

# **Application Programming Interface (API) Technical Guidance**

Minimum Viable Capability Release (MVCR) 2



April 2025

Office of Systems Engineering and Architecture

Office of the Under Secretary of Defense  
for Research and Engineering

Washington, D.C.

Distribution Statement A. Approved for public release. Distribution is unlimited.

## **Application Programming Interface (API) Technical Guidance**

Minimum Viable Capability Release (MVCR) 2

Office of Systems Engineering and Architecture  
Office of the Under Secretary of Defense for Research and Engineering  
3030 Defense Pentagon  
Washington, DC 20301  
[osd-sea@mail.mil](mailto:osd-sea@mail.mil) | Attn: Software Engineering  
<https://www.cto.mil/sea>

Distribution Statement A. Approved for public release. Distribution is unlimited.  
DOPSR Case # 25-T-2035.

Approved by  
Principal Deputy Executive Director, Systems Engineering and Architecture

**API Technical Guidance Change Record**

Date	Publication	Change
October 2023	Initial release.	
July 2024	Minimum Viable Capability Release (MVCR)-1	New Sections 5 and 6. API Team review and updates.
April 2025	Minimum Viable Capability Release (MVCR)-2	New Sections 7,8, and 9.

## Contents

1	Introduction.....	1
1.1	Purpose and Scope.....	1
1.1.1	API Open Standards .....	3
1.2	Intended Audience.....	4
1.3	Document Relationships.....	4
1.4	DoD Landscape .....	5
1.4.1	API System Development Paradigm .....	6
1.4.2	Interoperability .....	8
1.4.3	Legacy Systems .....	10
1.4.4	Other API Terms .....	10
2	Data Access .....	13
2.1	System Access Design Considerations.....	13
2.2	Data Access Example.....	15
2.3	API Stakeholder Roles .....	16
3	API Project Governance .....	19
4	Cybersecurity.....	22
4.1	Importance of Securing APIs in DoD.....	23
4.2	API Cybersecurity Challenges .....	23
4.3	Cybersecurity Best Practices .....	24
4.3.1	Implement Robust Authentication and Authorization Mechanisms.....	25
4.3.2	Ensure Input Validation and Output Encoding.....	25
4.3.3	Encrypt and Protect Data in Classified Environment .....	26
4.3.4	Monitor and Log for Early Threat Detection and Response.....	26
4.3.5	Securing the API Gateway.....	26
4.3.6	Ensure API Security Testing and Compliance in the DoD.....	28
4.3.7	Implement an API Governance Process .....	29
5	Design and Implementation Principles .....	30
5.1	Common Data Model .....	30
5.2	Open Standards and Protocols.....	30
5.3	Design for Security Compliance .....	31
5.4	Developmental Testing and Validation Processes.....	32
5.5	Collaboration and Communication.....	33
5.6	API Parameters for Pagination, Sorting, and Filtering.....	34
5.7	API Metrics .....	35

## Contents

6	Development, Security, and Operations (DevSecOps).....	37
6.1	DevSecOps Enabling Technologies .....	38
6.2	Monitoring and Logging.....	39
7	Testing .....	41
7.1	Software Testing.....	41
7.1.1	Unit Testing .....	41
7.1.2	Integration Testing.....	42
7.1.3	Application Testing .....	43
7.1.4	Other Types of Testing .....	44
7.2	Test Environments.....	44
7.2.1	Ephemeral versus Persistent Environments.....	45
7.2.2	Test Data.....	46
7.2.3	Testing Suite Infrastructure .....	46
7.3	Testing Skill Sets.....	47
7.4	Contract Testing .....	48
7.5	AI-Enabled Testing .....	49
7.5.1	Test Generation.....	50
7.5.2	Verifying Correctness.....	50
7.6	Zero Trust Testing .....	51
7.6.1	API Authentication and Authorization .....	51
7.6.2	API Replay Attack Vulnerabilities .....	51
7.6.3	Securing APIs with ZT .....	51
7.6.4	API Testing and Development Environments Are Assets.....	52
7.6.5	API Monitoring.....	52
8	Enhancing API Efficiency and Accessibility Using Software Developer Kits and Libraries .....	53
8.1	Introduction of SDKs and Libraries .....	53
8.2	What Is an SDK? .....	54
8.3	Origin of SDKs.....	56
8.3.1	Benefits of an SDK.....	57
8.4	Support and Maintenance .....	57
9	Real-Time System APIs .....	58
9.1	General RTS API Principles.....	58
9.1.1	RTS Examples .....	59
9.1.2	RTS Operational Regime.....	61
9.1.3	Mission Planning Considerations .....	61
9.1.4	Constraints on Real-Time Systems.....	62

9.1.5 Transport Considerations.....	62
9.1.6 RTS Isolation from API Service.....	62
9.1.7 API Adapters and Gateways.....	62
9.1.8 RTS API Pattern Variations.....	64
9.2 RTS-Tailored API Recommendations.....	65
9.2.1 Data Visibility and Accessibility .....	65
9.2.2 Format.....	66
9.2.3 Security.....	67
9.2.4 Reusability .....	68
9.2.5 Reliability .....	69
9.2.6 RTS Testing.....	70
9.2.7 Scalability .....	71
9.2.8 Sorting .....	72
10 Data Standards .....	74
10.1 Plumbing Data Standards for APIs.....	74
10.2 Content Data Standards for APIs .....	77
11 Conclusion .....	81
Appendix A: API Project Governance Considerations.....	82
Appendix B: Common API Vulnerabilities and Threats .....	85
Appendix C: API Security Challenges.....	87
Glossary .....	89
Acronyms.....	103
References.....	105

## Figures

Figure 1-1. API Context Diagram from a System Perspective .....	2
Figure 1-2. API Context Diagram from a Data Perspective .....	3
Figure 1-3. Document Relationships .....	5
Figure 1-4. APIs in a System of Systems Context.....	7
Figure 1-5. Interoperability Concepts and their Relationships .....	8
Figure 1-6. Component Relationship Diagram .....	9
Figure 1-7. API Ecosystem Terms.....	11
Figure 2-1. Notional UAV System Hierarchy and Mission-Critical Data .....	15
Figure 2-2. Notional Example of Data Communications and API Scope.....	16
Figure 4-1. DevSecOps Infinity Diagram.....	22
Figure 9-1. Example of UAV Video Streaming RTS System .....	61

## Contents

Figure 9-2. Generalized RTS API Pattern .....	63
Figure 9-3. Fully Decoupled Pattern (Pattern A) .....	64
Figure 9-4. Fused to External Application (Pattern B) .....	64
Figure 9-5. Fused to RTS (Pattern C) .....	64
Figure 10-1. Many Aspects of Interoperability: An Analogy .....	74
Figure 10-2. Hierarchy of Data Models for Shared Understanding.....	78

## Tables

Table 2-1. System Access Design Considerations.....	14
Table 2-2. API Stakeholder Roles.....	17
Table 3-1. High-Quality API Attributes and Benefits .....	19
Table 9-1. RTS Example Components .....	60
Table A-1. API Project Governance Considerations .....	82

This page is intentionally blank.

# 1 Introduction

In the rapidly evolving landscape of modern warfare, the U.S. Department of Defense (DoD) relies on advancements in technology to maintain a competitive edge in joint warfare capabilities. Central to these advancements are software application programming interfaces (APIs). An API is a structured, software -level data transport-supported mechanism for system developers to discover and for programmed systems to request and send/receive information and functionality to and from the same or other systems. Properly designed APIs promote interoperability, security, and scalability.

Interoperability, “the ability to act together coherently, effectively, and efficiently to achieve tactical, operational, and strategic objectives” (CJCSI RSI 2019), is a priority of the Joint Chiefs of Staff. Interoperability is crucial to modern software, joint warfighting, artificial intelligence (AI) superiority, and achieving the Deputy Secretary of Defense Data Decrees (DepSecDef May 2021).

APIs are essential to interoperability (Brady and Dianic 2022). APIs facilitate data sharing, collaboration, and the seamless integration of systems and capabilities across different branches and units within the Department and with allies (e.g., NATO). Other key concepts of APIs include sensor fusion and Internet of military things (IoMT) operational integration, emerging technology adoption, rapid prototyping and experimentation, ecosystem development and innovation, and protection of critical and emerging technologies.

APIs facilitate seamless communication among diverse software systems and enable the creation of sophisticated, integrated applications. Composability of APIs allows numerous capabilities to be aggregated rapidly into new and distinct capabilities.

The Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) office of Systems Engineering and Architecture (SE&A) led the development of this document in collaboration with the OUSD for Acquisition and Sustainment (A&S); a team involving Joint All-Domain Command and Control (JADC2); and an API Technical Guidance work group, which includes members from the Services, other government agencies, industry, universities, and academia focusing on improving DoD software engineering.

## 1.1 Purpose and Scope

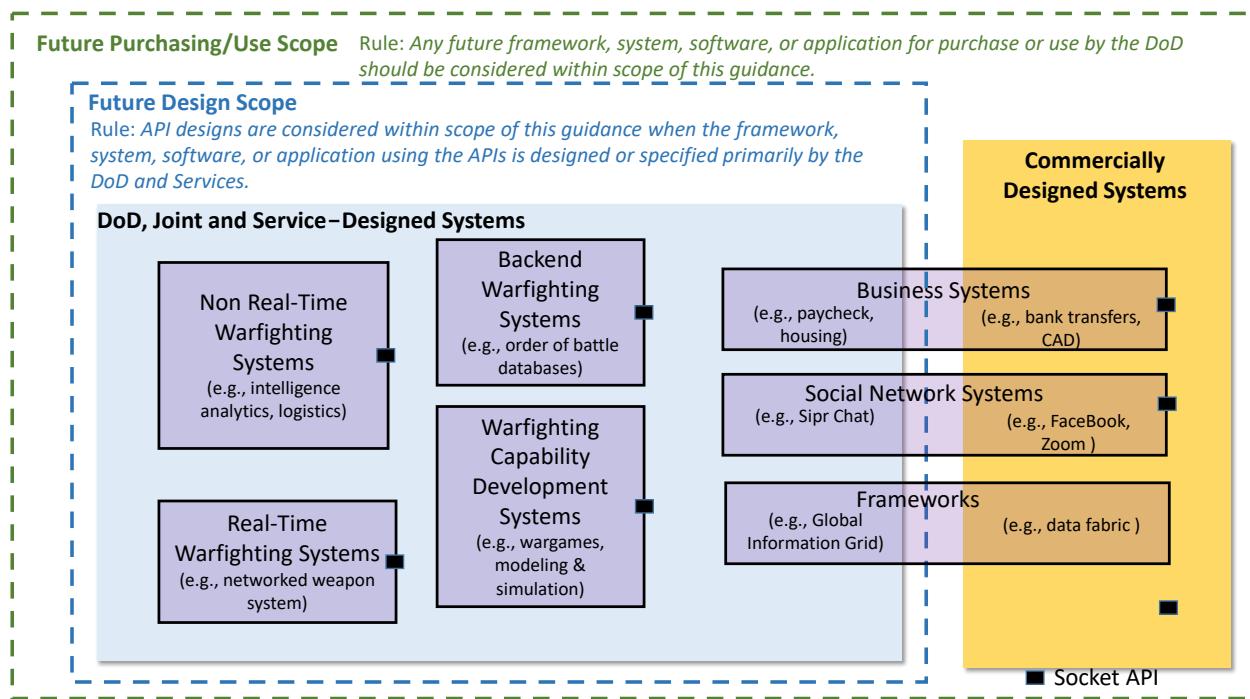
This guide provides an overview of API concepts in software development, including definitions (Section 1), governance (Section 2), and cybersecurity (Section 3). In Section 4 on design and implementation principles, the guide recommends the Department move toward an “API-first” approach. Sections 5 and 6 discuss development, security and operations (DevSecOps) and testing, also central to software engineering.

This MVCR-2 includes new content on software developer kits (SDKs) and libraries (Section 7), real-time system (RTS) APIs (Section 8), and data standards (Section 9).

Future versions of this guidance will include additional topics which the work group is continuously tracking through a topic backlog. Prior to new work, the work group meets to revise, prioritize, and review the topic backlog and identify leads for the writing task for each topic.

This document includes use cases, lessons learned, and best practices from DoD and industry. Although other guides exist, this guide emphasizes the importance of enhancing and advancing the DoD warfighting capabilities of the near future to support the Combined JADC2 (CJADC2) vision and to secure information interoperability across the DoD. (See also Appendix C: API Security Challenges for more detail about the CJADC2 vision.) The guide describes an API framework to help programs define their technical baseline for delivering future systems that support the DoD enterprise and warfighter mission requirements.

Figure 1-1 illustrates the scope of APIs covered in this guidance from a system perspective.



**Figure 1-1. API Context Diagram from a System Perspective**

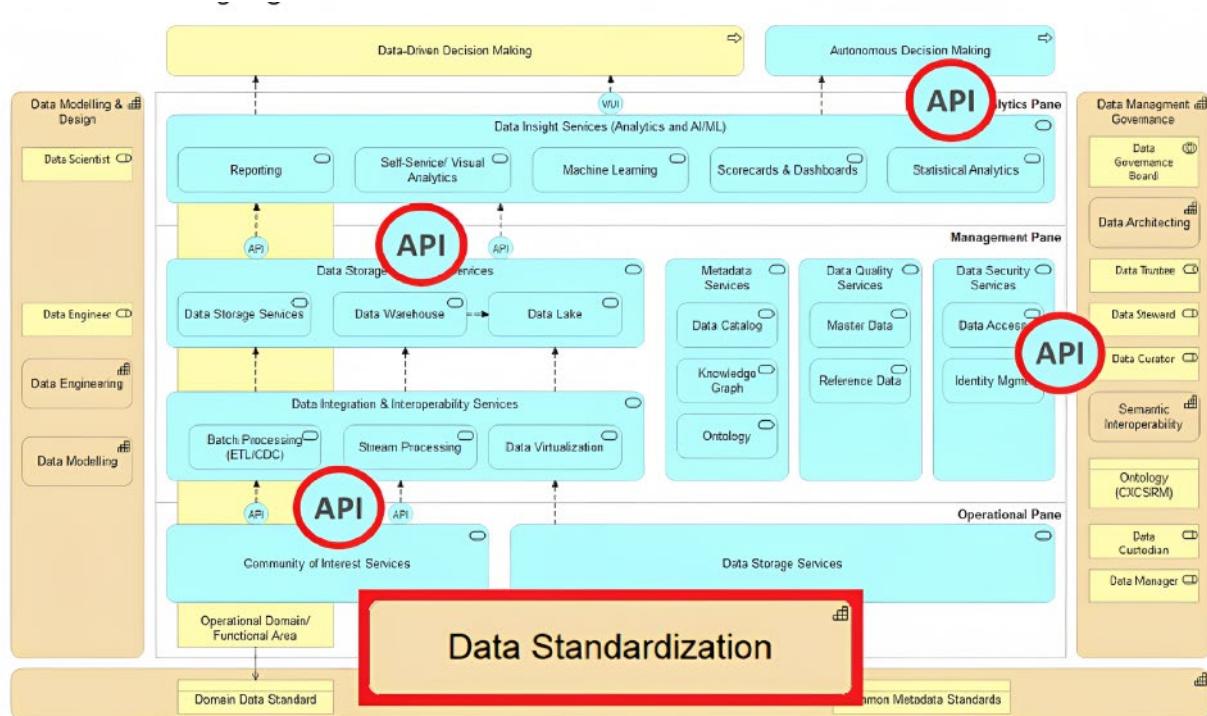
This figure views an API as a socket connection between two systems. In general, a socket (e.g., Web, Berkley, Windows, Unix, Linux, Java) is an abstract representation for the local endpoint of a network communications path. This perspective dichotomizes API ecosystems into those designed for the DoD and commercial industry.

Four types of DoD-specific warfighting systems are non-real-time (e.g., intelligence analytics, logistics); real-time (e.g., networked weapons); back-end (e.g., order of battle); and capability development (e.g., wargames, modeling and simulation); however, the commercial API ecosystems such as business systems (e.g., paychecks); social networks (e.g., SIPR chat, other ChatOps); or other frameworks (e.g., Global Information Grid) are also partially in scope. Each such system contains one or more open or proprietary API socket interfaces connecting with other systems. The future design scope of APIs includes the four DoD systems and, in part, the four commercial systems.

### 1.1.1 API Open Standards

Moving forward, the DoD should maximize APIs developed using open standards and minimize the proprietary data interfaces the governance does not control. These requirements must be clearly addressed within the acquisition processes and contracting language to avoid proprietary, contractor-specific, vendor lock-in integrations throughout the program life cycle.

The scope of APIs also can be seen from a data perspective as shown in Figure 1-2.



Source: (NATO 2023)

**Figure 1-2. API Context Diagram from a Data Perspective**

This figure views an API as an automated data standard between two services. In general, a data standard is any documented agreement on the representation, format, definition, structuring, tagging, transmission, manipulation, use, and management of data (EPA 2023). An automated

data standard or API can reside at various levels including between autonomous decision making and data insight/analytic services (e.g., reporting, machine learning, statistical analysis); analytics services and storage services (e.g., data warehouse, data lake); data integration and interoperability services (e.g., batch or stream processing or data visualization) and community of interest services; or data management and governance and the management plane (e.g., data quality and security). Thus, any API in use by, designed by, or specified by the DoD, Joint, or Services is considered within scope of this guidance.

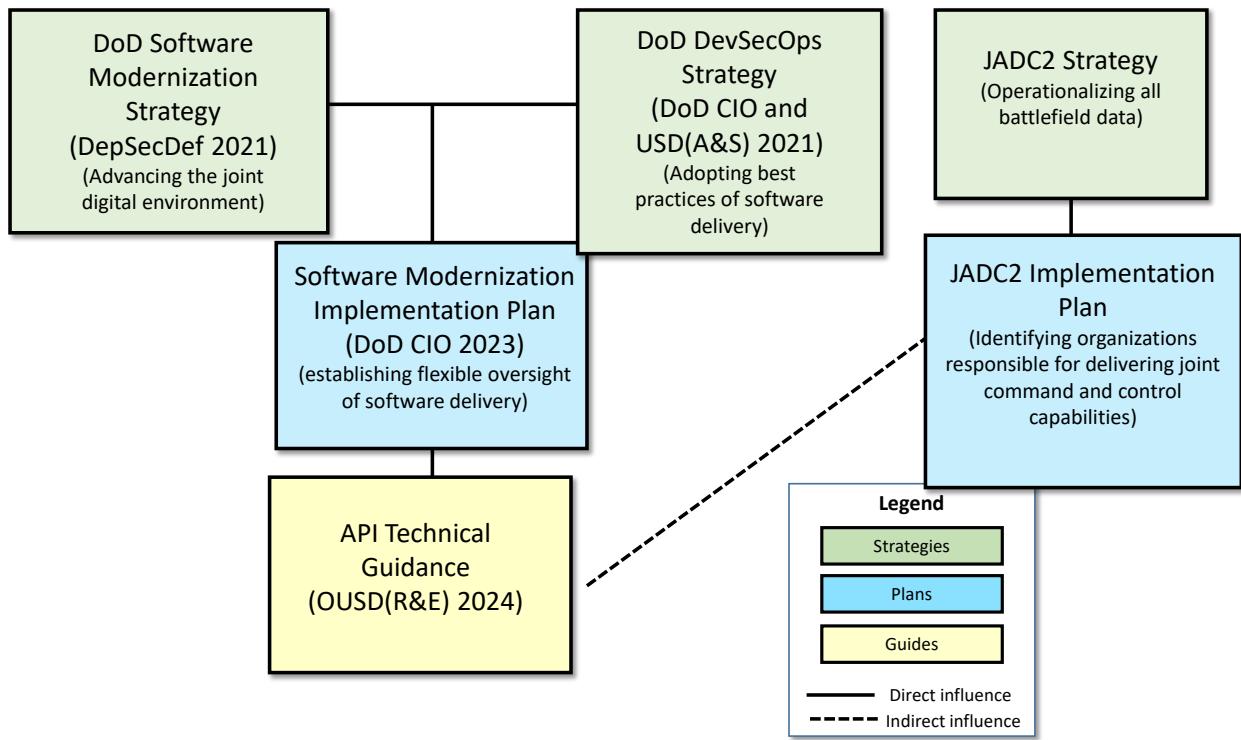
## 1.2 Intended Audience

This document is intended for a range of stakeholders involved in the design, development, deployment, and management of APIs across the DoD, industry, and academia, including the following:

- Architects, designers, developers, and testers responsible for designing and implementing APIs and data frameworks
- Program Managers (PMs) responsible for overseeing API development and deployment
- Security professionals responsible for ensuring the security and compliance of APIs
- Policy professionals responsible for maintaining policy for the DoD
- Acquisition professionals responsible for creating acquisition guides, pathways, and policy
- Operations and support staff responsible for maintaining and monitoring APIs

## 1.3 Document Relationships

Figure 1-3 shows other guidance documents related to this topic and their relationships.

**Figure 1-3. Document Relationships**

This guide is primarily related to three DoD strategies for advancing the joint digital environment: DoD Software Modernization Strategy (DepSecDef November 2021), the DevSecOps Strategy (DoD CIO and USD(A&S) September 2021), and the JADC2 Strategy (Hoehn 2022).

The guide discusses adopting best practices (DoD CIO October 2023) and operationalizing battlefield data (Hoehn 2022). The guide is influenced directly by the Software Modernization Implementation Plan (DoD CIO 2023) and the JADC2 Implementation Plan (DoD 2022) and indirectly by close collaboration with the Joint Staff.

## 1.4 DoD Landscape

Brady and Dianic (2022) observed that “in contrast to commercial industry and modern web economy. DoD lacks a coherent API ecosystem.” They note that 21<sup>st</sup> century businesses know investing in an API strategy pays significant dividends, and properly designed APIs or exposed interfaces (with controlled access) promote interoperability, security, and scalability.

The DevSecOps strategy (DoD CIO October 2023) also notes the challenge of having to rely on few vendors for certain interfaces: “DoD must acknowledge a lock-in posture; recognizing vendor lock-in, *and* recognizing product, version, architecture, platform, skills, legal, and mental lock-in also exist” (DoD CIO and USD(A&S) September 2021).

This document considers the following DoD API goals:

- Combined Joint All-Domain Command and Control (CJADC2)
- Modular Open Systems Approach (MOSA)
- DoD Data Strategy (September 2020) VAULTIS Goals which must be achieved to become a data-centric DoD.
  - Make Data Visible - Consumers can locate the needed data.
  - Make Data Accessible - Consumers can retrieve the data.
  - Make Data Understandable - Consumers can recognize the content, context, and applicability.
  - Make Data Linked - Consumers can exploit data elements through innate relationships.
  - Make Data Trustworthy - Consumers can be confident in all aspects of data for decision making.
  - Make Data Interoperable - Consumers have a common representation/ comprehension of data.
  - Make Data Secure - Consumer data is protected from unauthorized use/manipulation.

#### 1.4.1 API System Development Paradigm

An “application program” is a software system “implemented to satisfy a particular set of requirements” (NIST SP 1800-21). APIs help organizations “connect the many different application programs used in day-to-day operations. For developers, APIs provide communication between application programs, simplifying their integration” (IBM API 2023).

An application program resides on a host hardware system, which may be part of a larger system, which in turn may be part of a larger system of systems (SoS). The application program calling the API can be entirely machine code or could have a user interface. Figure 1-4 shows System 1 and 3 application programs connecting with System 3 end point (end user) using API messaging. For purposes of this document, these systems are divided into two types of systems from an API standpoint: **organic systems** (e.g., System 2) contain the data and functionality (herein just called data) needed by the broader enterprise, and **external systems** (e.g., Systems 1 and 3) are those that need and request access to the organic data. “Organic data” in this context merely indicates that the data resides on the system, not necessarily that the system was the first originator of the data.

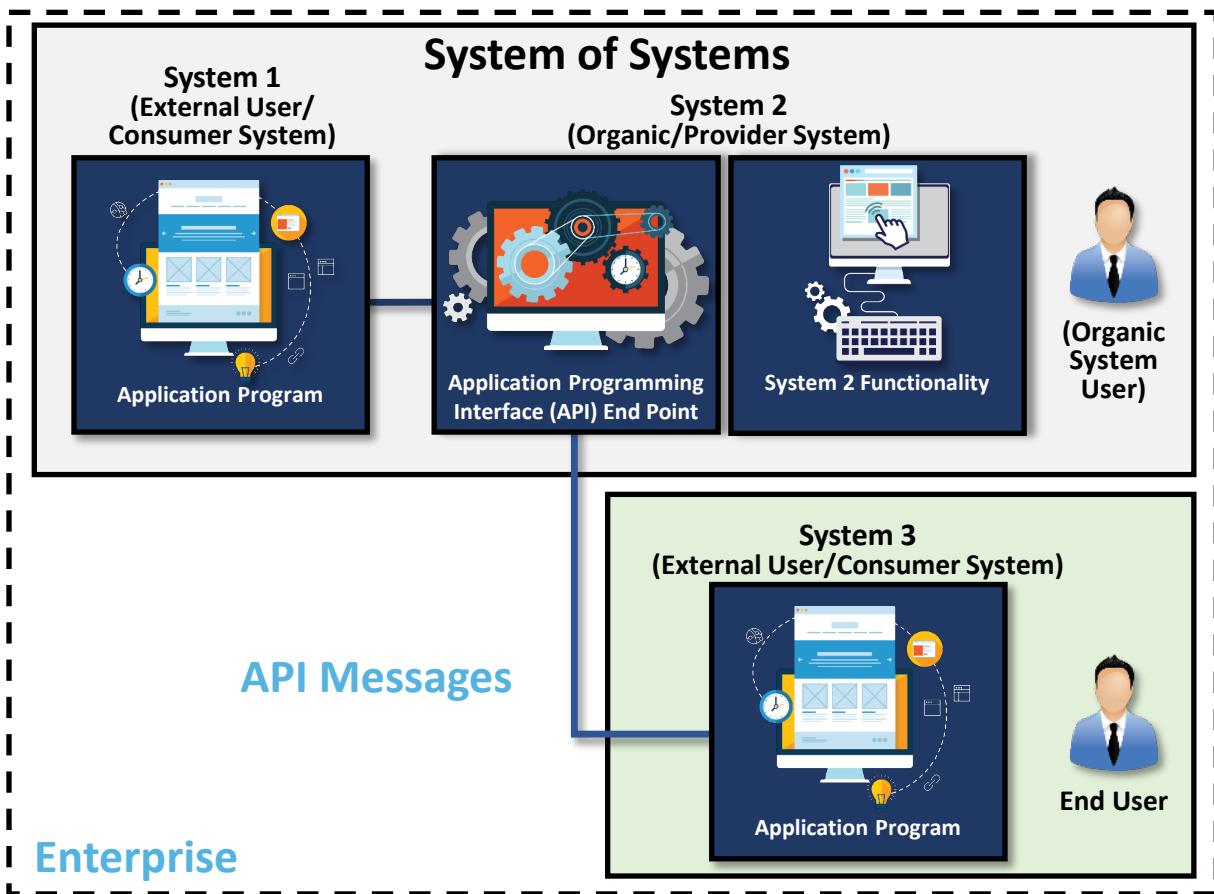


Figure 1-4. APIs in a System of Systems Context

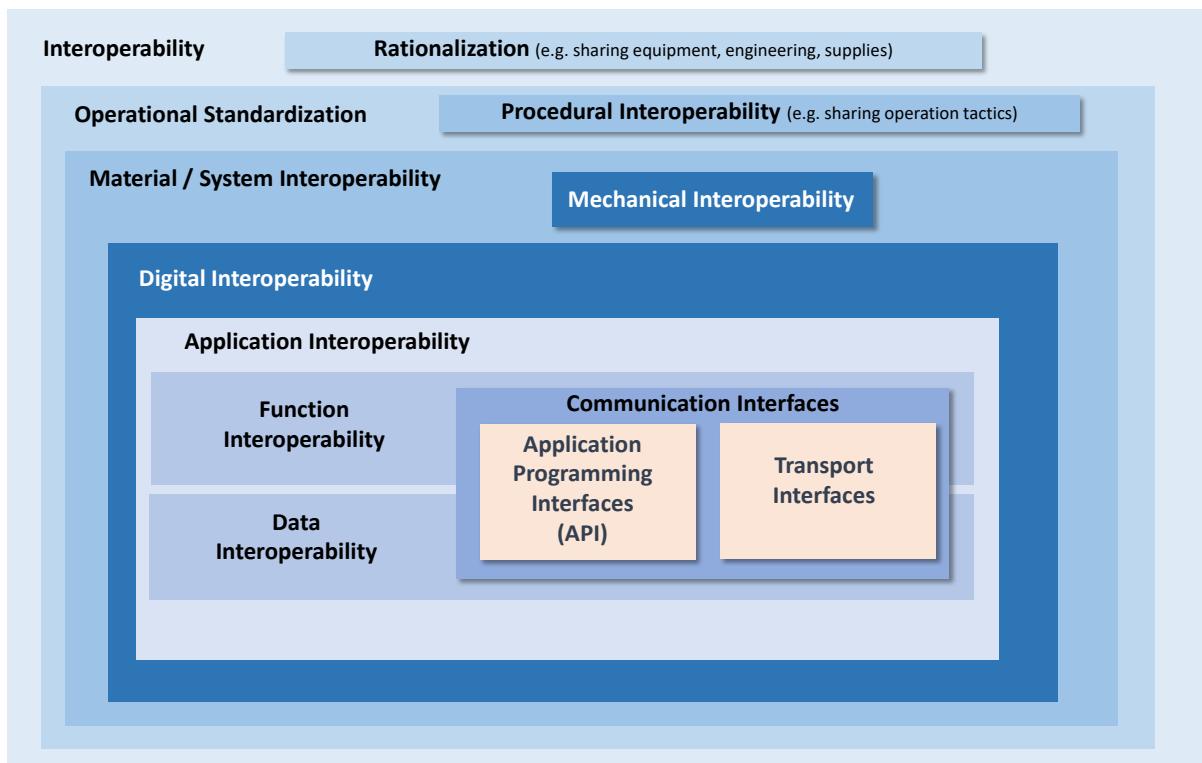
An API may be used in or developed for a single system or for an SoS environment. Within a single system, the API may meet the same needs as the application program. In a broader SoS context, the same API may or may not meet the needs of external systems and use cases.

For DoD, the API challenge is that each system is acquired via contracts from a specific commercial industry, military industry, or government provider. APIs and application programs in these systems may be designed by a single provider or alternatively by a consortium of providers and stakeholders. These designs should follow guidance from section 1.1.1, focusing on open standards-based APIs; however, the actual APIs and application programs implemented into code by providers are for a specific system or set of systems.

A crucial influence on the design of the APIs is the overall development environment in which they are used. Principles of security, trust, dependencies, test, and production all must be considered (see Section 5).

### 1.4.2 Interoperability

Interoperability is referenced in many contexts and forms, so establishing where APIs reside among the different concepts is worthwhile. Figure 1-5 shows how the common terms relate to one another.



**Figure 1-5. Interoperability Concepts and their Relationships**

Data strategies, architectures, and frameworks are an integral part of interoperability and often use APIs. APIs are used in conjunction with broader frameworks, which could be called models, ecosystems, capabilities, protocols, and messaging standards (ISO/IEC 7498-1 1994) (Computer Networking Notes 2023).

Some military standard tactical data links communicate with each other where the messages are predetermined and no programming interface is needed (e.g., Link 16). Figure 1-6 shows the relationship among many kinds of systems and components. These relationships are valuable in establishing a lexicon relating to APIs. For example, remote procedure calls are considered a form of software messaging, but APIs have a bit broader reach, spanning into data store, application software, and software messaging types of components. In the context of this document, any system large or small supporting an operational, business, or enterprise activity can be considered a mission system. APIs do not cover data transport, only going as far as covering the software messaging that interface with the data transport systems.

## 1. Introduction

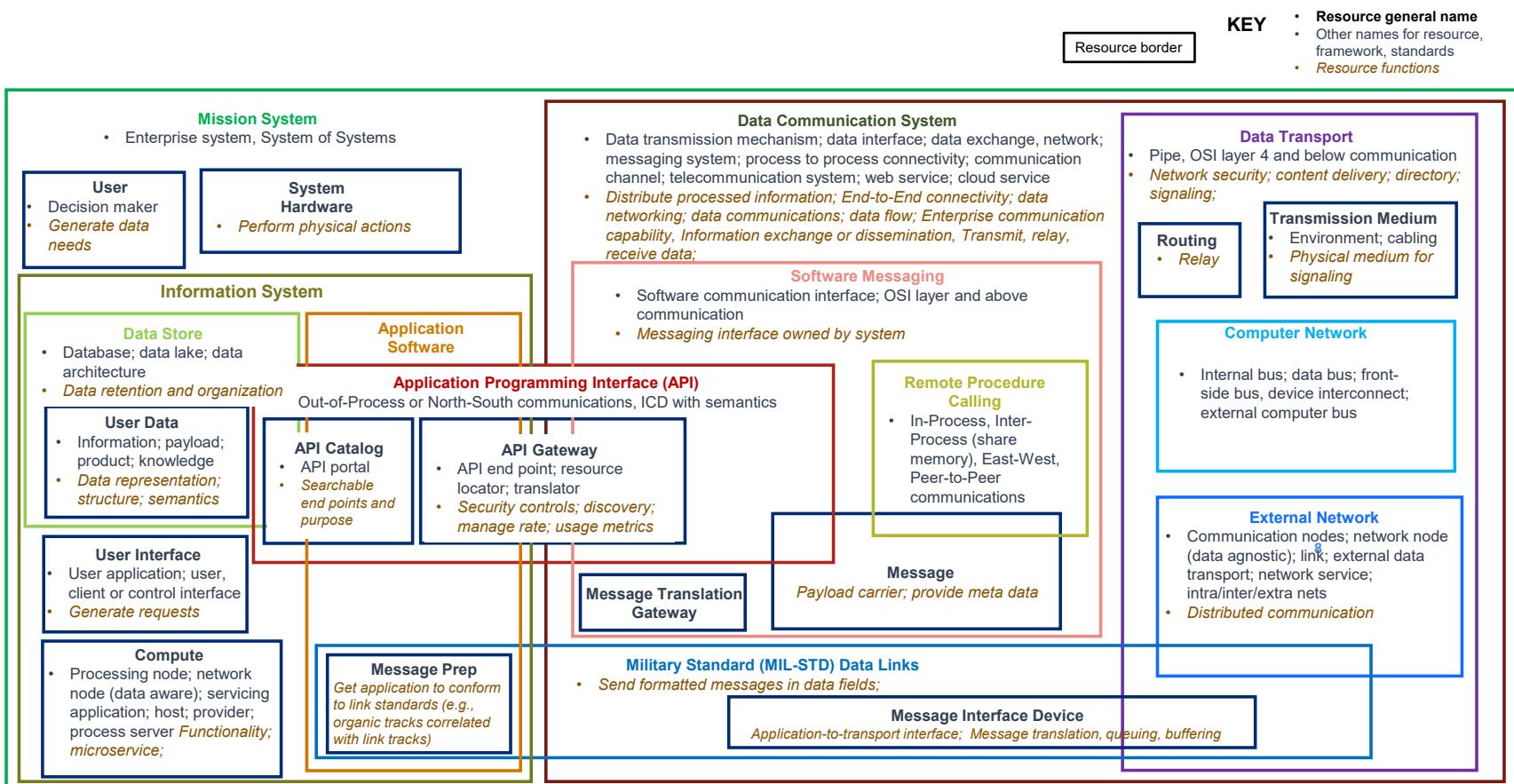


Figure 1-6. Component Relationship Diagram

### 1.4.3 Legacy Systems

The DoD has many legacy systems that are costly and challenging to maintain. Updating these systems with APIs can extend their life. Complete replacement of the systems at once may be costly and time consuming, so some programs have discovered inexpensive and creative techniques to integrate APIs with these legacy systems, ultimately improving capabilities, lowering risk of failure, and extending the life span into the future. (See Table 3-1 for a list of some ideal characteristics for an API.)

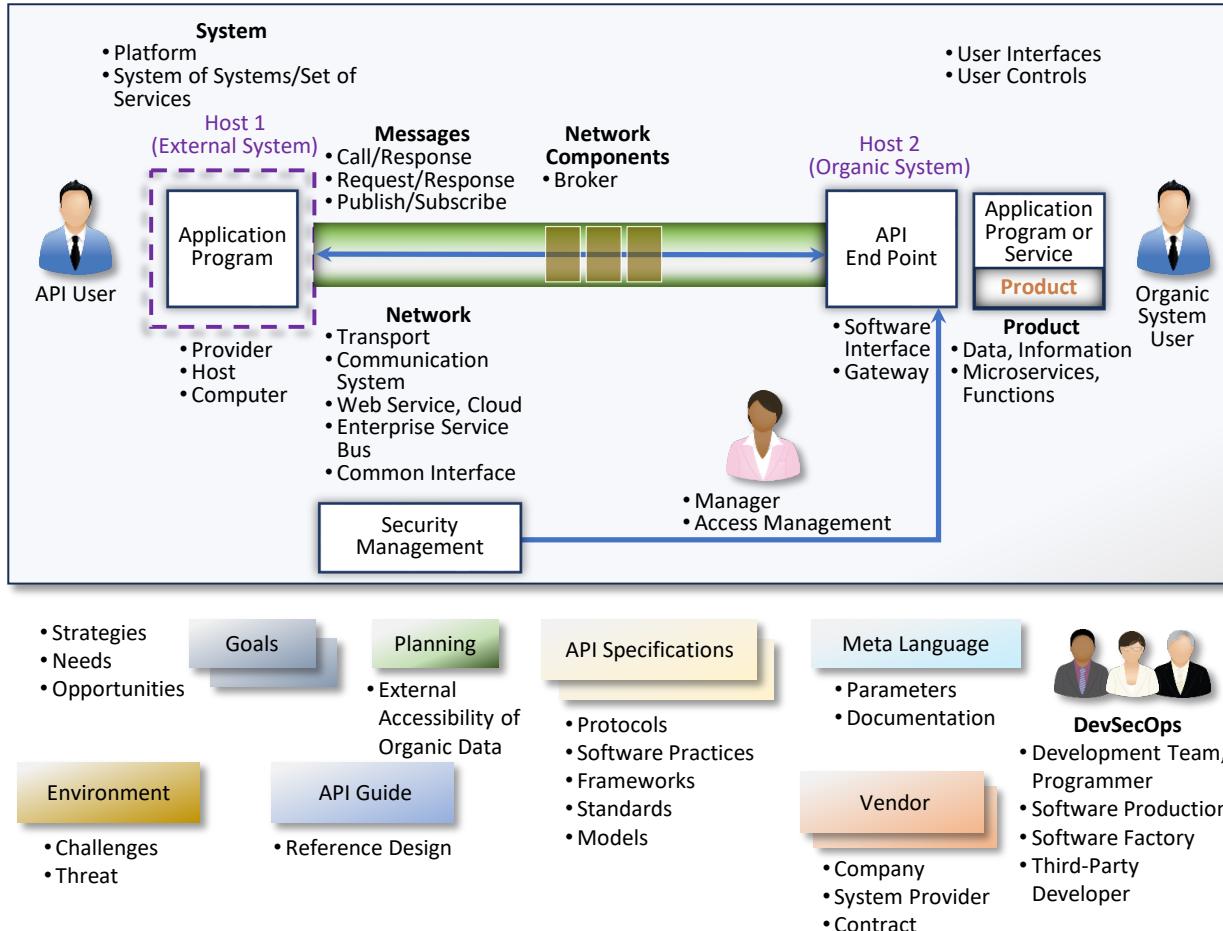
In the objective future state, software processes will start with a discovery phase to identify the existing APIs relevant to the application, their definition, and access pathway (API service discovery). Some new applications will require APIs to be extended or to be made more available, or new APIs will be needed and added. To reach a more desirable end state, an incremental phased change approach will most likely be needed to employ applications and APIs (including the deprecation of older API versions). This effort will take time and sustained enterprise governance.

Although cost and schedule are key considerations for sustaining legacy systems, any new APIs should follow the guidance in section 1.1.1 to leverage open standards-based APIs.

### 1.4.4 Other API Terms

Figure 1-7 shows other terms the community may be familiar with that relate to the terms in this guide. The figure depicts the user (API consumer) connecting to the product (API provider) over a network (transport) to consume data and information services.

## 1. Introduction



**Figure 1-7. API Ecosystem Terms**

In terms of the DoD, the user might be a soldier OCONUS in theater using their Android Tactical Assault Kit (ATAK) APIs via a handheld android or ruggedized laptop device to access various data services (e.g., Blue Force Tracking, Red Force Tracking, weather, or terrain topology) via the Joint Operational Edge (JOE) cloud to support critical situational awareness (SA) information needed for their mission. Future versions will elaborate on the March 6, 2025, Secretary of Defense memo “Directing Modern Software Acquisition to Maximize Lethality.”

APIs have become critical enablers in an era of software-defined warfare, where there is a strong emphasis on agility, security, rapid innovation, and seamless integration, all of which are inherently supported and facilitated by APIs.

The Software Acquisition pathway and contracting strategies envisioned in the above referenced memorandum incorporate APIs as described below:

- Software Acquisition Pathway: The Software Acquisition pathway emphasis on Agile methodologies naturally lends itself to the use of microservices, typically exposed and interconnected through APIs.
- Commercial Solutions Openings (CSOs): Modern commercial products and services, particularly Software as a Service (SaaS) offerings, heavily rely on APIs for access and integration.
- Other Transaction Authorities (OTAs): Focused on fostering innovation, rapid prototyping, and seamless integration, OTAs make the use of APIs highly likely, often essential, for project success.

The manager in the diagram (bottom left) could also be accessing management services via the JOE from a CONUS-based SIPRNet system at the Pentagon or other DoD facility to evaluate system metrics and operational status of the family of systems hosting the ATAK APIs (product) at the edge.

## 2 Data Access

For the purposes of this guide, “data” is defined as information stored, transmitted, or received in an electronic format.

Data can take on many purposes, including to inform and control. Data may be described as existing at the application software level and all the way down to hardware signal level. At the highest levels, it can be considered to have major flows in one direction, but at lower levels (OSI, system interface levels) it may flow bi-directionally to support the major flows.

One of the main purposes of API is to make data available to other systems and users. A user’s access may be considered one of three types:

- **Security Access:** authentication, authorization (e.g., identity, credential, and access management (ICAM))
- **System Access:** practical system, network, and software messaging design (including APIs)
- **Operations Access:** traffic management and network availability

The user’s overall data access is determined by the most restrictive of the three contributing types, that is,

$$\text{Data Access} = \min(\text{Security Access}, \text{System Access}, \text{Operations Access}).$$

Although literature sources on APIs cover security access and operations access extensively, generally they do not address system access. System access is important for large enterprises made up of many federated systems as in the case of the DoD.<sup>1</sup> Federated systems own their own behavior but work together and share input and output of certain data to achieve common goals. Because of the breadth of DoD federated systems, thorough planning should be conducted across various system stakeholders, regarding system access design.

### 2.1 System Access Design Considerations

System access (the designed-in access by external systems to organic systems) can enable the DoD enterprise but also comes with many design decisions and costs (Table 2-1).

---

<sup>1</sup> The Department’s infrastructure supporting data, analytics, and AI technologies will remain federated. The Department will continue to centralize some decisions and services while others remain decentralized to address unique mission needs (Data, Analytics, and Artificial Intelligence Adoption Strategy (DoD June 2023)).

**Table 2-1. System Access Design Considerations**

<b>Consideration</b>	<b>Description</b>
Performance	The more API objects provided to external community, the greater the demands on organic system's processors, memory, and communications.
Security	The more API objects we share with the external community, the more security measures, controls, and monitoring will be required.
Safety	In organic systems where safety is a key concern, increasing the number of API objects, controls, and externally accessible information can negatively affect performance. It may also introduce new user interactions and behaviors, potentially compromising the system's safety.
Configuration Management	As the organic system provides more API objects to the external community, the organic system must ensure configuration control for those objects. This places a heavier burden on the organic system to keep the external community in sync with the changes to include consideration of the update methods and the criticality to the external community.
Organization Operation	Who manages the configuration changes, who debugs external system issues
Data Abstraction	Generally, the more detailed a system's data is, the more the data is tied to the details of the system. For example, if a component such as an actuator is changed from hydraulic to electric then the system hydraulic system pressure information would no longer be valid. The data abstraction concept gives system interactions an advantage when systems remain more abstract with respect to each other. Figuring out what data will serve the external community in the long term best and with least cost impact is a significant consideration.
Decentralization	The more data provided to the external community by an organic system, the more dependent the external systems may become on the organic system's data. If communications between the system were to disappear there could be an adverse effect on the external systems capability. An alternative would be to provide the external system community with a model of the organic system in case communications are not available.
Testing	What is the testing approach for organic testers compare with testers higher up in the mission echelon? Will the organic testers have access to detailed information as it happens and the external community will only be able to test the mission critical data?
Enterprise Data Planning	Given that API's only talk between applications and don't necessarily solve the communication problem between federated systems, how extensive should the critical mission data planning be? Who will decide the breadth of users considered in the GECP? See Data Access Example section later and the discussion about data access scope versus API data access scope.
Cost	All the above considerations can add or subtract cost from the overall design and operation of the organic system. Design decisions on the organic system can also have cost implications for the external systems whether in design, operations, or updates.

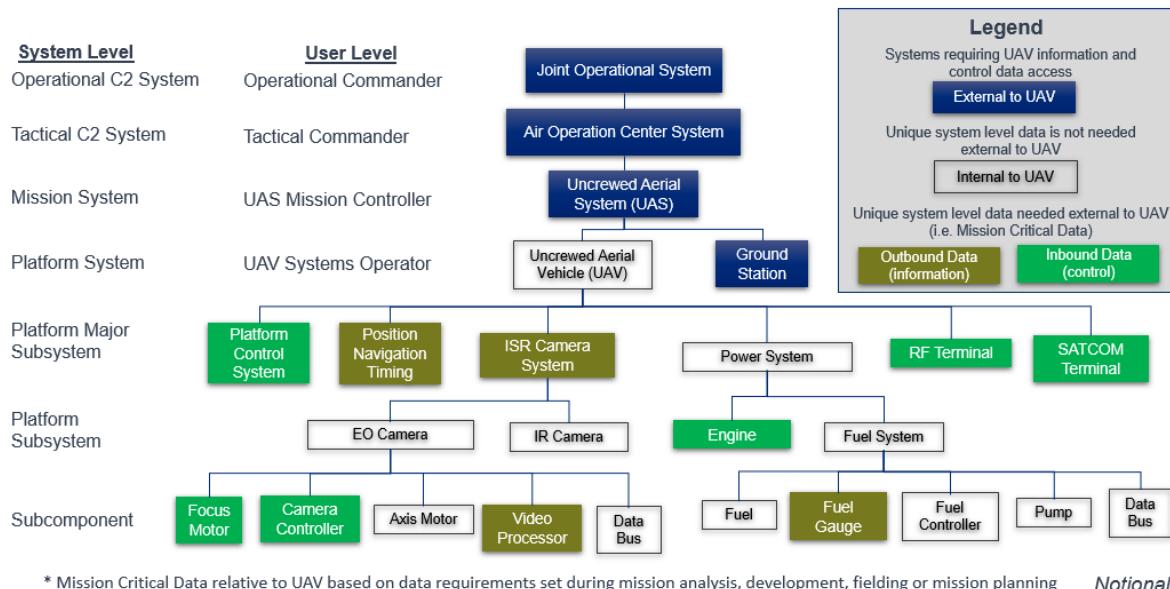
In general, this guidance provides the reader with best practices and concepts to consider when designing organic system access; however, this guidance does not currently discuss how to weigh organic system benefits and costs against side external system benefits and costs. For example, providing 100 API methods from an organic system may cost more to create, validate, and maintain than 10 API objects.

The direct benefits to the organic system or program supporting this API may not be calculable. The cost-benefit analysis for the external system or enterprise is likely to be different. The cost of several systems needing to maintain configuration control with 100 APIs from the organic system may be much higher than if 10 APIs' objects were designed-in; however, the benefits of “using” the organic data are probably much more obvious to the external system designer.<sup>2</sup>

This section has discussed the importance of considering both the organic and external system benefits and costs. The following sections present a data access example and discuss stakeholder roles.

## 2.2 Data Access Example

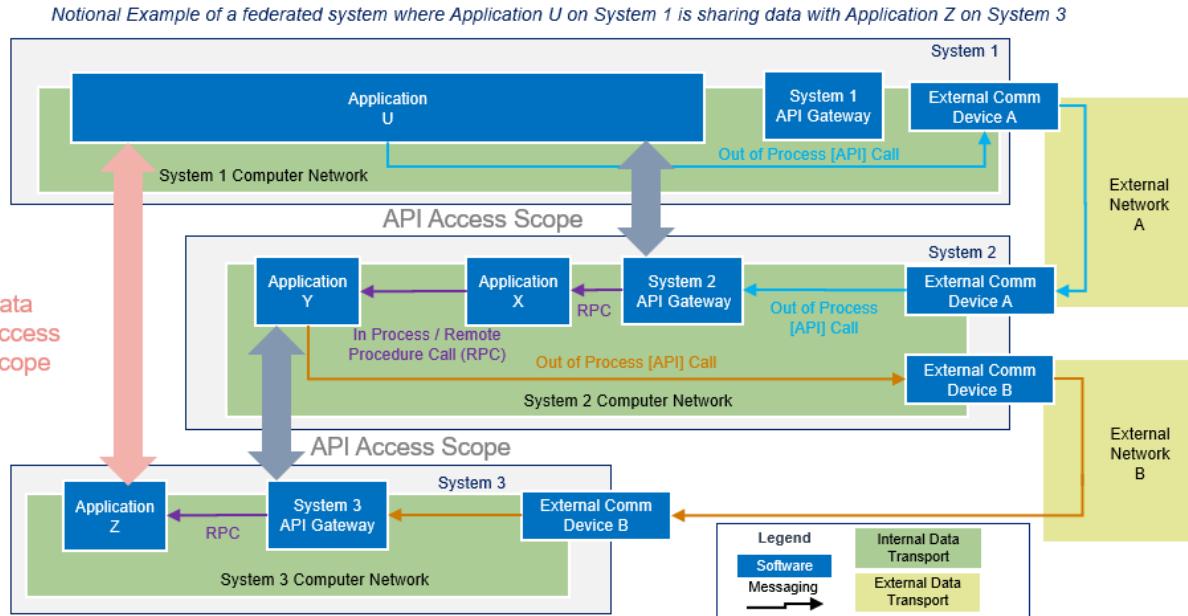
As shown in Figure 2-1, a higher echelon operational commander’s ability to see a supply voltage for a camera focus motor on an uncrewed aerial vehicle (UAV) operating in their force may not be good use of system design, communication, or operational resources, but the status of the ISR camera system and its video feed may be. Theoretically the camera voltage data could be designed into hierarchy of systems, including buffering the information between systems, for the information to be supplied all the way up the chain. Given the expansive nature of DoD systems, however, limiting the scope of System Access by way of mission-critical data analysis beforehand is considered good practice. Mission-critical data analysis is considered outside the scope of this document as will be explained next.



**Figure 2-1. Notional UAV System Hierarchy and Mission-Critical Data**

<sup>2</sup> A future version of this guidance will provide a General External Consumer Planning (GECP) appendix, or something similar. Table 2-1 lists system access considerations. See also information on real-time systems (RTS) in Section 9.

Although APIs support data access among systems, in the case of federated systems or systems separated by external networks, their scope is more limited than data access or system access. In order for data to traverse between systems, several APIs could be at work (Figure 2-2).



**Figure 2-2. Notional Example of Data Communications and API Scope**

In this notional example, the data access scope is wider than the API access scope. System 3's application request of System 1's data must be received and processed through an intermediary, system 2's computer, because the two external networks are different and the systems that support them are federated with respect to each other.

## 2.3 API Stakeholder Roles

The API ecosystem may be made up of many stakeholders representing various systems, components and functions depending on their system's mission (see Figure 1-7). To promote successful implementations of their own and other's mission systems, these API stakeholders need to be aware of all of these viewpoints when developing their enterprises, systems, APIs, and applications.

The system that these stakeholders represent or own can come in several different forms:

1. **External System:** the mission system making the request to the organic system's API. Note the request can be to access data or control the organic system (herein simply called requesting data).
2. **Organic System:** the mission system that has the desired data and that provides an API endpoint either via software on the system or via a separate, but associated gateway.

3. Communication System: The general communication infrastructure in-between the external “data requestor” system and organic “data providing” system.

In the API ecosystem context, the “API User” is considered **the primary API stakeholder** (Table 2-2). The reason for this “user-centric” view is that APIs exist fundamentally to support consumers. The User role, for example, could be an operations planner which needs to consume data from their internal planning system (organic to them) and just as importantly, consume data made available via APIs (external to them).

For reasons of performance, security, safety, configuration management, organizational operation, data abstraction, decentralization, and cost, DoD systems and their APIs the Providers should anticipate what mission critical data will likely be needed to support both current and likely future external system missions and use cases. Mission critical data will be discussed more in the Data Access section.

The “Tester” is also an API consumer but may not need to consume or control data in the same ways that are needed for the primary User stakeholders. The distinctions between Testers and User are important when it comes to API end-point system design. To promote well-considered system designs, testing data and mission critical data (data needed by the external eco system) should not be assumed to be one in the same. For example, detailed testing data can be provided outside of the API functionality and so as to not burden the API design. The distinctions between testing and mission will be elucidated in the future General External Consumer Planning (GECP) Appendix.

**Table 2-2. API Stakeholder Roles**

API Stakeholder Roles	Owned System	API Stakeholders and Objectives
API User (API Consumer type 1)	External System	Operations planner, analyst, decision maker, operator, or Non-Person-Entity (NPE) (e.g., AIs; automated decision aides) which are concerned with accessing external system end-point data in order to support operations. NPE recognizes the possibility for automating some non-human decision making. Multiple API personas <sup>3</sup> can apply to this role.
Tester (API Consumer type 2)	External System	Component testers, system testers, mission testers, operation configuration personnel that are concerned with accessing external system end-point data in order to test their mission systems. In the development life cycle, the Tester is considered to come in one-step prior and possibly after User operations. The Tester may also be involved in supporting the Gatekeeper role.
Organic System User	Organic System	Operations planner, analyst, decision maker, operator, or NPE which are concerned with performing the organic system missions

<sup>3</sup> A “persona” is a user role customized for a business case.

<b>API Stakeholder Roles</b>	<b>Owned System</b>	<b>API Stakeholders and Objectives</b>
Provider (Developer type 1)	Organic System	Enterprise architects, system designer, API developer and built-in test developers that are focused on providing data from the organic system to the external API consumer community.
Applier (Developer type 2)	External System	Enterprise architect, mission engineers, application developer, built-in test developers focused on setting up the external mission system to support the Consumer needs.
Gatekeeper	Organic System, Communication System	Security designers and operational security teams whose purpose is to assure only valid external users received necessary organic system data.

There are two types of users, the API User and the Organic System User. API users focus on accessing data from external systems. Organic system users focus on the needs of the organic system. In the API context, the “organic system” means the system that has the data of interest that needs to be provided to the external community. Traditional system development focuses on the Organic System User needs (i.e., the organic system mission requirements) and less on external community data needs; however, this guide advocates for the organic system development efforts also to have a Provider on the team who can consider the broader community needs as well.

In the context of APIs and RTSs, the Provider also should consider the Organic System User needs. As mentioned later in the RTS chapter, the API end-point functionality should not interfere with the tight performance requirements of the organic RTS.

## 3 API Project Governance

This section discusses how DoD API project teams should create a governance process to guide development, implementation, and updates to their unique API ecosystem or framework. A DoD API project is any new or existing DoD-funded project that is creating APIs or updating their API. These projects could be system development programs for which APIs are but one aspect of the system, or the projects could focus on interoperability standards or API frameworks. All these use cases will be referred to in this guidance as “API projects.”

An API project governance model is the application of rules, policies, and standards to the project API ecosystem. Governance includes how the API project organization should share, encourage adaptation, administer, adjudicate, and update the API in support of both internal and external stakeholders. The API project governance process will evolve as technologies, standards, best practices, and organizations evolve, but the process focuses on open and fair governance, not specific technical recommendations. Appendix A provides considerations for developing an API Governance process. These considerations do not mean one size fits all. Developers must be free to implement APIs as needed for mission requirements. A governance process and enforcement can ensure the API follows best practice and demonstrates high-quality attributes, resulting in the benefits noted in Table 3-1.

**Table 3-1. High-Quality API Attributes and Benefits**

Attribute	Benefit
<b>Reusable</b>	Can reuse existing components. Developers need to build components only once and will not end up duplicating code. They can spend more time on tasks that benefit the business, like building new services.
<b>Reliable</b>	The APIs reliably are available and function as documented.
<b>Interoperable</b>	Can be used with approval in all use scenarios by applications that would benefit operations.
<b>Discoverable</b>	Developers can easily find existing API artifacts and reuse them in future designs.
<b>Scalable</b>	Can have small or large number of elements and the APIs can serve many diverse users.
<b>Consistent</b>	API remains consistent even when implemented by different developers and across the entire DoD solution space.
<b>Easy to Use</b>	The API is easy to understand and implement in many diverse use cases.
<b>Clear</b>	The API vision, design, and documentation are clear. Helps keep everyone involved in the API program. Misunderstandings about API goals or designs can cause API programs to fail.
<b>Secure</b>	Security is built into the foundation of the API development and deployments. API interfaces includes classification metadata support (NSA Access rights and handling, information security metadata, NSA Guidance for implementation of REST – in DISR).
<b>Compliant</b>	Well-managed and visible exception pathways (Sindall 2020)
<b>Complete</b>	Life cycle use of API is well thought through and provisioned (Sindall 2023).

Attribute	Benefit
<b>Loose/Low Coupling</b>	Interoperability is achieved without unnecessary knowledge of the operation of the service application/system.
<b>High Cohesion</b>	An API only requires and returns data/classes required to perform the requested action, reducing complexity and context.
<b>Robustness</b>	The API can withstand or overcome adversity

The API project team should create a GECP document to ensure the system supports the external stakeholders and considers development and maintenance in a balanced way over the course of the system life cycle (see also Section 2, Data Access).

In scenarios in which a DoD Component's API project will have external consumers, typically the API project would require a memorandum of understanding (MOU) to onboard new consumers. An MOU outlines the terms and conditions under which one party (the API provider) will make the API available to the other party (the API consumer) and defines the responsibilities and obligations of both parties.

MOUs can be used to establish a variety of agreements related to APIs, such as:

- Defining the scope of the API, including the data and functionality that will be made available, and any limitations or restrictions on its use.
- Describing the use and security of the data as well as privacy measures that will be implemented to protect the data transmitted through the API.
- Specifying the terms of service for the API, including any fees or charges for its use, and the consequences for violating the terms of the agreement.
- Establishing the process for resolving disputes or addressing issues that may arise during the use of the API.

Having a clear and well-defined MOU in place can help to ensure that the use of an API is smooth and successful and can help to prevent misunderstandings or disputes between the API provider and consumer. Helpful MOU resources include the Department of Homeland Security MOU writing guide (DHS 2015) and the Test Resource Management Center MOU Template (TRMC 2019).

Governance should also define and describe details on how the API versioning strategy will work including a communication plan to frequently update consumers on coming version updates, new features, and expected changes to the APIs. The API versioning strategy should include details on how the API project will leverage metrics to track users across all API

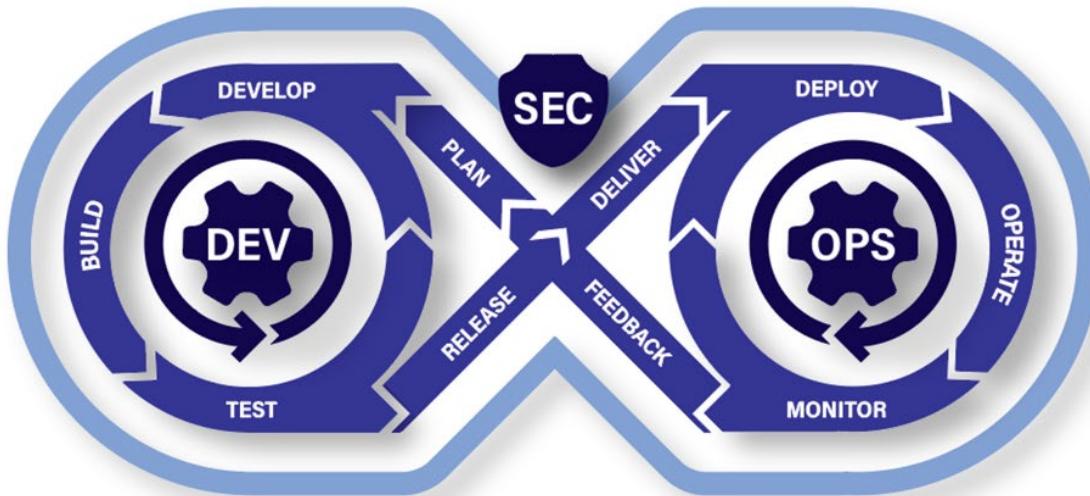
### 3. API Project Governance

versions along with how and under what circumstances versions will be deprecated once consumers have migrated onto newer versions.

Maintaining simultaneous multiple API versions typically requires additional infrastructure and labor costs, so API projects may be limited regarding the number of simultaneous API versions they can maintain and under what circumstances they are deprecated. They should communicate this information up front to their API consumers as part of their API governance.

## 4 Cybersecurity

The Department is evolving toward DevSecOps (DevSecOps Reference Architecture 2023), which combines DevOps and integrates cybersecurity principles with an added emphasis on security at each of the DevOps life cycle phases (Figure 4-1).



Source: (DevSecOps Reference Architecture 2023)

**Figure 4-1. DevSecOps Infinity Diagram**

Zero Trust (ZT) is a security framework and enabling capability supporting DevSecOps. ZT challenges the traditional approach of trusting entities within a network by default. Instead, it assumes that no entity, whether a user, device, or application, should be trusted automatically. ZT adopts a “never trust, always verify” approach, in which every access request is thoroughly authenticated, authorized, and continuously monitored, regardless of the entity’s location or network environment (DevSecOps Reference Architecture 2023).

All APIs need to satisfy and support the requirements of the intended authority to operate (ATO) and align with the intended ZT security posture of the DevSecOps automated pipeline, its automated tests, analysis activities, and security posture.

ZT principles encourage the following:

- Incorporating strong identity and access management practices to ensure that only authorized entities have access to resources.
- Implementing granular micro-segmentation to enforce strict access controls and prevent lateral movement within the network.

- Leveraging automation and continuous security practices to continuously monitor and enforce security policies.
- Integrating data-centric security measures, such as data labeling and encryption, to protect sensitive information.

By integrating security practices into the DevOps process, DevSecOps helps organizations build and maintain a strong security posture, which aligns with the principles of ZT (NIST SP 800-207).

Additional insights regarding security best practices are also possible by comparing the components found in the Component Relationship Diagram (Figure 1-6) against the security practices and mechanisms in Section 4.3. This component to security concept mapping will be provided in a future release.

## 4.1 Importance of Securing APIs in DoD

The increased interconnectivity amplifies the attack surface and introduces new cybersecurity challenges that can potentially compromise mission-critical systems and operations. APIs are susceptible to various attacks, including injection attacks, authentication and authorization issues, availability, and data breaches. Consequently, securing APIs has become an urgent priority for the DoD and other organizations that depend on them (DoD CIO CMMC); (OWASP REST Security Cheat Sheet 2024). Following are best practices for defending against API threats in the context of emerging technology trends and the evolving battlespace.

## 4.2 API Cybersecurity Challenges

The Department faces numerous API cybersecurity challenges as it seeks to maintain and modernize its systems. APIs play a crucial role in facilitating communications across a wide range of vital systems, which rely on the secure and timely exchange of information between various Components in support of mission objectives. Cybersecurity challenges include:

- Ensuring the confidentiality, integrity, and availability of API communications and maintaining effectiveness of resilient systems.
- Securing and enhancing data exchange and interoperability capabilities for legacy systems using open standards-based APIs.
- Securing the infrastructure that supports APIs (e.g., gateways, proxies, load balancers, content delivery networks (CDN)).
- Ensuring secure transfer of information between weapon system platforms supported by APIs.

Addressing these challenges requires a comprehensive approach involving the development, deployment, requirements (e.g., cyber performance requirements), and ongoing management of APIs. This approach includes implementing secure coding practices, conducting regular security and threat-based testing, and continuously monitoring API activity for potential threats. By prioritizing API security, the DoD can ensure the continued effectiveness of its systems while safeguarding sensitive information from adversaries. See section 4.3 for API security best practices.

APIs are a critical component of modern software systems, enabling communication and data exchange between different applications and services; however, APIs also can be a source of vulnerabilities and threats if they are not properly secured. In the context of the DoD, APIs, along with many interfaces, can be vulnerable to attacks given the sensitive nature of the data and systems they are used to access.

NIST provides an API security Special Publication (SP) 800-204, “Security Strategies for Microservices-based Application Systems,” and additional in-depth coverage of the topic in supplemental SPs specific for API security: SP 800-204A, 800-204B, and 800-204C. Although the document is titled for microservices, the “microservices are packaged as APIs” (SP 800-204, section 4.1) to support complex data tasks. The SP specifically addresses API security threats and associated mitigations. Specific threats to APIs include most attacks that normal applications experience including “injection, encoding and serialization attacks, cross-site scripting (XSS), Cross-Site Request Forgery (CSRF), and HTTP verb tampering” (NIST SP 800-204 2019). See also Appendix C: API Security Challenges.

### **4.3 Cybersecurity Best Practices**

To mitigate API vulnerabilities and threats, it is important to implement strong API security measures, such as authentication and authorization mechanisms, encryption of data in transit and at rest, input validation to prevent injection attacks, and continuous monitoring of API traffic and logs. In addition, programs should stay up to date with the latest security best practices and regularly review and update API security measures to ensure they remain effective. See also Appendix B: Common API Vulnerabilities and Threats.

API cybersecurity should be designed using ZT principles (NIST SP 800-204), secure coding practices, and defensive measures to protect against attack techniques associated with API environments. Following the best practices in this section can help programs address the challenges DoD faces in securing its systems and data, resulting in more resilient and secure critical systems and operational data against potential threats.

Programs should consider the following seven security-focused best practices when planning to implement an API project.

### **4.3.1 Implement Robust Authentication and Authorization Mechanisms**

Authentication and authorization (A&A) in an API and data service environment is expected to require internal A&A centralized architecture because of the sheer number of APIs and services interacting. Associated dependencies are realized from using loosely coupled and smaller application bases. The following security controls are recommended:

1. Robust authorization services to ensure availability and timeliness for access decisions.
2. Sensitive data will not use traditional embedded API keys in requests.
3. Digitally signed authentication tokens.
4. If an API key is used, restrictions on where the API and applications may be used.
5. A&A token expiration times as short as possible.
6. A&A token keys produced dynamically in the API and service process using variables represented from the exchange environment.
7. Integration with ZT, for single use or very short time to live (TTL) tokens.
8. Software Bill of Materials (SBOM) to identify and mitigate cybersecurity threats.
9. Software Assurance Maturity Model (SAMM) – an open framework that improves API security.

### **4.3.2 Ensure Input Validation and Output Encoding**

Input validation and output encoding are crucial for secure data exchange. They provide a foundation for mitigating API cybersecurity threats, especially within the ZT framework and the specific security challenges the Department faces. Input validation involves verifying and sanitizing user input to prevent malicious code injection and to ensure that only valid and expected data is processed. Output encoding, on the other hand, converts untrusted data into a safe format to prevent XSS attacks and the execution of malicious code in users' browsers. By implementing robust input validation and output encoding practices, the DoD can enhance the security of its APIs, protect sensitive data, and reduce the risk of unauthorized access or manipulation. These measures align with the principles of ZT.

### 4.3.3 Encrypt and Protect Data in Classified Environment

Encryption and data protection play a critical role in mitigating API cybersecurity threats, particularly in a highly classified environment within the DoD and in alignment with the ZT framework. In a highly classified environment, sensitive information must be securely transmitted and stored to prevent unauthorized access or disclosure. Encryption ensures that data is transformed into an unreadable format, making it inaccessible to unauthorized individuals. By implementing strong encryption algorithms and secure key management practices, the DoD can protect the confidentiality and integrity of its data, even in the event of a breach. This protection aligns with the principles of ZT.

### 4.3.4 Monitor and Log for Early Threat Detection and Response

Monitoring and logging are essential components for early threat detection and response in the context of API cybersecurity threats. By implementing robust monitoring and logging practices, the DoD can continuously monitor its APIs and systems for suspicious activities, unauthorized access attempts, and potential security breaches. This monitoring enables the timely detection of threats and facilitates a rapid response to mitigate the impact of any potential breaches. In addition, comprehensive logging allows for detailed analysis and investigation of security incidents, aiding in the identification of vulnerabilities and the implementation of necessary remediation measures. These monitoring and logging practices align with the principles of ZT.

### 4.3.5 Securing the API Gateway

The API gateway is an architectural pattern that enables monitoring. According to the F5 technology company Glossary, an API gateway is “a component of the app-delivery infrastructure that sits between clients and services and provides centralized handling of API communication between them” (F5 Glossary n.d.). This pattern allows for an intermediary between clients and the API implementations to monitor and redirect traffic as needed and to provide discoverability, authentication, and authorization capabilities. In the case of a denial-of-service attack (or even an unintentional system overload), an API gateway can be configured to drop traffic from some clients.

Since the API gateway is the primary component to effective API implementation, management, and security, it should be equipped with infrastructure services appropriate to mitigate the growing list of threats. At the least, these enterprise services should include service discovery, authentication and access control, load balancing, caching, application aware health checking, attack detection, security logging and monitoring, and circuit breakers (NIST SP 800-204).

Specific security strategies for API gateway include (NIST SP 800-204):

1. Integrate an identity management application to provision credentials before API activation.
2. Ensure API gateway has a connector to generate an access token for a client request.
3. Securely (HTTPS, SSH, OpenSSH, Type1 NSA) channel all traffic to a monitoring and analytics solution for attack detection and performance degradation.
4. Ensure distributed API gateway deployments and microgateways (gateway customized and closer to API and service) include a token exchange service between gateways. Edge gateways should have tokens with broader permissions, and internal microgateways should have more narrowly defined permissions, enabling a least privilege paradigm.

#### **4.3.5.1 Securing API Service Discovery**

Securing API service discovery is crucial for maintaining the availability and proper location of services as needed. This consideration is particularly vital in the virtual and cloud environments where they might need to be replicated and relocated for various reasons, including security considerations. It is essential that service discovery not only facilitate the connection between the clients and services but also ensure the authenticity and integrity of the services provided.

Loose coupling should be used, and APIs should not include self-registration and deregistration capability. If an API and associated service crash or they are unable to handle requests, the inability to perform deregistration affects the integrity of the data and information-sharing process. Reliance on API local cache data should be considered based on the needs of the operating environment, other data, and system-specific factors.

#### **4.3.5.2 Implementing Circuit Breakers in API and Services to Prevent Cascading Failures**

Circuit breakers are a common technical implementation for minimizing cascading failures. Circuit breakers prevent delivery to an API and associated service that is failing. This technique helps prevent security attacks such as denial of service and brute force attempts (i.e., login attempts, erroneous inputs, and code injects). Three options for deploying circuit breakers are client-side, server-side, and proxy implementation. Proxy implementation is recommended because it avoids placing trust on clients and APIs.

#### **4.3.5.3 Data Integrity Assurance**

Data integrity assurance is the assurance that digital information is uncorrupted and can be accessed or modified only by those authorized to do so (e.g., system database administrator).

There are four types of data integrity to consider: entity, referential, domain, and user-defined. Data integrity assurance is another important consideration for APIs and their underlying data.

#### **4.3.5.4 Other API Security and Release Recommendations**

With new APIs, versions, and associated services (Corbo 2023):

1. Canary release tactic should be employed, thus limiting the number of requests and use of a new API. This is to protect failure and erratic behaviors of both clients and API as use cases may not meet the expectations despite extensive testing.
2. Usage monitoring of existing and new version should steadily increase traffic to new version.

#### **4.3.5.5 Session Persistence**

1. Session information needs to be stored securely.
2. Internal API and service authorization tokens must not be provided to the user.
3. User session tokens must not be provided beyond the gateway used for policy decisions.

#### **4.3.6 Ensure API Security Testing and Compliance in the DoD**

By conducting API security and threat-based testing, the DoD can proactively identify and address potential security risks, ensure compliance with security standards and regulations, and enhance the overall security posture of its systems and data. These testing practices are crucial for maintaining the integrity, confidentiality, and availability of APIs within the DoD environment.

Static application security testing (SAST) and dynamic application security testing (DAST) should be conducted as part of the continuous integration/continuous deployment (CI/CD) pipelines, and other manual testing (e.g., penetration) should be conducted before initial deployment and at regular intervals (e.g., monthly, quarterly, yearly) based on program security needs.

Interactive application security testing (IAST) combines the benefits of SAST and DAST and analyzes the application's runtime behavior to provide a context-aware analysis of potential security vulnerabilities.

##### **4.3.6.1 Types of API Security Testing Relevant to the DoD**

Several types of testing are relevant for DoD APIs, including (1) functional testing, which ensures APIs perform as intended and handle inputs correctly; (2) penetration testing, which simulates real-world attacks to identify vulnerabilities and weaknesses; (3) fuzz testing, which

involves sending unexpected or malformed data to an API to uncover potential security flaws; and (4) security code reviews, which involve analyzing API source code to identify security vulnerabilities.

#### **4.3.6.2 Common Tools and Techniques for API Security Testing in a Defense Context**

Tools and techniques to ensure compliance in API security testing include OWASP ZAP, Burp Suite, JMeter, Postman, and Nessus, which help identify vulnerabilities and weaknesses in APIs. These tools and techniques allow DoD to proactively identify and address potential security risks, ensure compliance with security standards and regulations, and enhance the overall security posture of systems and data.

#### **4.3.6.3 API Security Testing and Compliance within the DoD**

DoD can follow several best practices to improve API security testing and compliance. Functional testing, penetration testing, fuzz testing, and security code are necessary to identify vulnerabilities and weaknesses in APIs. Regularly updating and patching APIs, as well as monitoring and logging API activities, can help a program detect and respond to potential security incidents. Adhering to industry standards and regulations, such as those outlined by the DoD and NIST, is a best practice for maintaining compliance.

#### **4.3.7 Implement an API Governance Process**

The security of the API created for a system relies on the cohesiveness of the API governance. Projects should build in and enforce governance throughout the development pipeline (top-down, not bottom-up) to reduce risk and achieve overall security objectives (see also Section 3, API Project Governance).

## 5 Design and Implementation Principles

To ensure the success of an API project, an “API-first” strategy emphasizes the key principles of modularity, scalability, and reusability. By adopting this strategy, a program places priority on the design and development of the API before implementing the underlying system. Doing so can make testing easier and separates the development of the API from the rest of the system it supports. This separation makes the system more modular. Creating a modular architecture enables scaling, reuse across various applications and platforms, and seamless integration, flexibility, and enhancements. This approach also encourages a more efficient and collaborative development process, allowing teams to work in parallel, with front-end and back-end developers focusing on the team’s areas of expertise. Ultimately, an API-first strategy sets the stage for a robust and adaptable system that can meet the evolving organizational and stakeholder needs. The following paragraphs discuss seven design and implementation principles that support an API-first strategy.

### 5.1 Common Data Model

It is essential to establish a Common Data Model (CDM) early in the design and implementation phase that can be used across all API endpoints. This CDM serves as a standardized schema or structure for organizing and sharing data, ensuring consistency and interoperability between different API components and applications. Defining a common data model creates an understanding of how data should be shaped and shared, enabling rapid unification of data and facilitating seamless integration between various systems and services. This promotes data consistency, reduces complexity, and enhances data interoperability, allowing different applications to communicate effectively and exchange information seamlessly.

Implementing a CDM not only streamlines the development process but also improves data quality, accuracy, and reliability, as it eliminates the need for data mapping and transformation between different systems. In addition, the CDM enables easier data integration with industry-wide standards and frameworks, facilitating collaboration and data exchange with external partners and stakeholders. Early and comprehensive adoption of a common data model establishes a solid foundation for the API ecosystem, ensuring consistency and interoperability throughout the entire system.

### 5.2 Open Standards and Protocols

In the design and implementation phase, the API project should leverage open standards and protocols to ensure compatibility and interoperability with other systems and applications. Adoption of widely accepted standards such as Open API, Extensible Markup Language (XML),

and JavaScript Object Notation (JSON) helps enable seamless communication and data exchange between different components of the API ecosystem.

REST provides a lightweight and scalable architectural style for designing networked applications, while JSON offers a simple, compact, and human-readable format for representing data. These open standards promote flexibility, allowing developers to easily integrate with the API, reducing the learning curve associated with understanding proprietary protocols.

In addition, incorporating Open Authorization (OAuth) as a security protocol enhances the security and trustworthiness of the API. OAuth enables secure and delegated access to protected resources by providing a standardized framework for authentication and authorization. OAuth ensures that only authorized users or applications can access sensitive data or perform specific actions within the API. This protection not only enhances the security of the API but also simplifies the integration process for developers, as they can leverage existing OAuth libraries and frameworks to handle authentication and authorization (OAuth 2.0 2023).

Embracing open standards and protocols will lead to an API ecosystem that is compatible with a wide range of systems and applications. This approach promotes interoperability, allowing the API to seamlessly integrate with existing infrastructure and enabling developers to leverage existing knowledge and tools. The use of open standards reduces vendor lock-in and fosters a collaborative development environment, as developers can easily understand and work with the API. Overall, adopting open standards and protocols is a key aspect of designing and implementing an API that is accessible, interoperable, and developer friendly.

Given the DoD's global operation environment, across all time zones, one important standard to consider is the use of the ISO-8601 (2019) date and time format, which supports a variety of flexible use cases and formatting options. ISO-8601 allows local date and time to be expressed using a time zone designator, with extreme precision. ISO-8601 is critical in supporting interoperability among theater, combatant commands, and continental U.S. C2 environments.

### 5.3 Design for Security Compliance

For the DoD, designing the API with security compliance in mind is critical. Incorporating security measures from the outset protects the API from unauthorized access and potential data breaches.

One of the main security protocols to consider is Secure Sockets Layer/Transport Layer Security (SSL/TLS), which provides encryption and secure communication between clients and servers. Implementing SSL/TLS provides data integrity, non-repudiation, and ensures that data transmitted between the API and its consumers is encrypted, preventing eavesdropping and unauthorized interception of sensitive information.

In addition, using industry-standard authentication and authorization protocols such as OAuth and JWT (JSON Web Tokens) enhances the security of the API. OAuth enables secure and delegated access to protected resources, ensuring that only authorized users or applications can access specific endpoints or perform certain actions. JWT provides a secure method for transmitting claims or assertions between parties, allowing for secure authentication and authorization.

API projects should design the API with security compliance in mind to establish a robust security framework that protects sensitive data and prevents unauthorized access. This framework not only safeguards the integrity and confidentiality of the API but also helps to satisfy industry and regulatory compliance requirements. It is important to stay up to date with the latest security best practices. Projects should regularly review and update security measures and address emerging threats and vulnerabilities.

Overall, incorporating industry-standard security protocols such as SSL/TLS, OAuth, and JWT into the API design ensures that security is a fundamental aspect of the API ecosystem, protecting data and the privacy of users (e.g., soldier personally identifiable information (PII), medical systems protected health information (PHI)).

### 5.4 Developmental Testing and Validation Processes

Establishing a robust testing and validation process to ensure the quality and reliability of the APIs is also essential. By implementing comprehensive testing methodologies including unit, integration, and section tests, issues or vulnerabilities can be identified and addressed before they reach the production environment. Three common testing concepts in a robust process are unit testing, integration testing, and end-to-end testing.

Unit testing involves testing individual components or units of code to ensure they function correctly in isolation. Unit tests help identify and fix bugs, validate the behavior of individual functions or modules, and provide a solid foundation for building more complex features.

Integration testing involves testing the interaction between different components or modules within the API to ensure they work together seamlessly. Integration tests help identify any issues that may arise when different parts of the API are combined, ensuring the overall functionality and reliability of the system. Integration testing between components and the APIs that support them is also necessary to validate that the APIs themselves are functioning correctly.

End-to-end testing validates the entire flow of the API, simulating real-world scenarios and user interactions. This type of testing ensures that all components, integrations, and dependencies work together as expected, providing a holistic view of the API's performance and functionality.

By implementing a comprehensive testing and validation process, any issues or bugs can be identified and addressed early in the development cycle, reducing the risk of encountering problems in the production environment. This helps ensure that the API functions as intended, delivers the expected results, and provides a positive user experience.

API projects should automate testing processes as much as possible, using tools and frameworks that facilitate test automation. Automation allows for faster and more efficient testing, enabling CI/CD practices. In addition, incorporating security testing, such as penetration testing and vulnerability scanning, helps identify and mitigate potential security risks. It is a best practice to download available security tools from approved artifact registries and trusted sources (e.g., NIWC PAC SCORE (2024), Platform One Iron Bank (2024)).

A robust testing and validation process is crucial for ensuring the quality, reliability, and security of an API. By conducting thorough unit, integration, and end-to-end testing, any issues can be identified and addressed early on, leading to a more stable and reliable API.

Data used to test APIs as well as results should be submitted to a centralized database such as cloud hybrid edge-to-enterprise evaluation and test analysis suite (CHEETAS) for testing and evaluating data from test range events or other.

### 5.5 Collaboration and Communication

Collaboration and communication among developers, architects, and other stakeholders are essential elements in the successful development and implementation of APIs. A collaborative culture with fluid communication paths and consistent internal feedback loops helps ensure that the API will meet the needs and expectations of all parties involved. In addition, this communication helps establish a culture of psychological safety (McKinsey & Company 2023) and professionalism that focuses on respect for all team members.

API development also requires close collaboration between consumers and producers. When changes occur, they should be closely coordinated with the consumers. API metrics should be thoroughly tracked to alert all interaction activities so the provider can understand who is using their APIs, how and when the APIs are being accessed, and the result codes. APIs that do not track their consumer metrics open themselves up to having malicious clients that can surveil or exploit an application. Consumers need to stay up to date on the latest changes to how the API works, while producers need feedback from consumers to ensure they are building the right thing. A communication plan facilitates frequent and iterative information flow to the consumers, provides consumer feedback, provides recommendations, and allows reporting on issues and bugs. In addition, holding weekly scrums or monthly “ask me anything” engagements further enhance the relationship. Lastly, fostering a culture of collaboration helps create a robust

feedback cycle that allows producers to better understand consumer needs and supports continuous improvement and iteration.

To facilitate collaboration and communication, teams can use tools such as API documentation, developer portals, ChatOps, collaboration software, and forums. API documentation serves as a comprehensive resource that provides information on how to use the API, its endpoints, parameters, and response formats. Developer portals act as a centralized hub where developers can access documentation, explore API features, and engage with the API community. ChatOps (2024) typically provides a highly interconnected environment across topic threads to obtain immediate answers from appropriate team members. Collaboration software typically provides an ecosystem of online tasking, communications, and meeting capabilities. Forums provide a platform for developers to ask questions, share insights, and provide feedback on the API. Projects should consider experimentation and pilots to establish which tools are most effective.

API projects should use these and other effective tools to create an environment that encourages open communication and collaboration. This open environment enables API developers, architects, and other stakeholders to share ideas, address concerns, and work together toward building a robust and user-friendly API.

Overall, encouraging collaboration and communication is crucial for the success of an API project. Fostering a collaborative environment and using tools facilitates effective communication, enhances feedback for all parties involved, and helps the project deliver a seamless experience for API consumers.

## 5.6 API Parameters for Pagination, Sorting, and Filtering

Pagination is essential when dealing with large result sets. By using parameters such as “limit” and “offset,” clients can control the number of items returned per page and navigate through the result set. This approach prevents overwhelming the client with a massive amount of data and improves performance by reducing the payload size. Pagination also prevents overwhelming the provider’s databases with massive requests when the underlying data contains many thousands or millions of records. Another benefit of pagination is its support in reducing the ability of malicious actors to consume compute resources (e.g., DDOS) or information (e.g., exfiltration) through large API requests.

Sorting is another important capability that can be achieved through API parameters. Clients can specify the sorting criteria using parameters like “sort” and “order.” This allows them to retrieve data in a specific order, such as ascending or descending based on a particular field. Sorting empowers clients to organize and analyze the data according to their requirements.

Use of filtering is critical for harnessing the power of APIs. Filtering allows retrieval of data based on specific criteria (e.g., datetime, domain, category, location) and reduces or eliminates the amount of irrelevant information retrieved. For larger data sets, filtering can significantly improve performance and efficiency of request responses, post-result processing, network communications, and system utilization.

By supporting pagination, sorting, and filtering capabilities through parameters, APIs can deliver a more efficient and personalized experience to clients. It is important for API developers to design and document these parameters effectively, ensuring that clients understand how to use them correctly and take full advantage of the API's capabilities.

Overall, API parameters are a powerful tool for enhancing the usability and efficiency of APIs. They enable clients to control the data they receive, navigate through large result sets, sort data according to their needs, and filter out irrelevant information. By incorporating these capabilities into API design, developers can provide a more flexible and user-friendly experience for API consumers.

## 5.7 API Metrics

Several metrics should be considered to measure API performance and effectiveness. Metrics help assess the performance, availability, and usage of APIs. Some API-specific metrics include:

1. Response Time: Measures the time it takes for an API to respond to a request. This metric helps understand the performance of the API and identify any bottlenecks or latency issues.
2. Error Rate: Tracks the percentage of API requests that result in errors. This metric helps identify any issues with the API's functionality or stability.
3. Availability: Monitors the uptime and availability of the API. This metric helps ensure that the API is accessible to users and identifies any downtime or service interruptions.
4. Usage, Traffic, and Growth: Measures the number of requests and the volume of data being processed by the API. These metrics help system owners understand the usage patterns and scalability requirements of the API.
5. Latency: Measures the time it takes for data to travel from the client to the API server and back. This metric helps identify any network or infrastructure-related issues that may impact API performance.
6. Rate Limiting: Monitors the number of requests per second or minute to enforce rate limits and prevent abuse or overload of the API.

7. Authentication and Authorization: Tracks the success rate of authentication and authorization processes for the API. This metric helps ensure the security and integrity of the API.
8. SLA Compliance: Measures the compliance of the API with the defined Service-Level Agreements (SLAs). This metric helps assess whether the API is meeting the performance and availability targets set for it.
9. API Retention: Allows system owners to retain a record of all API interactions, which can be useful for various purposes such as investigating anomalous behavior, identifying performance bottlenecks, or compiling evidence from nefarious activities.
10. API Memory Usage: Allows system owners to understand the amount of memory required to support an API and can help identify potential memory leaks, and ensure that the API can handle a large volume of requests without running out of memory.

These metrics will provide valuable insights into the performance, usage, and reliability of the APIs. Projects should regularly monitor and analyze these metrics to identify areas for improvement and to make data-driven decisions that optimize the API's performance and improve user experience.

It is important to define specific goals and thresholds for each metric based on system requirements and consumer expectations.

## 6 Development, Security, and Operations (DevSecOps)

There are several key documents to consider when planning and implementing DevSecOps:

1. The DoD Enterprise DevSecOps guidance (DevSecOps Reference Architecture 2023) is the Department’s primary guidance to develop and deliver secure software using modern practices. The guidance outlines principles including API security testing that aligns with industry best practices and supports the DevSecOps culture.
2. NIST SP 800-204 (e.g., Microservices) (2019), NIST SP 800-207 (e.g., ZT) (2020) and other industry standards also help ensure APIs meet operational security requirements.
3. DoD Instruction 5000.87, “Operation of the Software Acquisition Pathway” (2020), defines DevSecOps as, “An organizational software engineering culture and practice that aims at unifying software development, security, and operations.” This instruction establishes policy for the software acquisition pathways and calls out DevSecOps as a recommended development strategy.
4. The Carnegie Mellon University Software Engineering Institute (SEI) described DevSecOps as “an entire socio-technical environment that encompasses the people in certain roles, the processes that they are fulfilling, and the technology used to provide a capability that results in a relevant product or service being provided to meet a need” (SEI May 2022a).

The processes and practices of DevSecOps are not significantly affected by whether the software in question is an API or not; however, APIs necessitate additional verification before release.

Implementation of a versioning strategy including backward compatibility, which will affect the DevSecOps approach or automated pipeline flow and processes. Backward compatibility between versions creates challenges for integrating and releasing new code; however, strategies exist to make this backward compatibility possible: releasing new APIs on unique resource locators (paths or ids), feature flags, content negotiation headers, creating optional fields on API requests or even maintaining multiple versions of APIs deployed at the same time. Coordination with consumers or clients may be needed to ensure everyone is aware of a deployment with a breaking change. If the team keeps their API always backward compatible, it can be released once all tests and acceptance criteria pass. How an API is versioned and developed will affect the deployment strategy, and the trade-offs must be considered.

Similarly, the architecture of an API will affect its deployment. If an API is served by a single physical machine, it may be simple to re-deploy a new version, but considerations about downtime and user experience then need to be considered. Conversely, if the API is designed in such a way that user requests are load balanced across several machines or a cluster, a team may

be able to perform a rolling release that deploys to one machine at a time. Again, this is intertwined with backward and forward compatibility of the API. In these ways, the use of APIs in a system will affect the DevSecOps pipeline and process.

The DoD Chief Information Officer (CIO) provides general documents on DevSecOps at <https://dodcio.defense.gov/library/>.

## 6.1 DevSecOps Enabling Technologies

This section addresses five key DevSecOps-related technologies: version control, secrets management, policy enforcement, embedded security, and the CI/CD pipeline flow.

In the DevSecOps Platform Independent Model (PIM) (SEI May 2022 ), a requirement of maturity level 1 (of 4) for a DevSecOps program is to have a source code repository and to maintain version control of that source code (these are requirements Dev\_7 and Dev\_7.1 in the model). This use of version control is essential to enabling a DevSecOps project. Version control allows for consistent change management and integration of new changes into the deployed version of the API. Understanding what files, versions, features, and defects are present in a version of an API is necessary for each step in the DevSecOps process (DoD CIO March 2021 ). In addition, being able to publish a new version of an API or roll back to a previous version is dependent on a version control strategy.

Secrets management is particularly important for producers and consumers of an API. It is common for an API to require authenticated and authorized access to call certain endpoints or certain functions. Developers of APIs should use well-known industry standards such as Multi-Factor Authentication and OAuth 2.0 for delegated authentication. Authorization services (e.g., Policy Enforcement Point (PEP), Policy Decision Point (PDP)) should be kept separate from the API and should compute decisions based on user roles, scopes, and policies. These should be based on the principle of least privilege, giving clients only the minimum necessary permissions.

Once authenticated and authorized, an API consumer will need to securely store and rotate credentials, secrets, and encryption keys, which can be commonly achieved through a secrets management provider. Examples of providers include Amazon Web Services (AWS) Secrets Manager, Azure Key Vault, and Hashicorp Vault. In addition, detailed auditing and logging of authorization failures should be implemented in a centralized service away from the API that does not collect any PII. A well-designed authorization mechanism with a centralized policy store is crucial for securing APIs and protecting data from unauthorized access.

The API is generally the most exposed portion of a code base or system to which users and malicious actors may have access. Baking in quality and security to the code that implements an API is crucial to both security of the system overall and to meeting requirements. For an API in a

weapon system, these are typically defined in a Contract Data Requirements List (CDRL), which references a Data Item Description (DID). The DID defines data format and content requirements or another interface document. Web-based or administrative APIs may provide requirements and data definitions in a different format.

In either case, a pipeline can ensure the DIDs or data format requirements are being met by the system as implemented. A team then can answer specific questions about the APIs such as, “Am I exposing any vulnerabilities through poor code conventions? Do the functions handle the nominal cases and edge cases as specified? Am I exposing any secret keys or tokens in the built software?” All these aspects are especially important for API development as the API is the user’s interface to the code (be it a human or non-person entity). Tools exist that combine these ideas as well as create documentation of user acceptance testing in a matrixed organization.

As discussed in the testing section, there are many recommended API test and evaluation roles, and having a robust CI/CD pipeline can free up developers’ time to fill these roles. In addition, pipelines can provide feedback on the progress, conformance, and compliance of software development and developed products. This can be in the form of “gates checks” on the status of test results, manual approvals, or other criteria before a new version is published. For APIs, these checks could include whether a new version has been approved by a dedicated test person.

## 6.2 Monitoring and Logging

APIs provide a unique opportunity to measure and monitor a system. This measuring and monitoring plays into collecting information during the operations cycle of DevSecOps as well as enabling ZT through monitoring and logging of users. In addition to enabling security through the feedback they provide; APIs can enable a better user experience if the logging or monitoring is able to determine common usage errors.

When considering APIs in a DevSecOps context, APIs can enable operations monitoring. According to the DoD DevSecOps reference architecture (DoD CIO August 2019) operations monitoring should be automated, and APIs provide a means for this automation. The metrics will be different for each system.

Monitoring API usage can help inform a project’s versioning strategy as identified in Section 2 (API Governance). If a breaking change is to be made, some users may have trouble upgrading and may need to stay on an older version of the API. If the project is willing to support multiple versions, monitoring can help inform when all or most users have switched to a new version and support for an old version can be dropped. In the context of deployed systems, this monitoring of multiple concurrent versions is crucial to make sure users in the field are not cut off.

APIs also help in monitoring a system by providing transparency of usage. Users that query a monitored API can provide valuable information for analysts, such as identifying user interaction patterns not obvious in the previous development cycles, detecting errors and incorrect usage, etc. In addition, as discussed in the Section 6 (Testing), monitoring and rate limiting (or an API Gateway that combines all these components together) can help prevent a denial-of-service attack. This protection provides a better experience to legitimate users.

When considering APIs for monitoring in a ZT context it is necessary to aggregate log data for building patterns of usage for consumers of the API. According to the Cybersecurity and Infrastructure Security Agency (CISA), an optimal ZT will make authorization decisions based on “incorporating real-time risk analytics and factors such as behavior or usage patterns” (CISA 2023, 23). To be able to identify and check against legitimate usage patterns, however, one must have a log of data sufficiently large to prove robust for analysis.

The CIO’s DevSecOps Reference Architecture recommends use of an API Gateway (see Section 4.3.7) to support continuous monitoring by way of its reverse proxy capability for inspection of all traffic traversing the gateway.

DevSecOps and Agile software development are based on the ability to receive feedback from the users and incorporate it into the project plan to continuously improve. The ability to collect metrics and information from an API can make them an asset to this end. While API development is largely the same as for any other software, the security and operations of an API may need to be tailored to meet requirements of different architectures and operational environments.

## 7 Testing

APIs, like any software, may have bugs, which is anything deviating from the expected functioning of the system. Vulnerabilities are a subset of bugs through which the system can be exploited (intentionally or unintentionally) to create significant risk for the operation of the system, people, and organization. What constitutes “risk” depends on the system component, data, personnel, and other factors (Mead and Woody 2017). APIs are particularly risky because they serve as system access points, so a bug or vulnerability in an API can enable bad actors to enter the system and cause it to behave unexpectedly. Preventing this from occurring requires a good design, implementation, and continuous testing of the software product to provide high assurance the system functions only as intended.

Testing is an integral part of the software development life cycle assuring the software product is working as expected and finds potential sources of risk from system vulnerabilities. Categories of testing include not just software but documentation, personnel, system integration, system-of-systems, and more (Firesmith 2015). This section will focus on software testing methods specific to APIs based on available data and best practices.

### 7.1 Software Testing

Software testing comes in two basic varieties: *manual* and *automated*. Manual testing provides qualitative feedback from the perspective of the user, whereas automated testing is useful for getting fast and repeatable feedback. Using automated tests has been demonstrated to significantly improve software quality and reduce manual labor in the long run, even though there is an upfront cost in creating the tests (Kumar and Mishra 2016) (Williams 2009). Some of the most common types of automated tests are unit testing, integration testing, and application testing (sometimes called *end-to-end [E2E] testing*). Below is a brief description of each type of testing and how it is used:

#### 7.1.1 Unit Testing

Unit testing is low-level testing focusing on a small part of the software system usually written by developers using a testing framework<sup>4</sup>. Unit tests can be written for both the interface and implementation of an API. Unit tests can be documented early in the design phase, added during development as developers discover new data paths, and added long after implementation to provide regression testing for any kind of refactor. Because it is infeasible to test all the possible

---

<sup>4</sup> A testing framework is a combination of practices and tools designed to help test more efficiently, usually in the form of libraries that assert code behavior.

combinations of inputs an API can have, developers can use two main strategies of unit testing for dividing the input space: deterministic and non-deterministic.

The deterministic strategy uses equivalence partitioning and boundary value analysis (Humble and Farley 2015). Equivalence partitioning divides a set of test conditions into different testable segments. Testing discrete segments provides the benefit of dramatically reducing the range of inputs to test while keeping the testing values relevant to what the software is supposed to do.

For example, the grades for an exam may be A (90-100), B (80-89), C (70-79), D (60-69), and F (0-59); each grade can be thought of as a different segment to be tested. In API unit testing, rather than testing the input range from 0-100, a project can test a single value from each segment (e.g., 5, 63, 78, 85, and 92). A testing segments approach reduces the range of inputs to test by providing a representative sample. Boundary value analysis tests the values between the partitions. So, in the case of the exams, unit tests would cover input values 0, 60, 90, and 100, which are at the edge. Combining these techniques provides a way to thoroughly test a variety of values without having to exhaustively test the entire input space.

The non-deterministic strategy uses a fuzz tester (OWASP 2024), an automated, non-deterministic type of testing tool that excels at generating a range of testable inputs in a short amount of time. Fuzzers vary in how structured their inputs are and how they change or mutate over time. Fuzzers also can vary from black-box (not considering the internals of the program) to grey-box (having some idea of the internal structure and the kinds of inputs it accepts). To increase the coverage of inputs and outputs in an API, a fuzzer should be combined with automated unit tests. There are several open-source fuzzing tools, such as RESTler (GitHub 2024) and Dredd (2024), which will consume an OpenAPI specification and automatically generate and execute test cases against an implementation.

### 7.1.2 Integration Testing

Integration testing is used to test whether multiple units of code developed separately (such as services, modules, or projects) work correctly when connected, or integrated, with each other. For example, an integration test for an API that relies on a database connection will provide a mock or test double<sup>5</sup> for the database and ensure that it functions correctly during its expected (or unexpected) behavior. Integration tests assume that the stubbed services and dependencies are accurate and faithful representations of the real services.<sup>6</sup>

---

<sup>5</sup> In this context a mock or test double is narrowly targeted to replicate the behavior of real software as it pertains to the test in a controlled way. This is short of a digital twin which should replicate a real software or system in a general way.

<sup>6</sup> These services may change over time unexpectedly, in which case a Contract Test (see section 6.4) will provide a valuable stopgap for the developer to ensure that mocks map to the real service's interface and behavior.

API integration tests should cover the issues commonly encountered in distributed architectures and systems (i.e., concurrency, fault tolerance, failure recovery) (Anderson 2020, 243). Integration testing also would include APIs that need to function in a denied, disrupted, intermittent, and limited (DDIL) environment. In some cases, these adverse environments require alternate code paths or, in the case of an API, a different implementation to maintain operation. Even when co-located with other software on the same physical system, components that interact through APIs can fail when multiple consumers try to use the API simultaneously. Orchestrating integration tests that target these cases can be important for systems (e.g., a database API) that need to be atomicity, consistency, isolation, and durability (ACID) compliant.

Fault tolerance and failure recovery are closely related. Tests for an API should ensure that an error-handling mechanism is exercised when the API processes bad data or is in an error state. Failure recovery is then activated to perform the steps to move past the error state and provide service to consumers. In some systems, failure recovery can be entirely manual or infeasible. Even in these cases, ensuring the API sends the correct signals or returns the right error code can be important. Tests that cover both fault tolerance and failure recovery provide robustness and availability to an API.

Two ways to test for fault tolerance are *stress testing* and *graceful degradation testing*. An API will go through a pipeline stage where different load parameters (# of threads, connections, etc.) are increased and decreased past the API's expected operating range to test its resilience. Stress testing also helps uncover potential vulnerabilities the system exposes when services are strained or taken past their expected working range by, for example, overfilling buffers and writing in over memory, or unexpectedly dropping transactions if the number of connections it can handle are exceeded. In graceful degradation testing, the system responds gracefully or in a “least surprising” way to its end users if the service cannot fulfill requests.

Chaos engineering is a more advanced version of graceful degradation testing in which faults are randomly injected into the system to test its resilience and error-handling capabilities.

### 7.1.3 Application Testing

Application testing verifies that the developed application in a representative environment meets the stated requirements. This type of testing examines the input and output of the API, including the other services it relies on. This is *live testing*, and it usually involves using *test data* in a *test database* in a *testing environment* (see Section 6.2). In this manner, application testing also can be considered *user acceptance testing*. Ideally, application testing is conducted during one of the later stages of the CI/CD pipeline, running a series of automated tests on a testing environment and assessing that both the *happy path* (i.e., expected behavior) and a few *bad paths* (i.e., unexpected behavior) are tested. Automating this step eliminates a potential manual checkpoint

to enable changes to be deployed quickly. Certain industries require a human to review changes before deployment, but this review can be performed after application testing and as the final step before deployment.

When considering application testing for APIs, it is important to consider adversarial users. This is also called threat-based cyber testing. Adversarial users can exploit an API in several ways. In addition to the previously mentioned issues with authentication, the Open Worldwide Application Security Project (OWASP) describes “unrestricted resource consumption” and “unrestricted access to sensitive business flows” as two of the top 10 risks for APIs. Depending on the vector, testing for these vulnerabilities may be performed as an integration test or an application test. Ultimately, ensuring all services react appropriately in the development environment is essential to catch errors induced by bad actors in production.

#### 7.1.4 Other Types of Testing

Other types of software testing exist, including coverage testing, DAST, SAST, IAST, and dependency scanning. Each of these tests provides an additional layer of verification by testing different parts of an application under varying conditions. Because software testing is context-dependent and exhaustive testing is not possible, developers and testers should be knowledgeable about the various testing frameworks and their appropriate use to optimize the amount of testing based on the requirements, quality attributes, and overall risk assessment of the API.

An effective API testing strategy, leveraging other tests (e.g., coverage, SAST, and dependency scanning foundation) provides programs with a strategic advantage and establishes a path to a successful deployment and high assurance. As Justin Searls from Test Double points out, “Nearly zero teams write expressive tests that establish clear boundaries, run quickly and reliably, and only fail for useful reasons. Focus on that instead” (Searls 2021).

## 7.2 Test Environments

When testing APIs, a test environment is required. The *test environment* is the set of systems that tests run on and the resources needed to create conditions as close to production as possible. Things that make up a test environment include hardware configuration of servers, operating system configurations, middleware, and services that support the application (such as databases and authentication systems) and the API system(s). According to the Software Engineering Body of Knowledge (SWEBOK), a test environment “should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials” (IEEE Computer Society 2023). The objective is to ensure that testing can be repeatedly and reliably performed such that the results of the tests provide confidence to developers and product owners that the system is ready for launch.

### 7.2.1 Ephemeral versus Persistent Environments

Two main forms of testing environment are *ephemeral* and *persistent*:

- An *ephemeral environment* is created to execute tests and is removed once tests are complete. It relies on the infrastructure-as-code (IaC) principle. This method of testing can save money and resources in a cloud-based or shared computing environment (as long as provisioning resources is not so expensive to do). Given that resources are used only while tests are executing, hourly or per-minute compute costs are minimized (Villalba 2023). An added benefit of using this environment is the decreased likelihood of configuration drift, which is the tendency of systems to change over time and which results in small software differences from the originally deployed system. Examples of ephemeral environments are short-lived containers, serverless components, and environments built through IaC that are rebuilt and destroyed in each continuous integration (CI) build cycle.
- A *persistent environment* is long-lived and is not decommissioned or spun down at the end of testing. This type of environment offers the benefit of pre-allocated resources to execute the tests, potentially leading to quicker test starts; however, since it is a long-lived infrastructure, continued configuration management becomes an issue that must be handled. This configuration management includes updates to the system and correctly setting the system's initial state. Having a known initial state ensures that tests can be consistently repeated, with the setup process often being the most time-consuming aspect. Ephemeral environments are generally not affected by prior states. Examples of persistent environments are Virtual Machines or servers that are specifically provisioned as "Test" servers.

When developing a plan for the test environment, the focus should include feedback and participation from developers, project managers, and applicable stakeholders. To adequately test the system against performance requirements, the plan should incorporate expertise and input from these participants and should be representative of the operational environment. Section 4.3.3 of the Cybersecurity Test and Evaluation Guidebook (DoD February 2020) provides helpful guiding questions about the who, what, when, where, and how of planning a test environment. The questions include considerations of test infrastructures (e.g., appropriate labs) that can have a long lead time to set up. In the DevSecOps Playbook (DoD CIO September 2021), the section titled "Play 2: Adopt Infrastructure as Code (IaC)" applies to both persistent and ephemeral test environments. The playbook provides information about the key benefits of IaC. For API testing, IaC has the benefit of providing a known state for starting a test. Many APIs are stateful, so testing requires setting up a known good (or bad) state. Automating the setup of ephemeral test environments is key to enabling developer velocity.

APIs frequently require a network to operate. As per the previous recommendation, the test environment should be as representative as possible, which includes making the networking environment similar to production, while at the same time putting safeguards so external systems are not affected by testing. This testing could include emulating the various network segments (e.g., local area network (LAN), wide area network (WAN), tactical data link (TDL)) to test properties such as reduced throughput, error rate, or high latency. This would be critical for APIs that are required to work in a DDIL context.

Containers<sup>7</sup> are another useful tool for quickly packaging and shipping an API to different environments and can be a crucial part of the testing strategy. Containers isolate processes that provide repeatable, sandboxed environments that can be packaged in a standardized format.

### **7.2.2 Test Data**

The U.S. *Department of Defense Data Strategy* (September 2020) goals are summarized by VAULTIS in Section 1.4. These goals should be applied to operational APIs, and the data used to test the APIs should be congruent with these goals. Especially, it is important to ensure realism in the environment and in the test data. The test data must have the same format (interoperable) and have real or representative values (understandable) that are available for API developers and consumers to use (accessible) while not leaking any sensitive data, such as PII, PHI data, or other data (secure). When securing data (e.g., test data), testers should encrypt data-at-rest and data-in-transit.

Often, test data may not be readily available. In these cases, it may be necessary to generate simulated test data. Again, this data should be trustworthy, and producers and users of the simulated data should be confident that it represents real-world data (DoD Issues New Data Strategy 2020). Conversely, for some systems such as those for machine learning (ML), environments may require so much test data that configuration management is a concern. Both the DoD Data Strategy (September 2020) and the Data, Analytics, and Artificial Intelligence Adoption Strategy (June 2023) provide evaluation criteria for data integrity and accessibility. When considering AI/ML systems specifically, data gathering, preprocessing, and use come with specific considerations that are out of scope of this guidebook.

### **7.2.3 Testing Suite Infrastructure**

An automated testing suite with slow tests can tempt developers to skip writing additional tests or running them, which defeats the purpose of the testing suite. As such, it is important that a project's test suite be both fast and reliable. What is *fast*? Some practitioners say that tests should

---

<sup>7</sup> Containers are a package of software and all dependencies that virtualize the operating system to allow execution in many different environments with minimal configuration of the host system.

take no longer than 10 seconds to run (Seemann 2012) based on research that 10 seconds is the maximum amount of time people will wait while maintaining focus on the task at hand; however, 10 seconds may be too fast for the project's needs or capabilities. If so, team members must agree on how long they are willing to wait without having to switch to another task. Achieving these agreed-on speeds requires configuring the test framework to use parallelization as much as possible.

Other testing strategies for longer running tests include nightly or weekly builds of a software system. These tests are run less frequently but still on a regular schedule. While not a replacement for on-demand testing, this strategy can help offload long-running tests or builds to servers overnight. An example of where this may be applicable is in an end-to-end test of a microservice based API. Each service behind the API has its own pipeline that runs quickly and maybe performs its own unit-level testing. Every night a more extensive pipeline to build the entire product can be run that would perform integration or regression testing of the API. This enables mitigation of the risks noted about testing with mock services but where deployment of the real ones can take too long or be too resource intensive to perform on a per-commit basis. In addition, tests such as DAST or fuzzing could be run indefinitely. Moving these to a nightly or weekly pipeline could enable a longer minimum time for these tests to run while allowing a fast developer workflow.

### 7.3 Testing Skill Sets

As mentioned in Test Environments (Section 6.2), when constructing a system verification or test plan, it is important to ensure that the appropriate laboratory or range environments are available. Equally important is having the correct personnel with the right skills to execute the planned testing. The specific skills needed for performing API testing depend on the specific technology stack and deployed environment the API uses.

While many skill sets are needed for testing an API or any IT system, this testing should be continuous when following a DevSecOps model. Therefore, all team members must be skilled in testing since it is a critical activity.

Given that the pace of development in a DevSecOps project should approach continuous delivery (CMU SEI 2022), it is necessary to perform testing as often as possible and in sync with the development cycle. Enabling continuous testing using a Lean, Agile, or Scrum framework can conflict with the complete set of skills required. The current DoD Cybersecurity Test and Evaluation Guidebook (DoD February 2020) proposes 26 possible test and evaluation roles as part of a system cybersecurity working group. For many projects, it might be infeasible to staff this many roles or include them all as members of the development team. Each role, however, is applicable to API test and evaluation; therefore, a project or program developing an API must

plan to engage dedicated or specialized test personnel in addition to conducting routine automated testing. Each testing role brings a different perspective to the design and building of the automated tests and also should be engaged when tests fail.

Test personnel, developers, and all other members of an Agile team should receive training in the Agile methodology, practices, and specific ways the team or program has tailored their own implementation. This training should help to promote cross-functional teams with all members pulling work (including testing) from a backlog. When it comes to cross-functional teams, for testing, software developers are not acceptable substitutes for actual API users. Therefore, it is important to have user representation on the development team or to test using actual end users. Complications can include intricacies in actual workflows as opposed to policies or procedures handed down as requirements. API users' domain knowledge is therefore a required skill set when testing APIs.

For additional information on T&E certification training and credentials, see the Defense Acquisition University (DAU) online catalog (iCatalog) of courses and schedules:  
<https://icatalog.dau.edu/onlinecatalog/CareerLvl.aspx>

#### **7.4 Contract Testing**

APIs are useful tools for decoupling software and allowing both modular construction and testing. Using APIs can allow multiple teams to work on different components of a software system independently. If each component talks to the other through a well-defined API, it can ease the burden of integration testing when the system is large enough to make it difficult to assemble all the needed operators and testers to perform a full end-to-end test. An example of such a system is a military satellite communications system. It is not reasonable to test the system only once the satellites are in orbit and users are deployed to the area of operations intended to be covered by the system.

In a general sense, an API is a contract between a software application's provider and consumer that specifies what the system will do. In 1986, Bertrand Meyer coined the idea of programming by contract or design by contract (Meyer 1997, 342). In this paradigm, a software engineer formally defines a specification of each function or method that the system exposes. These specifications include noting preconditions, postconditions, and invariants (Kutztown 2014). Preconditions are the criteria required before a function is executed. These are things that the service or API provider expects to be true before a function is called. Postconditions are the state or set of criteria that must be true after a function is executed. Invariants are the data or states that, regardless of the operation or transformation applied by the function, will not change.

The benefits of the formal specification of a contract when testing an API are twofold:

- Before any implementation of an API is provided, a well-defined API contract makes it possible for the development team to write example client code that uses the functions or endpoints. This API user testing eliminates any time spent implementing irrelevant methods or API calls that are hard to use from the client perspective.
- During integration testing of different software modules, some systems can be hard to simulate in the test environment, the target system can be expensive to operate on (e.g., quantum computing hardware) (IBM Qiskit 2024), or the target system is not even built yet. In these cases, having an API contract that accurately represents a software module or library can help the software's producers and consumers conduct the integration testing.

How contracts are defined in a particular language varies. In Java, using Javadoc comments to document the parameters, return values, exceptions, and function's purpose is a common (though less formal) way of documenting a contract. A variety of tools and programming languages offer varying levels of formality for defining contracts that can help enable the verification of API usage.

The use of the OpenAPI (Swagger 2021) specification to define an API aligns well with the contract programming paradigm of specifying preconditions, such as the domain of inputs and a description of the endpoint. OpenAPI also allows a contract developer to specify the return from an endpoint, including the return code, a description of what that return code means, the schema of any returned data, and examples of the data. OpenAPI potentially falls short in its ability to specify invariants of a function. For instance, in a REST API, GET requests should be idempotent; however, there is no way to document what an endpoint may or may not change in terms of states outside of a text description.

## 7.5 AI-Enabled Testing

While a nascent field, AI-enabled testing of all software, APIs included, is growing. The following sections discuss some more traditional methods as well as some of the latest research in AI (e.g., natural language processing (NLP)), ML (e.g., supervised learning), as they are being used for API testing. This field of research is nascent, and as more tools come online it will be another option for testing. The general guidance is to start with more traditional, proven methods of testing as described in this document and elsewhere and then to move into these newer methods as needed for a given program or project.

Several types of relatively traditional testing such as random argument testing or fuzzy logic for assertion-based testing, can be enhanced through the use of AI or ML (Alakeel 2015).

### 7.5.1 Test Generation

One proposed technique is using NLP, such as OpenAPI, to enhance API specifications (Kim 2023). These plain-text descriptions often provide examples, parameters, and input domain information, and some might infer new rules that have not been documented. Researchers extracted these descriptions via an NLP model and parsed them for additional API rules that were not previously written. These new rules are validated against the document and injected as new rules. This novel method could be wrapped up as a validation stage in a CI pipeline to ensure every API change provides standardized, accurate specifications.

In another case, researchers used a large language model (LLM) to generate test data for an API successfully from an OpenAPI specification. The researchers thoughtfully used prompt engineering to generate useful response data sets. The researchers explained, “We design[ed] a prompt specific to the testing endpoint and status code being evaluated. This prompt includes relevant information about the scenario, such as the API endpoint, the expected status code, and any additional instructions for generating suitable test data or verifying the response” (Nguyen, et al. 2023). Applying LLM in this way can be helpful for finding corner cases where existing tools, such as fuzzers, may fail to find them.

### 7.5.2 Verifying Correctness

With AI-enabled testing, there is a need to verify the correctness of the output of the models. Research has shown that foundational<sup>8</sup> models can fail to identify well-understood errors in code (Sherman 2024). In addition, as with any test method, there is a need to integrate the use of the tool or method into the existing test pipeline. For example, Cornell University researchers have proposed a process for LLMs called Assured LLM-Based Software Engineering (Alshahwan, Automated Unit Test Improvement using Large Language Models 2024b). This strategy puts some guardrails on the generation of code by models that safeguard against regression and ensure a measurable enhancement in code coverage. This is especially important for API testing because regressions in API capabilities are the most visible to the product’s end user. With this set of guarantees in place, commercial companies have improved the deployment of their systems using these types of tests (Alshahwan, Assured LLM-Based Software Engineering 2024a); however, they are still in the early experimental phases of using this strategy and are targeting improvement of their test cases, not replacing the foundational work of writing tests.

---

<sup>8</sup> Large language models trained on a diverse data set and not fine-tuned to a specific domain.

## 7.6 Zero Trust Testing

To consider the nexus of ZT, ZTA, and API testing, this guide uses five of the seven basic tenets of ZT from NIST SP 800-207 (2020):

### 7.6.1 API Authentication and Authorization

Continuing to move up the stack from configuration and infrastructure, a team testing an API should next look for common broken authentication issues within the API. Broken authentication is the second of the top 10 security risks to APIs according to OWASP (Yallon 2023). This type of testing should occur in unit, integration, and application tests, depending on how authentication is integrated with the API. Tests should focus on finding flaws in an API's implementation that would allow a user to assume the identity of someone else. In this context, testing for *non-repudiation* is useful.

Broken authorization is another dimension that needs to be tested in ZT architectures. Authorization verifies that the user has the right permissions to use a resource. With ZT, users should be given the minimum permissions to accomplish the intended function and no more (i.e., the principle of least privilege). When testing an API for broken authorization, testers should try to find flaws in what a user is able to request or modify through the API. For example, a tester can determine whether an attacker can modify the ID of an object that is sent in an API request, which indicates a broken object level authorization (BOLA) (Yallon 2023). These types of tests generally can be handled at the unit level, but they might require integration testing if the API offloads the authentication and authorization to a central service, which is a recommended best practice (Yallon 2023).

### 7.6.2 API Replay Attack Vulnerabilities

Replay attacks, in which a malicious actor reuses an expired credential, are similar to broken authentication. Testing against this type of vulnerability ensures that any tokens or keys used are short-lived and valid only for connecting to the API in the current session.

### 7.6.3 Securing APIs with ZT

Testing that the data exchanges between an API and its users are secured (e.g., through encryption) is specialized testing that requires a particular skill set. As with authentication and authorization, the accepted best practice is to use libraries and functions provided for this purpose and not implement them from scratch. To verify that an API is appropriately implementing a secure connection falls into the integration testing category and may require capturing network traffic between the user and the API to ensure that a secure handshake was made and the correct protocol options were used. This type of security testing, while necessary

for ensuring compliance with ZT, is admittedly specialized and may require dedicated personnel trained in this area.

#### **7.6.4 API Testing and Development Environments Are Assets**

Developers must have an infrastructure in place to run tests and pipelines to enable the testing process. Testing, preproduction, and other environments must be accounted for and interoperate with the tools that enforce ZT (Sanders 2021).

#### **7.6.5 API Monitoring**

APIs provide a unique opportunity to enable the monitoring of the systems where they are deployed. When considering a ZT architecture, it is important that, once the system is in place, it be continuously evaluated for security-policy violations. In ZT there is not a single set of rules defined once. ZT security policy is context aware and adaptable (NIST SP 800-207 2020, 3.3.1). As such, it may be necessary to pull different types of information from different parts of an information system to determine if security has been compromised. APIs can enable this determination by allowing access to the information about API access, including who accessed what and when. This ability to log user activity should be considered in the design phase, but it also should be exercised in the testing phase.

## 8 Enhancing API Efficiency and Accessibility Using Software Developer Kits and Libraries

### 8.1 Introduction of SDKs and Libraries

As an API consumer, API provider integration can be a time-consuming process taking weeks to months after initially obtaining API credentials and completing whatever API governance paperwork (e.g., SAAR, MOU, SLA) that might be required by the provider (see Section 3, Project Governance, for more information).

It is important to understand that in the commercial world companies are focused on providing customer satisfaction through making access and use of their API as easy and efficient as possible. Quick and easy transition of new customers using a company's API provides a rapid initial and ongoing income stream.

The use of SDKs and libraries is a critical component of rapid integration with APIs, enabling developers to quickly integrate with APIs through simple tools and instructions. Directly deploying libraries via language-specific package managers, such as pip for Python, further streamlines this process; however, the lack of incentive for service providers to create SDKs can pose challenges for technical leadership within the DoD.

Without SDKs, developers may face difficulties integrating APIs into existing systems or applications, leading to potential delays and increased costs. This is particularly problematic in the context of modern API deployment within the DoD, where mission requirements are typically prioritized over revenue or customer satisfaction.

To address this challenge, technical leadership should consider explicitly outlining the need for SDKs or requiring service providers to develop SDKs as part of their Statement of Work (SOW), as well as exploring alternative options for supporting rapid integration, such as distributing SDKs and libraries as downloadable bundles for the most popular languages. In addition, technical leaders can explore alternative solutions such as open-source SDKs or partnering with other organizations to develop SDKs.

Ultimately, the goal is to ensure that the DoD can effectively leverage modern APIs to meet mission requirements while minimizing risk and optimizing resources. By addressing this challenge, technical leadership can help the DoD stay at the forefront of technological innovation and maintain its competitive edge.

Although DoD is seeing an increasing number of programs excelling with a culture, acquisition strategy, and adequate funding that includes the provision of SDKs and libraries across the

program life cycle, a significant number of programs have overlooked SDKs and other software modernization best practices and will be challenged to adapt.

## 8.2 What Is an SDK?

Amazon Web Services describes a software development kit as follows:

A software development kit (SDK) is a set of platform-specific building tools for developers. You require components like debuggers, compilers, and libraries to create code that runs on a specific platform, operating system, or programming language. SDKs put everything you need to develop and run software in one place. Additionally, they contain resources like documentation, tutorials, and guides as well as APIs and frameworks for faster application development. (AWS SDK 2024)

An SDK typically includes various libraries specific to popular development languages (e.g., Java, Python, C#). A developer would add that library to their application and then access the API via an object instantiation within their source code. Most modern integrated development environments (IDEs) or editors include code-editing features (e.g., completion, hinting, documentation). Code completion features allow developers to rapidly write code and be more productive without advanced knowledge of an object's underlying methods or attributes.

### Software Test and Production Environments

In the DoD, for a SOAP API-based ecosystem, there is typically a silver or test API integration environment in which developers can ensure the API is able to properly authenticate and communicate with the system. Silver or test environments are typically at an unclassified (e.g., IL2-IL5) network and usually contain synthetic or de-identified data.

Once everything is working, the system will provide gold or production access containing live data. Programs should consider which API environments will be needed and how they will be used early in the program life cycle and should ensure these decisions are explicitly called out in the SOW and Request for Proposals (RFP) during the acquisition phase.

### API Personas Supporting Test and Production

Defining API personas is a crucial step in designing and developing APIs (Pedro 2023). API personas help programs understand the needs, preferences, and behaviors of target users and consumers. Personas ensure developers are creating APIs that meet their specific requirements and testing and validating API functionality across each persona's use cases.

Here are some techniques to discover and define API personas:

- Conduct user research: Gather data on the target users by conducting surveys, interviews, and focus groups. Ask questions about their current workflows, pain points, and goals, as well as their expectations and requirements for an API.
- Analyze usage data: Look at data on how existing APIs are being used. Identify patterns and trends, such as which endpoints are most popular, which users are using which features, and what types of errors are occurring.
- Identify user roles: Define the different roles or types of users that will be using the API. For example, user roles may include developers, product managers, or business analysts.
- Create user journey maps: Map out the steps that each persona takes to achieve their goals using the API. Identify pain points and opportunities for improvement.
- Prioritize personas: Determine which personas are most important to the API project and focus on meeting their needs first.
- Develop persona profiles: Create detailed profiles of each persona, including their goals, pain points, motivations, and behaviors. Use these profiles to guide the API design and development process.

By following these techniques, project teams can create API personas that accurately reflect the needs and preferences of the target users and develop APIs that meet their specific requirements.

Here are some examples of a few typical personas:

- Data Provider/Curator
- API Developer for App
- Data Consumer
- Data Platform
- Non-Person Entity (NPE)

An example of a robust and mature API ecosystem is the AWS Developer Tools – SDKs and Programming Toolkits for Developing and Managing applications on AWS (AWS Tools 2024). The AWS API currently supports 13 different programming languages (e.g., C++, Go, Java, JavaScript, .NET, Python).

### Tools Commonly Found in SDKs

Various software development tools and building blocks are commonly found in SDKs (AWS SDK 2024):

- API libraries
- Debuggers
- Compilers and Interpreters
- Profilers
- Code samples
- Deployment tools
- Integrated development environment (IDE)

Other tools sometimes found in SDKs include:

- API testing/interoperability tools

### 8.3 Origin of SDKs

The concept of a Software Development Kit (SDK) emerged in the early days of computer programming. An SDK is a set of tools, libraries, documentation, code samples, and processes that assist software developers in creating applications for specific platforms.

The first SDKs were created by computer hardware and operating system manufacturers to allow developers to create applications that could run on their systems. In the 1960s and 1970s, companies such as IBM, Digital Equipment Corporation (DEC), and Hewlett-Packard (HP) provided SDKs for their hardware and operating systems.

The introduction of personal computers in the late 1970s and early 1980s led to the development of SDKs for popular platforms such as MS-DOS, Windows, and Macintosh. In the 1990s, the rise of the Internet and the World Wide Web led to the creation of SDKs for web-based applications.

Today, SDKs are available for a wide range of platforms and technologies, including mobile devices, game consoles, virtual reality systems, and cloud computing platforms. They are used by developers to create applications for these platforms, and they provide a standardized way for developers to access the features and capabilities of the platform.

### 8.3.1 Benefits of an SDK

SDKs offer several benefits to software developers, including (source: NIPRGPT.mil (CAC-enabled website)):

1. Faster development time: SDKs provide pre-built code, libraries, and tools that developers can use to build applications quickly and efficiently. This reduces the time and effort required to write code from scratch.
2. Improved code quality: SDKs often include code samples, documentation, and best practices that help developers write high-quality code. This can reduce the number of bugs and errors in the final application.
3. Consistent user experience: SDKs provide a standardized way to access platform features and capabilities, ensuring that applications have a consistent user experience across different devices and platforms.
4. Simplified integration: SDKs can simplify the process of integrating applications with third-party services, such as social media platforms, payment gateways, and analytics tools.
5. Access to new technologies: SDKs provide access to new technologies and features that may not be available through other means. For example, an SDK for a virtual reality platform may provide tools for creating immersive experiences that would be difficult or impossible to create otherwise.
6. Reduced development costs: SDKs can help reduce development costs by providing pre-built code and tools that would otherwise need to be developed from scratch. This can save time and money, allowing developers to focus on creating unique and innovative features for their applications.

## 8.4 Support and Maintenance

The SDK will need to be maintained and updated as the APIs change or as APIs are added or deprecated. If building from scratch, the project team should consider developing the SDK using an iterative, Agile approach starting with an MVP versus trying to create the entire SDK in one step.

## 9 Real-Time System APIs

The API guidance in the other chapters can be applied to systems in general as scoped in Figure 1-2, API Context Diagram from a System Perspective. However, providing APIs for real-time systems (RTS) presents some additional challenges and certain concepts and recommendations of the general API guidance must be tailored.

### 9.1 General RTS API Principles

An RTS is defined here as:

A designed set of resources operating within an operational regime (a collection of behaviors, procedures, and conditions) that provides a quality of service that includes one of the following time constraints:

**DETERMINISTIC:** the resources must act, respond or deliver with very high probability within a minimum or predictable period of time and margin of error (i.e., requirements for deadlines, latency and/or jitter), or;

**INTERACTIVE:** have functionality that appears to update interactivity from a human perspective or;

**SIMULATED TIME:** as a simulation, must execute in 1:1 time.

Resources could be software, computers, components, networks, communication systems, people, equipment, and materials.

This chapter will help DoD programs and developers to know when they have a system that may fall under the tailored RTS API guidance provided below. The chapter will provide terminology, frameworks, and unique characteristics to help identify and distinguish RTS and API use cases. Identifying these use cases will guide programs in adapting many of the general and tailored aspects of APIs as possible for the benefit of the system and DoD enterprise as a whole. The identified systems will require additional care from the project's outset, from design through implementation, testing, and deployment. The guidance is structured to help the deployed RTS be responsive to the API User community as well as to the organic system RTS users. The RTS API guidance will stay focused on the API-related capabilities rather than deal with more specific RTS design problems (e.g., asynchronous vs. synchronous; event-driven vs. call-response; etc.).

All RTS can have quality of service (QoS) goals that could be met with enablers such as real-time operating systems (RTOS), high-performance processors, high-bandwidth communication

systems or refactored networks. QoS could also be achieved in some use cases by selecting a sufficient number of operators, vehicles, or servers, to support the system. The QoS could also include other goals that affect timeliness such as resilience, graceful degradation, resolution, and memory usage.

The deterministic RTS has a deadline aspect. A deadline is a finite window of time in which a certain task must be completed. Deadlines in general may be (1) Hard – a “can’t miss” deadline (2) Soft – a “may miss” deadline; or (3) Firm – a “no gain when miss” deadline. The deterministic RTS definition does not define the penalty outright but is considered part of the QoS specified by the system. Note the deterministic RTS definition does not define a time. Some systems may have time requirements in the milliseconds (e.g. aircraft control sub-systems), whereas others may require determinism in hours (e.g. weather systems that periodically report).

Interactive RTS, the second type of RTS mentioned in the definition, could be built on components that are not deterministic though they often rely on deterministic real-time systems as a data source. These systems (e.g., some video delivery services) give the appearance of being in real-time, but the system can be implemented using non-deterministic technology approaches, such as ones that statistically or usually appear to meet the user’s expectations. Consequence for failure is acceptable, which overlaps with the soft deadline QoS.

Simulated Time RTS are simulations that model real-world systems. They may be 1-to-1 in time, or they may act “faster than real-time.” The techniques for achieving this simulated time are usually processor resource dependent and may involve other pseudo simulated components, which are quite unlike the technology required for two previous RTS types. Simulated Time RTS considerations and guidance are not considered in the below RTS API tailoring.

### **9.1.1 RTS Examples**

RTS can come in many forms. They may be classified as different types of components and fall into different operational regime. For example, in Table 9-1, the “transport” component can support an RTS, but its quality of being real-time is dependent on the mission system goals and regime. With this variability in mind, the table lists three types of systems, providing RTS examples where possible.

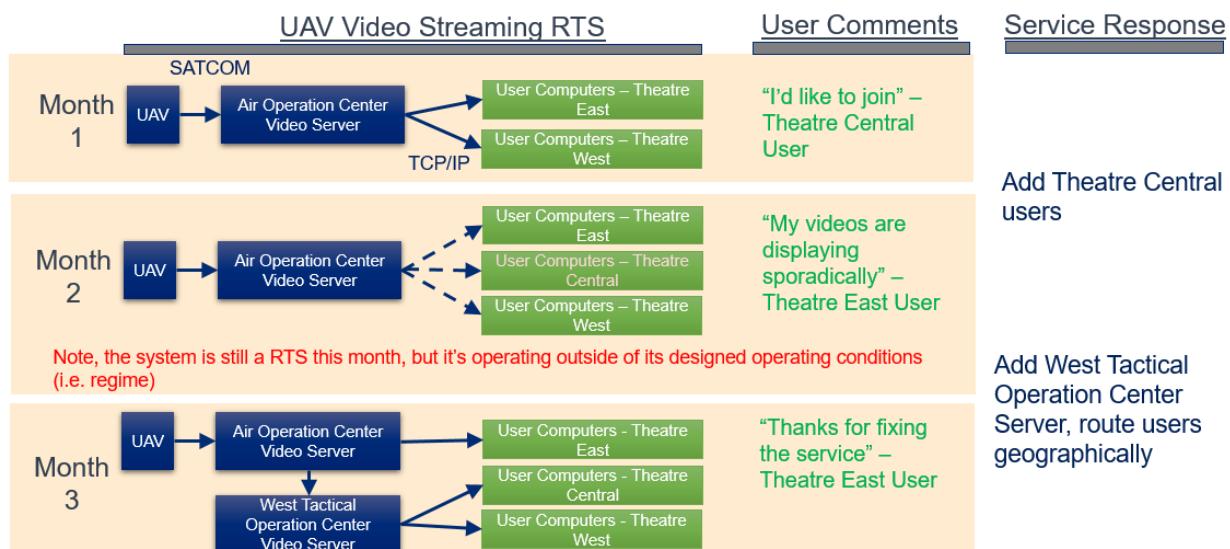
**Table 9-1. RTS Example Components**

<b>System Component</b>	<b>System Example</b>	<b>Operational Regime Type</b>
Mission System	Natural Language Processor for Intelligence	Real-time
User	Unmanned Aerial Vehicle (UAV) Operator	N/A
System Hardware	UAV Sensors	Could be real-time or non-real-time
Information System	DoD Intelligence Information System (DODIIS)	Could be real-time or non-real-time
Client System	UAV Flight Control System	Real-time
Process Server	Common Operating Picture (COP) Server	Real-time
Data Store	UAV Image Database	Could be real-time or non-real-time
Application Software	UAV aerodynamic settings application	Non-real-time
User Data	UAV Location and Fuel Level	Could be real-time or non-real-time
User Interface	UAV Flight Control System Interface	Real-time
Compute	Google Tensor Processing Units (TPUs)	Non-real-time
API Catalog	Facebook and AT&T APIs	Non-real-time
Message Prep	Cooperative Engagement Capability (CEC) Cooperative Engagement Processor (CEP)	Real-time
API Gateway	Amazon API Gateway	Non-real-time
Data Communications Systems	Warfighter and/or SIPRNET Chat	Real-time
Software Messaging	Chat	Could be real-time or non-real-time
Remote Procedure Calling	Image Processor	Could be real-time or non-real-time
Message	Army Unified Network Operations (UNO)	Non-real-time
Message Translation Gateway	HUNTR tactical datalink translation gateway	Real-time
Message Interface Device	Multifunctional Information Distribution System (MIDS) terminal	Real-time
Military-Standard Data Links	Tactical Data Link (TDL) & MADL	Real-time
Data Transport	DoDIN Transport	Could be real-time or non-real-time
Routing	Static or Dynamic	Could be real-time or non-real-time
Transmission Medium	Twisted Pair Cable	Could be real-time or non-real-time
Computer Network	PCI & Universal Serial Bus (USB)	Could be real-time or non-real-time
External Network	Wired/Ethernet	Could be real-time or non-real-time

The following paragraphs discuss terminology and framework concepts that will be used in the subsequent API guidance tailoring.

### 9.1.2 RTS Operational Regime

The operational regime concept mentioned in the RTS definition suggests the prescribed nature of RTS and the need to protect the RTS from unwanted outside influences. An RTS is designed to support specific functions and goals that meet specific time requirements and it does so within an operational regime. This does not necessarily imply that when delivered the system will meet those requirements, but the system is still considered an RTS because of its design intent. Figure 9-1 illustrates an example of a UAV RTS video streaming service that goes from initially meeting expectations of the RTS to not meeting expectations as the installed user base grows. On Month 2, once the RTS service goals drop below expectations, the RTS is modified to accommodate the additional users and return to intended operational performance levels.



**Figure 9-1. Example of UAV Video Streaming RTS System**

### 9.1.3 Mission Planning Considerations

Following from the regime discussion, mission planning is the immediate consideration for the operational user. With regard to operations involving an RTS, the project should establish the traffic going into an RTS at the beginning of the mission planning phase. The RTS may not be able to sustain the traffic if not planned appropriately. If the RTS has issues supporting the requested API payloads, delivery and the value of the RTS content may be compromised as well.

#### **9.1.4 Constraints on Real-Time Systems**

Real-time systems differ from the normal asynchronous traffic in internet or internet-like systems. For example, systems cannot achieve consensus in bounded time in the presence of faults on an asynchronous network (Fischer, Lynch, & Paterson 1985). In practice, many systems can achieve consensus in a reasonable amount of time but do not automatically meet RTS QoS requirements.

Furthermore, RTS typically operate on fixed hardware. Whereas conventional systems often can be scaled up by purchasing more cloud resources (servers, bandwidth, etc.), RTS may have additional design and operational regime constraints which limit the use of many scaling solutions. Much of modern, scale-out thinking that applies to the current popular conception of APIs may not apply to RTS.

#### **9.1.5 Transport Considerations**

RTS often use specialized or dedicated transport systems to ensure deterministic message delivery. Intruding on to a bus or network with arbitrary API traffic may be disruptive.

#### **9.1.6 RTS Isolation from API Service**

This concept suggests that general API guidance mandates are not useful inside of an RTS. RTS generally operate with constrained resources (e.g., processing power, memory, or transport links operating in real-time). APIs operating within an RTS and serving an external user community would consume these resources and would likely affect the responsiveness (or quality-of-service (QoS)) experienced by an RTS primary user. Thus, the concept advocates for removing the API service from within the RTS to maximize responsiveness while also following best practices in standardizing and minimizing APIs to the extent possible to ensure timely end-to-end mission execution. In the subsequent sections several “RTS API Patterns” that use an API gateway will be discussed as a framework for discussing the subsequent RTS API tailoring.

#### **9.1.7 API Adapters and Gateways**

API implementations generally require an “API Adapter” to perform the API end-point software messaging function. This adaptation function can be hosted by the organic compute resources or may be handled by a gateway external to the system’s main compute resources. An RTS may present special challenges for APIs in that a resource-constrained RTS may struggle to implement the primary data-handling functions. Given these challenges, the general recommendation is for RTS to off-load processing and use an API Gateway in order to insulate resource-constrained RTS elements. An API Gateway is the primary component for effective

RTS API implementation, management, and security and is critical for decoupling RTS from external interference. This section discusses the use of an API Gateway as “Pattern A.”

There are three components to a generalized and gateway-supported RTS architectural pattern: the RTS, the API Adapter, and the external application (Figure 9-2).



**Figure 9-2. Generalized RTS API Pattern**

First, the RTS is the system of concern, comprising one or more parts and all of which have time response responsibilities to one another. RTS is often considered a source of data, such as telemetry, sensor data, or some other data for consumption outside of the RTS. The RTS also may be thought of as a sink (or source) for control, where the RTS will receive control commands from outside. Some RTS are designed as a remote-controlled device, but other RTS might take occasional control commands as well. An RTS could serve as a sink (or destination) for data, but care must be taken that delayed data does not affect the correctness of the RTS itself, as everything outside of the RTS itself cannot be constrained deterministically or in timely manner.

The second generalized component is the API adapter, which does not itself have deterministic real-time constraints. It brokers or interprets API calls and works within the constraints of the RTS to serve those API calls. The API provider may be entirely separate from the RTS, or it may be a built-in, fused component of the RTS. When the API provider is separate, it is an API gateway.

The third and final component of the pattern is the external application. The external application could be a presentation system to a user, a data collection system, or even another RTS decoupled from the system of concern. The external application typically is thought of as a sink for data or a source of control for the system of concern.

### 9.1.8 RTS API Pattern Variations

In the fully decoupled pattern, illustrated in Figure 9-3, all roles from the basic pattern are independent actors communicating over some form of transport, such as an asynchronous network.



**Figure 9-3. Fully Decoupled Pattern (Pattern A)**

The RTS interacts with some external API gateway. External applications access the API gateway to, for example, present data in a human-understandable format or archive data for long-term record keeping. There might be separate APIs published by the gateway, with the RTS side tailored for minimal resource burden and the external application side tailored for ease of use or some other criteria. This pattern is structurally complex because there are so many moving parts, but the RTS and the external application layers can be oblivious of one another, maximizing flexibility; however, a third-party gateway may not suit the domain of the application.

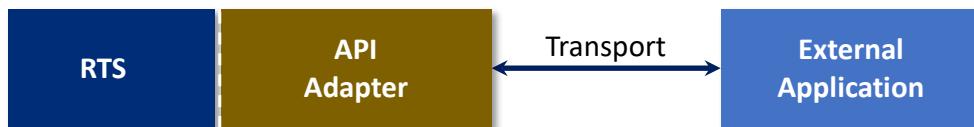
The fused-to-external-application pattern (Pattern B) in Figure 9-4 illustrates a simplified situation in which the external application itself publishes a singular API and the RTS interacts with that API.



**Figure 9-4. Fused to External Application (Pattern B)**

Structurally, this is much simpler than the fully decoupled pattern; however, it lacks flexibility in that each new external application may have its own API. This pattern is suitable for prototypes or other situations where the number and diversity of external applications is limited.

Another alternative is to fuse the API Adapter to the RTS. This pattern, illustrated in Figure 9-5, has many of the same caveats as fusing the API to the external application.



**Figure 9-5. Fused to RTS (Pattern C)**

The API presented to the external application does not make deterministic guarantees. Implementers must take care that API invocations initiated from the external applications do not interfere with deterministic RTS behavior.

## 9.2 RTS-Tailored API Recommendations

To adapt and accommodate the RTS to APIs, the following paragraphs tailor the general API guidance provided elsewhere in the document. Otherwise the general API guidance applies to RTS as well.

### 9.2.1 Data Visibility and Accessibility

#### 9.2.1.1 Applicable API Stakeholder Roles

This tailoring covers general API guidance under the topic of data and metadata accessibility, visibility, linkage, and trustworthiness and is provided for stakeholders in the roles of **Providers** and **Appliers** (Table 2-2). The guidance is intended to enable them to provide effective system solutions for the primary benefit of the organic system users, API users, and testers, described as “Consumers.”

#### 9.2.1.2 Target General API Guidance

As covered in Section 2 on Data Access: “***One of the main purposes of API is to make data available to other systems and users.***” Situational-awareness is a key concept for the data consumer and more so for RTS because the RTS API user is generally involved in use cases that are time critical and provide data to upper-echelon decision makers, allowing them to make spur-of-the-moment decisions. It is important that RTS APIs and the systems they support acknowledge and embrace their importance in providing the situational awareness by promoting consumers that can:

- Retrieve and locate the needed data.
- Exploit data elements through inherent relationships.
- Be confident in all aspects of data for decision making.

#### 9.2.1.3 RTS Considerations

Some RTS use cases may not perform adequately by:

- Making ALL data accessible and visible or;
- Ensuring inherent data relationships are developed *in situ* or extensively in real time; or

- Ensuring *every data item* is trustworthy, for example.

Where digital information in a real-time environment may be corrupted, providing relevant metadata (e.g., date or time the data was produced, the origin and authoritativeness of the data, or other relevant auditing information to assist in validating quality and integrity) along with the data item of interest provides a measure of “implicit” data integrity assurance.

#### **9.2.1.4 RTS Tailored API Recommendations**

“*Balance*” and “*tailoring*” to the specific RTS use cases should guide RTS design. Here the recommended GECP document should play a role. Consider targeted tailoring options based on the specific RTS domain and use cases:

System design should consider only ***making the minimal amount of data*** accessible or visible necessary for the data consumers to perform their decision-making goals.

Human factors are important in general and more so for RTS use cases. System design should include an assumption of how the data will be presented [displayed] to the data consumer. In a balanced-fashion, ***carefully consider the efficient display*** of linked data that could be presented in real-time. Where possible, system design should consider presenting data that are inherently related along with one another to aid the consumer in augmenting their decision-making goals by correlating the related data items themselves and looking for data consistency.

Integration testing and end-to-end testing using realistic test data should be performed before fielding to ensure the RTS is meeting its design requirements.

In Pattern A for this API principle, the API Gateway collects and makes available to external applications. For Pattern B, all external applications collect data directly from the RTS. For Pattern C, data and metadata directly from the RTS may interfere with QoS constraints.

### **9.2.2 Format**

#### **9.2.2.1 Applicable API Stakeholder Roles**

This tailoring covers general API guidance under the topic of data formatting and is provided for stakeholders in the roles of ***Providers*** and ***Appliers*** discussed earlier in the Roles section (Table 2-2) (e.g., API Developers and Application Developers). The guidance is offered to enable developers to provide effective data formatting solutions for the primary benefit of the Organic System Users, API Users, (and Testers as User-Proxies before system deployment).

### 9.2.2.2 Target General API Guidance and RTS Considerations

Some RTS use cases may not perform adequately by strictly prioritizing that data consumers have the ability to recognize the content, its context, and applicability in real-time.

For example, some data transform functions may be too computationally expensive to perform on a resource-constrained RTS. JavaScript Object Notation (JSON) requires memory, CPU, and network bandwidth that an RTS might not have. Format conversions from a simple timestamp as in integer unit count (like seconds since some epoch) to a proper ISO 8601 format may be computationally burdensome, affecting real-time performance.

### 9.2.2.3 RTS Tailored Format Recommendations

The Common Data Model (CDM) (Microsoft April 6, 2022) recommends standardized schemas or structures for organizing and sharing data, ensuring consistency and interoperability among different API components and applications. This model also promotes that the data be understandable to consumers and that they can recognize the content, context, and applicability for their needs.

System design should consider RTS performance and when possible, make data interoperable. Seek to strike-a-balance between RTS-performance and data interoperability and prioritize which may be more important. Reliance on CDM principles, (which should be understood at early in the project) are even more important for RTS. Therefore, system design should also consider how the data will be presented (e.g., displayed) to the data consumer. Human factors are important in general; more so for RTS. Carefully consider the efficient display of data that is to be presented in real-time. Integration testing and end-to-end testing using realistic test data should be performed before fielding.

In Pattern A for this API principle, the gateway understands native format from RTS. For Pattern B, all applications must understand native format from RTS. In converting native data formats (which may be generally unreadable) into understandable, portable formats, RTS implementation should also exercise balance and consider the system performance results in a positive user experience.

## 9.2.3 Security

### 9.2.3.1 Applicable API Stakeholder Roles

This tailoring covers general API guidance under the topic of security and is provided for developer stakeholders in the roles of *Providers* and *Appliers* discussed earlier in the Roles section (Table 2-2). The following guidance is intended to enable developers to provide effective security solutions for system Gatekeepers.

### 9.2.3.2 Target General API Guidance

The security recommendations discussed in Section 4 should be performed in general, whether they are for RTS APIs or not. RTS, however, presents special challenges: a resource-constrained RTS may struggle to implement these security best practices alongside the primary data-handling functions the RTS performs. The RTS should consider off-loading some of that additional security processing and **use an API gateway** to insulate resource-constrained RTS implementations.

### 9.2.3.3 RTS Considerations

Incorporating security measures from a project’s outset protects the API from unauthorized access and ensures that the consumer data is protected from unauthorized use or manipulation. Designing an API with security compliance in mind is critical; ensuring API security for RTS presents special challenges.

### 9.2.3.4 RTS Tailored Security Recommendations

Project teams should rely on the project’s security guidance to determine what data must be secure. Some RTS use cases may not perform adequately by strictly (and perhaps unnecessarily) making **ALL** data secure or anonymous. Performing consent management unnecessarily on ALL data without first considering the project’s security posture may have a negative impact on RTS performance, limiting its effectiveness to the data consumer.

System design for RTS should consider securing only the necessary data for the data consumers to perform their decision-making goals. Integration testing and end-to-end testing using realistic test data should be performed before fielding.

## 9.2.4 Reusability

### 9.2.4.1 Applicable API Stakeholder Roles

This tailoring is provided for stakeholders in the roles of **Providers** and **Appliers** discussed earlier in the Roles section (Table 2-2). Reusability is the characteristic ease with which Appliers can make use of APIs for current and future applications across the DoD enterprise. Testers also benefit from reusability in that test cases can be reused.

### 9.2.4.2 Target General API Guidance

Reusability is a key element of high-quality APIs. Ensuring API reusability for RTS presents special challenges. Some RTS use cases may not perform adequately if an “API-first” strategy is not emphasized at the project outset. Therefore, ensure an “API-first” strategy is in-place up-

front to promote the desired real-time performance. Approaches to consider include the following:

- Consider design reviews during the design / development life cycle.
- Conduct performance and metric testing and prediction during the design cycle.

Putting these measures in place during the design and development life cycle promotes an environment for a successful RTS project and appropriate RTS performance.

#### **9.2.4.3 RTS Considerations**

In general, fusing the API Provider with either the external application (Pattern B) or RTS (Pattern C) confounds reusability. For example, resource-constrained RTSs may struggle in performing their primary data-handling functions while attempting to implement the best-practices for reusability. In these instances, project teams should consider using an API gateway.

#### **9.2.4.4 RTS Tailored Reusability Recommendations**

As merited by the specific use case, by off-loading some processing using an API gateway can serve to insulate resource-constrained external applications (for Pattern B) or RTSs (for Pattern C). Using a gateway promotes and ensures reusability by decoupling sources and sinks; (refer to Pattern A of Figure 9-3).

### **9.2.5 Reliability**

#### **9.2.5.1 Applicable API Stakeholder Roles**

This tailoring covers general API guidance under the topic of “reliability” and is provided for stakeholders in the roles of *Providers* and *Appliers* discussed earlier in the Roles section (Table 2-2) The guidance is intended to enable developers in to provide reliable API solutions. The Organic System Users, API Users, Gatekeepers, and Testers all benefit from this achieving this goal.

#### **9.2.5.2 Target General API Guidance**

API documentation is a form of *contract* which provides data to the external API users. Throughout the development process, ensure terms of “*the contract*” are met by having review processes in-place and by verifying through testing.

### 9.2.5.3 RTS Considerations

Reliability, however, must be considered from both the external user standpoint and that of the organic RTS system. Reliability therefore means considering the API external support and the RTS QoS guarantees.

### 9.2.5.4 RTS Tailored Reliability Recommendations

Key test considerations for reliable RTS operation and API support include: contested environment testing (i.e., simulating conditions to those of a contested environment), “graceful degradation” under load, fault tolerance and failure recovery (resiliency), and integrated-performance (integration-testing); all should be part in ensuring reliability during a comprehensive RTS development cycle. If conflicts are identified in the design or testing, the project team should consider the performance and stakeholder tradeoffs and possibly reassess the design. Testing of RT systems can ensure the reliability of RT systems.

## 9.2.6 RTS Testing

### 9.2.6.1 Applicable API Stakeholder Roles

The importance of testing general systems (i.e., non-RTS) was covered earlier in the “Testing” (Section 7). Tailoring suggested here for RTS is to promote an understanding for all stakeholders of the challenges of RTS and to promote the successful development and implementations of RTS mission systems. API stakeholders for RTS need to be aware of all these viewpoints when developing their enterprise systems. The guidance offered here should aid developers when providing effective solutions for the primary benefit of all API users and stakeholders.

Subsequent testing will verify the developed solutions are provided effectively to the primary API stakeholder: the User/Consumer. As discussed in Section 2.3, the “user-centric” view holds that APIs exist fundamentally to support consumers. For RTS, effective deployed solutions are closely tied to system performance.

### 9.2.6.2 Target General API Guidance

Testing performance for RTS is **crucial** and should be emphasized throughout their development, given the **critical** nature of RT systems.

### 9.2.6.3 RTS Considerations

For RTS that generally involve ***constrained resources*** (e.g., processor memory or speed, input/output bandwidth, the number of interfaces), **testing the performance and time-constraints** much be considered to ensure the RTS provides the quality of service expected. Resources must act, respond or deliver with very high probability within a minimum or

predictable period of time and margin-of-error (i.e., deadlines, latency and/or jitter). In other instances, the RTS must have functionality that appears to update interactivity from a human perspective. Examples include real-time embedded systems like aircraft- or spacecraft flight-software; or their human/computer interfaces.

#### **9.2.6.4 RTS Tailored Testing Recommendations**

Key test considerations include: integrated-performance (integration-testing), fault tolerance (resiliency), “graceful degradation” under load, and failure recovery all should be part of a comprehensive RTS development cycle.

APIs can improve testing by restricting the types of interactions. Note however that the realities of RTS can confound testing. It may be awkward or impossible to fully automate testing end-to-end. For example, if we expect some specialized hardware to emit API calls, that can be difficult to implement into a typical continuous integration pipeline.

### **9.2.7 Scalability**

#### **9.2.7.1 Applicable API Stakeholder Roles**

This tailoring covers general API guidance under the topic of *scalability* and is provided for stakeholders in the roles of *Providers* and *Appliers* discussed earlier in the Roles section (Table 2-2). The guidance helps developers to provide scalable solutions for the primary benefit of all API users/stakeholders.

#### **9.2.7.2 Target General API Guidance and RTS Considerations**

Scalability is a key element of high-quality APIs. Recall that scalability is the ability of a system or application to handle increased workloads or users *without a significant decrease in performance*. Performance of RTS requires specially consideration during their design and implementation. Therefore, API scalability for RTS presents special challenges. Some RTS use cases may not perform adequately by offering or supporting a large number of elements, due to RTS performance expectations. Some RTS systems designs may be challenged by factoring-in the notion of API Scalability options. Therefore, RTS designs should *seek to strike a balance* between RTS performance and API Scalability; *prioritize* which may be more important in the given use case. It is good practice to perform integration testing throughout development and end-to-end testing using realistic test data before deployment.

#### **9.2.7.3 RTS Tailored Scalability Recommendations**

An RTS is usually resource constrained to keep the RTS in the task-initiating role. Then the RTS can manage how much work they perform to satisfy needs rather than burdening a resource

constrained RTS from the outside. Keeping API implementations outside of the RTS means the organic system can scale just like a non-RTS.

### 9.2.8 Sorting

#### 9.2.8.1 Applicable API Stakeholder Roles

This tailoring covers general API guidance under the topic of data sorting and is provided for stakeholders in the roles of **Providers** and **Appliers** discussed earlier in the Roles section (Table 2-2). The guidance helps developers to provide effective data sorting solutions for the primary benefit of the Organic System Users (i.e., the organic system meets the RTS QoS requirements).

#### 9.2.8.2 Target General API Guidance

**Sorting** (considered here to also include the related concepts of “*filtering*” and “*pagination*”) is an important capability that can be achieved through API parameters. Sorting empowers clients to organize and analyze the data according to their requirements.

#### 9.2.8.3 RTS Considerations

Filtering is critical for harnessing the power of APIs. For larger data sets, filtering **can significantly improve performance and efficiency** of request responses, post-result processing, network communications, and system utilization. Some RTS use cases may not perform adequately if returned datasets are large (e.g., sorting a massive dataset could lead to RTS performance issues). **Special attention should therefore be given to performance and efficiency for RTS.**

Pagination is essential when dealing with large result sets, especially in presenting the data to the user. This approach prevents overwhelming the client with a massive amount of data and improves performance by reducing the payload size.

Organic system designers (i.e., the Providers) should consider whether data sets need to be sorted. If the data sets are reasonably sized and sorting will not observably affect RTS performance, then the organic system should proceed with sorting the data. If the data sets are massive and sorting would significantly affect RTS performance, consider off-loading the sort-function (so it is not strictly performed in real-time) to a gateway; provide the data unsorted, letting the external application side handle; design the RTS database with an intrinsically sorted structure; or considering increasing hardware performance to adequately provide the desired data sort of the large data set.

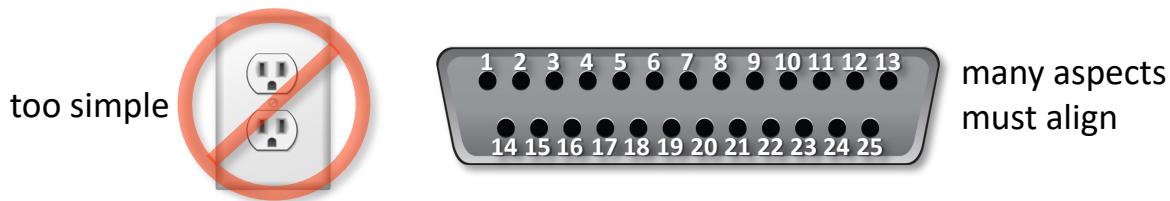
#### **9.2.8.4 RTS Tailored Sorting Recommendations**

Insulating the RTS with an API gateway decouples it from concerns about sorting, filtering, and pagination.

Integration testing and end-to-end testing using realistic and appropriately sized test data should be performed before fielding.

## 10 Data Standards

Data standards are essential for API interoperability. Standards help define the interoperability match between the consumer that invokes the API, and the provider that responds through the API. There are many aspects to this interoperability match, and the consumer and provider must be aligned on every aspect – employing either the same solution, or a satisfactory transformation or mediation. An old-fashioned parallel printer plug is a good analogy: One bent pin and your printer does not work (Figure 10-1).



**Figure 10-1. Many Aspects of Interoperability: An Analogy**

Data standards for API interoperability may be usefully divided into *plumbing data standards* and *content data standards*.

- A plumbing standard is an interoperability solution that works the same regardless of the data's meaning. For example, HTTP is a plumbing data standard; it works the same for target lists and Sunday school attendance.
- A content standard defines the data needed to inform an operational decision. For example, the Airspace Control Order (ACO) message is a content standard; it specifies the many facts that must be shared to deconflict airspace usage.

This distinction is useful in part because it determines the proper source of data standards for API interoperability. Industry experts are often the most competent source to define plumbing standards. These standards should be evaluated for their ability to meet DoD requirements and chosen accordingly. Content standards are essentially the exclusive purview of the DoD, to ensure the user is supported and the mission may be executed. If you need a content standard like the Airspace Control Order, you must go to military subject matter experts (SMEs).

### 10.1 Plumbing Data Standards for APIs

Data plumbing may be divided into runtime aspects and build-time aspects:

- Runtime aspects include: networking (TCP, UDP, etc.), data serialization (XML, JSON, ASN.1, etc.), protocol (HTTP, FTP), interaction style (REST, stateful), authentication, authorization, confidentiality, integrity, nonrepudiation

- Build-time aspects include: machine-processable documentation for discovery, human-readable documentation for understanding, an API registry making both available.

Plumbing data standards greatly simplify API interoperability by defining a known, reusable solution for the many aspects of interoperability. However, there is a problem in that there are many to choose from.

There are many aspects of interoperability, and many standard solutions for each aspect, generating very many permutations that can be chosen by the developers of an API provider. The result is something like an N-squared web of pairwise arrangements, in which learning how to use one API does not help with the next.

The obvious approach is to impose a single standard on each aspect of interoperability, but this is not feasible for a large enterprise like the DoD. There is a legitimate need for different choices to support mission requirements. One size will not fit all.

The workable approach is enterprise convergence, relying on consensus-based standards and widely agreed-to formats and/or architectures to reduce interoperability risks. An enterprise convergence point is an interoperability aspect with one or two preferred solutions that will work for almost everyone, at least one of which must be implemented by every producer and consumer, in addition to any other tailored solutions chosen for mission-specific requirements. Candidate convergence point standards for data plumbing are a small subset of the many standards found in the DoD IT Standards Registry (DISR). These could include:

- Networking: TCP/IP
- Network protocol: HTTPS
- Data serialization: XML or JSON
- API protocol: OpenAPI or OpenData

Given these convergence points, an API provider would be required to implement one version of each API using HTTPS, either XML or JSON, and either OpenAPI or OpenData. The provider would be free to implement other versions of an API as needed for mission-specific requirements – perhaps using ASN.1 serialization over UDP networking. These and other example convergence point plumbing standards are described below, citing the versions mandated in the DISR baseline as of the date of this publication. Developers should reference the DISR for the versions mandated at the time of development.

- **OpenAPI Specification (OAS)** – OAS is a mandated open standard that describes RESTful APIs, supports the REST API framework, and is typically written in YAML or JSON. It defines structure and syntax in a way that is programming language agnostic,

allowing developers to quickly identify and understand the service capabilities (For more information on OAS, see the [OpenAPI Initiative](https://www.openapis.org) (<https://www.openapis.org>)).

- DISR Reference
  - OpenAPI Specification v3.1.0
- Previously known as Swagger
- Written for HTTP APIs
- **Simple Object Access Protocol (SOAP)** is an application communication protocol predating REST that is language, platform, and transport independent.
  - Messages must be encoded in XML
  - Transport is available in HTTP, SMTP, UDP and more.
  - **DISR References:**
    - SOAP Message Transmission Optimization Mechanism, W3C Recommendation, 25 January 2005
    - SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), W3C Recommendation 27 April 2007
    - SOAP 1.2 Part 2: Adjuncts (Second Edition), W3C Recommendation 27 April 2007
    - Web Services Addressing 1.0 - SOAP Binding, W3C Recommendation 9 May 2006
    - OGC Web Coverage Service 2.0 Interface Standard - XML/SOAP Protocol Binding Extension, v.1.0.0, 2010-10-27
- **Open Data Protocol (ODATA)**
  - OData is a specific implementation of REST that helps programs focus on the business logic when building Restful APIs (DISR)
  - Data can be represented in JSON or in the XML-based Atom/ AtomPub
  - **DISR Reference:**
    - OData Version 4.01. Part 1: Protocol, OASIS Standard, 23 April 2020

One emerging technology **NOT** currently listed in the DISR is a **Graph Database**, which provides an alternative to REST and SOAP. While a Graph Database may provide simplicity, it is not approved as it carries its own unique security vulnerabilities, in addition to common security threats:

- **GraphQL** is a language for APIs that collects all the data needed for a single request, not just by accessing the properties of one resource but also by following references in between them ([GraphQL | A query language for your API](#)).
  - Supports use cases with low-bandwidth
- **SPARQL** is a Resource Description Framework (RDF) [reference link] family of query languages capable of accessing native RDF databases or RDF middleware.

While APIs primarily focus on communication protocols, request-response patterns and authentication, data standards primarily focus on data representation, structure, and semantics. Semantics are defined as the philosophical and scientific study of meaning in natural and artificial languages ([Semantics | Definition & Theories | Britannica](#)).

Data standards apply to the content and format of the data itself. Data is typically exchanged, stored, or analyzed. When designing an API, one crucial aspect is choosing a standardized data format for responses. Two common formats are XML eXtensible Markup Language (XML) and JavaScript Object Notation (JSON).

- **XML:**
  - XML is a machine-readable markup language that is more verbose than JSON but is still prevalent in the Defense domain.
  - To conform to the XML data standard:
    - Define an XML schema (XSD) that specifies the structure, data types, and constraints.
    - Ensure that API responses adhere to the defined schema.
- **JSON:**
  - JSON is lightweight, human and machine-readable, and widely used for APIs. It represents data as key-value pairs and nested structures.
    - To conform to the JSON data standard:
      - Ensure that API responses follow the correct JSON syntax.
      - Use consistent field names and data types.
      - Validate JSON data against a schema (such as JSON Schema) to ensure correctness.

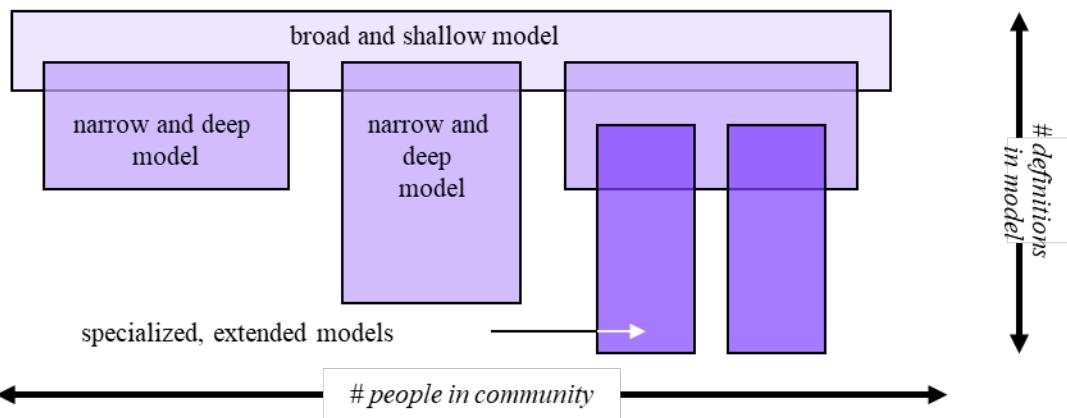
## 10.2 Content Data Standards for APIs

There are no industry standards for the data required to inform operational decisions. These content standards must be defined by military SMEs. (These standards may *employ* industry standards, as described below; they cannot be *replaced by* industry standards.)

Some of this content is *machine-to-machine* data, intended for automated processing, and never examined by a human. The software passing this data through an API cannot work correctly unless the developers have a compatible understanding of the data that passes through. The developers obtain this understanding from the content data standard that defines the API's data payload.

This necessary shared understanding does not mean that all developers understand all data. A single content data standard covering all information requirements and understood by all participants is never feasible in a large enterprise.

The optimum approach for DoD content standards is a hierarchy of data models: Small models that are understood by large communities on top, reused and extended as needed below by larger models that are understood by smaller communities. This approach divides knowledge about data into the communities that require it, producing the most value at the smallest cost of learning (Figure 10-2).



Source: (Renner 2005)

**Figure 10-2. Hierarchy of Data Models for Shared Understanding**

Examples of content data standards following this approach include:

- National Information Exchange Model (NIEM) is a standards-based framework for defining any number of content data standards. It is free and open-source, managed by OASIS under its NIEMOpen project.
  - NIEM offers a core data model, extended by several domain models, and further extended as needed for any particular content data standard. NIEM has a process for harmonizing the differences in these models, when the benefit exceeds the cost.
  - NIEM-conforming data specifications are machine-readable and support automated data validation. NIEM-conforming instance data can be translated between any supported data format; at present, XML, JSON, and RDF.

- NIEM-based data models have an ontology representation (in RDFS, OWL, and SHACL). NIEM-based instance data in RDF is a knowledge graph.
- Universal Command and Control (UC2) harmonizes several military content data standards into a common language for API payloads. UC2 uses the NIEM framework to specify the content and meaning of those payloads. (UC2 also offers a data plumbing stack containing many candidates for enterprise convergence points.)
- USMTF (MIL-STD-6040) is a content data standard defining a large number of data payloads for operational data exchange requirements. The latest version of USMTF uses the NIEM framework to specify those payloads.
- NATO Core Data Framework (NCDF) is another standards-based framework for defining content data standards. It employs the NIEM technical architecture with its own reusable core and domain data models.
- Military or Defense Standard (U.S. MIL-STD or Allied Tactical Data Link Publication (ATDLP) in NATO): a document that establishes uniform engineering and technical requirements for military-unique or substantially modified commercial processes, procedures, practices, and methods (DAU 2024). The ASSIST database, hosted by the Defense Logistics Agency (<https://assist.dla.mil/>), provides a repository of U.S. military and NATO standards and serves as the official source for specifications and standards used by the DoD.
  - According to DAU, defense standards include the following types:
    - Data Standards
    - Interface Standards
    - Design Standards
    - Manufacturing Process Standard
    - Test Method Standards
  - Some standards overlap several categories. For example, MIL-STD-6016G, "Tactical Data Link (TDL) 16 Message Standard," not only describes the message sets, but also defines the message and transmission/ receipt protocols, format of the message, system implementation specifications, and a common data element dictionary (DISR Record/ Applicability column). NATO splits these types of technical standards into two parts. For example, STANAG 5516 and ATDLP 7.0 documents into 2 parts. For example, STANAG 5516 and ATDLP 7.04 combined are the NATO standards for Link 16.

- The TDL community is developing Appendices to some of the TDL standards, that includes a logical XML representation of the standard that can be used for providing situational awareness but does not include all the rules for transmit receive rules or implementation requirements.
- EXI – Efficient XML Interchange provides a new approach to data optimization through combining Information Theory with Formal Language Theory to provide a highly efficient XML integration capability.
  - Compared with Tactical Data Format, XML packets alone are 10x-100x larger, while EXI is nearly on par, and in some cases significantly more efficient than Tactical Data Format.
  - Optimizes open source and COTS so they can be used everywhere.
  - XML alone typically encounters performance and efficiency problems in support of DDIL environments.

To have a coherent API ecosystem that promotes interoperability, DoD must adhere to standards for both the development of APIs as well as the response messages that supports their understanding.

## 11 Conclusion

This document has underscored the significance of existing APIs in supporting improved interoperability in DoD systems. APIs are essential to interoperability, facilitating data sharing and integration between diverse systems and applications. As technology evolves, so will APIs, bringing enhanced capability and functionality. Embracing these future technologies will not only enhance the DoD's agility, efficiency, and effectiveness but also will empower U.S. warfighters with the most advanced tools and information to succeed in their missions.

Adopting APIs will be challenging. A concerted effort of clear communications, comprehensive training, and active stakeholder engagement will help the Department to overcome cultural and other challenges for enhanced joint mission interoperability. Success may require changes to organizational constructs, acquisition processes, and the acquisition pathways. It is vital to involve all stakeholders, from leadership to developers and end-users. A culture of openness and collaboration with a focus on data standards will help ensure programs develop future systems equipped to leverage APIs for enhanced interoperability. With the above in mind, the DoD is investigating expansion of current DAU training to include APIs and these challenges.

While enabling access to sensitive data and functionalities, APIs can be potential targets for malicious actors. The DoD is prioritizing robust security measures, such as authentication, authorization, encryption, and continuous monitoring, to safeguard the confidentiality, integrity, and availability of its APIs. Privacy considerations data anonymization and consent management with regard to warfighter and non-warfighter personal information should be integral to API design and implementation to protect this vital information.

This document underscores the need for the DoD to proactively embrace future technologies while effectively managing the cultural shift required for their adoption. By recognizing the potential of future technologies, implementing robust change management strategies, and prioritizing security and privacy, the DoD can successfully adopt APIs. This effort will not only enhance operational capabilities but also will help ensure warfighters and programs developing future systems are equipped with the most advanced and interoperable tools to achieve their mission objectives.

## Appendix A: API Project Governance Considerations

The API project team should consider the following factors (Table A-1) when designing or updating an API framework, ecosystem, model or standard for DoD use.

**Table A-1. API Project Governance Considerations**

Factor	Consideration
<b>API Strategy</b>	The API project should communicate its approach to interoperability before starting system development. The intent of this document is to establish a vision and initial plans to facilitate understanding by stakeholder. This includes envisioned near-term capability and possible evolutions for the future. This would also include a definition of scope of the API, describing what would be considered valid solution space for the API now and in the future. This strategy should be updated as the program evolves.
<b>Use Cases</b>	The API project should provide comprehensive use case descriptions and diagrams of how the API is intended to be used. This allows the potential consumers to validate if the API meets the intended requirements. This may not preclude them from using it in new ways but is another way to describe how the API structure came to be what it is and more quickly understand its design. This also describes the problem space chosen for the API in the strategy and allows current and future stakeholders a way to communicate new and unforeseen needs.
<b>API Contracts</b>	The API project should describe Service Life Agreement (SLA) rules that should be followed for use of the API including Non-Functional Requirements (NFRs). The API contract should also provide API usage description information including inputs and outputs (Sindall 2020).
<b>Hosting</b>	The API project should describe how the API services are to be hosted and their required reliability.
<b>API Project Performance and Design</b>	The API project should describe what performance criteria the API functions and responses must meet.
<b>API Service Registry and Discovery</b>	The API project should describe how the API service discovery process will work. This will include a discussion of service catalog, registry categorization, and interaction styles (e.g., REST, stateful).
<b>Scalability</b>	The API project should describe scalability requirements.
<b>Transport</b>	The API project should describe what type of transport services the API will use. These descriptions include the required network communication frameworks (e.g., HTTPs, FTPs); data serialization (e.g., XML, JSON, ASN.1, etc.); and network confidentiality, integrity, and nonrepudiation approaches.

Factor	Consideration
<b>Security</b>	The API project should describe the general security framework in which the API resides and what parts of the security framework need to be instituted. This at least includes the required authentication, authorization, and “need to know” checks.
<b>Design</b>	The API project should describe how stakeholder’s inputs are taken into account for the project during the initial and follow-up updates. Also describe the API design patterns, caching requirements, data retrieval function, and data semantics. Information models used and fault tolerant flows (Sindall 2020).
<b>Quality Reviews</b>	The API project should describe how API ecosystem design and implementation quality reviews will be done to ensure the API meets the API standard and has the desired attributes. Ensure governance rules are met before deployment (Sindall 2023).
<b>Testing</b>	The API project plan should describe how the API is tested before being deployed as well as the processes used to quickly and consistently test updates via automation to the API. This may include automated governance checks (Sindall 2020).
<b>Deployment</b>	The API project should describe the deployment approach such as API Library versus several libraries/formats. Also, describe the registration, setup, configuration sequences that allow quick onboarding.
<b>Feedback</b>	The API project should describe how stakeholders and developers can provide improvements, suggestions, report issues, and raise concerns about API not performing as documented.
<b>Versioning and Backward Compatibility</b>	<p>Establishing a versioning and backward compatibility plan is critical for ensuring changes to the APIs do not break existing integrations. Implementing a comprehensive versioning strategy from the outset provides a clear upgrade path for API consumers and avoids any workflow disruptions.</p> <p>One key consideration is the use of semantic versioning, a widely adopted three-part version numbering scheme (i.e., major.minor.patch). Major version changes indicate significant changes that may break backward compatibility. Minor version changes indicate new features or functionality that are backward compatible. Patch version changes indicate bug fixes or minor updates that are fully backward compatible. Use of this scheme communicates API changes and provides predictable upgrade paths for users. Thus, existing integrations will continue to function as expected, while also allowing for the introduction of new features and functionality.</p> <p>Another key consideration is planning for backward compatibility. Backward compatibility ensures that existing integrations continue to function correctly, even when changes are made to the API. This can be achieved by maintaining existing endpoints, providing fallback mechanisms, or implementing versioning strategies that allow for multiple versions of the API to coexist.</p> <p>Planning for both versioning and backward compatibility from design to implementation ensures that the API remains stable, reliable, and functional over time. This builds trust and confidence with API users and ensures that the API continues to meet their needs and expectations.</p>

<b>Factor</b>	<b>Consideration</b>
<b>Updating</b>	The API project should describe how an API framework is updated and issue any new release of the updated API implementation. This creates a robust API versioning approach (Sindall 2023).
<b>Deprecating</b>	The API project should describe how the API should handle planning and timing of removing aspects of the API that are no longer desired or have been replaced by improved functionality.
<b>Tracking Use</b>	The API project should describe how the project can track use of the API, if applicable. Where applicable this can help create new use case and business case support for the changes.
<b>Telemetry</b>	For further information on metrics to track performance, scalability, security, and tracking use, see metrics section in Design and Implementation section.

## Appendix B: Common API Vulnerabilities and Threats

The following are some of the most common API vulnerabilities and threats:

- Injection Attacks - Injection attacks occur when an attacker sends malicious input to an API with the intent of executing unauthorized commands or accessing sensitive data. Common types of injection attacks include SQL injection, XML injection, and command injection. Injection attacks can be particularly dangerous in the context of the DoD, as they can be used to gain unauthorized access to sensitive systems and data.
- Cross-Site Scripting (XSS) Attacks - XSS attacks occur when an attacker injects malicious code into a web page or API response, which is then executed by a user's browser. This can allow the attacker to steal sensitive data or perform unauthorized actions on behalf of the user. XSS attacks can be particularly dangerous in the context of the DoD, as they can be used to compromise user accounts and gain access to sensitive systems and data.
- Denial-of-Service (DoS) and Distributed Denial of Service (DDoS) Attacks - DoS attacks occur when an attacker floods an API with requests in an attempt to overwhelm the system and prevent legitimate users from accessing it. DoS and especially DDoS attacks can be particularly damaging in the context of the DoD, as they can disrupt critical systems and services.
- Insufficient Authentication and Authorization - Insufficient authentication and authorization can occur when an API does not properly verify the identity of users or restrict access to sensitive data and systems. This can allow unauthorized users to access sensitive data and systems, potentially leading to data breaches and other security incidents.
- Insecure Data Storage - Insecure data storage can occur when an API stores sensitive data in an unencrypted or otherwise insecure manner. This can allow attackers to steal sensitive data, such as passwords and other credentials, and use it to gain unauthorized access to systems and data.
- XML External Entity (XXE) Attacks - XXE (XML External Entity) attacks are a type of injection attack that can be used to exploit vulnerabilities in APIs that process XML data. XXE attacks are a significant API vulnerability and threat because they can be used to gain unauthorized access to sensitive data and systems. In the context of the DoD, XXE attacks can be particularly dangerous as they can be used to compromise critical systems and services. To mitigate the risk of XXE attacks, it is important to implement strong API security measures, such as input validation to prevent injection attacks, and to use secure XML parsers that are not vulnerable to XXE attacks.

- Insecure Data Transmission - Unsecure data transmission is an API vulnerability and threat that occurs when data is transmitted over a network in an unencrypted or otherwise insecure manner. This can allow attackers to intercept and read sensitive data, such as passwords and other credentials, and use it to gain unauthorized access to systems and data. To mitigate the risk of unsecure data transmission, it is important to use strong encryption mechanisms, such as TLS (Transport Layer Security), to protect data in transit.
- Improper Access Controls and Authorization Flaws - Improper access controls and authorization flaws are an API vulnerability and threat that occur when an API does not properly restrict access to sensitive data or functionality. This can allow attackers to gain unauthorized access to systems and data, potentially leading to data breaches and other security incidents. To mitigate the risk of improper access controls and authorization flaws, it is important to implement strong access controls and authorization mechanisms, such as role-based access control (RBAC) and attribute-based access control (ABAC).
- Security Misconfigurations and Improper Error Handling - Security misconfigurations and improper error handling are an API vulnerability and threat that occur when an API is not properly configured or when errors are not handled in a secure manner. This can allow attackers to exploit vulnerabilities in the API and gain unauthorized access to systems and data. To mitigate the risk of security misconfigurations and improper error handling, it is important to implement strong security configurations and to properly handle errors in a secure manner.
- Insider Threats and Unauthorized Access - Insider threats and unauthorized access are API vulnerabilities and threats that occur when individuals with authorized access to an API misuse their privileges or when unauthorized individuals gain access to the API. This can lead to unauthorized disclosure, modification, or destruction of sensitive data, as well as disruption of services. To mitigate the risk of insider threats and unauthorized access, it is important to implement strong access controls, such as RBAC and least privilege principles. Regular monitoring and auditing of API activities can also help detect and prevent unauthorized access. In addition, implementing strong authentication mechanisms, such as multi-factor authentication (MFA), can further enhance security and protect against unauthorized access.

## Appendix C: API Security Challenges

### C.1 Injection Attacks and Their Impact on Mission-Critical Systems

Injection attacks pose a significant threat to mission-critical systems within the DoD context. These attacks involve the introduction of malicious data or code into a system, exploiting vulnerabilities to manipulate system behavior, compromise data integrity, or gain unauthorized access. In the DoD context, where mission-critical systems are integral to intelligence, command and control of military forces, weapons systems, and fulfilling military requirements, the impact of injection attacks can be severe. They can disrupt operations, endanger operator safety, compromise sensitive information, and potentially jeopardize national security. The DoD's cybersecurity initiatives aim to mitigate such threats through secure coding practices, automated security testing, and continuous monitoring; however, the evolving nature of injection attacks and the complexity of DoD systems present ongoing challenges.

### C.2 Authentication and Authorization Issues in a Multi-Domain Environment

Authentication and authorization in a multi-domain environment within the DoD context present unique security challenges. Authentication, the process of verifying the identity of a user, device, or system, and authorization, the process of granting or denying access rights to resources, are critical for maintaining the security and integrity of DoD systems. In a multi-domain environment, where resources and users are distributed across various domains, ensuring consistent and secure authentication and authorization becomes complex. This complexity can lead to potential vulnerabilities, such as unauthorized access or privilege escalation. The DoD addresses these challenges through robust MFA, RBAC, and ABAC mechanisms, along with continuous monitoring and auditing; however, the dynamic nature of multi-domain environments and the evolving threat landscape continue to pose significant challenges.

### C.3 Data Breaches and Protection of Sensitive Information

Data breaches and the protection of sensitive information are significant security challenges within the DoD context. The DoD manages vast amounts of sensitive data, including classified military information, personnel records, and intelligence data. Data breaches can lead to the exposure of this sensitive information, with potential impacts on national security, operational effectiveness, and the privacy of personnel. The DoD has experienced significant data breaches in the past, highlighting the importance of robust data protection measures. These measures include data encryption, secure data handling practices, and continuous monitoring for potential threats. However, the complexity of the DoD's information systems, the sophistication of adversaries, and the evolving nature of threats continue to pose challenges to the protection of sensitive information within the DoD.

#### C.4 Service Discovery Threats

The service discovery threats outlined in the “Service Discovery Threat Model for AD HOC Networks” by Adrian Leung and Chris Mitchell highlight security challenges within the DoD context. Ad hoc networks, which are dynamic and vulnerable, present unique security and privacy challenges. Service discovery, the process of finding and connecting to available services, is particularly susceptible to threats in these networks. The DoD relies on secure and reliable service discovery mechanisms to ensure the availability and integrity of critical services. However, the dynamic nature of ad hoc networks and the potential for malicious actors to exploit vulnerabilities in service discovery protocols pose significant challenges to the DoD’s ability to maintain secure and resilient communication and information exchange. Implementing robust security measures, such as encryption, authentication, and intrusion detection systems, is crucial to mitigating these threats and ensuring the security of DoD operations in ad hoc network environments (Leung and Mitchell 2023).

Other related threats include:

- Service Spoofing
- Passive Listening
- Data Alteration

#### C.5 Cascading Failure

Cascading failure, a phenomenon where the failure of one component triggers a chain reaction of failures in interconnected systems, poses significant cybersecurity challenges within the DoD. In the context of API cybersecurity threats, cascading failures can occur when a vulnerable API is exploited, leading to the compromise of other interconnected APIs or systems. This can have severe consequences for the DoD, as it relies on a complex network of interconnected systems and APIs to support critical operations. The potential for cascading failures highlights the importance of implementing robust security measures, such as secure coding practices, vulnerability scanning, and continuous monitoring, to prevent and mitigate the impact of API cybersecurity threats and ensure the resilience and integrity of DoD systems.

## Glossary

Unless otherwise noted, the following definitions are suggested by the authors of this guide for their relevance to APIs. The definitions are not intended to be authoritative in all contexts.

Term	Description	Source
abstraction	The process of simplifying complex systems or concepts by focusing on essential features while hiding unnecessary details.	<a href="https://www.imedpub.com/articles/abstraction-simplifying-complexity-in-software-engineering.php?aid=50800#:~:text=Abstraction%20is%20a%20fundamental%20concept%20in%20software%20engineering%20that%20helps,an%20facilitating%20scalability%20and%20maintainability">https://www.imedpub.com/articles/abstraction-simplifying-complexity-in-software-engineering.php?aid=50800#:~:text=Abstraction%20is%20a%20fundamental%20concept%20in%20software%20engineering%20that%20helps,an%20facilitating%20scalability%20and%20maintainability</a>
academia	The world of education and research, typically associated with universities and scholarly activities.	
access controls	Security measures and policies that determine who is allowed to access or modify certain resources or data.	
acquisition	The process of obtaining or procuring something, often used in the context of acquiring assets or technology.	
Adaptive Acquisition Framework	A set of acquisition pathways to enable the workforce to tailor strategies to deliver better solutions faster.	<a href="https://aaf.dau.edu/">https://aaf.dau.edu/</a>
acquisition pathways	The various routes or methods used to obtain resources or technology, typically within a business or organizational context.	
acquisition processes	The procedures and steps involved in acquiring resources, technology, or assets, often including planning, procurement, and implementation.	
algorithm	A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation.	<a href="https://www.merriam-webster.com/dictionary/algorithm">https://www.merriam-webster.com/dictionary/algorithm</a>
application program	A software system implemented to satisfy a particular set of requirements.	(NIST SP 1800-21)

## Glossary

Term	Description	Source
application programming interface (API)	A structured, software level, data transport supported mechanism for system developers to discover and for programmed systems to request and send/receive information and functionality from the same or other systems.	
API discovery service	A service allowing application developers (the Appliers in the API context) to find existing API artifacts and reuse them in future designs. API discovery methods include: service catalogs, registry categorization, Interface Control Documents, and interaction styles (e.g., REST, stateful). Contrast with Data Discovery.	
API gateway	A data-plane entry point for API calls that represent client requests to target applications and services. It typically performs request processing based on defined policies, including authentication, authorization, access control, SSL/TLS (Secure Sockets Layer/Transport Layer Security) offloading, routing, and load balancing.	<a href="https://www.nginx.com/learn/api-gateway/#:~:text=An%20API%20gateway%20is%20a%20data%2Dplane%20entry%20point%20for,%2C%20routing%2C%20and%20load%20balancing">https://www.nginx.com/learn/api-gateway/#:~:text=An%20API%20gateway%20is%20a%20data%2Dplane%20entry%20point%20for,%2C%20routing%2C%20and%20load%20balancing</a>
API-first	Prioritizing the APIs that support your application and focusing on the value they can deliver to your business, rather than just scrambling to deliver a single application and creating an API as an afterthought.	<a href="https://www.postman.com/api-first/#:~:text=Being%20API%2Dfirst%20means%20prioritizing,an%20API%20as%20an%20afterthought">https://www.postman.com/api-first/#:~:text=Being%20API%2Dfirst%20means%20prioritizing,an%20API%20as%20an%20afterthought</a>
application aware health checking	An API monitoring method that checks your API and alerts you when it notices something is amiss. A diagnostic tool for your code base that can help you find problems before they become more significant headaches than they need to be.	<a href="https://testfully.io/blog/api-health-check-monitoring/">https://testfully.io/blog/api-health-check-monitoring/</a>
architects	Professionals responsible for designing the overall structure and organization of software systems or IT infrastructure.	
artificial intelligence (AI)	A machine's ability to perform the cognitive functions usually associated with human minds.	<a href="https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-ai">https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-ai</a>
attack detection	Detecting suspicious API traffic. Attack detection is of utmost importance in today's digital landscape. With the increasing reliance on APIs for data exchange between different applications and systems, it has become crucial to ensure the security and integrity of these interactions.	<a href="https://nonamesecurity.com/blog/how-to-detect-suspicious-api-traffic/">https://nonamesecurity.com/blog/how-to-detect-suspicious-api-traffic/</a>

## Glossary

Term	Description	Source
authentication	Carefully and comprehensively identifying all related users and devices. Typically requires client-side applications to include a token in the API call so the service can validate the client.	<a href="https://www.techtarget.com/searchapparchitecture/tip/10-API-security-guidelines-and-best-practices">https://www.techtarget.com/searchapparchitecture/tip/10-API-security-guidelines-and-best-practices</a>
backward compatibility	The ability of a RESTful API to handle requests from the clients that use an older version of the API without breaking or returning errors. Reduces the friction and cost for the clients to upgrade to the new version of the API. Also preserves the trust and reliability of the web service, as the clients can expect the API to work as intended.	<a href="https://www.linkedin.com/advice/0/how-do-you-design-restful-api-supports#:~:text=Backward%20compatibility%20is%20the%20ability,new%20version%20of%20the%20API">https://www.linkedin.com/advice/0/how-do-you-design-restful-api-supports#:~:text=Backward%20compatibility%20is%20the%20ability,new%20version%20of%20the%20API</a>
canary release	A technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.	(Sato 2014) <a href="https://martinfowler.com/bliki/CanaryRelease.html">https://martinfowler.com/bliki/CanaryRelease.html</a>
circuit breakers	Wrapping a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. Usually involves a monitor alert if the circuit breaker trips.	(Sato 2014) <a href="https://martinfowler.com/bliki/CircuitBreaker.html">https://martinfowler.com/bliki/CircuitBreaker.html</a>
code review	The act of consciously and systematically convening with one's fellow programmers to check each other's code for mistakes; shown to accelerate and streamline the process of software development as few other practices can.	<a href="https://smartbear.com/learn/code-review/what-is-code-review/">https://smartbear.com/learn/code-review/what-is-code-review/</a>
common data model (CDM)	Contains a uniform set of metadata, allowing data and its meaning to be shared across applications. In addition to the uniform metadata, a CDM includes a set of standardized, extensible data schemas that include items such as entities, attributes, semantic metadata, and relationships. Once all the elements of the CDM are defined, methods to access and operate on the data are developed so all applications can use these same, standardized procedures.	<a href="https://www.synopsys.com/glossary/what-is-common-data-model.html">https://www.synopsys.com/glossary/what-is-common-data-model.html</a>
communication frameworks	Software libraries or protocols that facilitate communication and data exchange between software components or systems.	
confidentiality	The principle of protecting sensitive or confidential information from unauthorized access or disclosure.	

## Glossary

Term	Description	Source
Conway's Law	Essentially the observation that the architectures of software systems look remarkably similar to the organization of the development team that built it.	<a href="https://martinfowler.com/bliki/ConwaysLaw.html">https://martinfowler.com/bliki/ConwaysLaw.html</a>
cross-site scripting (XSS)	A type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.	<a href="https://owasp.org/www-community/attacks/xss/">https://owasp.org/www-community/attacks/xss/</a>
cross-site request forgery (CSRF)	An attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated.	<a href="https://owasp.org/www-community/attacks/csrf">https://owasp.org/www-community/attacks/csrf</a>
(Agile) culture	At an enterprise level, moving strategy, structure, processes, people, and technology toward a new operating model by rebuilding an organization around hundreds of self-steering, high-performing teams supported by a stable backbone.	<a href="https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/doing-vs-being-practical-lessons-on-building-an-agile-culture">https://www.mckinsey.com/capabilities/people-and-organizational-performance/our-insights/doing-vs-being-practical-lessons-on-building-an-agile-culture</a>
cybersecurity (API)	Strategies and solutions to understand and mitigate the vulnerabilities and security risks of Application Programming Interfaces (APIs). APIs are a critical part of modern mobile, software as a service (SaaS), and web applications and can be found in customer-facing, partner-facing and internal applications. By nature, APIs expose application logic and sensitive data such as Personally Identifiable Information (PII) and because of this have become a target for attackers. Without secure APIs, rapid innovation would be impossible.	<a href="https://owasp.org/www-project-api-security/">https://owasp.org/www-project-api-security/</a>
data	Information stored, transmitted, or received in an electronic format. It can take on many purposes, including to inform and control. Data may be described to exist at the application software level and all the way down to hardware signal level. At the highest levels, it can be considered to have major flows in one direction, but at lower levels it may flow in two directions to support the major flows.	
data anonymization	The process of removing particular pieces of private information that could be used to identify a person in data.	<a href="https://www.splunk.com/en_us/blog/learn/data-anonymization.html">https://www.splunk.com/en_us/blog/learn/data-anonymization.html</a>
data discovery	The process of users (the API Users in the API context) finding data stores, databases, and applications that will help them make decisions or control system functionality. Contrast this with API Service Discovery.	

## Glossary

Term	Description	Source
data fabric	An architecture and set of data services that provide consistent capabilities across a choice of endpoints spanning hybrid multicloud environments. A powerful architecture that standardizes data management practices and practicalities across cloud, on premises, and edge devices.	<a href="https://www.netapp.com/data-fabric/what-is-data-fabric/">https://www.netapp.com/data-fabric/what-is-data-fabric/</a>
data mesh	Principles to help address changes in the data landscape and speed of response to change: (1) domain-oriented decentralized data ownership and architecture, (2) data as a product, (3) self-serve data infrastructure as a platform, and (4) federated computational governance.	<a href="https://martinfowler.com/articles/data-mesh-principles.html">https://martinfowler.com/articles/data-mesh-principles.html</a>
data semantics	The meaning and interpretation of data, often defined through metadata and ontologies.	
data serialization	The process of converting structured data into a format suitable for transmission or storage, such as JSON or XML.	
data visualization	The presentation of data in graphical or visual formats to facilitate understanding and analysis.	
data-centric	Focusing on the data itself as the central element in system design and decision making.	
deadline	Deadline is a finite window of time in which a certain task must be completed. Deadlines may be: <ul style="list-style-type: none"><li>• Hard – can't miss deadline</li><li>• Soft – may miss deadline</li></ul> Firm – no gain when miss deadline	
deprecating	Phasing out or marking a software feature or API as obsolete, typically with the intention of removing it in future versions.	
deterministic	An application is <b>deterministic</b> if its timing can be guaranteed within a certain margin of error. Deterministic behavior provides a measure of reliability that the communications output will not only be correct but will happen in a specified time.	

## Glossary

Term	Description	Source
developmental test and validation	The process of verifying whether the specific requirements to test development stages are fulfilled, based on solid evidence. In particular, test validation is an ongoing process of developing an argument that a specific test, its score interpretation, or use is valid.	<a href="https://assess.com/test-validation/#:~:text=Test%20validation%20is%20the%20process,interpretation%20or%20use%20is%20valid">https://assess.com/test-validation/#:~:text=Test%20validation%20is%20the%20process,interpretation%20or%20use%20is%20valid</a>
DevSecOps	A software engineering culture that guides a team to break down silos and unify software development, deployment, security, and operations. Success in adopting DevSecOps requires buy-in from all stakeholders, including: leadership, acquisition, contracting, middle-management, engineering, security, operations, development, and testing teams. Stakeholders across the organization must change their way of thinking from "I" to "we," while breaking team silos, and understanding that the failure to successfully deliver, maintain, and continuously engineer software and its underlying infrastructure is the failure of the entire organization, not one specific team or individual.	<a href="https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Playbook_DoD-CIO_20211019.pdf">https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Playbook_DoD-CIO_20211019.pdf</a>
digital modernization	The process of updating and adapting an organization's digital systems and technologies to meet current and future needs.	
ecosystem	A community of interconnected components or entities that interact and influence each other, often used in the context of software or technology.	
emerging technology	New and cutting-edge technologies that have the potential to disrupt existing industries or create new opportunities.	
encoding attack	An attack in which malicious data or scripts are embedded in input to exploit vulnerabilities in a system.	
encryption	The process of converting data into a secure and unreadable format to protect it from unauthorized access.	
enterprise standards	Established guidelines, protocols, and best practices that an organization follows to ensure consistency and interoperability across its systems.	
error handling	The process of identifying, reporting, and managing errors or exceptions in software to ensure robustness and graceful degradation.	

## Glossary

Term	Description	Source
experimentation	The systematic testing and exploration of new ideas, features, or solutions to gather data and make informed decisions.	
fault tolerant	The ability of a system to continue functioning or provide degraded service in the presence of faults or failures.	
federated system	A federated system refers to a collection of interconnected but autonomous systems or components that work together to achieve a common goal. These systems maintain their independence while sharing data and resources as needed.	<a href="https://securiti.ai/glossary/federated-system/">https://securiti.ai/glossary/federated-system/</a>
filtering	The process of selecting or extracting specific data or information from a larger data set based on predefined criteria.	
firewall	A network security device or software that monitors and controls incoming and outgoing network traffic to protect against unauthorized access or threats.	
frameworks	Predefined structures and libraries that provide a foundation for building software applications.	
functional testing	Testing that focuses on verifying that the software functions according to specified requirements and performs its intended tasks.	
fuzz testing	Also called fuzzing, an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities. A fuzzing tool injects these inputs into the system and then monitors for exceptions such as crashes or information leakage. Fuzzing introduces unexpected inputs into a system and watches to see if the system has any negative reactions to the inputs that indicate security, performance, or quality gaps or issues.	<a href="https://www.synopsys.com/glossary/what-is-fuzz-testing.html">https://www.synopsys.com/glossary/what-is-fuzz-testing.html</a>
General External Consumer Planning (GECP)	The planning process performed by organic system programs and their Provider stakeholders to research and plan for what data will be made available to the Applier, API User and Tester stakeholders, how that will be delivered, and how it will be configuration managed. The process includes considering the API Users mission critical data. Practical ways for providing the Tester's unique data needs and CONOP will also be considered and planned. If the tester needs slightly exceed the API User needs, an adjustment to the API may be practical solution.	

## Glossary

Term	Description	Source
	Otherwise, other more direct data extraction, telemetry or data source removal means may more practical. The CONOP can also depend on the data layer available within the organic system where the lower layers may be better accessible via specific tools and means adapted to the lower layer. A GECP document or appendix may be provided in future releases of this API Guidance document.	
(API) governance	The processes and controls implemented to manage, monitor, and maintain APIs (application programming interfaces). It involves defining standards, policies, and guidelines for API design, development, deployment, and usage. The goal of API governance is to ensure consistency, security, scalability, and reliability across all API.	<a href="https://nonamesecurity.com/learn/what-is-api-governance/">https://nonamesecurity.com/learn/what-is-api-governance/</a>
Hypertext Transfer Protocol (HTTP)	A fundamental protocol used for communication on the World Wide Web. It defines the rules and conventions for transferring text, images, videos, and other resources between web servers and web clients (typically web browsers). HTTP operates as a request-response protocol, in which a client (usually a web browser) sends an HTTP request to a web server, and the server responds with the requested content or an error message.	
HTTP verb tampering	An attack in which an attacker manipulates the HTTP request method (HTTP verb) to perform unauthorized actions on a web application.	
identity and access management	The practices and technologies used to manage and secure digital identities and control access to resources.	
implementation	The process of translating a design or plan into a working system or application.	
industry	A specific sector of economic activity, characterized by similar products, services, and business practices.	
information model	A conceptual representation of data and its relationships within a system or domain.	
injection attack	An attack in which malicious code or input is injected into an application, potentially leading to data breaches or system compromise.	
integration	The process of combining different software systems or components to work together as a unified whole.	

## Glossary

Term	Description	Source
integrity	The assurance that data or resources have not been altered or tampered with in an unauthorized or malicious manner.	
intelligence analytics	The process of collecting, analyzing, and interpreting data to gain insights and make informed decisions in intelligence and security contexts.	
interface	A point of interaction between different software components, allowing them to communicate or exchange data.	
interoperability	The ability of different systems or components to work together and exchange data seamlessly.	
jitter	Jitter is a measure of how much a response (or update) time deviates from the upper limit across multiple iterations of an event.	Collins, D.; 10 July 2019; <a href="https://www.motioncontroltips.com/deterministic-real-time-control-what-does-it-really-mean-in-motion-control-applications/">https://www.motioncontroltips.com/deterministic-real-time-control-what-does-it-really-mean-in-motion-control-applications/</a>
Joint	In a military context, collaborative efforts involving multiple branches or Services.	
landscape	The overall view or context of a situation, often used to describe the broad environment in which a system operates.	
latency	Latency is the amount of time between an event and system's response. Latency should be normalized and low in a real-time system.	Collins, D.; 10 July 2019; <a href="https://www.motioncontroltips.com/deterministic-real-time-control-what-does-it-really-mean-in-motion-control-applications/">https://www.motioncontroltips.com/deterministic-real-time-control-what-does-it-really-mean-in-motion-control-applications/</a>
lateral movement within network	A tactic used by attackers to move horizontally within a network after gaining initial access, often to escalate privileges or reach valuable targets.	
legacy system	An older or outdated computer system, software, or technology that may still be in use but is typically less efficient or secure.	
load balancing	The distribution of network traffic or computing workloads across multiple servers or resources to ensure optimal performance and reliability.	

## Glossary

Term	Description	Source
logging	The process of recording events, actions, or transactions in a system for monitoring, troubleshooting, and auditing purposes.	
logistics	The planning, management, and coordination of resources, often in the context of supply chain management.	
machine learning (ML)	A subset of artificial intelligence (AI) that involves the development of algorithms and models that allow computers to learn from data and make predictions or decisions.	
maintaining	The ongoing process of keeping a system or application operational and up to date.	
mental lock-in	A cognitive bias in which individuals become overly committed to a particular idea or approach, making it challenging to consider alternative solutions.	
metrics	Quantifiable measures used to assess and evaluate the performance, effectiveness, or quality of a system or process.	
micro-segmentation	Network security strategy that divides a network into smaller, isolated segments to improve security and control.	
modeling and simulation	The use of mathematical models and computer simulations to replicate real-world processes or systems for analysis and experimentation.	
monitoring	The continuous observation and tracking of a system's performance, behavior, or security.	
networked weapons	Military or defense systems that are interconnected and can communicate with each other for coordinated operations.	
nonrepudiation	The assurance that a user cannot deny the authenticity or origin of a message or action they initiated.	
open standards and protocols	Publicly available and widely accepted specifications for communication and data exchange.	
operational	Related to the day-to-day activities and functions of an organization or system.	

## Glossary

Term	Description	Source
organization constructs	The structural elements, roles, and relationships within an organization.	
pagination	The practice of dividing large sets of data or content into smaller, manageable pages for display or retrieval.	
penetration testing	A security assessment where ethical hackers attempt to identify vulnerabilities in a system by simulating real-world attacks.	
performance	The ability of a system to execute tasks efficiently and meet specified criteria.	
platform	A software or hardware environment that provides a foundation for building and running applications.	
policy	A set of rules, guidelines, or principles that dictate decisions and actions within an organization or system.	
proactive threat detection	The practice of identifying and mitigating security threats before they can cause harm or damage.	
program managers	Individuals responsible for overseeing and managing projects or initiatives within an organization.	
providers	Entities that offer services, resources, or solutions to others.	
Quality of Service (QoS)	Quality of Service referenced in this document is the set of performance guarantees provided by a system. For example, a real-time system may guarantee a response to a message with some fixed time, or a network might guarantee that some fraction of the bandwidth between two endpoints for some service in particular is no less than some fraction of the total bandwidth.	
rapid prototyping	The process of quickly creating a working model or prototype of a product or system to test and validate concepts.	
Real-time Operating System (RTOS)	A specialized computer operating system that provides convenient access to hardware and efficient management of resources (task, memory, I/O) with strict time constraints. Examples of RTOSs are: VxWorks, Green Hills Integrity or QNX Neutrino, Red Hat RTOS, Space X extension??	

## Glossary

Term	Description	Source
real-time system (RTS)	A designed set of resources operating within an operational regime (a collection of behaviors, procedures and conditions) that provide a quality of service that includes one of the following time constraints a) DETERMINISTIC: the resources must act, respond or deliver with very high probability within a minimum or predictable period of time and margin of error (i.e. requirements for deadlines, latency and/or jitter), or b) INTERACTIVE: have functionality that appears to update interactivity from a human perspective or c) SIMULATE TIME: or as a simulation, must execute in 1:1 time. Resources could be software, computers, components, networks, communication systems, people, equipment, and materials.	
recursive	A process or function that calls itself to solve a problem by breaking it down into smaller, similar tasks.	
Representational State Transfer (REST)	An architectural style for designing networked applications, often used with HTTP, emphasizing stateless communication and resource-based URLs.	
requirements	The specifications and criteria that define what a system or product must accomplish or include.	
Reverse Conway's Maneuver	Adapting an organization's structure to align with desired software architecture.	
reverse proxy	A server that acts as an intermediary between client requests and one or more backend servers, often used for load balancing, security, and caching.	
scalability	The ability of a system or application to handle increased workloads or users without a significant decrease in performance.	
secure key management	The practices and processes for generating, storing, and protecting encryption keys.	
security breaches	Unauthorized access or incidents that compromise the confidentiality, integrity, or availability of data or systems.	
security compliance	Adherence to security standards, regulations, and policies to protect against security threats and vulnerabilities.	

## Glossary

Term	Description	Source
security posture	The overall security status and readiness of an organization or system.	
sensitive information	Data that, if disclosed or compromised, could result in harm to individuals or organizations.	
sensor fusion	Combining data from multiple sensors or sources to improve accuracy and reliability in various applications, such as navigation or surveillance.	
serialization attack	An attack that manipulates the serialization process of data to exploit vulnerabilities.	
Services	For the U.S. Department of Defense, the following Military Services: Army, Navy, Air Force, Marine Corps, Coast Guard, Space Force, and other supporting Components.	
session persistence	The ability to maintain session state or data across multiple interactions or requests from a user.	
socket	An abstract representation for the local endpoint of a network communications path	
social network systems	Online platforms and communities where users can connect, share, and interact with others.	
software delivery	The process of planning, developing, testing, and deploying software applications or updates.	
sorting	Arranging data or elements in a specific order, often numerical or alphabetical.	
stakeholders	Individuals or groups with an interest or concern in a project, system, or organization.	
statistical analysis	The process of analyzing data using statistical methods and techniques to draw conclusions or make predictions.	
strategic objectives	Long-term goals and plans that guide an organization's overall direction and decision making.	
support staff	Personnel responsible for assisting users, maintaining systems, and providing technical support.	

## Glossary

Term	Description	Source
system of systems	A collection of interconnected and interdependent systems that work together to achieve a common goal.	
tactical	Pertaining to short-term, practical, and on-the-ground decisions and actions.	
telemetry	The remote monitoring and measurement of data, often from distant or inaccessible locations.	
thread detection and response	The identification and mitigation of threats or malicious activities in computer systems, networks, or applications.	
time to live (TTL) tokens	Tokens or data elements with a specified lifespan or expiration time.	
tracking use	Monitoring and recording how resources or services are used or accessed.	
transport systems	Infrastructure and technologies used to move people, goods, or data from one place to another.	
unauthorized access attempts	Efforts to gain unauthorized entry to a system, application, or resource.	
vendor lock-in	A situation in which a user becomes dependent on a specific vendor's products or services, making it difficult to switch to alternatives.	
versioning	The practice of assigning unique version numbers to software or data to manage changes and updates.	
vulnerabilities and threats	Weaknesses or flaws in systems, applications, or processes that can be exploited by threats or attackers.	
warfighting	The conduct of military operations and strategies in armed conflict.	
wargames	Simulated military exercises or games used for training and strategic planning.	
zero trust (ZT)	A security model that assumes no trust by default and requires strict authentication and authorization for all users and devices, regardless of their location or network access.	

**Acronyms**

A&A	Authentication and Authorization
ABAC	Attribute-Based Access Control
AI/ML	Artificial Intelligence/Machine Learning
API	Application Programming Interface
A&S	Acquisition and Sustainment
ATDLP	Allied Tactical Data Link Publication
CDM	Common Data Model
CI/CD	Continuous Integration/Continuous Deployment
CJADC2	Combined JADC2
CSRF	Cross-Site Request Forgery
DAST	Dynamic Application Security Testing
DevSecOps	Development, Security, Operations
DISR	DoD Information Technology Standards Registry
DoD	Department of Defense
DoD CIO	Department of Defense Chief Information Officer
DoS	Denial-of-Service
GECP	General External Consumer Planning
HTTP	Hypertext Transfer Protocol
IAST	Interactive Application Security Testing
IoMT	Internet of Military Things
JADC2	Joint All-Domain Command and Control
JSON	JavaScript Object Notation
MFA	Multi-Factor Authentication
MIL-STD	Military Standard
MOSA	Modular Open Systems Approach
MVP	Minimum Viable Product
NIST	National Institute of Standards and Technology
NFR	Non-Functional Requirement

## Acronyms

OSI	Open System Interconnection
OUSD(A&S)	Office of the Under Secretary of Defense for Acquisition and Sustainment
OUSD(R&E)	Office of the Under Secretary of Defense for Research and Engineering
PHI	Protected Health Information
PII	Personally Identifiable Information
PM	Program Manager
RBAC	Role-Based Access Control
R&E	Research and Engineering
REST	Representational State Transfer
RTOS	Real-time Operating System
RTS	Real-time system
SAST	Static Application Security Testing
SE&A	Systems Engineering and Architecture
SLA	Service-Level Agreement
SLA	Service Life Agreement
SoS	System of Systems
SSL/TLS	Secure Sockets Layer/Transport Layer Security
TTL	Time to Live (Tokens)
URL	Uniform Resource Locator
XML	Extensible Markup Language
XSS	Cross-Site Scripting

## References

- Alakeel, Ali M. 2015. "Using Fuzzy Logic Techniques for Assertion-Based Software Testing Metrics." *Scientific World Journal* (April).  
<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4427827/>.
- Alshahwan, Nadia, et al. 2024a. *Assured LLM-Based Software Engineering*. Cornell University.  
<https://arxiv.org/abs/2402.04380>.
- Alshahwan, Nadia, et al. 2024b. *Automated Unit Test Improvement using Large Language Models*. Menlo Park, California: Meta Platforms Inc.  
<https://arxiv.org/pdf/2402.09171.pdf>.
- Anderson, Ross. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd Edition*. Wiley. <https://www.wiley.com/en-us/Security+Engineering%3A+A+Guide+to+Building+Dependable+Distributed+Systems%2C+3rd+Edition-p-9781119642787>.
- AWS SDK. 2024. *What Is an SDK?* Amazon Web Services (AWS). Accessed October 10, 2024.  
<https://aws.amazon.com/what-is/sdk/#:~:text=SDKs%20reduce%20the%20time%20and,required%20to%20create%20new%20applications>.
- AWS Tools. 2024. "Tools to Build on AWS: Tools for Developing and Managing Applications on AWS." *AWS Developer Center*. Accessed October 11, 2024.  
<https://aws.amazon.com/developer/tools/>.
- Brady and Dianic. 2022. *Data Interoperability*. PowerPoint presentation, DoD Enterprise DevSecOps Community of Practice. <https://software.af.mil/wp-content/uploads/2022/05/05-12-2022-DevSecOps-CoP-Slides-API-2.0-FINAL.pdf>.
2024. *ChatOps*. Accessed October 14, 2024.  
<https://www.techtarget.com/searchitoperations/definition/ChatOps#:~:text=ChatOps%2C%20sometimes%20known%20as%20conversation,development%20and%20IT%20operations%20tasks>.
- CISA. 2023. *Zero Trust Maturity Model, Version 2.0*. Cybersecurity Division, Cybersecurity and Infrastructure Security Agency (CISA), 23. [https://www.cisa.gov/sites/default/files/2023-04/zero\\_trust\\_maturity\\_model\\_v2\\_508.pdf](https://www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf).
- CJCSI RSI. 2019. *CJCSI Rationalization, Standardization, and Interoperability (RSI) Activities*. Chairman of the Joint Chiefs of Staff (CJCS).  
[https://www.jcs.mil/Portals/36/Documents/Doctrine/allied\\_doctrine/cjcsi2700\\_01g.pdf?ver=fIHUSEemnzzQUMj\\_sjoOnQ%3D%3D](https://www.jcs.mil/Portals/36/Documents/Doctrine/allied_doctrine/cjcsi2700_01g.pdf?ver=fIHUSEemnzzQUMj_sjoOnQ%3D%3D).
- CMU SEI. 2022. "Continuous Delivery." [https://cmu-sei.github.io/DevSecOps-Model/?\\_gl=1\\*i7p6w5\\*\\_ga\\*MTM1NjUzMjk1My4xNzA5MTI4MDM5\\*\\_ga\\_87WECW6HCS\\*MTcwOTE0NDg1MS4xLjAuMTcwOTE0NDg1MS42MC4wLjA.#Diagrams\\_7b8cab04-0f58-4b71-88b6-8eef5374734e](https://cmu-sei.github.io/DevSecOps-Model/?_gl=1*i7p6w5*_ga*MTM1NjUzMjk1My4xNzA5MTI4MDM5*_ga_87WECW6HCS*MTcwOTE0NDg1MS4xLjAuMTcwOTE0NDg1MS42MC4wLjA.#Diagrams_7b8cab04-0f58-4b71-88b6-8eef5374734e).
- Computer Networking Notes. 2023. "OSI Seven Layers Model Explained with Examples." *Computer Networking Notes*. August 8. Accessed October 23, 2023.

- <https://www.computernetworkingnotes.com/ccna-study-guide/osi-seven-layers-model-explained-with-examples.html>.
- Corbo, Anthony. 2023. "What Is Data Integrity?" *Built In*. January 3. Accessed October 23, 2023. <https://builtin.com/data-science/data-integrity>.
- DAU . 2024. "Specifications and Standards: Defense Standard Procedures, Practices, and Methods." *Defense Acquisition University (DAU) Acquipedia*. Accessed October 10, 2024. [https://www.dau.edu/acquipedia-article/specifications-and-standards#:~:text=A%20defense%20standard%20\(often%20referred,procedures%2C%20practices%2C%20and%20methods](https://www.dau.edu/acquipedia-article/specifications-and-standards#:~:text=A%20defense%20standard%20(often%20referred,procedures%2C%20practices%2C%20and%20methods).
- DAU. 2025. *iCatalog Courses and Schedules*. Defense Acquisition University (DAU). Accessed April 23, 2025. website: <https://icatalog.dau.edu/onlinecatalog/CareerLvl.aspx>.
- DepSecDef. May 2021. *Creating Data Advantage*. Memorandum, Deputy Secretary of Defense (DepSecDef). <https://media.defense.gov/2021/May/10/2002638551/-1/-1/0/DEPUTY-SECRETARY-OF-DEFENSE-MEMORANDUM.PDF>.
- DepSecDef. November 2021. *DoD Software Modernization Strategy, Version 1.0*. Deputy Secretary of Defense (DepSecDef). <https://media.defense.gov/2022/Feb/03/2002932833/-1/-1/1/DEPARTMENT-OF-DEFENSE-SOFTWARE-MODERNIZATION-STRATEGY.PDF>.
- DevSecOps Reference Architecture. 2023. *Modern Software Practices*. Chief Information Officer (CIO) Library. <https://dodcio.defense.gov/library/>.
- DHS. 2015. *Writing Guide for a Memorandum of Understanding*. Department of Homeland Security (DHS), Office of Cybersecurity and Communications. Accessed October 10, 2024. <https://www.cisa.gov/sites/default/files/publications/Writing%2520Guide%2520for%2520a%2520Memorandum%2520of%2520Understanding%2520%2528MOU%2529.pdf>.
- DoD CIO and USD(A&S). September 2021. *DoD Enterprise DevSecOps Strategy Guide, Version 2.1*. Washington, D.C.: Department of Defense (DoD) Chief Information Officer (CIO) and Under Secretary of Defense for Acquisition and Sustainment. [https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Strategy%20Guide\\_DoD-CIO\\_20211019.pdf](https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Strategy%20Guide_DoD-CIO_20211019.pdf).
- DoD CIO. September 2021. *DevSecOps Playbook, Version 2.1*. Department of Defense (DoD) Chief Information Officer (CIO). [https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Playbook\\_DoD-CIO\\_20211019.pdf](https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Playbook_DoD-CIO_20211019.pdf).
- DoD CIO. CMMC. "DoD Cybersecurity Maturity Model Certification (CMMC)." [10:21 AM] Stark, Michael T CTR OSD OUSD R&E (USA).
- DoD CIO. March 2021 . *DoD DevSecOps Fundamentals Guidebook: DevSecOps Tools & Activities, Version 2.0*. Department of Defense (DoD) Chief Information Officer (CIO). <https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOpsTools-ActivitiesGuidebook.pdf>.

- DoD CIO. August 2019. *DoD Enterprise DevSecOps Reference Design, Version 1.0*. DoD Chief Information Office (CIO). <https://dodcio.defense.gov/Portals/0/Documents/>.
- DoD CIO. October 2023. "Modern Software Practices." <https://dodcio.defense.gov/library/>.
- DoD CIO. 2023. *Software Modernization Implementation Plan*. Restricted distribution, DoD Chief Information Officer (CIO).
- DoD. 2022. "DoD Announces Release of JADC2 Implementation Plan." *defense.gov*. March 17. Accessed October 23, 2023.  
<https://www.defense.gov/News Releases/Release/Article/2970094/dod-announces-release-of-jadc2-implementation-plan/>.
- DoD. February 2020. *DoD Cybersecurity Test and Evaluation Guidebook Version 2.0, Change 1*. Department of Defense (DoD). <https://www.dau.edu/sites/default/files/2023-09/Cybersecurity-Test-and-Evaluation-Guidebook-Version2-change-1.pdf>.
- DoD. September 2020. *DoD Data Strategy*. Washington, D.C.: Department of Defense (DoD). <https://media.defense.gov/2020/Oct/08/2002514180/-1/-1/0/DOD-DATA-STRATEGY.PDF>.
- DoD. June 2023. *DoD Data, Analytics, Artificial Intelligence Adoption Strategy*, *Department of Defense*. Department of Defense (DoD).  
[https://media.defense.gov/2023/Nov/02/2003333300/-1/-1/1/DOD\\_DATA\\_ANALYTICS\\_AI\\_ADOPTION\\_STRATEGY.PDF](https://media.defense.gov/2023/Nov/02/2003333300/-1/-1/1/DOD_DATA_ANALYTICS_AI_ADOPTION_STRATEGY.PDF).
- DoD Issues New Data Strategy. 2020. *defense.gov*. October.  
<https://www.defense.gov/News Releases/Release/Article/2376629/dod-issues-new-data-strategy/>.
- DoDD 8140.01. 2020. *DoD Directive 8140.01, Cyberspace Workforce Management*. Washington, D.C.: Deputy Secretary of Defense.  
<https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodd/814001p.pdf>.
- DoDI 5000.87. 2020. "Operation of the Software Acquisition Pathway ." Office of the Under Secretary of Defense for Acquisition and Sustainment .  
<https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500087p.PDF>.
- DoDI 8140.02. 2021. *DoD Instruction (DoDI) 8140.02, Identification, Tracking, and Reporting of Cyberspace Workforce Requirements*. Department of Defense (DoD) Chief Information Officer .  
<https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/814002p.PDF>.
- DoDI 8510.01. 2020. *DoD Instruction 8510.01, Risk Management Framework (RMF) for DoD Information Technology (IT)*. Washington, D.C.: DoD Chief Information Officer .  
[https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/851001p.pdf?ver=qE\\_E2HGN\\_HE4Blu7161t1TQ%3D%3D](https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/851001p.pdf?ver=qE_E2HGN_HE4Blu7161t1TQ%3D%3D).
- Dredd. 2024. *Dredd-HTTP API Testing Framework*. Accessed June 21, 2024.  
<https://dredd.org/en/latest/>.
- EPA. 2023. "Learn about Data Standards." *U.S. Environmental Protection Agency (EPA)*. June. Accessed October 23, 2023. <https://www.epa.gov/data-standards/learn-about-data-standards>.

- standards#:~:text=What%20are%20data%20standards%3F,use%2C%20and%20management%20of%20data.
- F5 Glossary. n.d. *API Gateway*. Accessed June 21, 2024. <https://www.f5.com/glossary/api-gateway>.
- Firesmith, Donald. 2015. "A Taxonomy of Testing: What-Based and When-Based Testing Types." *SEI Blog*. [insights.sei.cmu.edu/blog/a-taxonomy-of-testing-what-based-and-when-based-testing-types](https://insights.sei.cmu.edu/blog/a-taxonomy-of-testing-what-based-and-when-based-testing-types).
- GitHub. 2024. *Microsoft/Restler-Fuzzer*. Accessed June 21, 2024. <https://github.com/microsoft/restler-fuzzer>.
- Hoehn, John R. 2022. *Joint All-Domain Command and Control (JADC2)*. IF11493, Congressional Research Service (CRS). [https://us-east-1-02900067-inspect.menlosecurity.com/safeview-fileserv/tc\\_download/d4a9166ec39f3f84b1592e54ac08c10706a8025adf7cfa5d118462b3395e2b74/?&cid=N17BBEED83A26\\_&rid=ed191390276891c293ed9aa049739f](https://us-east-1-02900067-inspect.menlosecurity.com/safeview-fileserv/tc_download/d4a9166ec39f3f84b1592e54ac08c10706a8025adf7cfa5d118462b3395e2b74/?&cid=N17BBEED83A26_&rid=ed191390276891c293ed9aa049739f).
- Humble, Jez, and David Farley. 2015. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston: Addison-Wesley.
- IBM API. 2023. "What Is an API?" *IBM*. Accessed October 23, 2023. <https://www.ibm.com/topics/api>.
- IBM Qiskit. 2024. *Qiskit Runtime - Create a Qiskit Runtime instance to run quantum programs, on Quantum Systems*. IBM. <https://cloud.ibm.com/catalog/services/qiskit-runtime>.
- IEEE 2675-2021. 2021. *IEEE 2675-2021: IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment*. C/S2ESC Software and Systems Engineering Standards Committee, IEEE Computer Society. <https://standards.ieee.org/ieee/2675/6830/>.
- IEEE Computer Society. 2023. "Software Engineering Body of Knowledge Version 3." *IEEE Computer Society*, July 19. [www.computer.org/education/bodies-of-knowledge/software-engineering/v3](https://www.computer.org/education/bodies-of-knowledge/software-engineering/v3).
- ISO 8601. 2019. *ISO 8601: Date and Time Format*. Geneva: International Organization for Standardization (ISO). <https://www.iso.org/iso-8601-date-and-time-format.html>.
- ISO/IEC 7498-1. 1994. *ISO/IEC 7498-1:1994. Information Technology: Open Systems Interconnection Basic Reference Model*. Reviewed and confirmed in 2000, Geneva: International Organization for Standardization (ISO). <https://www.iso.org/standard/20269.html>.
- Kim, Myeongsoo, et al. 2023. "Enhancing Rest API Testing with NLP Techniques." *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. <https://doi.org/10.1145/3597926.3598131>.
- Kumar, Divya, and K. K. Mishra. 2016. "The impacts of test automation on software's cost, quality and time to market." *Procedia Computer Science* 79: 8-15. <https://doi.org/10.1016/j.procs.2016.03.003>.

- Kutztown. 2014. *Invariants, Preconditions, and Postconditions*. Kutztown, Pa.: Kutztown University Department of Computer Science. Accessed July 2024.  
<https://www.kutztown.edu/Departments-Offices/A-ComputerScienceInformationTechnology/Documents/Student%20Resources/DocumentationInvariants.pdf>.
- Leung, Adrian, and Chris Mitchell. 2023. "Service Discovery Threat Model for Ad Hoc Networks."  
<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a400c6ac8890e523d3ee3cd17074ad9db0799085>.
- McKinsey & Company. 2023. "What Is Psychological Safety?" *McKinsey & Company*. July 17. Accessed October 24, 2023. <https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-psychological-safety>.
- Mead, Nancy R., and Carol C. Woody. 2017. *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance*. Boston: Addison-Wesley.
- Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. 2nd Edition. Prentice Hall.
- Microsoft. April 6, 2022. *Common Data Model*. <https://learn.microsoft.com/en-us/common-data-model/>.
- . 2024. "IntelliSense in Visual Studio." *AI Skills Challenge Sep 24-Nov 1, 2024*. January 18. Accessed October 10, 2024. <https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022>.
- NATO. 2023. *A Data Centric Architecture for the Alliance*. Working paper AC/322 (CP/1)WP(2023)0021 (INV), Consultation, Command and Control Board (C3B) Communications and Information Services Capability Panel (CIS CAP), North Atlantic Treaty Organization (NATO).
- Nguyen, Cuong, Huy Bui, Vu Nguyen, and Tien Nguyen. 2023. "An Approach to Generating API Test Scripts Using GPT." *Proceedings of the 12th International Symposium on Information and Communication Technology (SOICT '23)*. Association for Computing Machinery.
- NIST. 800-95. <https://csrc.nist.gov/pubs/sp/800/95/final>.
- NIST SP 1800-21. 2020. "NIST Special Publication (SP) Mobile Device Security: Corporate Owned Personally-Enabled (COPE)." *NIST Computer Security Resource Center*. Edited by Joshua, et al. Franklin. National Institute of Standards and Technology (NIST). September. Accessed October 1, 2023. <https://csrc.nist.gov/pubs/sp/1800/21/final>.
- NIST SP 800-204. 2019. *Security Strategies for Microservices-Based Application Systems*. National Institute of Standards and Technology (NIST), Department of Commerce. <https://csrc.nist.gov/pubs/sp/800/204/final>.
- NIST SP 800-207. 2020. *Zero Trust Architecture*. National Institute of Standards and Technology (NIST). <https://csrc.nist.gov/pubs/sp/800/207/final>.
- NIWC. 2024. *Naval Information Warfare Center (NIWC) Pacific (PAC)*. Accessed October 29, 2024. <https://www.niwcpacific.navy.mil/>.

- OAuth 2.0. 2023. *OAuth 2.0*. Accessed October 23, 2023. <https://oauth.net/2/>.
- OUSD(A&S). 2019. *DoD Digital Modernization Strategy: DoD Information Resource Management Strategic Plan FY19-23*. Washington, D.C.: Office of the Under Secretary of Defense for Acquisition and Sustainment (OUSD(A&S)).  
<https://media.defense.gov/2019/Jul/12/2002156622/-1/-1/1/DOD-DIGITAL-MODERNIZATION-STRATEGY-2019.PDF>.
- OUSD(A&S). 2022. *Guidance for Programs to Create API Strategy*. Washington, D.C.: Office of the Under Secretary of Defense for Acquisition and Sustainment (OUSD(A&S)).  
[https://aaf.dau.edu/storage/2023/05/Program-API-Strategy-Template\\_22May2023\\_v2.docx](https://aaf.dau.edu/storage/2023/05/Program-API-Strategy-Template_22May2023_v2.docx).
- OWASP. 2024. *Fuzzing*. Accessed March 8, 2024. [owasp.org/www-community/Fuzzing](https://owasp.org/www-community/Fuzzing).
- OWASP REST Security Cheat Sheet. 2024. *REpresentational State Transfer (REST) Security Cheat Sheet*. OWASP. Accessed October 24, 2024.  
[https://cheatsheetseries.owasp.org/cheatsheets/REST\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html).
- Pedro, Bruno. 2023. "Build a Successful API by Understanding User Personas." *Postman Blog*. May 16. Accessed October 11, 2024. <https://blog.postman.com/build-successful-api-by-understanding-user-personas>.
- Platform One. 2024. *Iron Bank*. Accessed October 29, 2024. <https://p1.dso.mil/services/iron-bank>.
- Renner, S. 2005. "Net-Centric Information Management." *10th Int. Command and Control Research and Technology Symposium, June 2005*. [https://us-east-1-029060369-inspect.menlosecurity.com/safeview-fileserv/tc\\_download/06c47709d88a12b838344780ccbdac0ce43e6c2ac099d385703b1b0f04c9593c/?&cid=NC6F6AE274792\\_&rid=1fd963d2fc4ec1d64b6c25a0a0b0](https://us-east-1-029060369-inspect.menlosecurity.com/safeview-fileserv/tc_download/06c47709d88a12b838344780ccbdac0ce43e6c2ac099d385703b1b0f04c9593c/?&cid=NC6F6AE274792_&rid=1fd963d2fc4ec1d64b6c25a0a0b0).
- Sanders, Geoffrey. 2021. *Integrating Zero Trust and DevSecOps*. Carnegie Mellon University Software Engineering Institute . <https://apps.dtic.mil/sti/trecms/pdf/AD1145432.pdf>.
- Sato, Daniel. 2014. "Canary Release." *martinfowler.com*. June 25. Accessed October 23, 2023. <https://martinfowler.com/bliki/CanaryRelease.html>.
- Searls, Justin. 2021. ""People love debating . . . ." X. May 14. <https://twitter.com/searls/status/1393385209089990659>.
- SecDef . March 2025. *Directing Modern Software Acquisition to Maximize Lethality*. Memorandum, Secretary of Defense (SecDef).  
<https://media.defense.gov/2025/Mar/07/2003662943/-1/-1/1/DIRECTING-MODERN-SOFTWARE-ACQUISITION-TO-MA>.
- Seemann, Mark. 2012. *TDD Test Suites Should Run in 10 Seconds or Less*. May 24. [blog.ploeh.dk/2012/05/24/TDDtestsuitesshouldrunin10secondsorless/](http://blog.ploeh.dk/2012/05/24/TDDtestsuitesshouldrunin10secondsorless/).
- SEI. May 2022 . "DevSecOps Platform Independent Model (PIM)." *Carnegie Mellon University (CMU) Software Engineering Institute (SEI)*. May. Accessed June 21, 2024.  
[https://www.sei.cmu.edu/our-work/projects/display.cfm?customel\\_datapageid\\_4050=344852](https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=344852).

## References

- . May 2022a. "Unlock the Potential of DevSecOps in Regulated and Constrained Environments." *Carnegie Mellon University Software Engineering Institute (SEI)*. May. Accessed June 18, 2024. [https://www.sei.cmu.edu/our-work/projects/display.cfm?customel\\_datapageid\\_4050=344852](https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=344852).
- Sherman, Mark. 2024. *Using ChatGPT to Analyze Your Code? Not So Fast*. Software Engineering Institute (SEI), Carnegie Mellon University. <https://insights.sei.cmu.edu/blog/using-chatgpt-to-analyze-your-code-not-so-fast/>.
- Sindall, Gemma. 2020. "An API Governance Model for Great APIs." *digitalML*. August 28. Accessed October 23, 2023. <https://www.digitalml.com/api-governance-model/>.
- . 2023. "What Is API Governance? 9 Best Practices for API Governance Success." *digitalML*. April 23. Accessed October 23, 2023. <https://www.digitalml.com/api-governance-best-practices/>.
- Swagger. 2021. *OpenAPI Specification, Version 3.1.0 February 2021*. Swagger. SmartBear. February. <https://swagger.io/specification/>.
- TRMC. 2019. *Memorandum of Understanding (MOU) Template*. Test Resource Management Center (TRMC). Derived from DoD Instruction 4000.19. Accessed October 10, 2024. <https://www.trmc.osd.mil/wiki/display/publicRCC/606+Range+Commanders+Council+Cybersecurity+Group+Guidebook?preview=%2F143951267%2F143951266%2FMOU+TEMPLATE.doc>.
- Villalba, Marcia. 2023. *Previewing Environments Using Containerized AWS Lambda Functions*. AWS Compute Blog. February 6. Accessed March 2024. <https://aws.amazon.com/blogs/compute/previewing-environments-using-containerized-aws-lambda-functions/>.
- Williams, Laurie, et al. 2009. "On the effectiveness of UNIT test automation at Microsoft." *20th International Symposium on Software Reliability Engineering*. <https://doi.org/10.1109/issre.2009.32>.
- Yallon, Erez, et al. 2023. *OWASP Top 10 API Security Risks – 2023*. Open Worldwide Application Security Project (OWASP). <https://owasp.org/API-Security/editions/2023/en/0x11-t10/>.

**Application Programming Interface (API) Technical Guidance**

Minimum Viable Capability Release (MVCR) 2, April 2025

Office of Systems Engineering and Architecture  
Office of the Under Secretary of Defense for Research and Engineering  
3030 Defense Pentagon  
Washington, DC 20301  
[osd-sea@mail.mil](mailto:osd-sea@mail.mil)  
<https://www.cto.mil/sea>

Distribution Statement A. Approved for public release. Distribution is unlimited.