulterior Software Engineering

Fowler, M. (2006). *Continuous integration*. Martin Fowler. https://martinfowler.com/articles/continuousIntegration.html

Fowler, M. (2018). *How to break a Monolith into Microservices*. Martin Fowler. https://martinfowler.com/articles/break-monolith-into-microservices.html

Government Accountability Office. (2019). *Defense acquisitions: Key considerations for implementing effective procurement processes*. GAO

Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes: Up & running: Dive into the future of infrastructure*. O'Reilly Media.

Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.

Kalske M., Mäkitalo N., & Mikkonen T. (2018). Challenges when moving from monolith to microservice architecture. In I. Garrigós & M. Wimmer (Eds.), *Current trends in web engineering: ICWE 2017 international workshops*. Springer.

Kuryazov, D., Jabborov, D., & Khujamuratov, B. (2020). Towards decomposing monolithic applications into microservices. *Proceedings of the 14th International Conference on Application of Information and Communication Technologies (AICT)*, 1–5. https://doi.org/10.1109/AICT50176.2020.9368605

Lewis, J., & Fowler, M. (2014). *Microservices: A definition of this new architectural term*. Martin Fowler. https://martinfowler.com/articles/microservices.html

Mazzara, M., & Meyer, B. (2017). *Present and ulterior software engineering*. Springer.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8), 114–117.

Morris, K. (2016). *Infrastructure as code: Managing servers in the cloud*. O'Reilly Media.

Murphy, A. (2022). *Integrated Combat System (ICS) Combat Management System (CMS) conceptual reference model* (NSWCDD/TR-22/48). Naval Surface Warfare Center Dahlgren Division.

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: Aligning principles, practices, and culture*. O'Reilly Media.

Newman, S. (2021). *Building microservices: Designing fine-grained systems*. O'Reilly Media.

Office of the Under Secretary of Defense for Research and Engineering. (2022). *Department of Defense prototyping guidebook*. Department of Defense.

Program Executive Office, Integrated Warfare Systems–Integrated Combat System, (PEO IWS ICS). (2024). *Combat system re-architecture strategy and approach*.

Program Executive Office, Integrated Warfare Systems. (2022). *Now, next, later – A technical strategy for evolving towards the Integrated Combat System*. Naval Sea Systems Command.

Richards, M. (2015). *Software architecture patterns*. O'Reilly Media.

Shalf, J. (2020). The future of computing beyond Moore's Law. *Philosophical Transactions of the Royal Society A*, *378*(2166), http://dx.doi.org/10.1098/rsta.2019.0061