Carnegie Mellon University
Software Engineering Institute

REFERENCE ARCHITECTURE FOR ASSURING ETHICAL CONDUCT IN LAWS

Andrew O. Mellinger, Eric T. Heim, Andrew Schellenberg, Emily Newman, Tyler Brooks, Charles Loughin

April 2025

DOI: 10.1184/R1/28700720

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Table of Contents

1 Introduction 1
1.1 Project Background 2

2 Context, Goals, Requirements, and Constraints 4
2.1 Functional Requirements 4
2.1.1 Mobility Requirements 4
2.1.2 Perception and Targeting Requirements 4
2.1.3 Human Interaction Requirements 5
2.1.4 Effect Requirements 5
2.2 RAI Principles 5
2.3 Conduct Assurance and Conduct Governors 5
2.3.1 Example RoE 6
2.4 Architectural Requirements 6
2.4.1 Human–System Interaction Goals 6

3 Quality Attributes 9
3.1 Testability 9
3.2 Observability and Transparency 9
3.3 Usability (Interaction Capability) 9
3.4 Maintainability 10
3.5 Modifiability 10
3.6 Modularity 10
3.7 Security 11
3.8 Performance 11
3.9 Portability 11
3.10 Availability 11

4 Design Decisions, Styles, and Patterns 12
4.1 Conduct Governor and Conduct Dissemination 12
4.2 Situational Awareness Blackboard (SAB) 12
4.3 Conduct Governor Wrapper Pattern 13
4.4 Interaction Styles 14
4.5 Hardware, Processes, and Threads 14
4.6 Data Annotating Versus Modifying (Checking Versus Constraining) 14
4.6.1 Modification 15
4.6.2 Annotation 15
4.6.3 Final Determination 16

5 Architecture Description with Component Views 17
5.1 Actuation 18
5.2 HSI and Communications 19
5.3 Health and Support 20
5.4 Navigation and Localization 20
5.4.1 Behavioral Planner 21
5.4.2 Map Manager 21
5.4.3 Movement, Goal, and Mission Planner 21
5.4.4 Obstacle Detection 22

5.4.5 Route Planner 22
5.4.6 Trajectory Planner 22
5.5 Perception, Targeting, and Response (PTR) Chain 22
5.5.1 Object Detector 24
5.5.2 Path Tracker 25
5.5.3 Entity Registrar 25
5.5.4 Activity Recognizer 26
5.5.5 Scene Understanding 26
5.5.6 Target Nominator 26
5.5.7 Responder 27
5.5.8 Battle Damage Assessment (BDA) 27
5.6 Sensing 27
5.6.1 Sensor 28
5.6.2 Validator 28
5.6.3 Processing 29
5.6.4 Fusion 29
5.6.5 Final Processing 29
5.7 Situational Awareness Blackboard and Conduct Disseminator (SAB&CD) 29
5.7.1 Situational Awareness Blackboard (SAB) 30
5.7.2 Conduct Disseminator (CD) 30

6 Variations and Alternatives 31
6.1 Combining Motion Tracking and Entity Registration 31

7 Related Works 32
7.1 UMAA 32
7.1.1 Sensor and Effector Management 33
7.1.2 Situational Awareness 33
7.1.3 Mission Management 34

Appendix A: Testing the Conduct Governance of LAWS Using a ROS-Based System Proxy 35
Study Introduction 35
Test System Details 35
Test System Design 36
Defining Conduct for the Conduct Governor and Conduct Dissemination 37
Designing the Situational Awareness Blackboard 39
Application of the Conduct Governor Wrapper Pattern 41
Selecting an Interaction Style 42
Decisions on Hardware, Processes, and Threads 43
Usage of Data Annotation and Modification 43
Implementation Architecture Description with Component Views 45
Implementation Actuation 45
Implementation HSI and Communications 46
Implementation Health and Support 47
Implementation Navigation and Localization 47
Implementation PTR Chain 47
Implementation Sensing 52
Implementation SAB & CD 52
Reflection on Functional Attributes 53
Reflection on Functional Requirements 53
Reflection on Architectural Requirements 55

Reflection on Quality Attributes 56
Study Conclusion 58

References **60**

Legal Markings **63**

Contact Us **64**

1 Introduction

Lethal autonomous weapon systems (LAWS) contain components that perceive the environment, identify targets, and choose lethal responses to meet mission goals. These decision-making systems must act in an ethical way in accordance with the rules of engagement (RoE), laws of war (LoW), and laws of armed conflict (LoAC) as specified by their commanders. This document describes a reference architecture that provides a set of patterns and techniques to reason about and instantiate a working system architecture that promotes assurance about the *conduct* of the LAWS during operation. This document does not describe a particular set of ethical values,[1] but—once decided and encoded as a formalism—these values can be systematically encoded, stored, decomposed, and communicated to various parts of the system.

Autonomous and artificial intelligence (AI) systems (such as expert systems) have been around for many years and have shown up on other weapons systems in the past. Modern AI and hardware developments are rapidly advancing machine learning (ML) technologies and bringing a wide range of new capabilities to weapon systems. This document focuses on the challenges and capabilities introduced by ML components and how their introduction into these systems and their advanced decision-making abilities require thoughtful design.

In the systems we discuss, autonomous decision-making results in physical interactions in the real world such as movement, actuators, and the firing of weapons. It is critical that the conduct of the system reflects and is limited to the ethics of the deployers of the systems. We call this adherence to ethics *conduct assurance.* We provide an example system decomposition that embodies these concepts directly into the system design itself by demonstrating how ethical values can be entered through configurations and controls, and how they can be observed at runtime. Additionally, this reference architecture decomposition promotes traceability of ethical requirements through implementation and testing.

The purpose of this architecture is to reason about the following:

- How to create systems that can embody and govern a set of ethical principles as intended. This goal implies that ethics are expressed through requirements and system design, and that they are observable during testing and operation. It also implies that the techniques and measures evaluators use can cover the entire set of potentially intended behaviors, further implying the need to define the scope of needed ethics.

[1] Ethics is a difficult, controversial, and ambiguous topic when it comes to artificial intelligence (AI) and autonomous weapon systems. Establishing a set of ethics for LAWS is beyond the scope of this document. However, for an example of a project that is working to better articulate and measure ethics, see the DARPA ASIMOV program at the following location: https://www.darpa.mil/program/autonomy-standards-and-ideals-with-military-operational-values.

- How to support a process that embodies and promotes the chosen set of ethical attributes. This goal implies that the process itself requires that developers consider and integrate ethics throughout the development and sustainment lifecycles so that that these devices can continue to embody the chosen ethics [IEEE 7000-2021].

In short, support for the use of an ethical model must be addressed pervasively and holistically both in *what* we build and *how* we build it.

In addition to the goals outlined above, this reference architecture supports the following:

- responsible AI (RAI) principles
- human–system integration (HSI), human–machine teaming (HMT), and trust
- traditional system and software engineering principles
- quality attribute scenarios
- Department of Defense (DoD) Directive 3000.09

Appendix A: Testing the Conduct Governance of LAWS Using a ROS-Based System Proxy provides the results from an experiment that attempted to follow the guidance in this document, including an evaluation of the impacts that resulted from implementing the patterns, techniques, and guidance that we describe in this reference architecture. Due to time constraints, we did not incorporate the feedback from that experiment into this document, but we have taken note of where we can improve the reference architecture for future versions.

1.1 Project Background

The Office of the Under Secretary of Defense for Research and Engineering (OUSD(R&E)) and the Carnegie Mellon University Software Engineering Institute (CMU SEI) jointly chartered the Center for Calibrated Trust Measurement and Evaluation (CaTE) program to assure the trustworthiness of AI systems and establish methods for the evaluation of operator trust in these systems. This reference architecture is a project to support the goals of that program.

During the pilot phase of the CaTE program in early 2024, we began evaluating LAWS by asking questions such as, “What does an example system look like? How many ML models are in a system? What traditional code supports these models? How do they interact with sensors and actuators? If one were to implement a conduct controller, what would it have to do?” When asking these questions, it became clear that we needed a sufficiently complex system operating in sufficiently complex use cases as a reference point to properly evaluate a LAWS. At that time, no complete LAWS was available that met the needs of the project, and we therefore set out to produce a fictitious one so we could reason about and evaluate practices, methods, and tools.

We started with the most challenging use case: that of a mobile ground system embedded in a military team operating in a contested urban environment with significant ongoing civilian activity. To build such a use case, we began with common patterns of autonomous grounds systems that included

perception, planning, and actuation components acquired from experience and partners. We then introduced the components of an automatic target recognition system. From this framework, we introduced components that specifically demonstrate the existing and emerging use of ML. Finally, we added patterns and components that support testability, observability, and transparency.

We did this background work with the intent of supporting the project of increasing testability, building an assurance case, and increasing user trust. CaTE specifically refers to the idea of *calibrated trust*, which is a term we use to refer to how much the operator trusts the LAWS in a specific operational mode and environment, rather than just a generalized sense of trust in something. We use the following definition of *calibrated trust*: "a psychological state of adjusted confidence that is aligned to end users' real-time perceptions of trustworthiness" [Gardner 2023]. We seek to understand, in detail, what aspects of LAWS behavior, design, and design processes positively or negatively impact the operators' trust.

In parallel to the development of this reference architecture, the CaTE team performed a baseline study on human-LAWS teaming interaction titled "Measuring Trust: Concept Testing and User Trust Evaluation in Autonomous Systems" [Hale 2025]. This study was performed, in part, to understand if operators anticipated the need for some sort of strong internal control over the actions of the LAWS. The experiments examined operator expectations for how a system might control behavior in a LAWS and what interactions the operator might have with such a system. Arkin uses the term "ethical governor" to describe such a system in his works (covered in Section 2.3), and we adopted that terminology. While the question often arose about the relative nature of ethics, the study identified the key finding in Table 1.

Table 1: Ethical Governor Findings

Do	Design, implement, and test an ethical decision-making framework that clearly explains system limitations to users.
Why	Users see an ethical governor as necessary for trust in autonomous systems, though they remain skeptical about its effectiveness and how ethics can be codified.
Impact	Without clear ethical guidance, operators may distrust the system's capabilities and decision-making, raising legal and ethical concerns for real-world applications of autonomous technologies.

This initial experiment identified many requirements such as the need for an ethical governor, strong situational awareness, strong human–system integration, and the need for traceability, all of which we take into consideration in the sections that follow. Based on the key findings, we have also expanded the capabilities for human-in-the-loop and de-escalation.

2 Context, Goals, Requirements, and Constraints

This section describes the requirements, constraints, and key drivers that we used to design the reference architecture.

2.1 Functional Requirements

While we aren't building a particular LAWS, having a set of motivating requirements promotes the design of the system. We chose a mobile ground system as the example for this reference architecture. Although these requirements are far from complete, they provide examples that give appropriate direction for our work.

2.1.1 Mobility Requirements

The system must function in a complex environment, and it must have some level of mobility. The motion of the system might violate ethical values such as the following: entering protected areas; causing damage by collision as an effector; and causing accidental damage, which should be avoided. Based on this scenario, we identified the following requirements that the system must perform:

- plan locations based on map intelligence and navigate to them
- identify static structures, obstacles, and entities
- navigate around static structures, obstacles, and entities
- navigate around and through dynamic obstacles and entities
- use the LAWS device as an effector when responding to threats

2.1.2 Perception and Targeting Requirements

Decision-making, especially with regards to lethal force, is highly dependent on how something perceives the world, whether it be a person or LAWS. Although far from exhaustive, the following set of items describes highly complex, perception-based tasks that influence such decision-making.

- identify stationary and dynamic entities
- identify and track dynamic targets across multiple viewings
- identify threats based on attributes such as "bearing weapons"
- identify threat levels based on activity
- identify non-threats
- request clarification from a user when identifying threats
- perpetrate attacks against threats when instructed to do so
- automatically perpetrate attacks against threats when pre-authorization to do so has been given
- identify sensitive situations where action would result in unintended consequences such as collateral damage

2.1.3 Human Interaction Requirements

Due to the ambiguous and complex nature of ethical decision making, interaction with operators and non-operators is highly important for complex LAWS. Therefore, the system must meet the following requirements in this area:

- ask for assistance from an operator when the system runs into ambiguity
- communicate with non-operators as an ability to de-escalate a situation
- communicate decision-making with an operator
- provide the operator with a way to monitor decision-making in all stages of the operation
- allow the operator to change the identification of threats and non-threats without breaking ethical rules
- operate in a human-acceptable, ethical framework

2.1.4 Effect Requirements

The system must meet the following requirements in this area:

- apply *lethal* effects when directed by mission goals or by the operator with the exception that the system shall not violate RoE or LoW
- apply *non-lethal* effects when directed by mission goals or by the operator

2.2 RAI Principles

The memo "Implementing Responsible Artificial Intelligence in the Department of Defense" describes five key principles for developing systems known as the RAI Principles [DoD 2021]. These key principles (responsibility, equitability, traceability, reliability. and governability) highly influence the quality attributes below, and we reference them in the sections that follow to emphasize and explain how different aspects of this reference architecture support the principles.

2.3 Conduct Assurance and Conduct Governors

The decision-making processes of a LAWS must adhere to RoE and LoW, which are written in natural language. These natural-language statements must be represented so that they can be translated into requirements, and, more importantly, into a formalism to configure the LAWS for different circumstances. Ronald Arkin offers good examples of how to address these issues in the 2007 paper, "Governing Lethal Behavior: Embedding Ethics in a Hybrid Deliberative/Reactive Robot Architecture" and in a subsequent paper from 2009 titled "An Ethical Governor for Constraining Lethal Action in an Autonomous System" [Arkin 2007; Arkin 2009]. In these papers, Arkin describes how to translate the natural language of RoE and LoW to configure the autonomous weapons systems using an ethical governor. In this reference architecture, we describe a system that can be configured using a formalism derived from these natural-language rules.

2.3.1 Example RoE

The creation of a formalism for LAWS to understand natural language is beyond the scope of this document. However, we offer an example of an RoE in this section to illustrate what a final architecture must take as an input. The RoE we provide below comes from the *Operational Law Handbook* and serves as an example of natural language that must be translated into a formalism that can be understood by a LAWS. The RoE is as follows:

> *a. You may open fire only if you, friendly forces or persons or property under your protection are threatened with deadly force. This means:*
>
> *(1) You may open fire against an individual who fires or aims his weapon at, or otherwise demonstrates an intent to imminently attack, you, friendly forces, or Persons with Designated Special Status (PDSS) or property with designated special status under your protection. [Lacey 2000]*

During the design of the formalism, these rules will need to be aggregated and then decomposed into features that the LAWS device understands such as object classes, behaviors, timing rules, and so forth. The applicable list of features depends on available capabilities and will change over time. Once we establish this decomposition and mapping, we can then implement the formalism in code. As an example, the rule above implies the need for, at minimum, the following requirements:

- The system shall be configurable for a set of predefined principles.
- The system shall establish traceability to show how it ingests, stores, and uses the principles at runtime.
- The system shall require user direction when situations diverge from the conditions described in the predefined principes. When lacking such direction, the system shall seek a safe state.
- The system shall communicate how it made decisions if it is functioning in a fully autonomous state, or it shall communicate how it made recommendations if it is functioning semi-autonomously or if it is dependent on a human-in-the-loop.

2.4 Architectural Requirements

For most projects, the goal of an architecture is to support requirements and quality attributes. For our purposes, the architecture is an output to help reason about practices, frameworks, tools, methods, and so on. Therefore, we have additional goals for the architecture beyond supporting quality attributes and requirements, which we outline in the sections that follow. In many cases, these requirements are themselves the quality attribute drivers listed in Section 3.

2.4.1 Human–System Interaction Goals

As mentioned in the example in Section 2.3.1, we are considering LAWS that work with an operator, function around other humans, and interact with adversaries. The study we introduced in Section 1.1—"Measuring Trust: Concept Testing and User Trust Evaluation in Autonomous Systems"—

produced 12 findings about how humans consider the idea of trust with regard to LAWS [Hale 2025]. These findings serve as guiding principles for the architecture we are describing.

This study is important to mention because we want to stress that human trust is a complicated and personal issue, and it is highly coupled to the operators, the environment, and real-world experiences. Because of this complexity, we can only go so far with abstract reasoning, and human feedback is essential to any human–system integration effort.

We provide summaries of the study's findings in Table 2 (see the study for full descriptions). They are listed in priority from highest to lowest as identified in the study.

Table 2: Priority of Findings with Respect to How Humans Consider Trust

1. Human authorization of force	Implement human-in-the-loop for all lethal decision-making and prioritize safety.
2. High-precision targeting	Design targeting systems that meet or exceed human performance.
3. Testing and reliability	Extensively test the system in varied environments and scenarios to ensure reliable performance in harsh conditions.
4. Situational awareness	Provide real-time, easy-to-understand data (e.g., video, audio, system location, orientation, projected path, objects of interest, obstacles, and battery status) to support situational awareness for both the system and operator.
5. Hardware redundancies	Build multiple redundant systems, especially for navigation and classification (e.g., camera, mic, speaker, GPS, lidar, IMU, and 3D camera) and total system shutdown (including remote and on-system kill switch).
6. Operator handover in uncertain contexts	Enable the system to communicate with the operator or to handover controls to the operator when facing uncertainty or issues that it cannot resolve.
7. User interface (UI) to support operator decision-making	Employ bounding boxes to create visual separation between target and background and numerical confidence metrics to support decision-making.
8. Ethical governor	Design, implement, and test an ethical decision-making framework that clearly explains system limitations to users.
9. De-escalation capabilities	Equip system with high-precision lethal and non-lethal weapon systems. Prioritize de-escalation via (1) communications; (2) non-lethal engagement; and (3) precision targeting to wound rather than kill.
10. Spatial rules-based framework for autonomous behavior	Implement a spatial framework allowing operators and commanders to define operational zones where levels of autonomy are permitted, obligated, or prohibited. Current zone recs are as follows: no-go, non-combat, combat, and liminal.
11. Visual and audio operator feedback	Equip the system with real-time, low-latency, high-fidelity cameras, microphones, and speakers.
12. Post-mission review and continuous improvement	Log mission data and conduct post-mission reviews to improve model performance and to identify and mitigate errors and unintended biases.

While the practical application for many of these findings will be realized through an algorithm or component, the architecture should facilitate their implementations, and it should be obvious how operators can address them. For example, with respect to "human authorization of force," developers must consider how to include the human in such authorization, how the human ultimately provides authorization, and how the communication occurs.

3 Quality Attributes

Reference architectures do not show actual systems but are intended to show how properties, such as quality attributes, can be promoted or inhibited in a more specific domain context. Consequently, some of these attributes refer to the reference architecture (e.g., modifiability), rather than an anticipated realized architecture (e.g., testability). To clarify, the reference architecture should offer options for modification, but, regardless of those modifications, the resulting instance architecture should be testable. In the discussions of each attribute that follow, we describe which attributes should form part of the reference architecture and which ones should form part of the instance architecture.

The following quality attributes express goals that the reference architecture should meet, or they correspond to goals of an instance architecture that the reference architecture should promote. They are listed in the following sections and ordered by priority, from highest to lowest.

3.1 Testability

In section 1.1, we discussed the background of the project in which this architecture was conceived. One of our primary goals was to understand whether different system designs would impact testability either positively or negatively. When we decided to make our own architecture, we chose to promote testability to emphasize the types of features and characteristics that testing, evaluation, validation, and verification (TEVV) personnel should look for and how they could leverage these features and characteristics for testing. This quality attribute aligns with the RAI principle of *reliability*.

3.2 Observability and Transparency

Observability and transparency provide the ability for developers, testers, and maintainers to understand what is going on in the system at various stages of the lifecycle. For development testing and evaluation, it is important to understand what data the ML components are ingesting as well as the data quality and consequent confidence scores. During operational testing, this information also helps testers understand how the system perceives and interacts with the operational environment and whether the operational environment matches the development environment.

During runtime, it is especially important that the autonomous system communicates how the components are performing and why the system is making the decisions it's making. This level of transparency requires an effective UI to communicate with the operators.

This attribute is also directly motivated by the RAI principle of *traceability*; during deployment, it supports the principle of *governability*.

3.3 Usability (Interaction Capability)

"Usability" is a commonly used quality attribute name. However, ISO/IEC 25010:2023 notes that "usability" has been replaced by the term "interaction capability." The definition of "interaction

capability" offered by ISO/IEC covers the interactions between user and system and includes the capability to be interacted with and the information exchange it entails, highlighting both behavior and information concerns [ISO/IEC 25010:2023].

The RAI principles and tenets all deeply require such a capability, whether it be through developing "warfighter trust"— the second of several important tenets outlined in the memo "Implementing Responsible Artificial Intelligence in the Department of Defense" [DoD 2021] —or through the design and development processes supporting the principles of *traceability* and *reliability*.

In Section 2.1.3, we discussed UI requirements, which are also central to the question of usability. For LAWS usability, developers should consider warfighters, commanders, maintainers, and testers as key stakeholders in its design and execution.

3.4 Maintainability

Due to the rapid growth of ML technologies, we expect that LAWS will require significant updates to their models and supporting infrastructure, especially as more advanced technologies become available. Additionally, some requirements may levy significant maintainability needs on the system such as rapid updates during deployment or specific component replacement like swapping out a model.

3.5 Modifiability

ISO/IEC 25010:2023 defines modifiability as the "capability of a product to be effectively and efficiently modified without introducing defects or degrading existing product quality" [ISO/IEC 25010:2023]. We use this definition because it focuses on the ability of the system to be modified without introducing defects or reducing quality. As we do with modularity (discussed next), we want to emphasize defects and qualities related to runtime concerns rather than change independence, which is a concern during development.

For this reference architecture, our goal is to enable a decomposition such that the system can split a component's responsibilities into two components in a pipeline. Or we want to enable merging two components into one in a combined interface while continuing to promote the other quality attribute goals. For example, in the perception, targeting, and response (PTR) chain, a single model—rather than two models—might handle some perception tasks together, as shown in the current organization.

We expect that this reference architecture will not be implemented in its entirety. Developers should choose a decomposition that supports some anticipated technological advances rather than trying to cover a wide variety of circumstances.

3.6 Modularity

ISO/IEC 25010:2023 defines modularity as the "capability of a product to limit changes to one component from affecting other components" [ISO/IEC 25010:2023]. We use this definition of modularity

to emphasize the independence of components and the isolation of changes. In general, in this reference architecture, we want to minimize coupling between the components and maximize internal cohesion to promote testability—our top-priority quality attribute as discussed in Section 3.1. As we mentioned above in Section 3.5 on modifiability, modularity is more closely related to development-time concerns rather than runtime concerns.

This reference architecture should promote modularity so that many different providers can deliver clearly defined, independent components for ease of development, testing, delivery and project management.

3.7 Security

In this reference architecture, we do not consider any special additional tactics for security beyond those that one would expect for a fielded system that contains controlled unclassified information (CUI) or classified information. In general, we expect that the reference architecture's highly decomposed nature will allow for independent security assessment, internal intrusion detection, and appropriate runtime checks.

3.8 Performance

We have no detailed performance requirements for this system except that it should meet as-yet undefined mission needs. While we haven't introduced any specific patterns or tactics to promote performance, we have done our best to achieve our other goals without inhibiting performance. We clarify what we mean by striking this balance between meeting goals and not inhibiting performance in Section 4.

3.9 Portability

We did not design this reference architecture for any specific platform, including hardware or software. We did not employ any specific portability tactics. Instead, we have kept the architecture platform agnostic to defer as many decisions as possible to the actual implementation instance. The architecture should minimize requirements about the invocation semantics but should explain how invocation semantics impact various design structures.

3.10 Availability

Ultimately, system availability will be highly important for the final system. The reference architecture should not explicitly inhibit availability, and it should not employ specific availability tactics. We expect that systems influenced by this reference architecture will attain availability through lower-level redundancy patterns, hardware choices, and so on.

4 Design Decisions, Styles, and Patterns

This chapter describes the initial design decisions and the results, styles, and patterns that we used to make this architecture. Using well-understood and accepted patterns facilitates engineering and development and contributes to the RAI principles of *responsibility*, *traceability*, and *reliability*.

4.1 Conduct Governor and Conduct Dissemination

As described in Section 2.3, the decisions and actions that LAWS makes and performs must adhere to the ethics chosen during deployment and operation. This includes, but is not limited to, fire control and movement.

Independent of the actual ethical model or rules chosen, the system must have a way to ingest, store, and reason over a set of formal rules of conduct. This reference architecture assumes no specific ethics but makes provisions for the configuration of a set of conduct rules that resides in a component called the *conduct disseminator* (CD), which is accessible to all other components (we outline the functions of the CD in Section 5.7). This accessibility implies the existence of a complete and unambiguous set of rules from which to create a formalism or technical specification that can be used by the system.

A centralized authority on the rules of conduct is necessary but not sufficient because we must also determine how each runtime component that makes up the system interprets and translates each part of the conduct rules. In current systems, there are many components in the perception and targeting chain for dealing with different aspects of sensing. Each of these components requires different details to control their behaviors. This proposed architecture covers these needs in a variety of ways. For example, the CD component covers ingestion, storage, decompositions, and dissemination. The decomposition functionality converts the higher-level rules of conduct into the specific configuration values needed by each component as described in this reference architecture. The mapping of conduct rules to components will probably not be trivial because one rule (as we explained in the example in Section 2.3.1) will most likely require the configuration of different validators, checkers, and models in concert. This system of ingestion, decomposition, and dissemination provides strong support for the RAI principles of *traceability*, *reliability*, and *governability*.

4.2 Situational Awareness Blackboard (SAB)

Decision-making performed by the LAWS in a complex combat environment relies on balancing many situational factors that operators and others may need to track over time. At the center of the decision-making process is a data store called the situational awareness blackboard (SAB), which contains all the data and metadata needed for each component to perform its decision-making. A centralized blackboard strongly supports the RAI principle of *traceability* because it illustrates how requirements manifest during runtime.

We designed our architecture to function as a blackboard system because the *blackboard pattern* describes a useful organizational structure for reasoning about how the system makes decisions [Nii

1986]. This component stores a variety of information from basic data such as imagery to any knowledge gleaned from this data that another reasoning component might need, such as bounding boxes, image segmentations, key points, motion tracks, entity identification, and more. While we don't require following the blackboard pattern in terms of the specific subcomponents, whatever is implemented to store and improve the data must address the same set of problems that the blackboard pattern addresses.

Although the system can pass this sort of data from component to component, having a central storage location is a key design characteristic that supports the patterns of ongoing data enrichment, system extensibility, increased monitoring, and inspection by operators for their situational awareness and their ability to understand the state of the LAWS.

The SAB and the CD are needed by all other components in the system and are usually referred together as the SAB&CD. We outline these components in the following section as well as Section 5.7.

4.3 Conduct Governor Wrapper Pattern

The facts that ML-enabled systems are data intensive (during training and deployment) and return probabilistic outcomes represent two key challenges when incorporating ML. Sensors require extra data engineering and processing to clean and prepare data in input, while the probabilistic outputs require some engineering processing logic to interpret the predictions against possible courses of action. To mitigate challenges with input data and output confidences scores, we recommend a design pattern that surrounds the ML component with functionality that validates all inputs and outputs with the validation results provided to other components to provide better context for their decision-making. Lastly, due to the critical nature of LAWS and the nature of ethical decision-making, we also recommend constraining the output of each ML component. Figure 1: Governor Wrapper Pattern illustrates this design pattern, commonly referred to as a "wrapper" pattern.

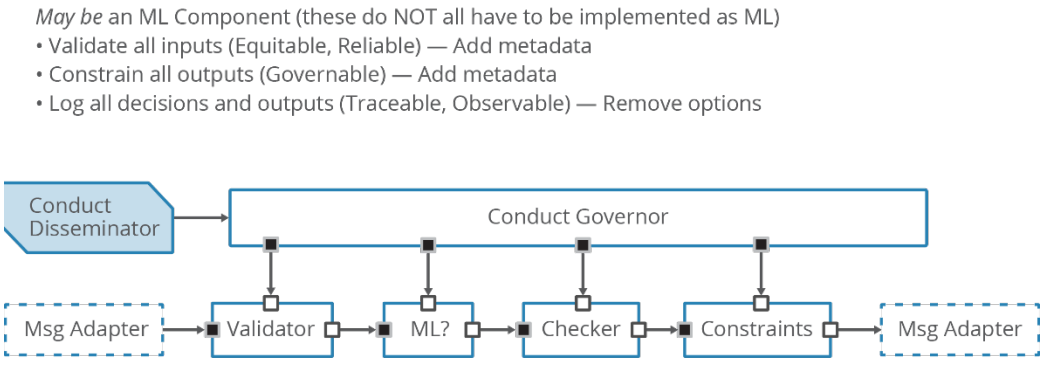

Figure 1: Governor Wrapper Pattern

Each conduct governor component interfaces with the CD, a component that occurs alongside the SAB. The CD contains all the RoE and LoW and knows how to apply them, as needed, to the specific component. For example, if an RoE specifies that certain structures are protected entities, then the CD

may communicate that a particular object detector class result should be marked as an invalid target. Or if these sorts of targets are hard to identify, it may increase the confidence score that the system needs before it is cleared to take an action on these types of objects. The governor component applies the rules and settings, as appropriate, to the validators, checkers, and constraints specific to those component types. This process provides strong support for the RAI principle of *governability*.

This reference architecture does not specify a particular invocation pattern (see Section 4.4 below). However, the Msg Adapter shown in Figure 1: Governor Wrapper Pattern provides an example of how the pattern could be introduced into an event-based system such as a robotic operating system (ROS).

4.4 Interaction Styles

This reference architecture does not specify a particular interaction style such as call–return or an event bus. We have intentionally left this specification out because we want to promote portability as much as possible, and the tradeoffs between various styles are highly dependent on the hardware architecture. Event-based architectures, while promoting extensibility and performance, lead to a highly decoupled system in which messages can be lost, and they can also inhibit direct component binding [Timperley 2022]. Direct call–return systems, with their higher coupling, increase direct responsiveness but may inhibit performance. However, direct call–return systems promote stronger guarantees that the system reflects events in real time compared with queued or event-driven systems.

4.5 Hardware, Processes, and Threads

This reference architecture does not specify any hardware, process model, or threading strategy. As mentioned in the discussion of the performance quality attribute (Section 3.8), deciding which hardware, processes, and threads to use depends on the mission constraints, and such considerations are beyond the scope of this document. The corresponding structure of the realized architecture will need to reflect those constraints, which we expect will also drive decisions regarding model architecture, model tuning and pruning approaches, and scheduling.

There are significant research and development efforts around the topic of hardware–software co-synthesis. This research might be helpful in making hardware decisions. Component decomposition based on technology is not available at the time of the writing. Consequently, we do our best to not limit the types of hardware, processes, and threading models that engineers might decide to use to build the system.

4.6 Data Annotating Versus Modifying (Checking Versus Constraining)

When realizing an architecture guided by this reference architecture, the architect will need to choose a balance between adding annotations to the data and modifying the data in flight. For example, if architects determine that input data does not lie within a particular gamut, they must decide between

simply annotating that detail when storing and communicating the data or adjusting the gamut once and storing it with the change applied.

4.6.1 Modification

Modification refers to the decision, after the system notes certain conditions during processing, to make changes to the data or even to omit the data from being passed downstream. For example, if a sensor validator determines that the input instances do not meet a minimum light threshold, then these instances may be rejected from the stream and not passed on. Of course, all anomalies should be noted and logged for later use.

Modification offers increased performance because the data requires no further processing or storage. Therefore, modification can reduce memory usage, communication usage, and CPU usage. In traditional software systems, modification is generally the preferred approach for increasing performance as opposed to annotation. However, because ML-based systems contain many more complexities and nuances than traditional software systems, modification may not be the best approach. The reason is that discarding or modifying data in an ML system might result in a process where properties in the original data are no longer available for other components. These losses eliminate the opportunity for later components to use information from previous sources (historical data, other modalities, etc.) to augment the data as they process it. However, in a case where data is removed, it may be useful to add an annotation or marker about that removal to enable other reasoning that might otherwise not occur because it would have required the removed data.

4.6.2 Annotation

Annotation refers to adding metadata to the data stream based on any conditions the system notes during data processing—such as during validation, application of the ML model, or output checking. We do not specify how the system should add these metadata, except that the system should make all annotations available to all downstream components.

Adding annotations may significantly increase system demands depending on the type of annotation. However, annotating the data instead of modifying it allows components to later make creative use of the data. Or when used in a complex calculation (such as ethical decision-making), any ambiguity or lack of confidence in the source data may result in a critical deciding factor. For example, object-tracking systems use a sequence of identifications of the same object across multiple frames to create object tracks. Multiple low-confidence identifications, which otherwise may have been discarded, may be combined to provide a higher confidence identification when aggregated. Alternately, if a misclassification occurs due to an adversarial attack (identified due to multiple classifiers voting), retaining the original data is still useful to the other classifiers in the voting set and may be used to identify a security monitor of the attack.

4.6.3 Final Determination

The question of whether to modify or annotate data directly impacts performance and capability. In practice, we expect systems to combine modification and annotation at varying degrees throughout the system to meet performance requirements and to allow for more complex reasoning. For example, sometimes data might be simply annotated when a validation is suspicious, or it might be entirely rejected when the data exceeds more extreme tolerances. The data might be annotated with a confidence level when it occurs above a certain threshold, but it might be entirely discarded when it occurs below the threshold.

Ultimately, this discussion highlights the importance of fully specifying the performance and functional requirements of a system (e.g., via a system requirement specification) and how different conditions should be handled. For example, developers need to decide how the system would handle a classification error due to sensor obfuscation differently than an error resulting from an adversarial attack. Because such decisions can have a tremendous impact on the capability and performance of the system, it is important that they are informed by well-executed, systems-engineering processes like those conducted in system safety assessments such as fault trees, failure mode, effects, and criticality analysis as well as system responses to short- and long-term failures and security analyses.

5 Architecture Description with Component Views

The following architecture description covers a variety of abstract systems and component relationships to identify the key decomposition decisions that support the requirements and quality attributes described above.

As mentioned in Section 4, we do not identify specific interfaces, formats, or interaction patterns in this reference architecture. Rather, we focus on the runtime organization and decomposition of human integration, perception and response, and navigation systems and their interactions.

In the following sections, we offer component views of the system. Each diagram uses the symbology outlined in Figure 2.

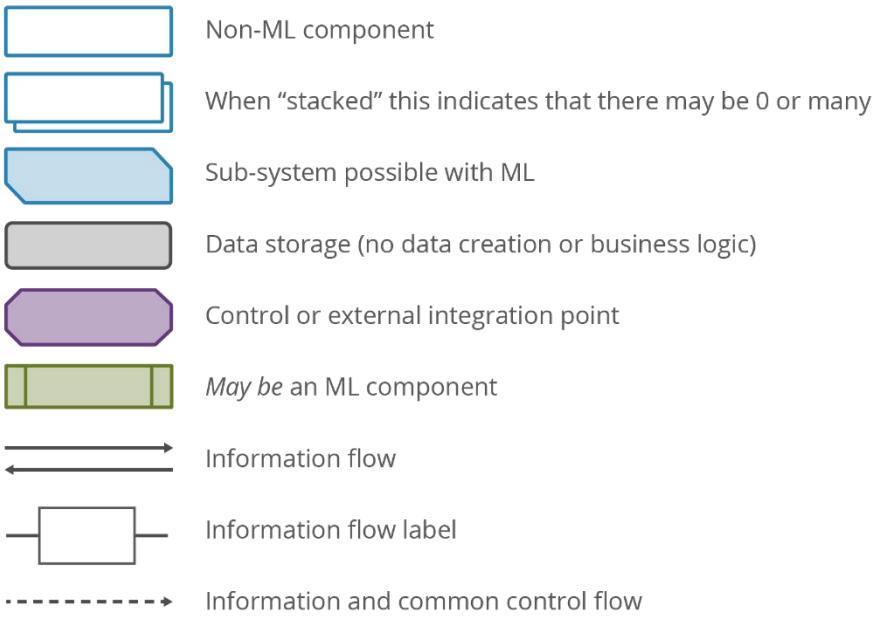

Figure 2: Legend

The symbol labeled "may be an ML component" embodies the wrapper pattern as described in Section 4.2 and may or may not contain ML components with associated traditional logic. It is unclear at this time which of these systems will consist of predominantly traditional algorithms, symbolic AI, ML, or ensembles of all three. We expect that combinations may vary depending on the available hardware resources, system requirements, and project schedule and cost.

Figure 3 shows the highest level of organization of the major component types. This decomposition is highly influenced by our need to reason about the decision-making chain for LAWS. The SAB&CD play a central role in the system and support all other decision-making. We also group the entirety of the PTR chain into one subsystem to emphasize the organization of those components and how they promote the conduct governor pattern and interact with it. Contrariwise, from a conceptual

perspective, navigation and localization is a well-understood area with autonomous systems development and is therefore allocated into one subsystem. The sections that follow explain each subsystem in detail.

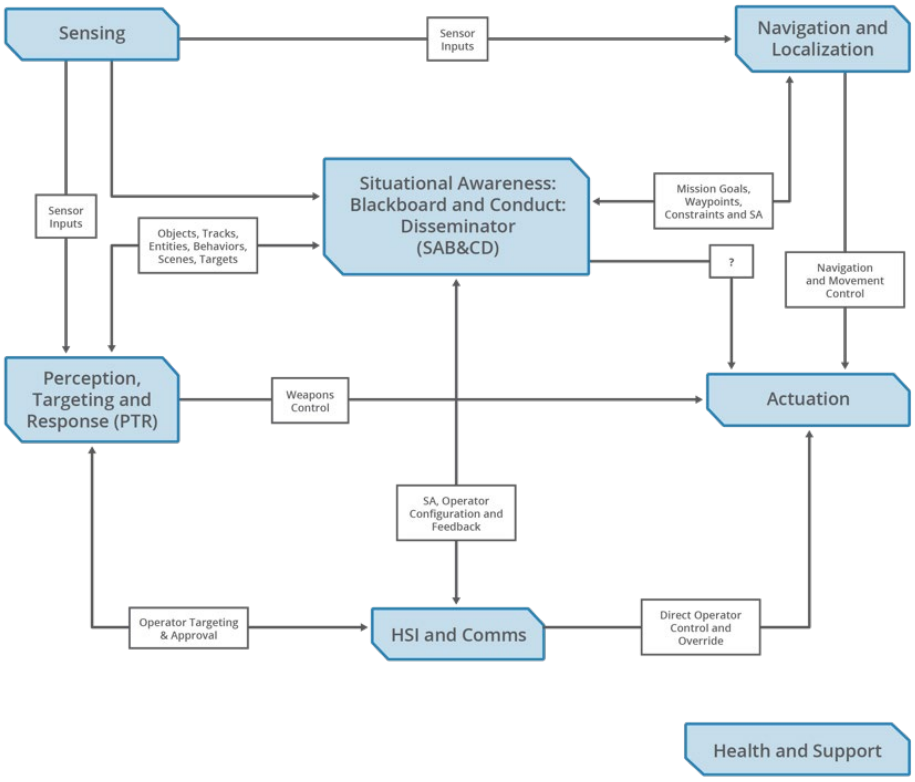

Figure 3: High-Level System View

5.1 Actuation

This area covers the set of hardware interaction points for engaging movement systems, sensor pointing, weapon systems pointing, firing, and all safety interlocks.

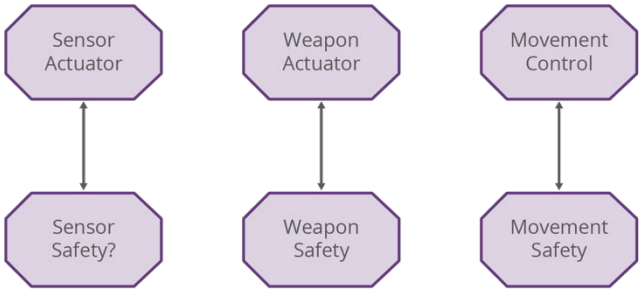

Figure 4: Actuation

For the purposes of this reference architecture, the components outlined in Figure 4 represent the existing, well-understood concepts of hardware actuation, and they serve to communicate how the ML components interact with the hardware.

As part of general safety and automation practices, each actuator is accompanied by safety controls to prevent harm and other unintended effects. Safety engineering practices offer significant insight into such safety controls, but discussion of these practices is beyond the scope of this document. We expect that safety components will include a wide variety of internal and external sensors such as motion limiters and collision detectors.

However, we also expect that additional requirements will be developed over time to increase user trust and that these will manifest in additional safety controls such as proper squad-level weapon handling. We don't expect such controls to directly result from the impacts of ML or the use of AI. Rather, they are likely to evolve to address the additional mission requirements that AI and ML will allow.

The flight limiters provided by aircraft systems are a good example of how developers can include robust safety in a system from architectural and design perspectives. These safety controls work when an autopilot (or a reinforcement learning system) generates commands to the flight control system. These commands go into the flight limiter system to ensure that the autopilot does not violate safety practices such as flying the aircraft into the ground, flying at too steep an angle of attack, or exceeding g-force limits [Bowers 2023].

5.2 HSI and Communications

LAWS often operate in complex, contested environments and require significant operator interaction. In addition, LAWS should provide feedback and direction to external users and potentially targets. Figure 5 outlines the components needed to deliver on these requirements.

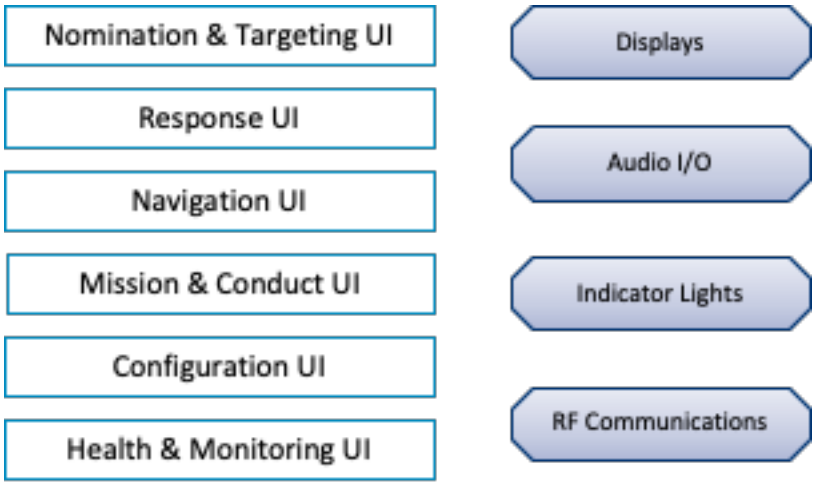

Figure 5: HSI and Communications

Like other hardware interaction points, this reference architecture does not introduce any approaches or new characteristics to these components. Their purpose in the diagram is to provide clarity about how ML components interact with the hardware.

5.3 Health and Support

The health and support components include items that perform general platform health monitoring for elements such as power, temperature, hardware diagnostics, and supporting tools that record telemetry and other logs including "black-box-type" functionality. For high-level functionality, we do not expect this monitoring to be different from existing complex systems. Any implementation will need to account for a significantly larger volume of telemetry and logging information for all the additional sensors, validators, checkers, ML algorithms, decision-making components, hardware, and power.

5.4 Navigation and Localization

The set of components outlined in Figure 6 covers the goal planning, route planning, trajectory planning, and other platform mobility requirements. This structure represents an amalgam of existing autonomy implementations but does not comprise a specific implementation. It demonstrates greater use of ML than we normally see in existing implementations for achieving autonomy.

The general purpose of this set of components is to translate mission goals and limits into robotic movements. The components achieve such movements through successive refinements of higher goals down to routes, waypoints, trajectories, and finally into actuator motion.

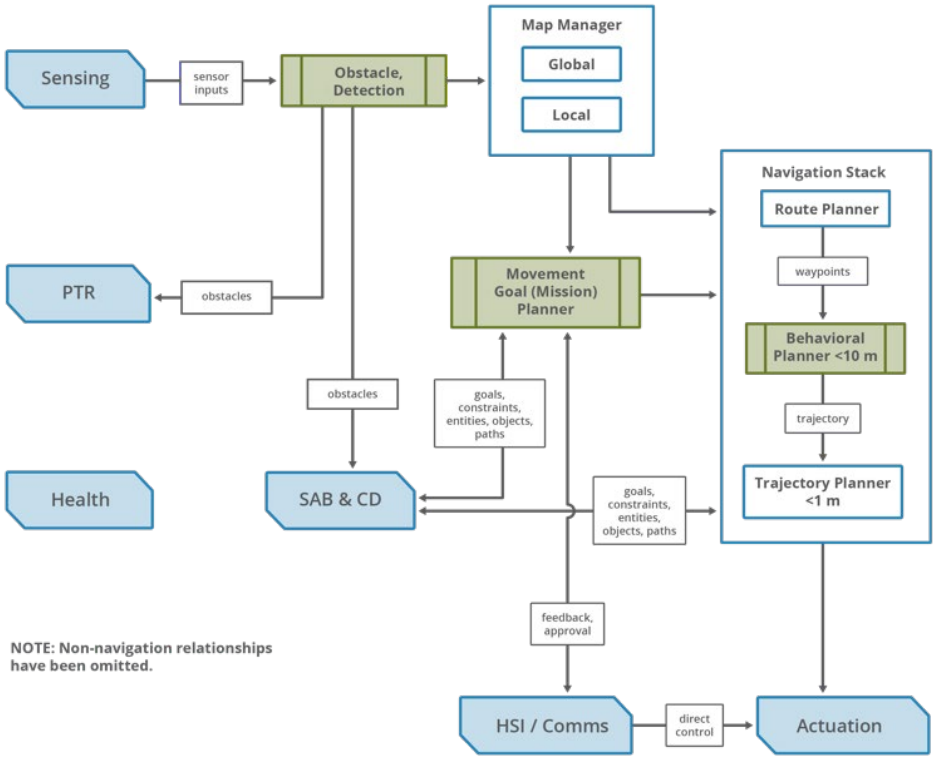

Figure 6: Navigation and Localization

5.4.1 Behavioral Planner

The behavior planner consumes course-grained waypoint information and computes the finer grained movements to achieve the next individual waypoint. This process produces a trajectory.

5.4.2 Map Manager

The map manager is responsible for configuring a geospatial global map including landmarks and other reference features. The local map portions result from what the navigation systems determine about the local terrain from various sensors. This data is separate from the SAB&CD due to representation, access needs, and different types of data. While the SAB&CD may include some types of obstacles, they usually reflect a PTR perspective rather than a navigational perspective.

5.4.3 Movement, Goal, and Mission Planner

This component is responsible for the highest-level navigation planning. It considers rough mission goals (e.g., survey an area, protect an area, perform search) together with the current map and conduct constraints (from the conduct disseminator) to identify goal locations. Commonly, this component is implemented as a behavioral tree. but its development remains an active area of ML research.

5.4.4 Obstacle Detection

Obstacle detection for navigation might leverage the perception components that are part of the PTR chain, but the goals of object detection for navigation may be different than those of PTR. We explicitly call it out here because of the possible variations and the need to directly couple object detection with mapping outputs.

5.4.5 Route Planner

The route planner computes coarse-grained route waypoints to meet the navigation goals produced by the mission planner. As with the goal planner, route waypoints must adhere to the conduct constraints that prevent waypoints from violating movement limits such as traversing a protected area.

5.4.6 Trajectory Planner

The trajectory planner consumes trajectory information and computes the appropriate control systems changes to achieve the trajectory. The process must take into account the physical conditions, environment, and so on.

5.5 Perception, Targeting, and Response (PTR) Chain

The grouping of components in Figure 7 includes basic object detection, path tracking and prediction, entity recognition, target nomination, determination of the appropriate course of action, and engagement of any effects.

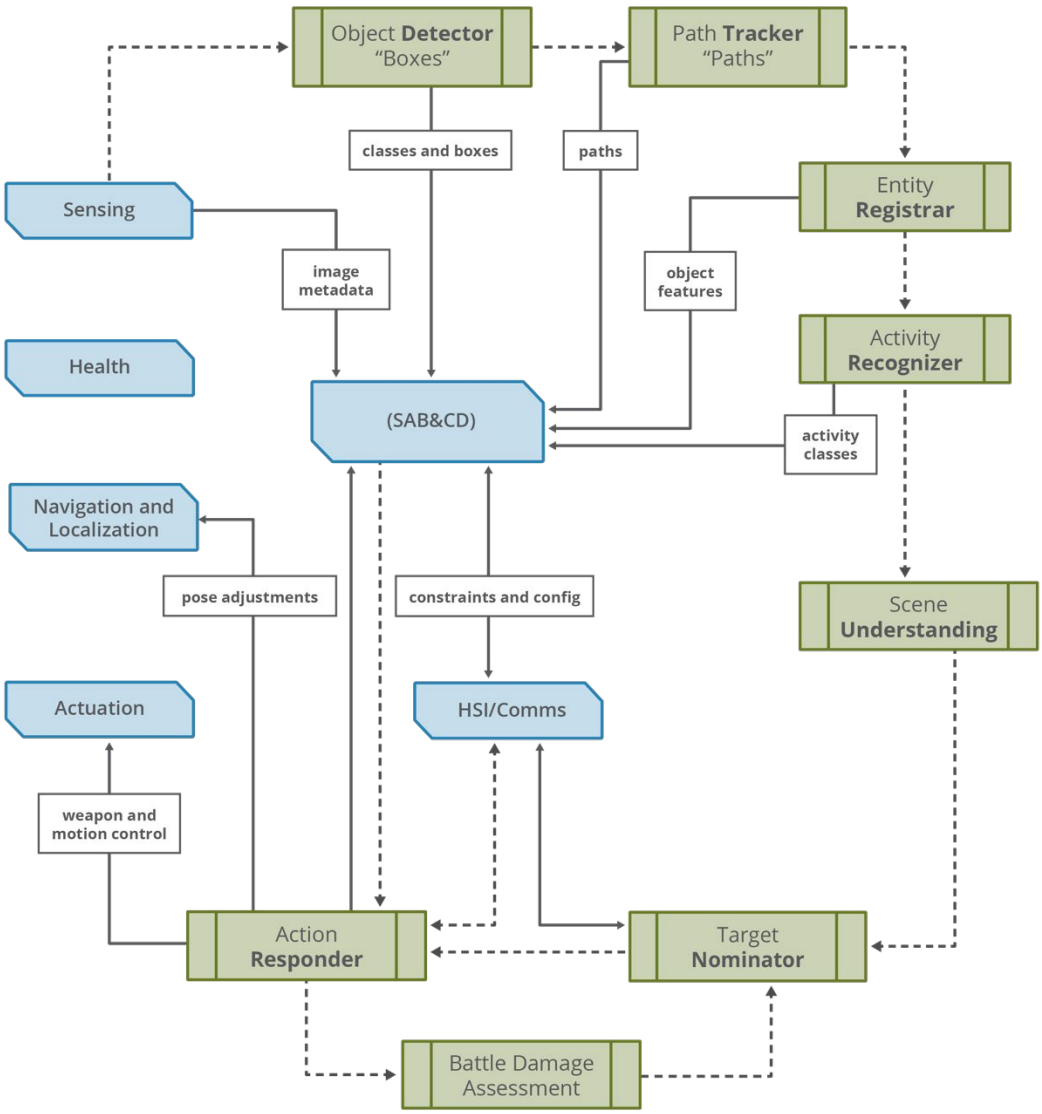

Figure 7: Perception, Targeting, and Response (PTR) Chain

This decomposition is intended to evoke a variety of different functions to progress from input instances (e.g., imagery) to an understanding of the environment (e.g., scene) for use in targeting and other decision-making. We created this decomposition to examine many of the different challenges that ML tries to address. Developers might choose to merge some of the capabilities provided by a set of components in this into a single component in particular implementations. For example, some of the initial object detectors such as Region-based Convolutional Neural Network (R-CNN) consisted of two parts: a region proposer and a classifier. In later detectors, such as Single Shot Detector (SSD) and You Only Look Once (YOLO), these two tasks were combined into one model. In the future, we expect that many of these steps may occur together or in a different order than we show. For now, we hope that this decomposition helps people understand the challenges posed by perception and targeting and how we might address them in a reasoned way.

Table 3 provides more information about these components and their function.

Table 3: Quality Attributes, Component Multiplicity, and Interaction Style for PTR Chain

Quality Attributes Impacted	This higher level of decomposition provides for *testability*, *modularity*, and *transparency* by providing clear separation between ML challenges and by placing clearly distinct outputs into the SAB. Inputs and outputs should follow specified formats, and they can be inspected for TEVV through the perception and targeting process. If components are combined as explained above, they are still encouraged to provide the appropriate outputs that would occur between the components to the SAB. This decomposition may inhibit *performance* due to the number of "steps" that occur in the pipeline. The API of the CD reflects the modules in the PTR chain and must be adjusted if a different decomposition is chosen. Any changes to composition will affect the various validators, checkers, and constrainers in the system, in addition to the outputs and formats stored in the SAB. These changes may inhibit transparency and testability if intermediate decisions or steps are removed.
Component Multiplicity	Each of the components can consist of more than one technique or approach. For example, object detection or path tracking might make use of multiple approaches to provide more than one output. Implementations are not limited to one configuration, and, in fact, many are required.
Interaction Style	As mentioned elsewhere, we do not specify a particular interaction style such as call–return or the use of an event bus. What is important to note is the rough flow of information from component to component, but also that all the information is accessible via the SAB and can be used by any component in a way that promotes observability and transparency.

5.5.1 Object Detector

This component performs initial object detection with the input stream. Object detection can result in rectangular bounding boxes, rotated bounding boxes, regions, segmentation maps, key points, and more. The primary output of this component is to provide the basic object identification and classification.

This component may also produce initial scene-oriented information such as the following:

- **Inputs**: Inputs include preprocessed data ready for perception, target identification, and action. All input instance frames should be preprocessed to meet input standards.
- **Outputs**: Outputs include object identification data such as boxes (axis-aligned or rotated), regions, segmentation maps, key points, features, and so on. They also include classification objects with confidence scores and metadata annotations from any validators and checkers.
- **Multiplicity**: The component might use one or more detectors to provide multiple (but aligned) output data with varying levels of confidence.
- **Validations**: The component includes checks to make sure input data meets data standards.
- **Checking**: Checks include calibration error detectors such as General Calibration Error (GCE) and others.

- **Constraints**: Constraints include limits on types of detections in a scene. For example, should validators indicate that the data is of poor quality (because of water drops, lighting, etc.), then the system may remove portions of the data to promote performance.
- **Examples**: Examples of detectors include R-CNN, YOLO, EfficientDet, DETR-based models, Segment Anything, Mask R-CNN, CondInst-based, SOLO-based, Mask2Former-based, region-based, and edge-based, among others.

5.5.2 Path Tracker

This component provides semantics for objects that the system is tracking across multiple instance frames to provide continuity of detection. It uses the current estimated position to identify a path based on a predicted path location. Methods for producing these results might differ in the dynamics models (i.e., they make different assumptions about motion).

This component produces the following information:

- **Inputs**: Inputs include detected objects in the current frame and tracking of previous objects.
- **Outputs**: The component matches objects in the current frame to previous frames or predictions of objects in future frames.
- **Examples**: Examples of trackers include Track RCNN, MOTSNet, PointTrack, and Track-Former, among others.

5.5.3 Entity Registrar

While the path tracker identifies paths based on locations of objects and predicts their previous or subsequent positions, it does not actually consider whether the object looks like an object in a previous frame. This component extracts features from particular objects to create the concept of an "entity" to augment tracking and to ensure that the object the system is tracking is the same from frame to frame. This task is particularly important in complex environments where the paths of objects cross and one occludes the other. The features used in entity registration ensure accuracy of the tracking.

The process of registration is essentially as follows:

- Extract features from the objects in the current image.
- Use some similarity metric to compare the features between current objects and previously registered objects.
- If the similarity is "close enough," then the system matches the current object and the registered object. (Components might calculate "close enough" statistically or by using some threshold, or they might also use a time metric.)
- Optionally, the system registers new objects or deregisters old objects.

Some of the feature detection capabilities may be available from the object detector for performance reasons, so there is some potential for increased coupling between these two implementations.

Generating features for all detected objects might inhibit the system's performance. However, since we are not interested in tracking every object, especially as some objects don't move, it might be unnecessary to generate features for all objects.

This component produces the following information:

- **Inputs**: Inputs include object detections and previous entity identification information.
- **Outputs**: The component creates new and updated entity features and identifications.

5.5.4 Activity Recognizer

This component can use the motion of the object (including orientation) to determine properties about it. For example, it can use key points from successive frames to determine behavior of individuals such as aggression or surrender. Understanding behavior is critical for a fully featured ethical governor to recognize situations such as "threatened with deadly force" and "otherwise demonstrates an intent to imminently attack." Because of the complex nature of behavior, this component will most likely rely on more than just a single ML model, or it might not function through a model at all. An expert system might be more appropriate for this component. Regardless, this component will produce predictions about objects in similar ways to other ML components.

- **Inputs**: Inputs include object detections, object paths, and object features.
- **Outputs**: The component identifies behaviors.
- **Examples**: Examples of activity recognizers include PytorchVideo, MMAction, OpenPose, HRNet, and AlphaPose, among others.

5.5.5 Scene Understanding

Often, it is not enough to acquire the properties of a single object to make decisions about how to categorize it or to respond to it. Therefore, its relationship to other objects becomes crucial to make such determinations. This component makes determinations about the object within the context of the scene and its relationship to other objects. For example, the component can determine whether an object is adjacent to other objects of a similar class. Alternatively, it can determine if an object is directing activity toward another object, whether it is responding to activity from an object, or whether it is showing aggression toward a protected entity.

5.5.6 Target Nominator

Joint Publication JP 3-60, "Joint Targeting," defines a target as "An entity or object that performs a function for the adversary considered for possible engagement or other action" [JP 3-60]. The target nominator component is responsible for evaluating objects as targets to engage based on this definition. The component makes the determination of whether to engage based on all available data, including the following: class, path, activity, behavior, situation in the scene, mission goals, targeting priorities, and ethical constraints.

In current weapons systems, this capability functions through the use of symbolic logic, and it might be considered an expert system. Currently, there are no known publicly available ML models that deliver this capability, and such a capability may not be a good candidate to develop using ML.

5.5.7 Responder

The responder elicits actions by the autonomous system based on nominated targets, the CD, and other situational data from the SAB. The responder generates a set of potential responses for all nominated targets, filters any restricted responses from the potential responses (based on the configuration from the CD), prioritizes targets based on the environment and goals, and executes the resulting action.

Some of the component's response generation may require operator approval. Requesting operator approval works through the HSI and communications subsystem, and it may incur delays that may invalidate some responses. Such delays might prompt the responder to constantly reevaluate response options until it receives the operator feedback or until it can choose a different response.

This component consolidates the autonomous response determination. We expect that, in these decision-making situations, there may be conflicting goals that the system must balance. The fact that there may be conflicting goals does not mean that other parts of the system can prevent action based on safety controls (e.g., weapons safety, collision avoidance), but the decision to engage a target must account for all variables in a consistent, auditable, and governable way.

5.5.8 Battle Damage Assessment (BDA)

This component combines historical data with up-to-date sensing and the intended effects to produce a BDA assessment. BDA is currently a hard problem for autonomy. For physical targets, projects such as XView 2 offer a better understanding of the challenges, even though that project is focused on overhead imagery [xView2 2025]. For human targets, little to no guidance is publicly available on quick assessment of target neutralization. We expect this area to be particularly challenging for some time.

5.6 Sensing

The group of components outlined in Figure 8 covers the sensors used for PTR and may also include those used for navigation or localization. Many different sensor configurations are available, with many variations of sensors, validators, filters, data-fusion components, and other processing tools. For example, architects might choose to implement a stereo camera with range finding as one unified device from a vendor or as a collection of independent components.

While capabilities for sensing are constantly evolving, many of the challenges are relatively well understood, and therefore the construction and investigation of these sensor pipelines are not the focus of this architecture. We assume that the system has properly processed any outputs from the sensing components for ingestion by the PTR chain.

The sensing group can have any number of sensors and sensor pipelines that feed into different parts of the PTR chain.

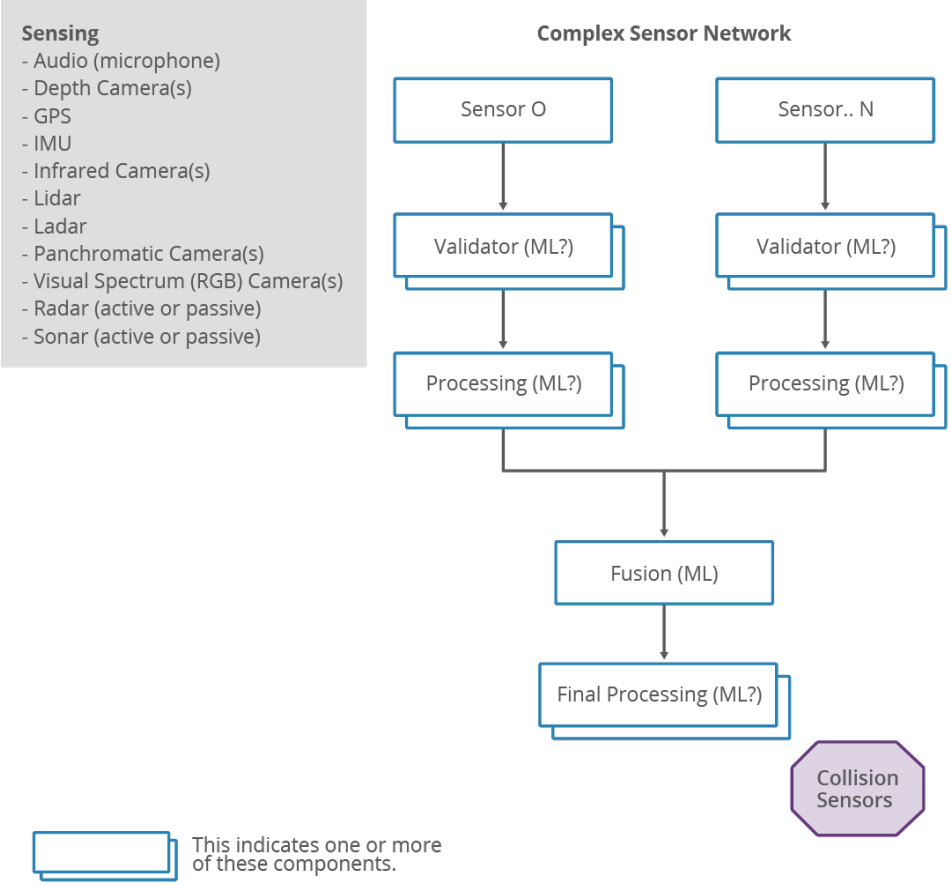

Figure 8: Sensing Components

5.6.1 Sensor

This component represents the hardware interface to the actual sensor. They include any sort of sensor such as cameras (including visual spectrum, infrared, and panchromatic), radar, lidar, ladar, sonar, IMU, GPS, audio, and so on.

5.6.2 Validator

It is common for systems to use a post-sensor validator that can detect issues such as water droplets, lens flare, or even hardware-based diagnostics such as out-of-bounds voltage. These validators may use traditional methods such as OpenCV. Each validator should augment the data stream with its findings.

5.6.3 Processing

After sensor validation, it might be appropriate to filter or transform the input stream before performing any further fusion or post-processing. This transformation may consist of a simple format or object conversion such as tfrecords, RGGB to RGB, HDF5, recordIO, numpy, pandas DataFrames, Spark DataFrames, and so on. Or it might include other augmentation such as whitening, centering, standardization, cropping, resizing, and change of precision, among others.

5.6.4 Fusion

After the system prepares the collection of related sensor data, it should unify it into one format or instance that the downstream components can interpret. This process may involve image alignment as well as augmentation. For example, a stereo camera can provide a single image with an attached depth map.

5.6.5 Final Processing

After the system fuses the images, it needs to prepare them to hand off to the PTR chain. These processing tasks are similar in nature to the processing we discussed earlier, but the results should be perception-ready data.

5.7 Situational Awareness Blackboard and Conduct Disseminator (SAB&CD)

The cluster in Figure 9 contains the situational awareness blackboard and conduct disseminator (SAB&CD), which comprises the central location that informs the system's current understanding of the world, mission goals, and rules of conduct. The SAB contains all the current state information about the surrounding environment including objects, tracking data, entities, activities, and behavior. It also contains any current mission goals as directed by the operator or external commander. The only information about the current state that is contained elsewhere is the terrain and mapping information used by navigation.

The CD contains the current set of ethical constraints that the system must apply when making decisions about how to interpret and interact with the world. The SAB and the CD, although loosely coupled, appear together because any component that needs to access one will also need access to the other.

Figure 9: Situational Awareness Blackboard and Conduct Disseminator

5.7.1 Situational Awareness Blackboard (SAB)

The SAB functions as a blackboard [Nii 1986], but it doesn't necessarily need to be implemented as one. However, the SAB should contain both data stores (labeled DB for databases) and logic to make sense of this data (i.e., relating, organizing, and curating) and to facilitate the interaction of other modules in the PTR chain.

There are four databases listed in Figure 9, but this structure is not required, and implementations may use different organizations. All data and metadata generated from the various components should be stored here.

5.7.2 Conduct Disseminator (CD)

This component stores the rules of conduct and ethical constraints and guidelines provided to the system, and it decomposes them into specific ML configuration settings needed by individual conduct governor components such as the following:

- class weights and biases
- confidence thresholds
- calibration error thresholds
- component input weights
- continuous time in view during tracking
- minimum separation distance between adjacent entities during scene understanding
- response priorities

The logic in this component may be complex and is directly tied to the other components it must support. For example, if the PTR chain uses a model in which object detection and tracking are combined instead of implemented as separate components, then the configuration may need to change.

6 Variations and Alternatives

The goal of any reference architecture is to demonstrate how different patterns and tactics work together to enable a certain set of properties in a particular domain. In this instance, the architecture must demonstrate its support of LAWS.

In this case, when we talk about variations, we don't mean varying how a component is implemented, but how the overall organization of the components and how the responsibilities may be partitioned differently. At this time, we are primarily interested in how the components of the PTR chain may change and how such changes affect the task of conduct assurance. We expect that ML components will take on additional functionality over time and some capabilities might merge, like when two-stage object detectors evolved into single-stage object detectors.

To this end, we provide an example in the following sections of how motion tracking and object registration may be combined.

6.1 Combining Motion Tracking and Entity Registration

As shown in Figure 10, any autonomous system decomposes the tasks of detection, tracking, and registration to promote efficiency, modularity, and ease of implementation, but upcoming models will combine these capabilities. Some examples include Track RCNN, MOTSNet, PointTrack, and TrackFormer [Shuai 2020, Porzi 2020, Xu 2020, Meinhardt 2022].

Combining detection and tracking simplifies the registration problem because the system can determine that tracked objects in subsequent frames are the same object. However, this functionality does not provide the ability to recall objects that have left the frame, so additional registration or entity history capabilities might be necessary to track objects that leave the frame and then return.

Also note that while these methods perform object detection or instance segmentation, they often **do not** identify the class of the objects they track. They only identify a unique ID for each tracked object. Development teams might need to develop additional functionality to also predict class information. The system can perform this activity in parallel, enabling some other performance options.

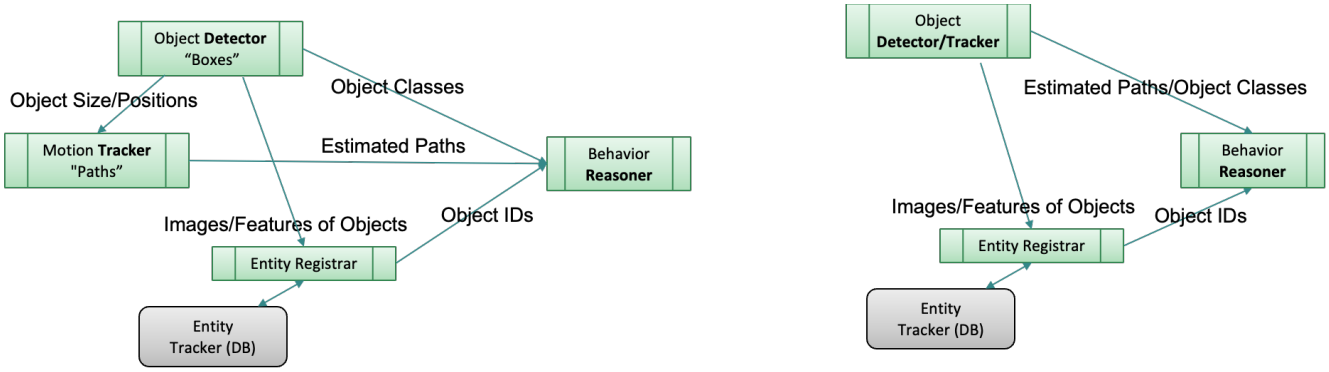

Figure 10: Example of Combining Detection and Tracking

7 Related Works

This section explains how this reference architecture relates to other reference architectures used in industry and how the concepts in this reference architecture can be added to others. Currently, we have only assessed it with regard to UMAA.

7.1 UMAA

The decomposition of components provided by the Underwater Maritime Autonomy Architecture (UMAA) is like those of many autonomous systems. Figure 11 shows the high-level functions and how they are allocated in the UMAA reference architecture [UMAA 2019].

When mapping our reference architecture to UMAA, the key additions in the PTR chain and the CD will introduce additional components in the mission management, sensor and effector management, and situational awareness groups. These groups are marked with a star in Figure 11: UMAA High-Level Functions following sections, we address how developers might allocate the PTR across the existing architecture, but that allocation raises a variety of questions about coupling and performance, so it might be necessary to revise the UMAA architecture to accommodate the extended ML functionality.

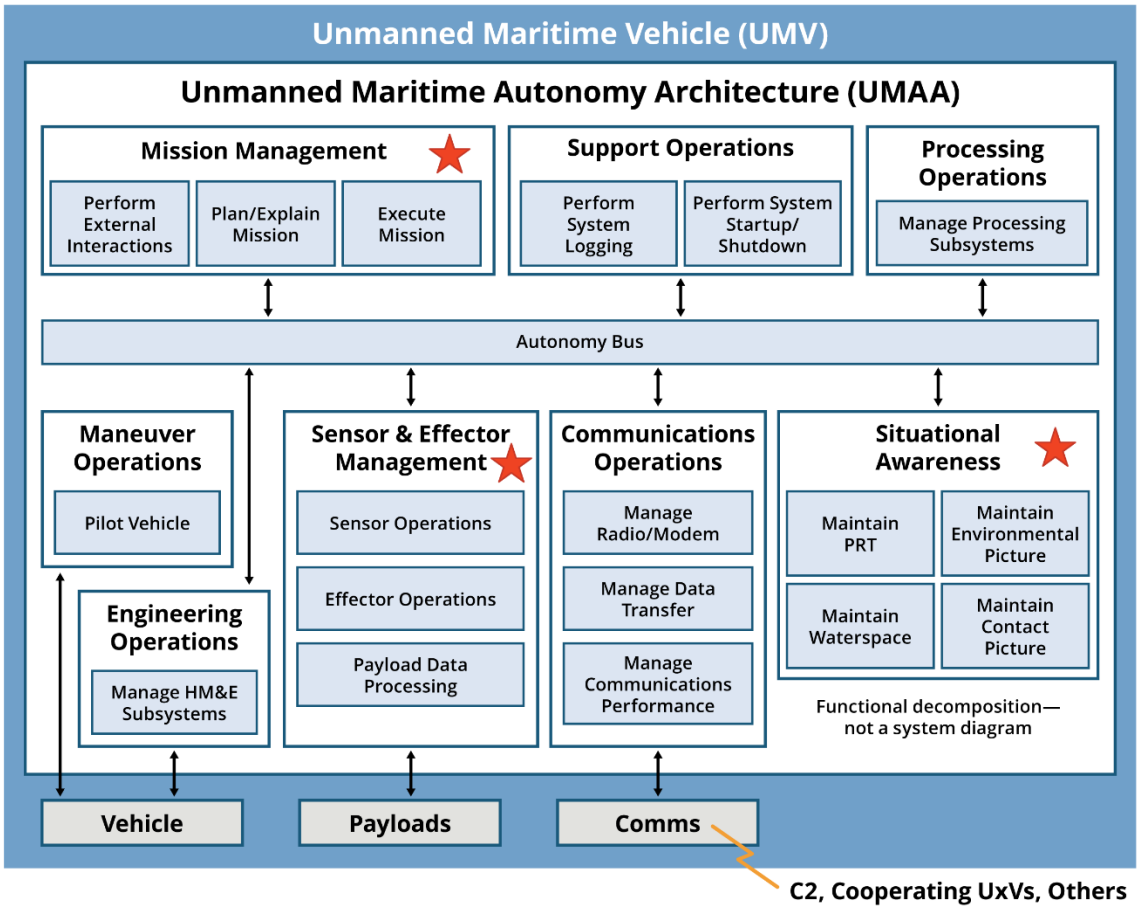

Figure 11: UMAA High-Level Functions

7.1.1 Sensor and Effector Management

Most PTR components will reside in this grouping including the object detector, path tracker, entity registrar, activity recognizer, scene understanding, target nominator, and action responder. Due to performance needs and the benefits of increased coupling, it may be more useful to place the SAB in this group.

7.1.2 Situational Awareness

Based on the description of this component, the SAB might reside here. However, for performance reasons (depending on the autonomy bus implementation), it may make sense for developers to embed the SAB in the sensor and effector group. Developers should perform a thorough evaluation of the performance demands of the ML components that interact with the SAB before deciding where to place the SAB.

7.1.3 Mission Management

Under the existing UMAA description, one would expect the CD to reside in this group because it is a mission-centric piece. Again, performance might be promoted by placing it elsewhere.

Appendix A: Testing the Conduct Governance of LAWS Using a ROS-Based System Proxy

This appendix contains the results of a study the SEI performed to evaluate the reference architecture described in the previous sections of this document. It discusses the insights we gained through the study as well as the practical challenges and tradeoffs involved in implementing the reference architecture. Finally, the appendix offers recommendations for future research and development of autonomous systems that align with the DoD's ethical principles for artificial intelligence (AI).

Study Introduction

Integrating machine learning (ML) components and pipelines into lethal autonomous weapon systems (LAWS) is still a nascent discipline. Almost all guidance and recommendations on proper engineering for such systems is theoretical and vague. For the Calibrated Trust Measurement and Evaluation (CaTE) pilot, we wanted to ensure that our recommendations or guidance were actionable and readily applicable for the testing, evaluation, validation, and verification (TEVV) community. In this spirit, members of the CaTE project team worked to instantiate the program's proposed reference architecture.

Over six months, the CaTE program adapted a simple Robot Operating System (ROS)-based robotics system to represent, as closely as possible, the architecture described by the reference architecture for assuring ethical conduct (RAAEC) in LAWS that we outline in this document. Following guidance from the RAAEC, we developed and integrated components of LAWS—primarily a perception, targeting, and response chain—into an existing ROS-based mobile platform. To understand the influence of AI on the system, we included several ML models to perform detection, classification, and tracking tasks.

Through six months of iterative development and testing, this research demonstrated that the RAAEC provides a structured yet flexible framework for enforcing conduct governance in autonomous systems. Key findings suggest that RAAEC-mandated components contribute significantly to system transparency, observability, and testability, while implementation flexibility allows for adaptation to different operational contexts. Further work is needed to refine human–system interaction (HSI), expand considerations for the user interface, formally evaluate quality metrics, and investigate the system across a full lifecycle.

Test System Details

The proxy system we adapted for this project is a Clearpath Robotics TurtleBot 4 (TB4), which is a commercial, off-the-shelf robotics platform designed as an open source, research-ready system. It features a mobile base, 2D lidar, and a depth camera. It utilizes the Robot Operating System 2 (ROS2) and includes a Raspberry Pi 4 as its onboard hardware controller. The manufacturer's system code provides basic functionality and APIs to extend the system's capabilities with custom code.

To extend the TB4's functionality to support the CaTE pilot's research, we developed an autonomy architecture with a central controller and multiple supporting subsystems, some of which contain ML models. We implemented this approach as a collection of ROS nodes to enable modular development and flexibility in testing and for integrating additional capabilities, such as automated threat recognition (ATR). The autonomy architecture distributes functionality between the TB4's onboard Raspberry Pi and an external laptop connected to the system via wireless communications. The entire ATR subsystem is hosted entirely on the external laptop, and it leverages the laptop's computational resources for image processing and model execution. Overall, the architecture provided a valuable platform for evaluating and testing governance and control mechanisms for LAWS.

Test System Design

This project followed the RAAEC document to develop a system on the TB4 platform that could demonstrate the viability of the reference architecture. The project focused on connecting key elements of the reference architecture, like the conduct disseminator (CD), situational awareness blackboard (SAB), and conduct governor wrapper pattern.

The RAAEC outlines several possible ML models a system might include, but our implementation followed the architecture shown in Figure 12. For the TB4 system, the team implemented a perception chain of detector, classifier, and tracker. We consolidated the elements for scene understanding, target nomination, and response selection into one responder logic tree.

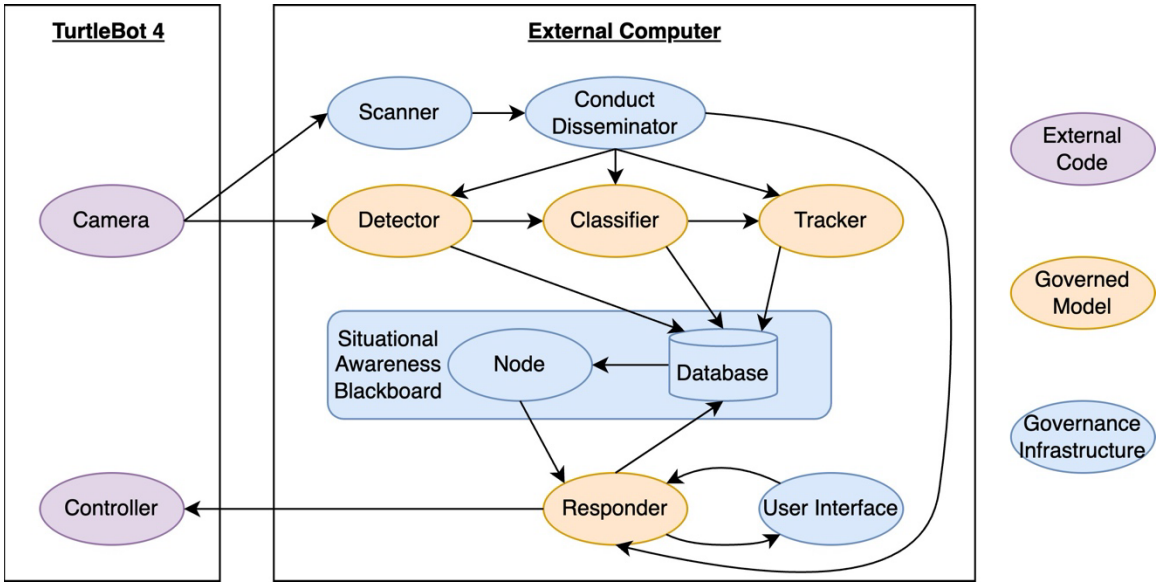

Figure 12: Simplified System Architecture

A scanner element monitored the camera's image stream for features that would trigger the CD to pass new configurations to all the models. Each model wrote data to a central database in the SAB, and a dedicated node read these databases to consolidate data and provide it to the responder. Depending on

the conditions, the responder either informed the user of its actions or requested their permission to perform an action.

The RAAEC outlines several design decisions that system architects will face when developing LAWS and outlines styles and patterns in support of those decisions. In some cases, the reference architecture requires the implementation to follow certain attributes, like including a CD, SAB, and the conduct governor wrapper pattern. With some system features—even with respect to required attributes—the RAAEC acknowledges that system architects have latitude for how to implement them in their specific system and mission context. The following sections detail how we made such decisions for the TB4 system, what styles and patterns we implemented, and the lessons we learned in this process.

Defining Conduct for the Conduct Governor and Conduct Dissemination

The RAAEC is predicated on the existence of a set of ethical values for the system to adhere to. It assumes that the ethical values are sourced from relevant sections of pre-existing, natural language sources, like formal rules of engagement (RoE) or codified laws of war (LoW). The RAAEC acknowledges that identifying possible or optimal sets of ethical values is an open challenge that is outside its scope.

Provided a set of ethical values, the RAAEC requires translating the natural language into formally expressed requirements. Although the RAAEC also acknowledges that detailing such efforts falls outside its scope, development teams can consult work by Arkin for an example of how they might perform the work. Two important reports include *Governing Ethical Behavior: Embedding Ethics in a Hybrid Deliberative/Reactive Robot Architecture* and *An Ethical Governor for Constraining Lethal Action in an Autonomous System* [Arkin 2007, 2009].

The CD component of the RAAEC acts as the central authority for these ethical values at runtime. The CD has two critical jobs, which are as follows:

- Given a set of conduct rules and a specific situation, it must understand what configurations to apply to the system.
- It must maintain awareness of the system's situation during execution and ensure that the appropriate configurations are applied correctly across all components.

Given that the process of sourcing and translating ethical values is still uncertain and undefined, development of the TB4 system began with the creation of a mission definition document. This document provided context for the experiment, and it included details on the operational environment and the ethical values it should exhibit. The mission definition described a set of system behaviors in reaction to a target and how those behaviors would depend on factors like the target's label and confidence, the proximity between targets, and what location (or zone) the system and target are in.

Using the mission definition document, we generated a total of 58 system-level requirements, 28 of which expressed some formalization of an ethical value. The mission definition described several

scenarios and how we expected the system to act in each one. These expectations became system requirements.

One mission scenario that we developed read as follows:

> *The participant sends the system into the combat zone. The system enters the zone. The system scans the zone and identifies Target C first. System requests permission to engage Target C. Participant approves this request and the system engages and eliminates the target. The system produces a report and then resumes its scanning. The system identifies Target D and emits a warning. The participant decides if this de-escalates the situation or not. If the situation is de-escalated, the target is considered surrendered. Else, the system requests permission to engage, which the participant should approve, and the system produces a report. After either case, the zone is considered clear.*

From this description, we generated the requirements listed in Table 4.

Table 4: Select Subset of System Requirements

Number	Requirement
2.1	System needs to recognize what zone it is in (of the 4 possible zones) and behave accordingly.
2.4	System must be able to identify zone based on visual tags within a zone.
3.1	System must have classification to label potential targets.
4.2	System must have object tracking to identify potential targets.
5.3	System must provide autonomous and manual kill chain modes.
5.6	System responses must include ignore, engage, and warn.
7.4	System must be able to request approval from user when appropriate.
7.8	System must be able to interact with the user to provide reports about engagement with targets.

We generated the full requirements list by repeating this approach with each scenario that we created. We then informally inspected and revised the requirements until they satisfactorily described a minimal system that could implement the reference architecture in the mission scenarios. This generation process allowed system development to move quickly and focused effort on the technical implementation of the RAAEC. While commentary about the application of standards is out of this experiment's scope, we want to note that development teams should follow all established standards for requirements development for operational systems.

The TB4 system defined the following four types of zones: combat, non-combat, liminal, and no-go. As a proxy for map, visual, or other localization methods, we defined zones using AprilTags [Olson 2011]. For each zone, the CD pushed new conduct configuration to all models that followed the conduct governor wrapper pattern (see "Application of the Conduct Governor Wrapper Pattern" for details). These configurations could modify the components surrounding a model and their parameters. The CD used ROS services to confirm successful configuration or to detect failures and lost requests.

This implementation proved that the CD supports responsible AI (RAI) principles by configuring components at runtime. The project's timeline and focus required simplifying assumptions—reducing mission complexity and ethical sources to a single, static document—to prevent it from evaluating the process of translating high-level rules to the proper runtime configurations.

Additional research is required to prove that the system supports RAI principles in the generation of conduct governance rules. The development of a real system should follow these strategies for managing ethical rules and tracking them to their configuration level and component implementation.

Designing the Situational Awareness Blackboard

The RAAEC requires the implementation of a situation awareness blackboard (SAB), and it recommends, as the SAB's name implies, that development teams use a blackboard pattern to implement it. This centralized storage location supports many characteristics including traceability, monitoring, data enrichment, and extensibility. As we stated in Section 4.2, it should contain "all the data and metadata needed for each component to perform its decision-making."

We deemed implementation of the SAB as crucial to evaluating the RAAEC. We designed the SAB as a database system with which all system components could interact as they ran. It served as both a form of logging and a facilitator for interactions.

While we depict the SAB as combined with the CD in the diagrams in the reference architecture, we do not require that they be implemented jointly. Therefore, for our study using the TB4 system, we opted to separate these components because all database options already ran as separate processes and did not require additional centralized management.

We implemented the TB4 system's SAB in two parts: as a PostgreSQL database and a situational awareness (SA) node. Figure 13: System Interactions with the SAB depicts these two elements of the SAB and how system nodes interacted with them.

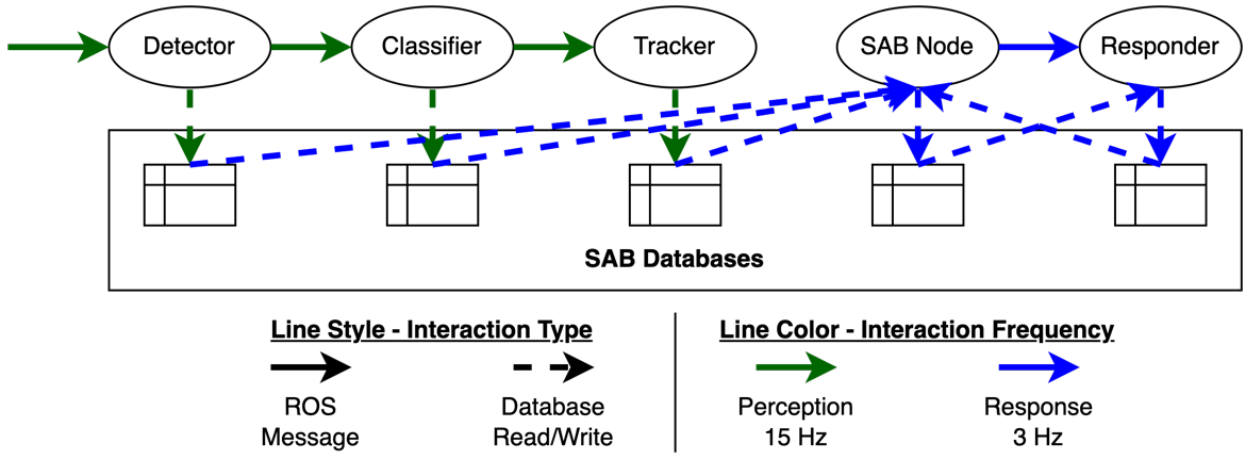

Figure 13: System Interactions with the SAB

The SAB database contained 16 tables that models would write to. We implemented it using PostgreSQL with the TimescaleDB extension because many queries operated on time-series data.

The SAB node would periodically read the models' tables for a specific time range. It would analyze the results to identify error patterns in a single model or correlations between them. The SA node would also combine the model outputs in the results to construct a more complete situational awareness, which the responder model could use.

In the TB4 system, the table rows for this model included information about whether their components submitted logs and whether any errors or warnings occurred (See "Application of the Conduct Governor Wrapper Pattern" for details on the components in the wrapper pattern and "Usage of Data Annotation and Modification" for a description of logging). The SAB node simply counted the errors and warnings for each model and deemed a model's results untrustworthy if the number of either exceeded a set threshold. In this way, it acted as a simple health monitor.

This monitoring was acceptable for the system's testing purposes, but insufficient for more complex use cases. System designers should consider what kind of monitoring their system requires and ensure that the SAB node can extract this information from models and interpret it. The system might require a health monitoring component that is separate from the SAB node and that acts as an ML component.

The SAB database and node also divided the system into two logical parts: perception and response. The perception pipeline ran at the camera's frames per second (FPS)—15 Hz—while the response was tuned independently to 3 Hz. A 3-Hz timer in the SA node would trigger it to read the detector, classifier, tracker, and responder model tables and generate its situational awareness table. With each update to the table, the node sent a notification message that triggered the responder to read the situation awareness table and run the responder model.

We developed and tested these logical parts independently so that development and testing could occur in parallel. By setting a lower frequency for the SAB nodes, the response side consumed less resources because the lower frequency reduced the number of ROS messages and database queries.

The RAAEC should encourage architects to consider where the work of constructing situational awareness should occur. In this implementation, one node was satisfactory to ingest and interpret the data that each model had independently generated. This node also served as a health monitor. But this centralized implementation may be inappropriate for other systems.

Designers should consider what situational awareness means in their system. That means they should consider which components require situational awareness, whether situational awareness means the same thing to every component, and whether they should construct situational awareness from the blackboard by a single source or through each component individually. The results of this consideration depend on the system's design and goals.

Application of the Conduct Governor Wrapper Pattern

The conduct governor wrapper pattern is one of the core elements of the RAAEC architecture. This pattern is applied to any component in the system that operates on data. Often, these elements are ML models, though using ML models to provide these features is not a requirement for the pattern.

We followed the pattern that the RAAEC describes in Figure 1. This pattern is intended to

- provide a standard interface for CD
- apply input validation, output checking, and model constraints in a uniform way
- adapt inputs to and outputs from whatever messaging format the system requires

The input adapter and validator feed into the model, and the model sends outputs to a checker, constrainer, and output adapter. A conduct governor block sits above to coordinate this data flow and apply new configurations received by the CD. This block is also responsible for logging data. The model block typically represents an ML model, but it can generally represent any data transforming process.

Figure 14 shows how we implemented this architecture in the TB4 system. All component objects are composed with a **ComponentLog** object to log information as a **Model** or **GovernedComponent** object runs (see "Usage of Data Annotation and Modification" for details). The **Governor** class contained any number of **GovernedComponents** and, paired with a single model, composes the **GovernedModel** class. **ModelData** and its subclasses provide an abstraction that allows **Validator**, **Checker**, and **Constrainer** classes to operate with typed information while decoupling them from any concrete **Model** class implementation.

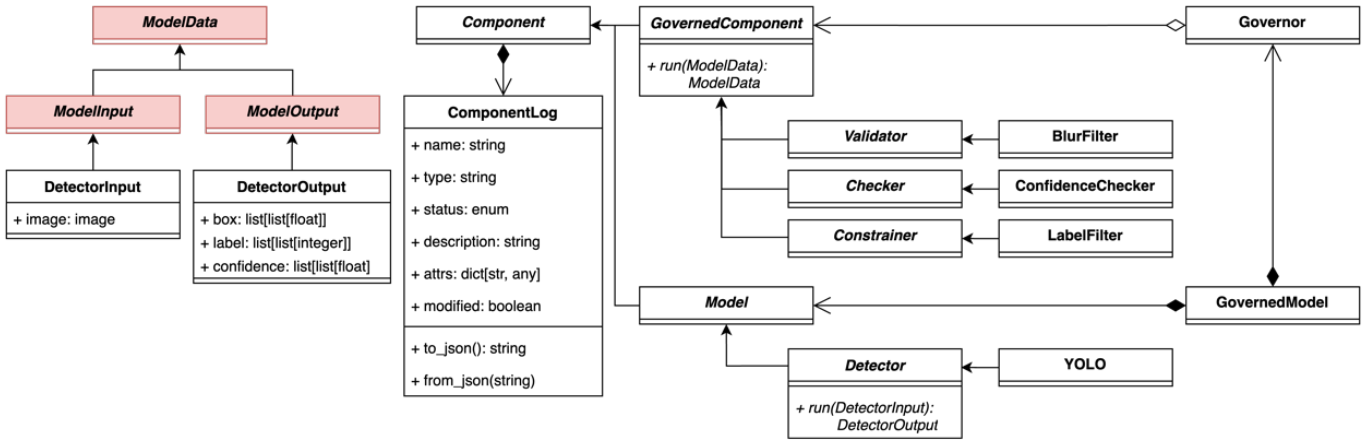

Figure 14: *Implementation UML*

The detector component provides an example of how we implemented a specific model. The detector used YOLO11 as its primary model. The **Detector** class generalizes YOLO by describing its input as **DetectorInput** and its output as **DetectorOutput**. We developed components such as **BlurFilter**, **ConfidenceChecker**, and **LabelFilter** to operate on these structures and could then operate on any future detection models added to the system.

Figure 14 shows the **ModelData**, **ModelInput**, and **ModelOutput** classes highlighted in red. The TB4 implementation did not explicitly use these classes and relied on Python's weakly typed nature to achieve a conceptually equivalent functionality. Other implementations of the RAAEC may find these intermediate classes useful or necessary to completely define the governor pattern.

System architects should consider exactly what governor components their system needs. It's possible that a model needs multiple validators, checkers, or constrainers. It is also possible that one model's checker or constrainer could be implemented as another's validator.

The conduct governor wrapper pattern is essential to the RAAEC because it enables the CD to configure the components and run governance around the system's models. As described, the RAAEC only emphasizes these two features, but system architects may find that extending the wrapper pattern is the best way to add other features the system requires.

Even though the pattern did not explicitly specify the following features, we found it advantageous or necessary in our development of the TB4 system to do the following:

- **Use generic data structures for types of models:** This work made governor components easier to develop and would allow drop-in replacement for different model implementations that served the same system function (e.g., swapping YOLO for FasterRCNN).
- **Correlate outputs across models:** It was necessary for the tracker and SAB to know which detector and classifier outputs to associate with a given track.
- **Design for models that could work on individual or batched inputs:** We assumed a single input for each model in our initial design of the implemented architecture, but the final version of the classifier worked on a batch of windows identified by the detector.

Selecting an Interaction Style

Modeling systems as graphs of connected components is an established strategy when designing or describing systems. Conceptually independent processes make up the nodes of the graph while the edges describe their interactions. The RAAEC uses this graph model to describe its reference system, but it doesn't specify restrictions on the implemented interaction style. It encourages designers to decide whether call response, message bus, or other communication strategies are the best solutions for the interactions of their components.

Interaction implementation is a core system-design decision that can only occur in coordination with and consideration of the system's hardware, processes, and threads (see "Decisions on Hardware, Processes, and Threads"). These decisions must balance project constraints like implementation difficulty and development time with system requirements like timing guarantees, message loss tolerance, and networking requirements.

With the freedom to select an interaction implementation, we chose the message bus system of data distribution service (DDS) via ROS2 for our TB4 system. The implemented system realized the nodes as separate processes that communicated via ROS topics and services.

ROS2 is commonly used in robotics research and was a natural choice for this project because it is well documented, has a prebuilt set of tools for system testing and introspection, and was preinstalled on the TB4 platform by the manufacturer.

System architects should perform a tradeoff analysis early in the design process to determine what interaction style is most appropriate for their system's RAAEC implementation. Because the TB4 system focused on evaluating the RAAEC, we did not perform a tradeoff analysis between ROS2 and other interaction styles.

Decisions on Hardware, Processes, and Threads

In the main body of this reference architecture, we acknowledge that we do not specify which hardware decisions, process models, or threading strategies that systems must use. While development teams should consult the RAAEC to decide on these design elements, development teams can't make final decisions without knowing the full context of specific system and mission constraints.

For our TB4 system, we decided to use the existing Create3 computer and Raspberry Pi 4B on the robotic platform. An external laptop ran the autonomy stack, receiving image streams from the platform and sending commands back.

This research implementation did not focus on performance optimization. Most processing and data storage was centralized on the external laptop for ease of development. When preparing real implementations, design teams should carefully consider a system's division on on-board and remote processing. Development teams can use performance testing and tradeoff analysis to identify the optimal balance.

We implemented each node as an independent process with ROS as the system middleware. These processes could spawn anywhere from one to three threads, with each thread managing a set of event-based callbacks. Typically, these callbacks executed upon the expiration of a timer or receipt of a message. These threads correlated to the sets of events that the system could safely run concurrently, with mutually exclusive callbacks placed in the same thread.

We designed a process model and threading strategy for our TB4 system for simple functionality, and we neither optimally designed nor particularly recommend this approach. The design is reflective of rclpy, the ROS client library for Python. Different message-bus or call-response libraries might allow other models and strategies. For example, other ROS client libraries, such as the client library for C++, allow process models where nodes can be composed into a single process. This approach has the advantage of message transportation via shared memory.

Usage of Data Annotation and Modification

Pipelines that transport data from a system's sensors and between its governed models are fundamental to the RAAEC. As data progresses, the RAAEC recognizes that system architects will require a data-handling approach. They may find it beneficial for pipeline components to annotate data with additional details or be permitted to modify the data directly—including removing it from the pipeline.

For example, a component that evaluates image blurriness in a computer vision pipeline could make use of an annotation approach that adds a quality score to the image as metadata that successive components read to adjust their operations. Conversely, a modification approach might allow sharpening the image if the blur is correctable or dropping the image from the pipeline if it is not.

Either action has an obvious impact on the implementation of the RAAEC and the governed models composing its pipelines. These options are not mutually exclusive, and the RAAEC allows the system's architects to balance the two approaches as necessary.

Our TB4 implementation used a combination of data annotation and modification approaches. While we could have designed a valid architecture for either approach alone, this mixed data-handling approach allowed us to evaluate the applicability of both recommendations.

We used data annotation between governed models and the SAB. As data flowed into an ML node and through the validator, model, checker, and constrainer component, these components would log governance information. At a minimum, this process documented that each component ran successfully. If the system modified data, it could record that fact along with a human-readable description and key-value metainformation about the change. Similarly, this logging structure could record any decisions to halt the flow of data from any component in the node.

We used a **ComponentLog** class to capture annotated information from each component of a governed model. Figure 14 includes this class's UML description. Each run of a governed model would generally record a log for each component that was written to the SAB.

Unique identifiers in a component log—like a user-defined name, the components type (validator, model, checker, or constrainer), and class name—provided satisfactory uniqueness to identify a component within a set. We provided a human-readable description for explainability, and we included a status enumeration, Boolean flag for data modification, and, optionally, a dictionary of attributes to support both runtime and post-hoc system analysis.

We used JSON serialization to extract the set of log information after each model run. This approach proved best for our implementation because of the optional presence of an arbitrary number of key-value pairs in an individual component's log. In the SAB's PostgreSQL databases, JSONB-typed columns encoded this data, and in ROS messages it could be passed as a string.

We allowed components of a governed model to perform data modification. Many components acted as filters and dropped data from the pipeline. For example, a blur and filter component on the system's detector model excluded images that lacked clear information. This component successfully prevented images that were out of focus or subject to motion blur from consuming resources.

However, the initial implementation lacked signaling to sequential nodes that information was dropped. Lack of signals created a problem for TB4's tracker model because it maintained internal counters to record when each track was last updated. When a sequence of blurry images triggered the detector to halt the perception pipeline, the tracker would not run, so it did not detect that new images had entered the system. This issue resulted in the preservation of tracks during a period when they should have been dropped.

When discussing data modification, the RAAEC should note the impact that removing data or halting its progression may have on downstream pipeline components. Noting this impact is especially important for components that expect to run at a specific frequency or maintain some sense of system time. The best approach will be system-dependent, but common solutions could also work. For example, guaranteed signal propagation is one solution. Once data enters a pipeline as a signal, the data may be nullified, but the signal must propagate to the end. Components on the pipeline must be designed to accept valid data or nullified data and forward it along. This ensures that each component runs. It still requires interactions to communicate between components, which can be expensive, but nullifying the data can minimize the amount of information transfer and allow a component to skip time-consuming model execution.

Alternatively, pipeline components could maintain independent timers as a fallback to guarantee that, if no data arrives to trigger an update normally, they still update their model at a desired frequency. This approach could reduce the number of messages the system needs to send by halting signal progression and make components independently robust. It could also introduce challenges for system timing and synchronization.

Implementation Architecture Description with Component Views

The following sections outline specific components and their relationships that support the requirements and quality attributes. This appendix gives a brief overview of each component, discusses whether it was included or excluded in the TB4 implementation of the architecture, and provides suggestions for future developers.

Implementation Actuation

The "actuation" component covers all software–hardware interactions in the system. The RAAEC states that best-practice safety engineering is assumed, but we imposed additional safety requirements on actuation functionality to further limit unsafe or unnecessary action.

On the TB4, there are limited options for actuating any part of the system. The TB4 employs a lightring composed of 6-LEDs, a speaker that can play sound for specified frequencies and durations, and differential drive wheels for basic mobility. For the purposes of the TB4 experiment, we considered the lightring and speakers to be the only available actuated system on the robot, and we used them to signal engagement with scenarios as appropriate. We descoped mobility system actuation, such as moving the TB4 or navigating it through the different zones, to focus our development effort on the governance system's architectural complexity.

In the TB4 architecture, we referred to this system component as the controller. The controller provided an interface that accepted a simple list of five different commands, which were as follows:

- **CMD_NO_OP:** A no-operation command that does nothing
- **CMD_NO_GO:** Signals that the robot should stand down from its perceived target
- **CMD_NEUTRAL_DETECTED:** Signals that the robot has detected a neutral target

- **CMD_FOE_DETECTED:** Signals that the robot has detected a foe, but does not automatically engage with the foe
- **CMD_ENGAGE:** Signals to the controller that the robot should engage its selected target

Notably, the controller node itself did not have its own governance and safety guardrails. The actuation and controller node was simple, and it did not have insight into the planning process beyond basic actuator status. However, we did not explore this avenue deeply as part of the Turtlebot experiment, and further experimentation with a more complex system is required.

Development teams implementing this architecture should consider including a simple interface with high-level commands to engage various parts of the actuation system. This approach simplified testing by allowing us to test the output of the responder separately from the actuation of the system. It also allowed for ease of reconfigurability, as we were able to quickly update different commands and modify their lightring and audio sequences as appropriate.

Implementation HSI and Communications

The RAAEC acknowledges that LAWS requires significant operator interaction, so the system must provide feedback and direction to external users and potentially targets. This requirement necessitates an interface between hardware and software. The reference architecture provides suggestions on components to include in the system. However, it explicitly does not provide guidelines on implementing these components.

We included a simplified version of these components in our TB4-based implementation. The TB4 platform's lightring and speaker (as described in "Implementation Actuation") served as indicators to operators that the system had reacted to a target.

We also included a user interface node in the TB4 system to support HSI. For some commands, the user had the option of approving an engagement action or denying an action and asking the robot to stand down. The user interface emulated a command-line-style input and prompted the user when it required an input before passing it back to the controller node.

The RAAEC recommends a variety of user-interface components, but we descoped the implementation of such components to emphasize other aspects of the architecture. Because the TB4 system implemented only a minimal user interface, this work provided limited commentary on the RAAEC's impact on systems' HSI and communications.

In our TB4 system, the user-interaction node had no insight into the system—the responder provided a prompt, and the user-interaction node returned the user's input back to the responder. We recommend that systems keep the interaction between user-input modules and the robotic system simple and secure.

Implementation Health and Support

The RAAEC states that health and support components should account for a significantly larger volume of telemetry and logging information for the additional system complexity a conduct governor introduces.

We did not explicitly support health and monitoring capabilities in the TB4 experiment. Telemetry and logging data were recorded to the database that served as the SAB.

Implementation Navigation and Localization

The architecture outlines many components that make up the navigation and localization stack. The stated goal of the navigation and localization subsystem is to translate mission goals and limits into robotic movements. The design describes a behavioral planner; map manager; movement, goal, and mission planner; obstacle detection capability; route planner; and trajectory planner. This set of components describes everything required to support robotic mobility and actuation in the system's environment.

Normally, navigation and localization are essential for mobile autonomous systems. However, the TB4 implementation of the RAAEC emphasized the perception pipeline and perception, targeting, and response (PTR) chain, so we descoped this functionality. Remote control by an operator provided the system with a suitable proxy for autonomous navigation. The system achieved localization to detect the system's zone with AprilTags placed in the environment rather than map-based methods.

Implementation PTR Chain

The PTR chain, as defined in the RAAEC, consists of basic object detection, path tracking and prediction, entity recognition, target nomination, determination of the appropriate course of action, and engagement of any effects. However, developers might not implement all these PTR chain capabilities for their specific system.

Figure 15 shows the PTR pipeline that we designed for the TB4 system. It contains an object detector, a classifier, a tracker, and a responder. Our scenarios involved the use of four life-sized standee cutouts representing the following classes: armed adult, unarmed adult, child, and robot quadruped. The object detector functioned as a high-level filter to identify the cutouts as objects in a scene. This information was then passed to the classifier, which classified each object as one of the four expected classes. The tracker used the output of the detector and responder to maintain knowledge of consistent, unique objects between frames. The responder used all this information to decide what action to take in the scene.

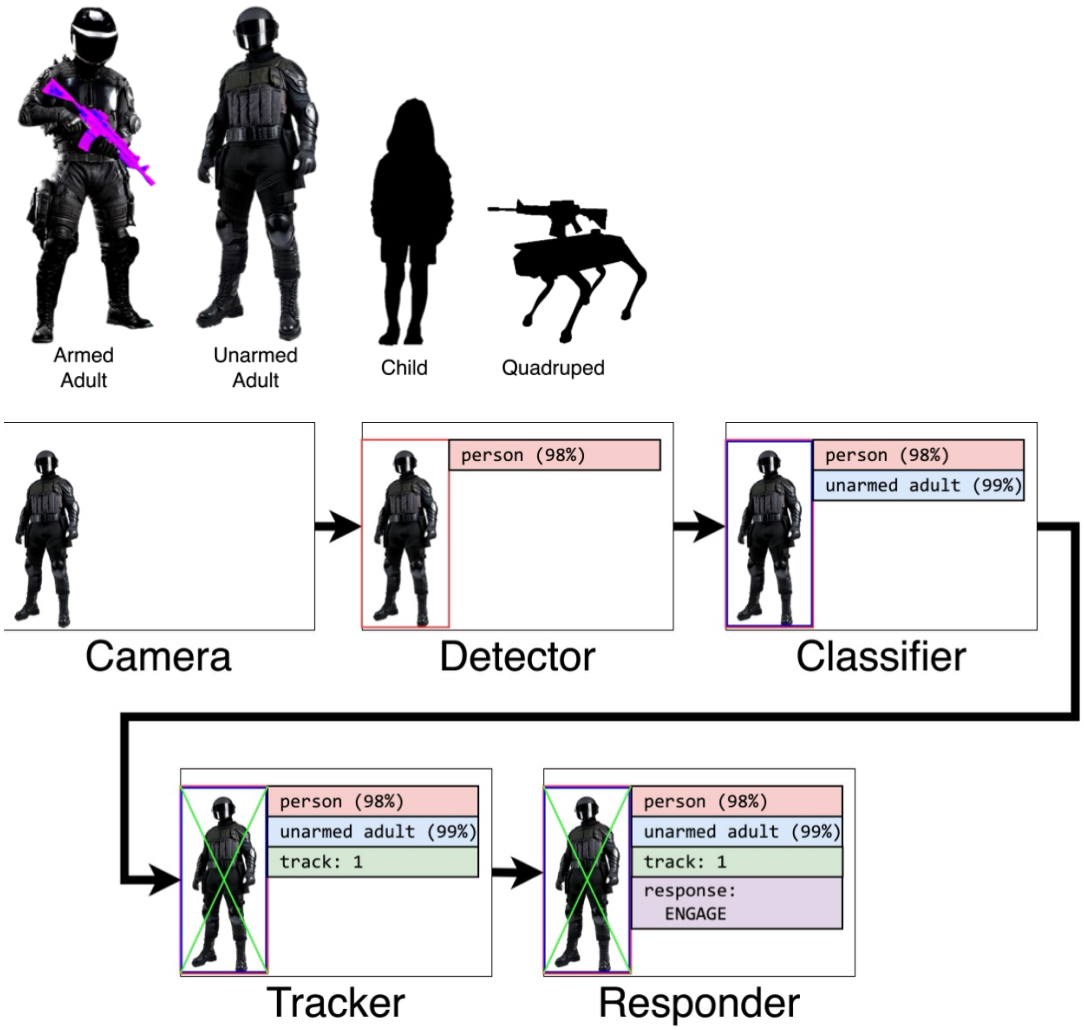

Figure 15: PTR Chain and Classification Targets

We expand on each of the PTR capabilities presented in the RAAEC in further detail in the sections that follow, and we discuss why we chose to implement or descope each capability.

Implementation Object Detector

The RAAEC states that this component's primary output is basic object identification and classification. For the TB4 experiment, we used a fine-tuned YOLO11n model from Ultralytics to perform detection and general classification of objects in the images provided by the camera stream [Khanam 2024]. We assigned objects a COCO label. We extended the model and class list slightly to include our robotic quadruped cutout, but the YOLO model was able to identify the other cutouts as humans without fine-tuning. The detector filtered out all classes that were irrelevant to our scenario before forwarding the object bounding-box information to the classifier.

We experimented with different validators, checkers, and constrainers on the detector. The final TB4 employed a blur filter, a confidence filter, and an object overlap filter. We implemented these filters as configurable components of the governor system with modifiable thresholds that users could update as appropriate to the mission scenario. While additional filters may be necessary in a real-world scenario, we found this minimum set of filters to be appropriate for testing the RAAEC attributes within the bounds of our test scenarios.

For the classifier, we used a MobileNetV2 model from Google, which we fine-tuned on our four classes of interest [Sandler 2019]. The classifier received the bounded objects from the detector and assigned classification with confidence scores to each frame before forwarding the necessary information to the responder.

We collected data for the fine-tuning by video recording the standee cutouts in conditions that were as similar to our real-world case as possible. We labeled this raw data using the Computer Vision Annotation Tool (CVAT) to produce model-specific datasets with annotations. We then split up the datasets into train, validation, and test buckets, as appropriate. We applied programmatic augmentations such as blur and lighting conditions to data during training to produce a more robust model capable of handling the variation inherent to real-world conditions.

Through our experiment, we found it valuable to use an off-the-shelf model to perform basic object detection. During the testing phase, the YOLO model enabled us to start testing the pipeline even while we were still training the specific target-classifier model. By isolating objects that may be relevant to the scene, we ensured the classifier was able to focus on classifying potential targets instead of irrelevant objects.

Further experimentation is required to determine whether system performance is drastically improved with this detector-classifier model. Additionally, it may be worth experimenting with the placement of the detector in the pipeline instead of the classifier and tracker. Our experiment performed detection-classification-tracking sequentially, but a system may benefit from performing detection-tracking-classification sequentially, parallelizing these processes, or establishing other approaches.

Implementation Path Tracker

The RAAEC requires that the path tracker component system track objects across multiple frames to maintain continuity. For the TB4 system, we determined that the system should track only objects classified by the scenario-specific classifier.

We implemented a tracker node in the TB4 system that used the OC-SORT tracker algorithm to maintain tracks of objects across different frames [Cao 2022]. The tracker node used the combined output of the detector and classifier nodes as input and maintained knowledge of tracks in the situational awareness database. The tracker did not maintain track information after an object left the scene.

The placement of the tracker and its reliance on the other perception components are important for designers to consider. Our system placed the tracker at the end, so it had the detector's label and confidence and the classifier's output vector for each bounding box in an update. We decided not to use these models' class information for the boxes when updating the tracker. While this data could be

useful for associating boxes to tracks, it could propagate uncertainty about a box's class from those models into the tracker.

Implementation Entity Registrar

The entity registrar, as defined in the RAAEC, extracts features from specific objects to ensure that the object the system is tracking is consistent from frame to frame. Inputs include object detections and previous entity identification information. Outputs include the new and updated entity features and identifications.

The TB4 system did not explicitly include this functionality. The system identified objects based on the track ID provided by the OC-SORT tracking algorithm.

Implementation Activity Recognizer

The activity recognizer uses the motion of the object, including orientation, to determine properties about it. This capability helps characterize the object's behavior in the scene. This component may rely on multiple ML models or a different construct entirely.

We deemed this component too complex for inclusion in our simplified TB4 experiment. For the sake of the experiment, we determined the behavior of each entity by its specific object classification. We used only four stationary cutouts on which we performed classification and created a simple class-response mapping. We considered using other entities that moved or made other gestures, but we ultimately deemed them out of scope for the experiment.

Implementation Scene Understanding

The scene-understanding component—as we describe it in the RAAEC—makes determinations about each object and its relationship to other objects in the scene.

We did not implement a designated scene-understanding component for our TB4 experiment. We achieved scene understanding through the combination of the responder's logic tree and the use of a conduct governance mechanism that calculated the overlap between object bounding boxes. This mechanism modified the behavior of the responder if two bounding boxes were found to be overlapping—in other words, if one entity was in very close proximity to another. This approach evaluated object relationships in a simplified way. We discuss the responder and its logic in further detail in the "Responder" section.

Implementation Target Nominator

In the RAAEC, we describe the target nominator as the component that is responsible for evaluating objects as targets to engage. This component should use all available data, including scene understanding and ethical constraints, to nominate a target.

We decided to combine the nomination functionality with that of the responder. The nomination and response logic used for the TB4 experiment was rudimentary by design, and we chose to reduce overhead and simplify the system to compensate for limited development time. The target nominator

capability itself was essentially a mapping between an identified class name and a potential action. We used conduct governance mechanisms to control what targets the system nominated. Before the system nominated it as a target, each entity had to pass through all the "gates" that conduct governance had in place.

We provide more details on the functionality of the combined nominator and responder in the following section.

Implementation Responder

The responder—as we explain in the RAAEC—uses nominated targets, the conduct disseminator, and situational data to elicit actions by the autonomous system. The responder should generate a set of potential responses for all targets, remove restricted responses based on the CD, prioritize targets based on the environment, and execute an action. The responder may require operator approval.

The responder node is essentially the end of the pipeline for the TB4 experiment. Once the scene has been processed by all the other pipeline components, the responder node will process all available information and decide how to proceed.

In the TB4 system, the responder node performed the roles of both the target nomination and the responder. The responder itself was triggered by a ping from the SAB to indicate that there is new information in the environment. For the TB4 experiment, the ping occurred whenever a new image arrived from the camera and successfully passed through the entire pipeline.

Upon receipt of this notification, the responder node queries the situational awareness database for the latest information from the scene. Based on the tracks present in the scene, the responder also queries the tracks table for further information on the visible scene objects. From this information, the responder node creates an internal understanding of the current scene and the objects in it, including each object's bounding box, the time it was first seen, the time it was most recently seen, its combined classification probabilities, and its combined detection probabilities.

We do not specify in the RAAEC what form the responder should take. It may be worthwhile to further explore what kinds of responders are available, especially as researchers are currently investigating responders as an open area of research. The responder that we used for the experiment took the form of a simple logic tree. Practical guidance on implementing a responder is limited.

Implementation Battle Damage Assessment

We provide a brief description in the RAAEC of a theoretical battle damage assessment (BDA) component that combines historical data with up-to-date sensor data to assess the scene and the results of the system's actions. We acknowledged that this is a difficult area for autonomy.

The TB4 experiment did not include a BDA component, and it did not make any assertions about BDA capabilities. We deemed this component as out of scope for the experiment.

Implementation Sensing

Sensing is a key component of any autonomous system. In the RAAEC, we address sensors for PTR, acknowledging that many possible sensor configurations and post-processing options exist. The RAAEC outlines five general steps for a sensing subsystem for a conduct-governed system with a PTR chain: the hardware interface, post-sensor validation, processing, sensor data fusion, and final processing.

We did not focus on processing of sensor data in the TB4 experiment. The TB4 platform's camera served as the primary source of data for the PTR pipeline. We maintained the manufacturer's default settings, but we reduced the FPS to 15 Hz and set the image resolution to 1080p to avoid overwhelming the perception pipeline.

In the RAAEC, we focused strongly on examples of image data processing, particularly in the examples provided in Section 5.6.4 and Section 5.6.5. Exploring other sensor modalities mentioned by the RAAEC –including depth sensing, microphone and sound, lidar, GPS, and sensor fusion—would require further work research an experimental platform that includes the appropriate modalities.

Implementation SAB & CD

The SAB and the CD inform the system's understanding of the world. The SAB should contain both data stores and logic to make sense of the data contained in those stores. It should also facilitate the interaction of other modules in the PTR chain. The CD component should store the rules of conduct and ethical constraints and guidelines provided to the system and decompose them into specific ML configuration settings.

We separated these two ideas into two distinct components in the TB4 system, which we explain in the following sections.

Implementation Situational Awareness Blackboard

The SAB took the form of a PostgreSQL database with numerous tables to handle the output of each of the pipeline components. Many tables stored time-series data and the TimescaleDB extension, which optimized read and write queries to them.

Rather than employing a centrally defined database schema, each component defined its own table definitions and schemas via a database module. Upon startup of the system, all components cleared their tables in preparation for a new session. Components interfaced with their tables directly to add all relevant results, add intermediate telemetry data, and conduct governance tracking data to the database associated with the timestamp of the original input image used to inform these decisions.

Implementation Conduct Disseminator

Conduct rules and ethical constraints were defined by configuration files specific to each pipeline component. The localization node handled the conduct dissemination functionality. The localization node maintained a list of all the possible nodes that accepted conduct governance reconfiguration.

When the system entered a new zone, the localization node updated all nodes with the conduct governance associated to the new zone.

The TB4 system configured conduct governance based on multiple configuration files for each component. For example, the detector, classifier, and tracker each had their own configuration files with their own conduct governance. The system also ended up having multiple layers of configuration with files containing references to other files. Development teams that intend to implement this system should consider simplifying the configuration file system and combining files in favor of one large configuration file. Or they can collocate the various configuration files. Additionally, the system should define and track the configuration schema in a central location because tracing all the files that are modifying the system can quickly become confusing.

Similarly, the database schema for the SAB should be well tracked in a central location. Common guidance for SQL databases states that the database schema should be well defined beforehand so large modifications are not required during development. We found that, during system development, the database schema will very likely be modified. Modification was handled by creating an abstraction layer between the database interface and the various pipeline components. This abstraction layer allowed us to apply Python-type hints to database queries and insertions, which proved useful when updating database schemas to include additional information. We found that the approach of using many tables was also useful because it allowed us to easily track all relevant telemetry data across the pipeline. Dedicated database engineers could certainly consider creating more complex schemas, but we were able to execute a functional database system even with the limited database experience available on our team.

Reflection on Functional Attributes

We used the functional and architectural requirements outlined in the RAAEC to design the reference architecture. In this section, we examine those functional and architectural requirements and discuss whether the TB4 system met or descoped each requirement.

Reflection on Functional Requirements

This section outlines the functional requirements of the system as defined in the RAAEC and how they manifested in the TB4 experiment. The RAAEC categorizes its functional requirements by mobility, perception and targeting, human interaction, and effects.

The RAAEC outlined a set of basic requirements for the mobility system of a LAWS. These included map-based planning and navigating to locations, identifying and navigating around static and dynamic entities, and responding to threats.

We descoped all mobility requirements in our TB4 experiment in favor of emphasizing the implementation and complexity of the PTR chain due to limited time as well as limited hardware capability. Test operators had the ability to manually control the TB4 with a controller, but this ability did not factor into the execution of zone localization or the PTR chain.

The RAAEC outlines the following functional requirements for the perception and targeting system, and the TB4 system addresses each of these requirements in the following ways:

- **Identify stationary and dynamic entities**: The detector-classifier-tracker pipeline identifies both stationary and dynamic entities, though it is configured to identify a limited set of entities as described in prior sections.
- **Identify and track dynamic targets across multiple viewings:** We interpreted this requirement as establishing the need to track targets as they leave and re-enter the view of the system. We descoped this functionality for the TB4 experiment because it involves a tracking algorithm and does not directly impact conduct governance. A fully fledged system should implement this capability.
- **Identify threats based on attributes such as "bearing weapons"**: We configured the TB4 system to classify the following four entities: an unarmed adult, an unarmed child, an armed adult, and an armed robotic quadruped. We therefore trained the classifier model to identify two classes that had specific attributes such as "bearing weapons."
- **Identify threat levels based on activity**: We interpreted this requirement as establishing the need for the system to interpret the behavior of the entity and factor in its movement and behavior accordingly. As we discussed in "Implementation Activity Recognizer," we descoped this requirement for the TB4 system.
- **Identify non-threats**: In the TB4 experiment, we generally considered the "child" and the "unarmed adult" entities to be non-threats based on the current zone of the system, and we trained the classifier to identify them as such.
- **Request clarification from a user when identifying threats**: As we described in "Implementation HSI and Communications," the TB4 system prompted the user to provide input when it was presented with a target that required such input.
- **Perpetrate attacks against threats when instructed to do so**: The TB4 emulated engaging a target through use of the available lightring and speaker.
- **Automatically perpetrate attacks against threats when pre-authorization to do so has been given**: We configured the TB4 was to automatically "engage" an armed adult in a combat zone through use of the lightring and speaker without user input.
- **Identify sensitive situations where action would result in unintended consequences such as collateral damage**: The scenarios we presented to the TB4 were very limited, and we therefore descoped collateral damage evaluation.

In the RAAEC, we recommended that these requirements be enumerated or given unique IDs and linked to in the sections that describe each architectural component. Doing so would give a more complete picture of how the presented architecture addresses each of these stated requirements.

Implementation details and guidance are covered in greater depth in "Test System Design Implementation Architecture Description with Component Views."

The RAAEC outlines high-level requirements for human interaction capabilities in the system. Many of these requirements address capabilities such as communicating to de-escalate a situation, asking for assistance, providing traceability, and operating within the framework of ethical rules.

The TB4 experiment did not exercise any of these requirements in depth. The TB4 system is very limited in its communication abilities because it employs only an LED lightring and a speaker. While there was a user input component to the system, we considered that component to form part of the PTR chain. In the event of uncertainty as defined by the conduct governance system, the TB4 defaulted to inaction. The decision-making process could be traced by the developers, but the team did not put significant effort into developing a user interface intended for human interaction.

The team did develop basic database visualization capabilities using a live Grafana dashboard. The dashboard was crucial to obtain insight into the performance of the system as it was operating as well as for debugging of system decisions. We recommend that system developers employ a similar method for visualizing SAB information and telemetry, and a dashboard visualization would also likely be useful to system operators.

The RAAEC defines the following effect requirements:

- Apply lethal effects when directed by mission goals or by the operator with the exception that the system shall not violate RoE or LoW.
- Apply non-lethal effects when directed by mission goals or by the operator.

The TB4 system addresses both requirements by emulating lethal and non-lethal responses via the lightring and speaker. We described the implementation of these effects in greater detail in "Implementation Actuation."

Reflection on Architectural Requirements

This section outlines the architecture requirements of the system as defined in the RAAEC and how they manifested in the TB4 experiment. The RAAEC includes these in addition to the stated functional requirements that focus on human-system interaction.

The RAAEC made use of the findings from "Measuring Trust: Concept Testing and User Trust Evaluation in Autonomous Systems," which was an SEI-led study that studied how to measure user trust in autonomous systems [Hale 2025]. These findings largely served to inform the creation of the architecture, and they are listed by priority in Section 2.4.1. In this section, we address each finding and discuss how they are connected to a component in the TB4 system implementation.

1. **Human authorization of force:** Human-in-the loop is enabled via the responder component and the user input component.
2. **High-precision targeting:** While important for a fully functioning system, we descoped this capability because it is not the focus of the TB4 experiment.

3. **Testing and reliability:** We tested the TB4 system using the scenarios we used to designed it. Reliability varied, but we found that the ability to tweak various parameters allowed for quick system performance improvements and increased system complexity.
4. **Situational awareness:** The RAAEC references situational awareness for both the system and the operator. In the system we designed, we addressed both via the PostgreSQL database and the Grafana dashboard as described previously.
5. **Hardware redundancies:** While crucial in a fully functional system, the TB4 system was limited to the hardware and capabilities of its Raspberry Pi computer and Create3 mobile base. We descoped this requirement because hardware was not a main focus of the experiment.
6. **Operator handover in uncertain contexts:** The TB4 system was able to detect uncertain contexts, but we did not configure it to facilitate operator handover in these situations. We descoped this capability to focus on the PTR chain.
7. **User interface (UI) to support operator decision-making:** We made use of ROS's visualization capabilities to display the camera feed with various annotations, including bounding boxes around detected objects with tracks and track IDs.
8. **Ethical governor:** The TB4 conduct governance system functions as the ethical governor.
9. **De-escalation capabilities:** As we described in "Implementation Actuation," the TB4 system maintained a command list that included "no-go" and "neutral-detected" as a stand-in for de-escalation behavior.
10. **Spatial, rules-based framework for autonomous behavior:** We fully implemented this capability in the TB4 system, and we updated conduct governance and behaviors based on current understanding of zone location. We defined four zones—no-go, non-combat, combat, and liminal—in the TB4 experiment.
11. **Visual and audio operator feedback:** As described above, control over available hardware was minimal. However, the TB4 system did make use of the speaker and lightring to provide visual and auditorial feedback to the operator.
12. **Post-mission review and continuous improvement:** We extracted and saved all data from the TB4 testing campaign for further analysis. The TB4 system fully supported this capability.

Reflection on Quality Attributes

As a reference architecture, the RAAEC is intended to promote specific quality attributes in a system. We identified 10 quality attributes as important for a LAWS. In the RAAEC, we acknowledge that, while system architects should consider all attributes, some attributes are independent of the reference architecture or dependent on system-specific context.

During development of the TB4 system, we consulted the RAAEC to evaluate some of its quality attribute claims. Because this project was short term and executed on an experimental platform, we can only comment on a subset of these attributes in detail.

The RAAEC describes the following quality attributes, listing them in order from highest to lowest priority:

1. **Testability**: The RAAEC should promote testability through its decomposition of the system into components and pipelines that can be tested as individual pieces. Through our work on the TB4 implementation, we found that decomposition did improve testability by allowing components to run independently and to interact with controllable test mock-ups of other components.
2. **Observability and Transparency**: The RAAEC should promote observability and transparency and enable developers, testers, and maintainers to understand system operation at various stages of its development lifecycle. Through our TB4 implementation, we found that the SAB element strongly promoted this attribute for the development and testing lifecycle stages.
3. **Usability (Interaction Capability)**: The RAAEC acknowledges that this is important for stakeholders like warfighters, commanders, maintainers, and testers. Our TB4 implementation lacked outside stakeholders to verify this attribute.
4. **Maintainability**: The RAAEC acknowledges maintainability is important given the rapid growth of ML technologies—including models, computing platforms, and support infrastructure. The TB4 implementation did not reach a level of maturity for an extended period and cannot verify this attribute.
5. **Modifiability**: The RAAEC should promote modifiability through the system's decomposition into components and pipelines. Through our work on the TB4 implementation, we found that decomposition and the use of interfaces (e.g., ROS messages, model data classes, and the PostgreSQL database) created a strong separation of concerns that effectively limited defects to individual components.
6. **Modularity**: The RAAEC should promote modularity, especially the ability for multiple vendors or developers to contribute components to the system through the system's decomposition and the conduct governance wrapper pattern. Through our TB4 implementation, we found that the decomposition effectively separated the system into individual components and subsets of components that development teams could develop in parallel. Similarly, the implementation of the conduct governor wrapper pattern led to governor components that were small and that development teams could also develop in parallel.
7. **Security**: The RAAEC acknowledges that security is important for LAWS given their likely proximity to classified information and export-controlled technology. The TB4 implementation did not implement any notable security features, and we therefore cannot comment on the RAAEC's influence on this attribute.
8. **Performance**: The RAAEC acknowledges that performance is important for LAWS given their likely need to operate in real time and on limited or novel hardware. We did not optimize any performance features in our TB4 implementation beyond reasonably functionality. This attribute was primarily influenced by the interface selected. Beyond allowing the freedom to select this interface, the RAAEC did not promote or inhibit this attribute.
9. **Portability**: The RAAEC acknowledges that portability is important for hardware, software, and model reuse across platforms. Through our TB4 implementation, we found that this attribute was primarily influenced by the interface selected. Beyond allowing the freedom to select this interface, the RAAEC did not promote or inhibit this attribute.

10. **Availability**: The RAAEC acknowledged that availability is important for system deployment and claims its features should not inhibit it. Our TB4 implementation did not reach a level of maturity comparable to system deployment, and we therefore cannot comment on the RAAEC's influence on this attribute.

In addition, one of our goals in developing the RAAEC was to find relationships between our recommendations, requirements, and quality attributes and the DoD RAI principles [DOD 2021]. Through our study, we found that the RAAEC supports those RAI principles as follows:

- **Responsible**: The RAAEC does not relate this principle to a specific quality attribute. It specifically cites the CD architecture element for this principle.
- **Equitable**: The RAAEC does not relate this principle to a specific attribute or architecture element.
- **Traceable**: The RAAEC relates this principle to quality attribute 2—observability and traceability. It specifically cites the SAB architecture element as supporting this principle.
- **Reliable**: The RAAEC relates this principle to quality attribute 1—testability. It specifically cites the CD architecture element as supporting this principle.
- **Governable**: The RAAEC relates this principle to quality attribute 2—observability and traceability. It specifically cites the CD and conduct governor wrapper pattern architecture elements as supporting this principle.

Study Conclusion

We found that the RAAEC provided useful guidance for the TB4 system. Decomposing the system to follow the reference architecture's component-interaction view was useful and directly applied to our ROS2 implementation. The RAAEC afforded flexibility even when making required decisions about interaction style, hardware, process models, threading strategies, and whether to annotate or modify data. That flexibility was necessary for us to integrate our implementation with the TB4 manufacturer's code. Without this freedom of interpretation, the RAAEC might over-constrain systems and inhibit collaboration, code, or hardware reuse.

In addition to these decisions, the RAAEC requires certain features. It does not specify the exact implementation of these features, but it does require the system to have a SAB and a CD and for all models to follow the conduct governor wrapper pattern. We found that it was easy to implement the SAB as a time-series database with PostgreSQL and a ROS node that synthesized data from tables in the database. The RAAEC depicts the SAB and CD as one component in its diagrams, but we found it advantageous to separate these components for development. Our implementation of the conduct governor wrapper pattern provided a modular and extensible interface for a variety of model types as well as validator, checker, and constrainer elements. The pattern also served as a useful interface for logging data because it flowed all the elements of the wrapper.

The RAAEC depicts a model system decomposed into multiple ML components. We implemented only a subset of the components described in the RAAEC for this project due to time and

technological constraints. The TB4 system included a detector, classifier, tracker, and responder, which implemented the functionalities of the detector, entity registrar, tracker, nominator, and responder as outlined in the RAAEC. We excluded other components, such as BDA and advanced actuation, because they were overly complex for this study.

The TB4 test system showed that the RAAEC was beneficial to the design process and technically feasible to implement. More research is required to explore all aspects of the RAAEC across multiple systems and all stages of the system development lifecycle. Even with our system's relative simplicity and complexity-limiting design assumptions, this project was the first step in proving the RAAEC's utility as a tool for TEVV practitioners to calibrate, measure, and evaluate trust in autonomous systems.

References

URLs are valid as of the publication date of this report.

[Arkin 2007]
Arkin, Ronald C. *Governing Lethal Behavior: Embedding Ethics in a Hybrid Deliberative/Reactive Robot Architecture*. Technical Report GIT-GVU-07-11. Georgia Institute of Technology. 2007. https://sites.cc.gatech.edu/ai/robot-lab/online-publications/formalizationv35.pdf

[Arkin 2009]
Arkin, Ronald C.; Ulam, Patrick; & Duncan, Brittany. *An Ethical Governor for Constraining Lethal Action in an Autonomous System.* Technical Report GIT-GVU-09-02. Georgia Institute of Technology. 2009. https://apps.dtic.mil/sti/citations/ADA493563

[Bowers 2023]
Bowers, Ryan & Thomas, John. Safety Implications of Autonomous Vehicles – System-Theoretic Process Analysis Applied to a Neural Network-Controlled Aircraft. *54th Annual International Symposium.* October 2023. http://psas.scripts.mit.edu/home/wp-content/uploads/2023/11/SFTE-Paper-STPA-for-autonomous-vehicles.pdf

[Cao 2022]
Cao, Jinkun; Pang, Jiangmiao; Weng, Xinshuo; Khirodkar, Rawal; & Kitani, Kris. *Observation-Centric SORT: Rethinking SORT for Robust Multi-Object Tracking*. arXiv. March 27, 2022. https://arxiv.org/abs/2203.14360

[DoD 2021]
Deputy Secretary of Defense. Memorandum: Implementing Responsible Artificial Intelligence in the Department of Defense. Department of Defense. May 2021. https://media.defense.gov/2021/May/27/2002730593/-1/-1/0/IMPLEMENTING-RESPONSIBLE-ARTIFICIAL-INTELLIGENCE-IN-THE-DEPARTMENT-OF-DEFENSE.PDF

[Gardner 2023]
Gardner, Carrie; Robinson, Katie; Smith, Carol; & Steiner, Alexandrea. *Contextualizing End-User Needs: How to Measure the Trustworthiness of an AI System.* Software Engineering Institute. December 18, 2024. https://doi.org/10.58012/8b0v-mq84

[Hale 2025]
Hale, Matt. *Measuring Trust: Concept Testing and User Trust Evaluation in Autonomous Systems*. Software Engineering Institute. 2025. [TODO: Link Pending]

[IEEE 7000-2021]
IEEE Standards Association. *IEEE Standard Model Process for Addressing Ethical Concerns During System Design*. IEEE Std 7000-2021. IEEE Computer Society. September 15, 2021. doi: 10.1109/IEEESTD.2021.9536679

[ISO/IEC 25010:2023]
ISO/IEC. *Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Product Quality Model*. ISO/IEC Standard 25010:2023. November 2023. https://www.iso.org/standard/78176.html

[JP 3-60]
Chairman of the Joint Chiefs of Staff. *Joint Targeting.* Joint Publication 3-60. September 2018. https://www.esd.whs.mil/Portals/54/Documents/FOID/Reading%20Room/Joint_Staff/21-F-0520_JP_3-60_9-28-2018.pdf

[Khanam 2024]
Khanam, Rahima & Hussain, Muhammad. *YOLOv11: An Overview of the Key Architectural Enhancements*. arXiv. October 23, 2024. https://arxiv.org/pdf/2410.17725v1

[Lacey 2000]
Lacey, Mike O.; Bill, Brian J.; Berrigan, Michael J.; Boehman, Michael P.; & Chiarella, Louis A. *Operational Law Handbook*. Judge Advocate General's School. Charlottesville, VA. 2000. https://apps.dtic.mil/sti/citations/tr/ADA377522

[Meinhardt 2022]
Meinhardt, Tim; Kirillov, Alexander; Leal-Taixe, Laura; & Feichtenhofer, Christoph. *TrackFormer: Multi-Object Tracking with Transformers*. arXiv. 2022. https://arxiv.org/abs/2101.02702

[Nii 1986]
Nii, H. Penny. *Blackboard Systems*. KSL 86-18. Knowledge Systems Laboratory, Stanford University. 1986. http://i.stanford.edu/pub/cstr/reports/cs/tr/86/1123/CS-TR-86-1123.pdf

[Olson 2011]
Olson, Edwin. AprilTag: A Robust and Flexible Visual Fiducial System. Pages 3400–3407. In *2011 IEEE International Conference on Robotics and Automation*. IEEE. May 2011. https://ieeexplore.ieee.org/document/5979561

[Porzi 2020]
Porzi , Lorenzo; Hofinger, Markus; Ruiz, Idoia; Serrat, Joan; Bulò, Samuel Rota; & Kontschieder, Peter. *Learning Multi-Object Tracking and Segmentation from Automatic Annotations*. arXiv. 2020. https://arxiv.org/abs/1912.02096

[Sandler 2019]
Sandler, Mark; Howard, Andrew; Zhu, Menglong; Zhmoginov, Andrey; & Chen, Liang-Chieh. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. arXiv. March 21, 2019. https://arxiv.org/pdf/1801.04381

[Shuai 2020]
Shuai, Bing; Berneshawi, Andrew G.; Modolo, Davide; & Tighe, Joseph. *Multi-Object Tracking with Siamese Track-RCNN*. arXiv. 2023. https://arxiv.org/abs/2004.07786

[Timperley 2022]
Timperley, Christopher S.; Dürschmid, Tobias; Schmerl, Bradley; Garlan, David; & Le Goues, Claire. ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems. Pages 112–123. In *Proceedings of IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE. March 2022. https://ieeexplore.ieee.org/document/9779703

[UMAA 2019]
Unmanned Maritime Autonomy Architecture (UMAA) Architecture Design Description (ADD). Version 1.1a, UMAA-INF-ADD. AUVSI. December 19, 2019. https://www.auvsi.org/sites/default/files/PDFs/UMAA/UMAA%20ADD%20Dec%2019%2C%202019%20v1.1.pdf

[Xu 2020]
Xu, Zhenbo; Zhang, Wei; Tan, Xiao; Yang, Wei; Huang, Huan; Wen, Shilei; Ding, Errui; & Huang, Liusheng. *Segment as Points for Efficient Online Multi-Object Tracking and Segmentation*. arXiv. 2020. https://arxiv.org/abs/2007.01550

[xView 2 2025]
Computer Vision for Building Damage Assessment. *xView2 Website*. April 9. 2025. [accessed]. https://xview2.org/

Legal Markings

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM25-0382

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu