AI/ML for Mission Processing Onboard Satellites

Mark J. Veyette Jr., PhD*, Kevin Aylor, PhD†, Dan Stafford‡, Michael Herrera§, Sajit Jumani¶, Cole Lineberry‖, Chris Macklen**, Eric Maxwell††, Randy Stiles‡‡, and Matt Jenkins§§
Lockheed Martin Space, Littleton, CO, 80127

As new emerging threats require faster than human-in-the-loop response times, next generation defense systems are requiring more autonomy, data processing, and decision making at the edge. These new systems are looking to artificial intelligence and machine learning (AI/ML) to provide higher levels of autonomous command and control. For space systems, onboard processing of advanced AI/ML algorithms, especially deep learning algorithms, requires a multiple magnitude increase in compute capability compared to what is available with legacy, radiation-tolerant, space-grade processors on current space vehicles. The next generation of space processors for AI/ML onboard will likely include a diverse landscape of heterogeneous systems including various combinations of CPUs, GPUs, FPGAs, and purpose-built ASICs. In this manuscript, we identify the driving requirements for AI/ML processing onboard; detail the similarities and differences between the ground, edge, and space environments for AI/ML; define a reference architecture and the services required to provide an end-to-end framework for developing and deploying AI/ML applications for space; and evaluate the hardware landscape for current and next-generation space AI processors.

I. Introduction

Artificial Intelligence (AI) for the space domain is critical to the United States' national security strategy. Developing and implementing mature and reliable Artificial Intelligence and Machine Learning (AI/ML) technologies is a key enabler for realizing modern Joint All Domain Operations, and requires collaboration across the Department of Defense (DoD), academia, industry, and allies to create and deliver these AI-enabled enhancements. Top military leaders are predicting that whoever masters the AI/ML techniques first will dominate the battlefield of the future [1, 2]. In 2020, when Congress asked then Secretary of Defense Mark Esper what the number one priority was for DoD technology modernization, he responded with, "For me, it's artificial intelligence. I think artificial intelligence will likely change the character of warfare, and I believe whoever masters it first will dominate on the battlefield for many, many years. It's a fundamental game changer. We have to get there first." [3]

To realize this, the DoD has developed an AI/ML strategy and is encouraging adaptable AI problem solving, enabling decentralized development and experimentation, restructuring to promote AI and space, and streamlining acquisitions to speed modernization. A relevant application of AI in space domain is "intelligentized" war where large scale awareness and communications is enabled by growing constellations that rely on AI to process sensor data, manage data traffic, control the satellites, and provide timely insights. The DoD is positioning itself as a leader in AI/ML and is pushing for greater investments in this area. The DoD AI Research, Development, Test and Evaluation (RDT&E) investments for the space domain are expected to see multifold growth (3x) over the coming years, achieving over $2B by 2025 (~10% of the AI budget) [4]. As the industry evolves to meet the DoD's AI/ML vision, DoD organizations will expect more sophisticated, intelligentized satellites that observe, communicate, and cooperate across domains, at scale. This will necessitate a more streamlined, standardized approach to AI development, deployment, demonstration, and sustainment.

*A/AI Machine Learning Engineer Stf, Engineering and Technology, 10475 Park Meadows Dr, Mail Drop: B870-3S07, Littleton, CO 80124.
†A/AI Machine Learning Engineer Sr, Engineering and Technology, 10475 Park Meadows Dr, Mail Drop: B870-3S07, Littleton, CO 80124.
‡Project Engineer Stf, Engineering and Technology, 10475 Park Meadows Dr, Mail Drop: B870-3S07, Littleton, CO 80124.
§A/AI Machine Learning Engineer Sr, Special Programs, 10475 Park Meadows Dr, Mail Stop: B870-3S07, Littleton, CO 80124. Now a Software Engineer at Microsoft.
¶Business Development Analyst Stf, Business Development, 12257 South Wadsworth Blvd, Littleton, CO 80124.
‖Software Engineer Stf, Special Programs, 12257 South Wadsworth Blvd, Mail Drop: B870-3S07, Littleton, CO 80124.
**A/AI Machine Learning Engineer Stf, Engineering and Technology, 9970 Federal Dr, Mail Drop: 31A, Colorado Springs, 80921.
††A/AI Machine Learning Engineer Sr, Engineering and Technology, 10475 Park Meadows Dr, Mail Drop: B870-3S07, Littleton, CO 80124.
‡‡Principal Research Scientist, Advanced Technology Center, 3251 Hanover Street, Mail Drop A014S, Palo Alto, CA 94304.
§§A/AI Machine Learning Engineer Sr Manager, Engineering and Technology, 10475 Park Meadows Dr, Mail Drop: S-4040, Littleton, CO 80127.

The purpose of this white paper is to derive the driving requirements for AI/ML processing onboard satellites (Section II), identify the different environments for executing AI/ML (Section III), define a reference architecture for the services necessary to develop for and deploy to these environments (Section IV), and evaluate the landscape of current and next-generation hardware for execution of AI/ML models onboard satellites (Section V).

II. Driving Requirements

In this section we define the driving requirements for developing an AI/ML application for onboard processing. These requirements can be difficult to meet for AI/ML systems running on state-of-the-art, scalable, cloud-based environments with mature infrastructure and tools and access to high-performance computing resources and GPUs. The task will be even more difficult to achieve on a satellite with limited compute, support, and infrastructure. We divide the driving requirements into three broader categories defined below: operational requirements, safety requirements, and user experience requirements. In Table 1 we provide a summary of our findings.

Operational requirements cover the ability of an AI/ML system to perform the desired task in the desired environment. End users desire AI/ML solutions that are able to process large amounts of data in real-time with a minimal hardware footprint and high accuracy. These solutions also need to be deployable to a wide variety of platforns and hardware architectures, scalable to variable inputs and throughputs across multiple processors and satellites, all while maintaining the desired level of performance.

Safety requirements involve ensuring AI/ML systems operate in an expected manner and are resistant to outside threats, both physical and digital. Deep learning models, in particular, are often described as black boxes where inputs go in and outputs come out and there is little insight into how the model is making its determination. The seemingly non-deterministic nature of AI/ML systems may cause some end users to be hesitant to trust AI/ML systems. AI/ML systems will also need to meet more standard software metrics for reliability and security as they suffer many of the same vulnerabilities that standard software systems do. There needs to be procedures in place to mitigate radiation induced faults, allow graceful failover, and ensure damage to the AI/ML system has limited impact on the spacecraft and mission.

User experience requirements focus on meeting the user needs for low cost AI/ML systems, developed rapidly, and which are easy to use and provide value over more traditional methods. AI/ML-based systems can be more costly to develop and maintain. End users need to see benefits from these systems worth the investment.

AI/ML Driving Requirements		
Requirement	**Parameters**	**Solutions**
Operational		
Performant	• Latency • Throughput • Duty Cycle • Startup time • Resource consumption	• Model/Pipeline optimization • Pipeline scheduling systems • Testing environment • Minimal runtime
Deployable	• Model architecture (supported ops) • Programming language • AI/ML Framework • Retargetability	• Multi-platform support • Multi-accelerator support • Containerization
Scalable	• Number of simultaneous I/O • Horizontal Scalability • Vertical Scalability	• Variable input/throughput • Distributed processing/reasoning • Orchestration
Sustainable	• Model update frequency • Fraction of data downlinked • MLOps artifacts size • Required update bandwidth	• MLOps pipelines • Onboard monitoring • Digital twins • Active Learning • Differential updates
Safety		

Secure	• Threat vulnerability • Threat detectability • Threat responsiveness • Model Robustness • Prediction redundancy	• Robust model development and testing • Secure model updates • Adversarial training • V&V • AI/ML-based cyber security
Reliable	• Fault tolerance • Robustness • Downtime • Duty cycle	• Model monitoring • Built-in test/fault mitigation • Multiple fallback options • Resource optimization
Explainable	• Model uncertainty • Model architecture/complexity • Explainability	• Explainable AI techniques • Continuous monitoring • Root cause analysis
User Experience		
Rapid	• Development time • Ease of update	• Model/Data repository • COTS development tools/services
High Value	• Non-recurring cost • Recurring cost • Improvement over other methods	• Automated pipelines • Reuse common solutions • Reduce duplication of efforts
Easy to use	• UX/UI intuitiveness • Ease of integration • Standards conformity	• User-centered design • Adopt standard interfaces/formats/methods

Table 1 Table of driving requirements for AI/ML and common solutions.

III. AI/ML Environments

The different AI/ML deployment environments can be defined by the unique challenges, hardware, and use cases found in said environments. Here we briefly define the various environments under consideration and refer to them as ground, edge, tiny, mobile, and space. This is not meant to be exhaustive, but instead to provide a baseline for discussing the similarities and differences in developing for and deploying to different environments.

A. Ground

The ground environment contains AI/ML systems deployed on desktops, laptops, servers, or on scalable, on-demand computing services (often referred to as cloud computing and includes services such as AWS and Azure). Computational power is greatest in ground environments and resource efficiency tends to be secondary to accuracy, especially in the cloud where it is easy to spin up additional resources as needed. Ground AI/ML systems are most frequently powered by x86 CPUs and NVIDIA GPUs, although other processor architectures (e.g., ARM CPUs, AMD GPUs, TPUs, and FPGAs) are seeing more applications in this environment. AI/ML systems in this environment tend to be deployed for command and control, closed-loop processing, or for analysis of collected data.

B. Edge, Tiny, & Mobile

The edge, tiny, and mobile environments contain a wide variety of devices focused on small form factors, low power consumption, and mobility. The edge environment can include computational units such as CPUs, GPUs, TPUs, FPGAs, ASICs, and many others, usually in the format of a single board computer (SBC), system-on-a-chip (SoC), or a microcontroller (MCU). Edge devices can be found in both unmanned/manned land, air, and sea vehicles. They can also be found in weapon systems, medical devices, drones, smart assistants, remote sensing stations, manufacturing equipment, and IoT devices. Limited available computational power, memory, training data, and other constraints

present challenges to overcome in this environment. Not only must an AI/ML system be small enough to fit within the available volatile and non-volatile memory, it must be able to perform inference within a desired level of latency and accuracy. This results in the need to balance various performance metrics during development. Additionally, the wide variety of computational units found at the edge do not all provide full support for AI/ML operations, if any at all. Deploying to the edge generally requires more investment integrating the AI/ML system with the target device than in the ground environment. In the tiny environment one can find devices focused on even lower power consumption and smaller form factor, such as bare metal (instructions are executed directly on logic hardware without an operating system) microcontrollers. Sensors, hand-held equipment, and other devices requiring a minimal form factor may employ tiny computational units. Devices in the mobile environment are based on SoCs built around an ARM-based CPU and may also contain GPUs, image signal processors (ISPs), digital signal processors (DSPs), and in some more modern devices, neural processing units (NPUs) or tensor processing units (TPUs). Devices found in the mobile environment include cellular phones and tablets; applications generally run in either iOS or Android OS (there are many other OS options in the mobile environment, but none are as commonly used as the two mentioned). As in the edge and tiny environments, the limited computational power and frequent need for real-time inference in the mobile environment present challenges for developing AI/ML systems that can operate with the desired level of performance. Due to the similarity of the edge, tiny, and mobile environments we frequently only refer to one of the environments in the following sections on the AI/ML workflow, unless an explicit difference needs to be highlighted.

C. Space

We consider the space environment to include AI/ML deployed on platforms in low Earth orbit (LEO) or further from the Earth. This can include unmanned space vehicles such as satellites, deep space explorers, and planetary rovers; or manned space vehicles like space shuttles, crew modules (e.g., the Orion spacecraft), and the International Space Station. The space environment shares many similarities with the edge environment but also presents unique challenges to overcome. Like the edge environment, processing power is limited by size, weight, and power (SWaP) in the space environment. In space, one must also contend with hardware that tends to lag several generations behind what is found in the edge environment (and even further behind what is possible on the ground) due to the additional challenges presented by operating in high-radiation environments. Thermal management also plays a more critical role in determining what kind of processing is possible in vacuum. Any onboard AI/ML system needs to be optimized to operate within many tight constraints (e.g., latency, accuracy, power and thermal loads, volatile/non-volatile memory demands) in order to meet mission requirements. Often the AI/ML system will need to share resources with other processing taking place onboard the space vehicle. Additionally, communication with devices in space is more limited compared to other environments. Even when a permanent connection is not possible, it is relatively easy to transmit data to and from devices in the ground, edge, tiny, and mobile environments compared to devices in space. In space, bandwidth and transmission windows are reduced and different strategies for monitoring AI/ML systems in space are required. All of these factors make the space environment the most challenging to deploy AI/ML systems in.

IV. Reference Architecture

Here we define the five steps of the AI/ML workflow (Prepare, Model, Optimize, Integrate, and Certify), identify the components and functions that can exist within each step, identify the differences and similarities when developing for the different environments, and survey existing functionality provided by FOSS and COTS software tools. Finally, we consider how this workflow can be automated through the concept of MLOps and how MLOps practices might be employed in ground, edge, and space environments.

A. Prepare

In the Prepare step of the AI/ML workflow, data are collected, stored, analyzed, and preprocessed. In some cases this step is completed via ad hoc scripts and notebooks to import, analyze, and ready the data for training; however, there is an ever-growing list of software tools to aide in data curation, management, and exploration. Figure 1 shows the functional diagram for the data preparation step.

Training data are often stored in a variety of formats including text files (e.g., CSV or other delimited text, raw text, fixed-width text, JSON, XML), binary files (e.g., HDF5, Excel, Python Pickle, TFRecord), image files (e.g., PNG, JPEG, TIFF, NTF), databases (e.g., MySQL, PostgreSQL, NoSQL), and object stores (e.g., S3). Broad and extendable support for different data formats and storage methods is essential for supporting a variety of uses cases with a single

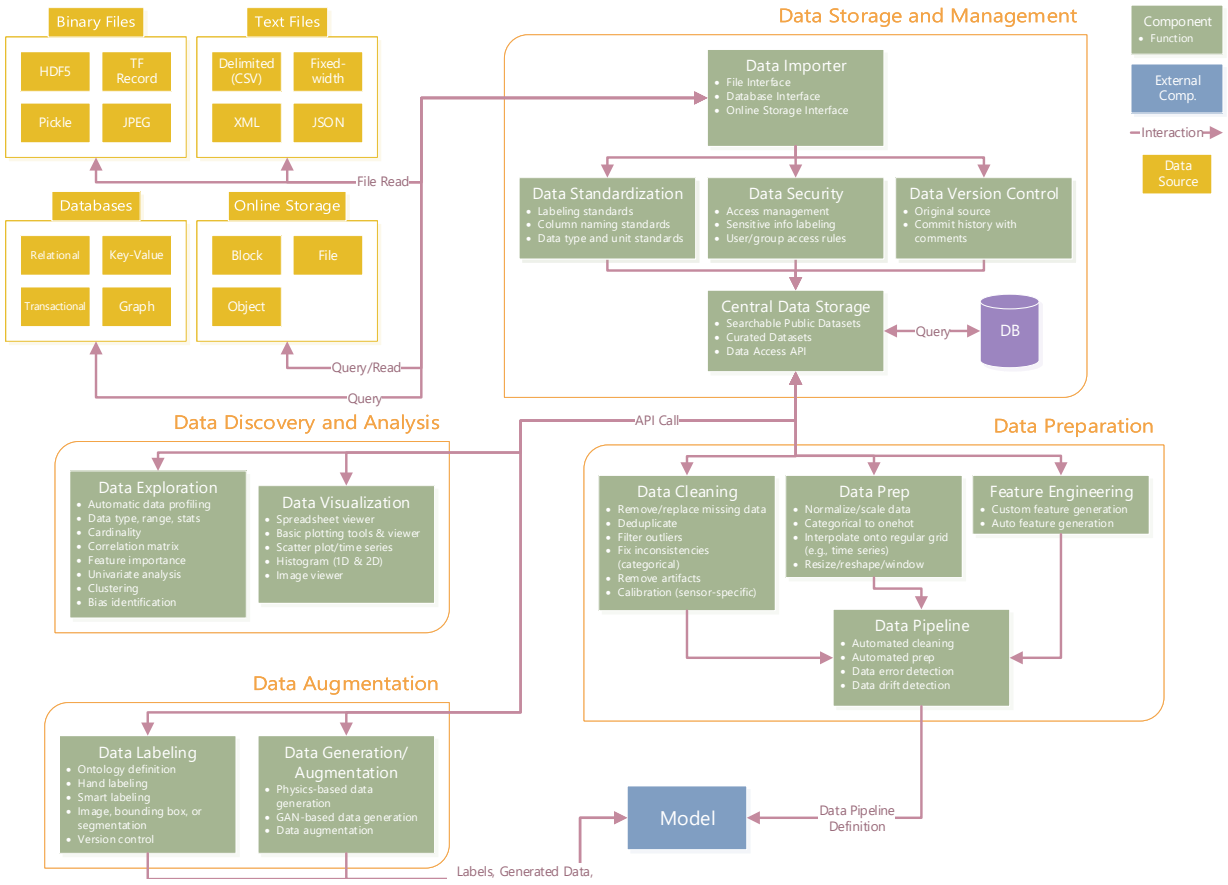

Fig. 1 Functional diagram of the Prepare step of the AI/ML workflow.

framework. A number of FOSS and COTS solutions exist and are implemented either as tools or APIs at the project level (e.g., DVC* or the tf.data.Dataset module†) or as an enterprise-level central dataset repository (e.g., Collibra‡).

Beyond storing the data, centralized dataset repositories provide additional features that aid in the management of datasets via data governance. Central dataset repositories can be tied to enterprise authentication services such as Active Directory to provide an easy-to-use and secure way for groups to manage access and modification permissions for their datasets. For unrestricted datasets, a centralized repository makes it easier for developers to find relevant datasets and offers a single source of truth for datasets. Standards can also be implemented at the enterprise level to ensure data is stored with consistent labeling, column naming, data type, data unit, etc. Data storage tools implemented at the project level, like DVC and TensorFlow's dataset API, can't offer the same enterprise-level features as a central data repository, but can provide other benefits such as being more portable to other systems, avoiding vendor lock, and not requiring enterprise licenses and management. Both enterprise-level and project-level data management solutions can offer data version control that tracks the original source and any modifications to a dataset, ensuring traceability of the data lineage.

The first step in most machine learning projects, once data is in hand, is exploratory data analysis (EDA). The goals of EDA are to understand the contents of the dataset, identify any trends that might be exploitable via AI/ML, and begin to shape the hypotheses and approaches to explore. The EDA step is often dataset-specific and requires a knowledgeable data scientist to generate visualizations and other analysis of the data including histograms of features or summary statistics, scatter plots comparing features, line plots of time series data, image visualization, metadata analysis, data distribution statistics, cardinality, missing or inconsistent data, correlations between features, univariate analysis, and clustering. Some of these analyses can be automated for some types of datasets. For example, the Pandas Profiling tool

*https://dvc.org/
†https://www.tensorflow.org/api_docs/python/tf/data/Dataset
‡https://www.collibra.com/

can automatically profile dataframes containing boolean, numerical, date, categorical, URL, path, file and image data[§].

The EDA step may be ongoing and revisited multiple times during the course of an AI/ML project. However, at any point, the developer can use information gathered via EDA to start cleaning and preparing the data for ingestion into a model. The purpose of data cleaning is to fix errors or other issues in the dataset that would hamper model performance. Data cleaning can include removing or replacing missing data, deduplication of data, filtering of outliers, fixing inconsistencies in labeling categorical labels, removing artifacts or other faulty data, and any number of sensor-specific calibrations such as flat-fielding image data. Data preparation takes the cleaned data and reformats it into a format better suited for ML models. Data preparation can include normalizing or rescaling the data, converting categorical data to one-hot encoding, interpolating onto a regular grid, resizing, reshaping, and windowing the data. Some data preparation steps may be applied in reverse to model outputs in order to transform the outputs back to the same view as the original data. Finally, feature engineering is the process of combing features of the dataset to build aggregate or derived features that might be easier for an ML model to learn from (e.g., combining the fuel efficiency and gas tank size of a car to calculate the maximum range). While feature engineering is often a manual, iterative, and experimental process, there are some tools that attempt to automate the search for good feature combinations (e.g., Featuretools[¶]). These three steps—cleaning, preparation, and feature engineering—may be performed at different times and the results stored off in intermediate datasets, or they may be combined into a data pipeline definition and executed on the raw data at training and inference time (e.g., scikit-learn Pipelines[‖]).

An additional step is sometimes required between preprocessing the data and training a model on the data. That is the data augmentation or generation step. For deep learning models that are prone to overfitting and often do not generalize well to data different from the training data, it is sometimes necessary to augment the training data or generate additional data based off the training data in order increase the robustness of the trained model. Data augmentation is typically applied to sensor data such as imagery or speech data where transformations to the data can mimic changes in how the data were collected. For example, changing the brightness of an image can mimic different lighting conditions or changing the speed and pitch of an audio sample can mimic changes in speaking speed. Data augmentation processes are typically defined by a transformation and an intensity value (which may be boolean) that is randomly chosen at training time for each sample in each new batch. For example, an augmentation process may be to rotate an image and the intensity is the degree of rotation, randomly chosen from a uniform distribution between -30 and $+30$ degrees. Data augmentation may occur before or after data cleaning and preparation; however, unlike data cleaning and preparation, data augmentation is non deterministic and is not applied to data at inference time. There are many data augmentation processes that can be applied independently or concurrently. The augmentation processes to apply to a dataset and how the intensity values are randomly chosen can be combined into a pipeline like Keras ImageDataGenerator[**]. Aside from scripted augmentation, data generation or synthesizing can be used to add additional training data. The data may be generated via physical modeling or DL-based approaches such as a GAN trained on the training data. The result of the Prepare step is a data pipeline, either manual or automated, which incorporates all of the data cleaning, preprocessing, augmentation and generation to produce the data used in the Model step to train models and test hypotheses.

B. Model

The Model step of the AI/ML workflow involves selecting an appropriate model for the application, defining the model architecture, and iterating over the process of training the model, tuning hyperparameters, and evaluating the model. Various components of the model selection, training, and tuning loop can be automated via approaches generally referred to as AutoML. Whether or not desirable results are achieved will determine if a new model needs to be selected or if the developer needs to return to the previous workflow and re-examine the available data. This workflow remains largely the same for ground, edge, or space, but constraints of the intended environment may limit the models available to choose from.

The model selection workflow, shown in Figure 2, starts with the developer identifying a model class, or classes, suitable to the desired task. The applicability of a model is dependent on multiple factors, including the type of data available, label availability, the problem objective, and constraints induced by the intended operational environment. For example, if labeled data is unavailable, the developer will generally not choose supervised learning models which require labeled data. The developer will then build the model architecture, usually with a machine learning framework

[§] https://github.com/pandas-profiling/pandas-profiling

[¶] https://www.featuretools.com/

[‖] https://scikit-learn.org/stable/modules/compose.html

[**] https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator

such as TensorFlow, and select a method for initializing the model parameters and hyperparameters (the parameters that are not directly learned from the data during the training process).

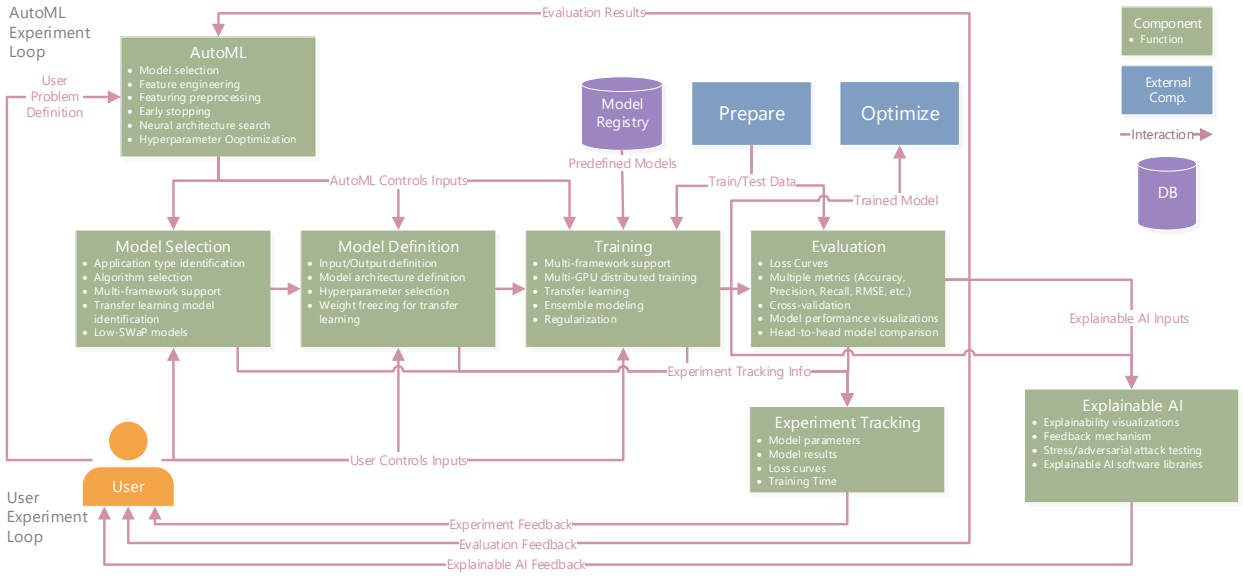

Fig. 2 Functional diagram of the Model step of the AI/ML workflow.

The next steps of the workflow involve training the model on the available data, evaluating results, and tuning model architecture or hyperparameters if necessary. Training refers to the process of providing a model with data such that it attempts to learn optimal model parameter values (weights) for inference. This is often done in conjunction with evaluation, where results (such as accuracy, precision, and recall) are evaluated to determine whether the model needs to be tuned and retrained. Tuning generally refers to the process of changing the architecture of the model, such as the number of layers in an neural network, or adjusting other hyperparameters, such as batch size or learning rate for training neural networks or the L_1 regularization parameter in a Lasso linear model. This process is iterative and usually requires machine learning expertise as the performance of the algorithm can be highly dependent on the choice of hyperparameters.

Various components of the model selection, training, and tuning loop can be automated via approaches generally referred to as AutoML. At a minimum, AutoML tools generally provide automated hyperparameter tuning through a standard grid or random search in a predefined hyperparameter space (e.g., scikit-learn's Grid Search module[††]). An exhaustive grid search can be too computationally intensive for some problems with large and high-dimension hyperparameter search spaces, especially when using K-fold cross validation. More advanced tools like auto-sklearn[‡‡] and AutoKeras[§§] can also automate the data preprocessing, model selection, and model architecture configuration (e.g., neural architecture search) steps and use more efficient search methods like Bayesian optimization. Some COTS ML platforms like Dataiku[¶¶] and H2O's Driverless AI[***] include even more automation features and offer near complete AutoML suites that guide users through an intuitive GUI to import their data, define their problem, and compare automatically generated models. However, these more advanced and intuitive AutoML suites are often limited in their application and work best on classification and regression of tabular data, with some extensions to a few time series, computer vision, and natural language processing applications.

Once multiple models have been trained and compared, an approach called ensemble modeling can be used to improve performance and robustness. Ensemble modeling can refer to approaches where many weak models are combined to create a strong model (e.g., bagging and boosting). These techniques are often directly implemented as their own model within a framework and the ensemble is built into the training scheme. However, ensemble modeling can

[††] https://scikit-learn.org/stable/modules/grid_search.html
[‡‡] https://automl.github.io/auto-sklearn/master/index.html
[§§] https://autokeras.com/
[¶¶] https://www.dataiku.com/
[***] https://www.h2o.ai/products/h2o-driverless-ai/

also refer to the technique of stacking a handful of strong models to average out each individual model's bias to create an even better performing model that is more generalized, such as stacking results from mulitple CNN architectures to achieve start-of-the-art performance on an image classification task.

The model selection workflow is similar whether the application is intended to run in a ground, edge, or space environment, but additional considerations may need to be taken into account based on the deployment environment. The field of machine learning has experienced rapid growth thanks to advances in computational power available on the ground and many machine learning-based applications leverage this power in both the training phase and the inference phase. If the intended environment is edge, mobile, or space, available computing resources are much more limited and can operate in vastly different ways than a standard CPU or GPU. Model performance, processing restrictions, and more must be considered when selecting a model to be deployed to the edge, mobile, or space. For limited compute environments, the developer has to consider: the size of the model, computational complexity, data rates, desired throughput, and many other factors. This will be dependent upon the available hardware and goals, as different hardware configurations will have varying amounts of compute power. Another consideration is model maintenance. Regular updates to the model can be difficult if the target hardware has limited uplink/downlink bandwidth. Limited downlink reduces the amount of new training data that can be collected to update the model. Active learning techniques to identify the usefulness of data in terms of the amount of information gained by including that data in a training dataset can help determine what data is most valuable to downlink. If the model is a large deep learning model, it may not fit the size or time constraints of the target hardware's uplink windows, but a simpler model might. When deploying to different environments, developers should consider the computational resources available, the system's availability for updating, as well as the required model maintenance.

Model evaluation is usually carried out alongside iterative model training and involves model analysis, head-to-head model comparison, and explainable AI. Applicable evaluation techniques differ based on the problem category and type of model chosen. Evaluation techniques result in a metric or set of metrics which are used to judge the performance of a model. The applicable metrics and evaluation workflow can also depend on the intended environment.

Models can be evaluated with various metrics depending on the category of models. Visualizing the training and validation loss functions over training iterations provides insight into how well a deep learning model has converged. For supervised classification models, popular metrics include but are not limited to accuracy, precision, recall, F1 score, and area under curve (AUC) of the receiver operating characteristic (ROC) curve. Visualizations like confusion matrices provide a more detailed overview of the performance of the model across different classes. For supervised regression models, metrics such as root mean square error (RMSE) and mean absolute error (MAE) provide a sense of the model performance across the entire dataset while residual plots can visualize performance as a function of input parameters. The same metrics generally would not apply to unsupervised models due to difference in output between the algorithms. Unsupervised model outputs are not compared to ground truth, so metrics for supervised models would not apply. Popular metrics for clustering (an application of unsupervised models), for example, include internal validity indices such as the Silhouette index, which aims to measure cluster cohesion and separation.

Explainable AI is an emerging field in machine learning that aims to provide better insight into how AI/ML systems make decisions. The broad acceptance of AI/ML systems can be limited by the inability of users to understand the system's decisions and actions [5]. By better understanding how AI models work, developers can design AI solutions to satisfy key performance indicators, correct errors, and mitigate bias [6] For example, a ship detection model may be learning the pattern of the ocean in the image rather than the characteristics of the ship itself. Catching this error via explainable techniques could aid developers correct the errors and ultimately produce a more reliable and trustworthy system.

One way to achieve more explainable AI is to choose models that are inherently more explainable than others due to their internal architecture. Simple models like decision trees, linear regression, and logistic regression are easier to interpret. For example, decision trees make a sequence of decisions to arrive at the final decision, where intermediate decisions can be analyzed directly to explain the output rationale. For these simpler models, there are often methods to calculate feature importance, which features have the biggest impact on the models decision, in an intuitive manner directly from the model. Neural networks, on the other hand, usually involve a large number of weights, all contributing in complex ways to a final output. Due to the sheer number of these weights and their non-linear relationships, it becomes extremely difficult to interpret how the inputs directly influence the output decision.

For complex models, there are some explainability techniques that help developers gain better insight into a models decision process and include explainability metrics, visualizations, and prediction explanation. Local Interpretable Model-agnostic Explainations (LIME) [7] and SHapley Additive exPlanations (SHAP) [8] are two black box explainability methods that are agnostic to the type of model used. They attempt to identify the most important input features

that lead to a specific output by perturbing the inputs and measuring the response at the output. For deep learning models specifically, gradient-based explainability methods like Class Activation Maps (CAMs) [9] and Grad-CAMs [10] can highlight regions in an image that had the most influence on the classification of the image. Darwin AI's GenSynth platform uses a counterfactual approach to identify which inputs (e.g., pixels) when removed result in incorrect classification [11].

If the intended environment is edge or space, additional evaluation and optimization techniques may be necessary. Performing thorough in-situ evaluation on models prior to deployment may not be feasible in some cases; however, some summary metrics such as number of parameters, model size in MB, number of operations in GFLOPS, and critical datapath length (CDL) [12] can still provide insight into the feasibility of deploying a model to a specific platform within a specific environment.

C. Optimize

Once a model has been trained, evaluated, and chosen for deployment, an optional next step is to optimize the model through various methods of compression and acceleration. The goal of the optimize step is to reduce the complexity of the model in order to reduce the size of the model on disk, the latency of running inference, and the power used per inference. For models deployed to ground/cloud environments where compute and power limitations are often of little concern beyond efficiency, the Optimize step is regularly skipped. For models deployed to edge/mobile and space environments, optimization can provide significant benefit and may, in some cases, be necessary to meet operational requirements. There are four major categories of model optimization: computation graph optimization, pruning, quantization, and hardware-specific optimization. Computation graph optimizations and hardware-specific optimizations often do not fundamentally change the results of the model and the functional mappings of input to output should be equivalent before and after these optimizations. Pruning and quantization, on the other hand, do fundamentally change the functional mapping of the model and can alter results.

Computation graph optimizations look at the network as a directed acyclic graph (DAG) and attempt to optimize the graph through elimination of no-ops and zero-dimension tensors, algebraic simplification, operator fusion, constant-folding, data layout transformations, and static memory planning [13]. Generally, these optimizations are not hardware-specific and can be applied to a high-level intermediate representation of the model such as Relay and ONNX; however, some operator fusions and layout transformations can by optimized based on the target hardware[†††].

When supplied with a training dataset, higher-level node pruning can be used to reduce the overall size of the computation graph without significantly reducing the accuracy of the model. These training-aware compression approaches attempt to trim away operations in the model that don't have a significant impact on the inference results, but add to the size and complexity of the model [14]. Node pruning alone can reduce the number of parameters in models like AlexNet and VGG-16 by an order of magnitude with negligible ($< 1\%$) impact on performance [15]. More advanced pruning techniques alter the entire architecture of the model to replace large, costly portions of the computational graph with smaller, more efficient subgraphs that approximate the replaced subgraph The Generative Synthesis approach applied in Darwin AI's GenSynth platform uses a generator-inquisitor pair to generate new models from a seed model that are subject to user-defined constraints and performance targets [16]. For models like ResNet-50 and InceptionV3, GenSynth can produce models with $1/3$rd-$1/7$th the number of parameters as the original model while maintaining accuracy within a few percent [17].

Quantization is a powerful approach to reducing the size and latency of a model by replacing the costly 32-bit floating-point operations typically required for training with 16-bit, 8-bit, or even down to 1-bit fixed point operations [18, 19]. Converting from FP-32 to INT-8 alone can reduce the size of the model to $1/4$th the original size [20]. As the dynamic range is significantly reduced when representing the model with 8-bit integers, there can be some degradation in performance, though it is usually minimal [18]. Training-aware quantization can often provide better accuracy results if a training dataset is available during the optimization step [21]. Latent AI's Efficient Inference Platform implements both post-training quantization and training-aware quantization for model compression. Without a training set, Latent AI's post training quantization can achieve bit-depths down to 3-bits without significantly reducing accuracy ($<2\%$) [22].

Hardware-specific optimizations can include elements of computation graph optimization, pruning, and quantization to tailor a model for inference on a specific device. The optimizations utilized will differ between hardware types (e.g. CPU, GPU, FPGA, TPU) and manufacturers (e.g. NVIDIA, Xilinx, AMD, Intel). The basic approach is to make the best use of hand-optimized kernels that have been designed for specific target hardware. This often is achieved through the

[†††] `https://www.onnxruntime.ai/docs/resources/graph-optimizations.html#extended-graph-optimizations`

use of deep learning libraries such as cuDNN for CUDA-based GPUs; oneDNN for Intel CPUs, GPUs, and FPGAs; Arm NN for Arm CPUs and GPUs; and MIOpen for AMD GPUs. These libraries can often be implemented as the backend for deep learning frameworks and provide optimized kernels for common deep learning operations such as multiply and accumulate and higher-level kernels for deep learning-specific fused operations such as 2D convolution + ReLU + Batch Normalization. For some edge devices, the trained model is executed on the device via a runtime inference API such as TensorFlow Lite or device-specific libraries such as Android NN API for Android devices and Core ML for iOS devices. For other target hardware, specialized SDKs are required to convert a trained model to a runtime inference engine. Vitis AI, for example, provides compression, quantization, and synthesization of deep learning models for running inference on Xilinx FPGAs[‡‡‡]. Similarly, TensorRT provides an all-in-one platform for graph-level optimization, quantization, and compilation of neural networks for NVIDIA GPUs[§§§]. Sometimes, hardware-specific optimizations happen in parallel with compilation of the runtime inference engine as part of the Integrate step of the workflow.

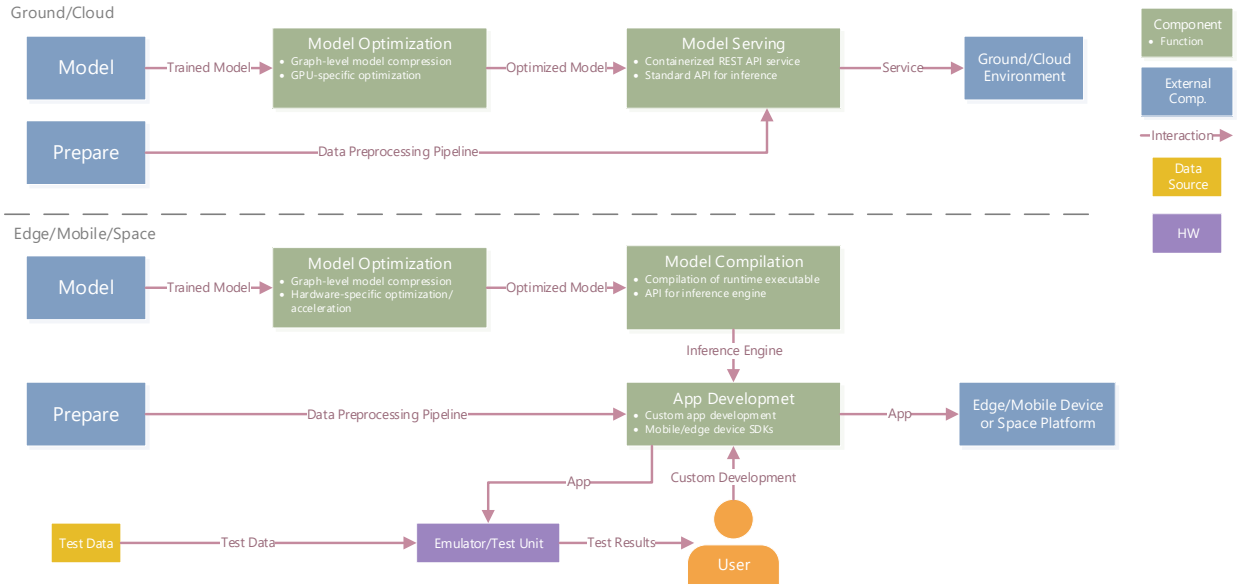

Fig. 3 Functional diagram of the Optimize and Integrate steps of the AI/ML workflow.

D. Integrate

In the Integrate step, the final steps are taken to bring a model from proof-of-concept to a deployed, full-fledged AI/ML solution. Broadly, this involves compiling the model into a runtime inference engine that can execute on the target hardware, productionizing the model by developing and integrating the support services required to operate in the target environment, and developing sustainment methods to monitor the data and model while in production. Figure 3 shows the functional diagram of the Optimize and Integrate steps.

There are many options for how to execute a model as a runtime inference engine. The simplest approach is to wrap an API around the model in the framework the model was developed in. This API can be made available through a RESTful service and deployed in the same environment used for training. Some frameworks have built-in support for this approach such as TensorFlow Serving[¶¶¶]. Another approach is to make use of a model deployment platform that orchestrates the deployment and sustainment of deep learning models to a scalable compute environment. Examples of tools that provide model deployment features include Dataiku[17], MLflow[18], and Modzy[19].

For edge/mobile and space deployments, additional steps are required to generate a runtime inference engine for

[‡‡‡] https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html
[§§§] https://developer.nvidia.com/tensorrt
[¶¶¶] https://www.tensorflow.org/tfx/guide/serving
[17] https://www.dataiku.com/
[18] https://mlflow.org/
[19] https://www.modzy.com/

a given model. Typically, the acceleration hardware and software libraries that are used for model training are not available on edge devices or space platforms. In this case, the model must first be compiled to run on the target hardware either by translating the model operations to low-level C or C++ code that can be directly compiled for the target hardware or through the use of a neural network runtime library that is already compiled to run on the target hardware and implemented through an API. Though the specific details of the process may differ across libraries, the overall flow is largely the same. First, the model is exported or converted to a format that can be read in by the runtime library. Next, as mentioned in Section IV.C, some libraries provide optimization features such as fusing operations and making use of highly optimized deep learning kernels supported on the target hardware. Finally, the runtime inference engine is exported as an executable on the target hardware. TensorRT[20], Vitis AI[21], and TFLite[22] are examples of such libraries that offer optimization and compilation in one.

While many of these runtime libraries are hardware-specific or offer limited support of different target hardware, there are more recent efforts to develop deep learning compilers that generalize the optimization and compilation of deep learning models for a wide variety of target hardware [23]. At the core of DL compilers is the intermediate representation (IR). DL compilers can implement multiple levels of IRs. For a high-level IR, the model is typically represented as a computation graph where non-hardware-specific optimizations can be made. Low-level IRs provide a more fine-grained representation of the computations that allow for more hardware-specific optimizations such as memory allocation, loop oriented optimizations, and parallelization. The IR can then be processed by one of many backends that replace low-level IR instructions with hardware intrinsics and kernels from available acceleration libraries. There can be an enormous number of parameters for tuning hardware-specific optimization prior to compilation. Automatically configuring the HW-specific optimizations is called auto-tuning and approaches vary based on the parameterization of the configuration, the cost model, and the search technique. TVM is Apache's DL compiler and implements the Relay IR for computation graph representation, a Halide-based low-level IR, and a machine-learning-based cost model for auto tuning [13]. Other DL compilers include nGraph [24] from Intel, Tensor Comprehensions [25] and Glow [26] both from Facebook, and TensorFlow's XLA [23] from Google.

Once the model has been compiled into an optimized inference engine, it can be deployed to the target hardware, but this is typically not the end of development. Beyond the inference engine, other support services and software infrastructure may need to be developed to aide in the execution and sustainment of the deployed model such as for data streaming and preprocessing, model result postprocessing, and data/model monitoring. This development effort can be significant for edge and space applications where the deployment environment is dramatically different than the development environment and standard tools and software do not exist. Beyond these services, additional features to support MLOps such as a model registry or feature store may be required (see Section IV.F for more info). All these services need to be developed, integrated, tested, and compiled for the target hardware before the entire system is ready for deployment.

For space deployments where the variability of the natural environment can directly influence the sensors and systems providing inputs to the deep learning model, of particular interest for support services is a robust sustainment approach to ensure a continuous level of model performance. Some cases one might want to monitor for are data drift where the distribution of the input data changes over time, model prediction drift where the distribution of model predictions change over time, model performance degradation where the accuracy of the model results or reported confidence decrease over time, and outliers where some individual inputs or clusters of inputs are significantly different than any inputs in the training dataset. Beyond incidental changes in data and model performance, deployed ML models are at risk of purposefully being tricked via adversarial attacks [27]. Additional safeguards such as model ensembles and V&V techniques can provide some protection against adversarial attacks. Finally, specific to space deployments, radiation-induced faults can affect model performance [28]. The limited operations used in deep learning models and the hardware used to accelerate them may offer opportunities for fault mitigation specific to inference with deep learning models.

E. Certify

The previous sections covered a standard workflow for development and integration of AI/ML systems with environment specific hardware and some of the necessary support services. Each of the components in an AI/ML system

[20] https://developer.nvidia.com/tensorrt

[21] https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

[22] https://www.tensorflow.org/lite

[23] https://www.tensorflow.org/xla

need to be tested in a suitable environment to validate key operational requirements are met and to assess the safety/risk of the system. The need for testing is critical for the space environment, where a malfunction or unintended decision can cost billions and/or cause casualties. For software/hardware bound for space vehicles such testing is common practice; a Technology Readiness Level (TRL) system, developed by NASA, is used to describe the amount of development/testing a system has undergone [29]. However, deployment of AI/ML carries additional risk that needs to be accounted for. The complex nature of AI/ML models, which are often treated as black boxes, makes the logic behind each decision opaque. The need for extensive testing is additionally highlighted by the fact that the processing hardware on space vehicles can be an entirely different type of processor than what was used to train the model. It is not straightforward to predict how a model will perform on an FPGA based on how it performed on a GPU or CPU. In any of the environments under consideration, as the level of risk grows so to should the level of trust needed to deploy, which requires rigorous and thorough testing of resource consumption, system/environment interactions, and security vulnerabilities.

Each of the components or artifacts in an AI/ML system should be tested under conditions as similar as possible to the intended deployment environment. For ground this is relatively straightforward as AI/ML systems are commonly run in this environment and well established methods already exist. The widely available compute in the ground environment not only reduces the need to optimize models over various physical constraints, it also allows for running shadow models (models that undergo testing on real-time data but whose outputs are only compared to the actual outcome and the current production version of a model, if one exists) with continuous monitoring. For the edge, mobile, tiny, and space environments, real-time testing is more difficult to achieve. The limited compute available in these environments makes it difficult if not impossible to deploy shadow models while the current version of a model is in operation. In this situation either downtime for the service needs to be allowed to run a shadow model until satisfactory operation is shown, or the shadow model must be run on the ground on equivalent hardware and data. With space deployments, the latter method is complicated by latency issues related to communication with satellites, and the inability to always be online.

AI/ML systems deployed on space vehicles need to achieve desired levels of performance while constrained on processing power, power consumption, heat generation, volatile/non-volatile memory usage, and other limitations introduced by the operational environment. The AI/ML system as a whole, including data ingestion/preprocessing, model inference, output postprocessing, data/model monitoring, and decision making need to be tested against the same metrics as the model inside it. Testing the model and AI/ML system components separately allows for identification of bottlenecks and performance issues while testing the system as a whole validates overall performance. For example, it is often possible to simplify a model through creative feature engineering, but if the creation of these features requires extensive processing or is inefficient, the AI/ML system may still require more than the available memory despite the model fitting within the constraints.

Additionally, it would be necessary to test the AI/ML system integration with the other systems it would be interacting with on the target device (e.g., the datastream from a sensor, another AI/ML system, onboard mission management, or constellation management). This is needed to not only ensure all software and hardware dependencies are met, but to also investigate for possible feedback loops. The various systems onboard a space vehicle can interact directly by transmitting messages to each other and indirectly through a chain of events and resource consumption. The same resources needed by the AI/ML system will often be shared with other software for mission related applications and vehicle or payload control. An AI/ML system that consumes more than the initially planned amount of resources can cause the AI/ML system or another system to fail, which can result in mission failure and, depending on the level of autonomous control and interaction, damage the space vehicle and other assets or loss of life. Thus there is a need to rigorously stress test AI/ML systems intended for these environments under the expected resource constraints and develop a detailed understanding of how allocation of resources for the AI/ML system impacts the additional systems onboard over the mission of a space vehicle. It will be desireable to deploy AI/ML systems across many types of devices, like an app on a cellphone or tablet, and it is necessary to develop profiles of performance and resource consumption for every environment the AI/ML system will be deployed in.

Beyond just meeting goals for performance and resource constraints there are other risk factors an AI/ML system must be tested against before deployment. Part of certifying an AI/ML system should include developing profiles of the training and testing data to detect anomalous data and bias, which ideally took place during the prepare and model steps of the workflow. Additionally, high-quality datasets for these environments can be difficult to come by and relying on augmentation or simulations potentially introduces unknown vulnerabilities. Questions, such as "How will the model respond to novel inputs?" or "Under what conditions do we expect the outputs of the model to be valid?" need to be addressed. The performance of an AI/ML system will naturally degrade over time whether due to data drift, changes in data pipeline, or corruption of the model weights in memory. Data drift refers to the tendency of the distribution of data to change over time. For example; the phase, amplitude, and relative position of the individual elements in a phased

array can change due to vibrations in the array or other environmental effects; the signal arriving at an array can be impacted by external mechanical and natural sources of noise. The impact of both of these examples is a received signal with statistically significant differences over time. Data drift can cause a model that once operated within requirements to perform no better than random guessing. Part of the certification process of an AI/ML system should be to understand how model degradation can occur, what are the signs it is happening, and what is the best course of action when it does occur. This testing process may involve simulating data drift to observe how the model performs on data distributions it was not trained on and from there define a boundary for when the model results should not be trusted. Methods to monitor the inputs to and outputs of a model for data drift and degradation will need to be implemented into the AI/ML system.

Deploying AI/ML onboard satellites also creates new potential vectors for cyber attacks. Machine learning models do not learn perfectly and sometimes the training of the model can result in the learning of non-salient features that, while informative, can be exploited to cause the model to make erroneous predictions [30]. The training data itself can introduce vulnerabilities both unitintentially through inadequate curation, or intentionally via a data poisoning attack (where the training data is corrupted by a malicious individual or group). The fact that machine learning models can be spoofed has become common knowledge in the AI/ML community, and an active area of research involves both developing methods to attack AI/ML systems as well as protect them [31–33]. Adversaries could leverage publicly accessible information pertaining to common model architectures and spoofing methods to produce erroneous results from AI/ML systems. AI/ML systems are also vulnerable to standard cyber attacks since it is still software. Overall, adding AI/ML increases the potential attack surface, and the certification of an AI/ML system should include probing the potential security risks the system presents.

Certifying the operation of AI/ML systems is a vital process for mitigating risks and producing products capable of meeting the demanding needs of end users. Standardization of testing and validation practices will allow for automation of certification in the an MLOps pipeline (discussed in more detail in the next section), produce products with consistent quality, and instill confidence in users. The certify step is the final hurdle to overcome before having a deployable product and also serves as an opportunity to demonstrate the quality of a product that has been developed.

F. MLOps

Up to this point we have covered the steps involved in developing a model for various environments. We will now discuss how these steps can be combined and automated through the DevOps practices of Continuous Integration (CI) and Continuous Delivery (CD). Continuous Integration involves building, testing, and packaging as new code is pushed to a source code repository. For AI/ML systems this can also include building artifacts needed for running in the target environment, building and testing compatibility with the target environment, and testing if training converges. The input to the CI stage will be an automated model pipeline and the output will be the same pipeline packaged for the target environment. Continuous Delivery involves pushing new packages, manually or automatically, to the target environment as they become available. There are two steps to Continuous Delivery with an AI/ML system. First a package containing an automated model pipeline will be pushed to the target environment. Then the automated model pipeline will build and serve a model as a service or application. Additionally, with AI/ML systems two more practices, Continuous Training (CT) and Continuous Monitoring, are added. Continuous Training involves training a model while it is in service to maintain an acceptable performance level, this is also implemented by the automated model pipeline. Continuous Monitoring involves monitoring the input data and results of the model inference for anomalies or drift. Collectively the four practices of CI, CD, CT, and CM are often referred to as MLOps. The goal of MLOps is to introduce automation and monitoring throughout the life cycle of an AI/ML system to ensure only a reliable and accurate service is deployed.

We present in this section a guideline for implementing MLOps in three distinct environments: ground, edge/mobile[24], and space. MLOps on ground is the simplest due to the availability of powerful computing resources and the relative ease with which monitoring can be conducted. We base our guidelines for MLOps on ground off of Ref. [34] and from this source extrapolate guidelines for the edge/mobile and space environments.

1. Ground MLOps

In Figure 4 we show the general steps and artifacts for MLOps on ground. MLOps on ground can be divided into two main components: the Develop phase and the Deploy phase. Continuous Integration and Continuous Training occur

[24] In this section we do not explicitly refer to the tiny environment (which consists of microcontrollers such as Arduino and other small devices with no OS on board) as it can be derived from the edge/mobile guidelines for MLOps.

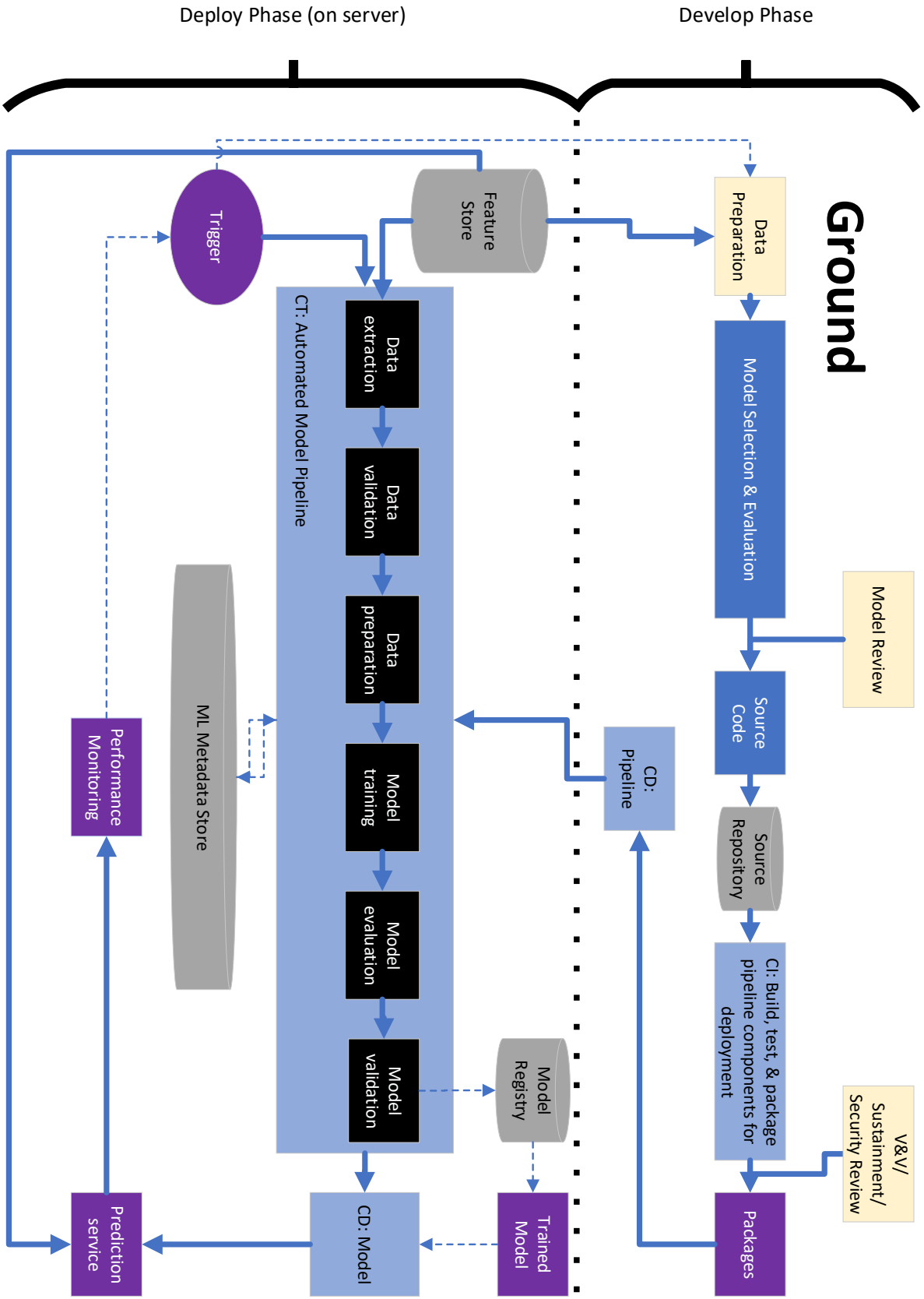

Fig. 4 Diagram of artifacts, pipelines, and processes needed to implement MLOps on a ground environment. Adapted from [34]

during the Develop phase and Deploy phase respectively. Continuous Delivery occurs across both phases where in the Develop phase an automated model pipeline is delivered and in the Deploy phase a model is delivered. Continuous Monitoring occurs in the Deploy phase.

The Prepare, Model, Optimize, and Integrate steps described in the previous sections occur during the Develop phase (Sections IV.A-IV.D). These workflows generally involve exploratory analysis and at any point the developer can be redirected to a previous step or workflow. Generally development begins with data preparation, and continues to model selection, training, and evaluation. In the data preparation stage the model inputs should be kept in a feature store that will also be accessible during the Deploy phase. A feature store is a centralized repository where you standardize the definition, storage, and access of features for training and serving [34]. After evaluation the model should be reviewed to ensure minimum standards are met. Upon meeting those standards the source code of the model is stored in a repository, from which it can be retrieved for deployment. At this stage, Continuous Integration, which envelops the model deployment workflow, begins. The model deployment workflow involves converting the model source code into a format compatible with the target processor and the development of artifacts to maintain and monitor the model. The artifacts of the AI/ML system are then tested to ensure proper execution and behavior.
These artifacts will generally consist of:

- An automated model pipeline which should perform the following processes:
  - Extract data from the feature store
  - Validate the data
  - Prepare the data for training
  - Train a new model
  - Evaluate the new model
  - Validate the new model's performance against previous versions
  - Incorporate the optimal model into the prediction service
- A model registry which stores previous versions of the model produced by the automated model pipeline.
- A metadata store to track previous executions of the automated model pipeline to help with reproducibility, comparisons, and error analysis.
- A monitoring system to track various statistics of the models performance.
- A trigger system to determine when to run the automated model pipeline or restart the Develop phase. The trigger can be determined by performance of the model and/or a schedule.

After development, the artifacts are tested to ensure proper function and performance. Next a final review is conducted before the packaged automated model pipeline is released for delivery to the target environment.

The major steps in the Develop phase are largely the same across the three environments and the differences at workflow level have been discussed in the previous sections. The main differences are the types of artifacts built during Continuous Integration, which we cover in the next sections.

During the Deploy phase, the packaged automated model pipeline from the Develop phase is built in the target environment. After the automated model pipeline is built it trains and delivers a model for a prediction service. The model is then monitored for a model performance or schedule based trigger to update the model either through the automated model pipeline or by restarting the Develop phase. After a new model is trained and validated to have optimal performance the prediction service is updated with the new model and the cycle continues. This phase varies the most across the three environments as the impact of limited available computation power and communication must be considered.

2. Edge/Mobile MLOps

Deploying an AI/ML system in the edge/mobile environment adds additional complexity to MLOps. Instead of the Deploy phase existing entirely on connected servers it is split across both a server and an edge or mobile device. For simplicity, for the remainder of this section we will refer only to edge devices as the MLOps practices are the same. How the Deploy phase is split across the server and edge device depends on the computational demands of the AI/ML system and the computational power of the edge device.

In Figure 5 we show a diagram of MLOps for the edge environment with three different levels of complexity for deployment. We refer to the simplest deployment as Deployment Lvl. 0. For Deployment Lvl. 0 the edge device serves only to collect data and perform minor processing before transmitting the data to the server hosting the remaining artifacts and services.

For Deployment Lvl. 1 the prediction service is either simple enough or the edge device is powerful enough to

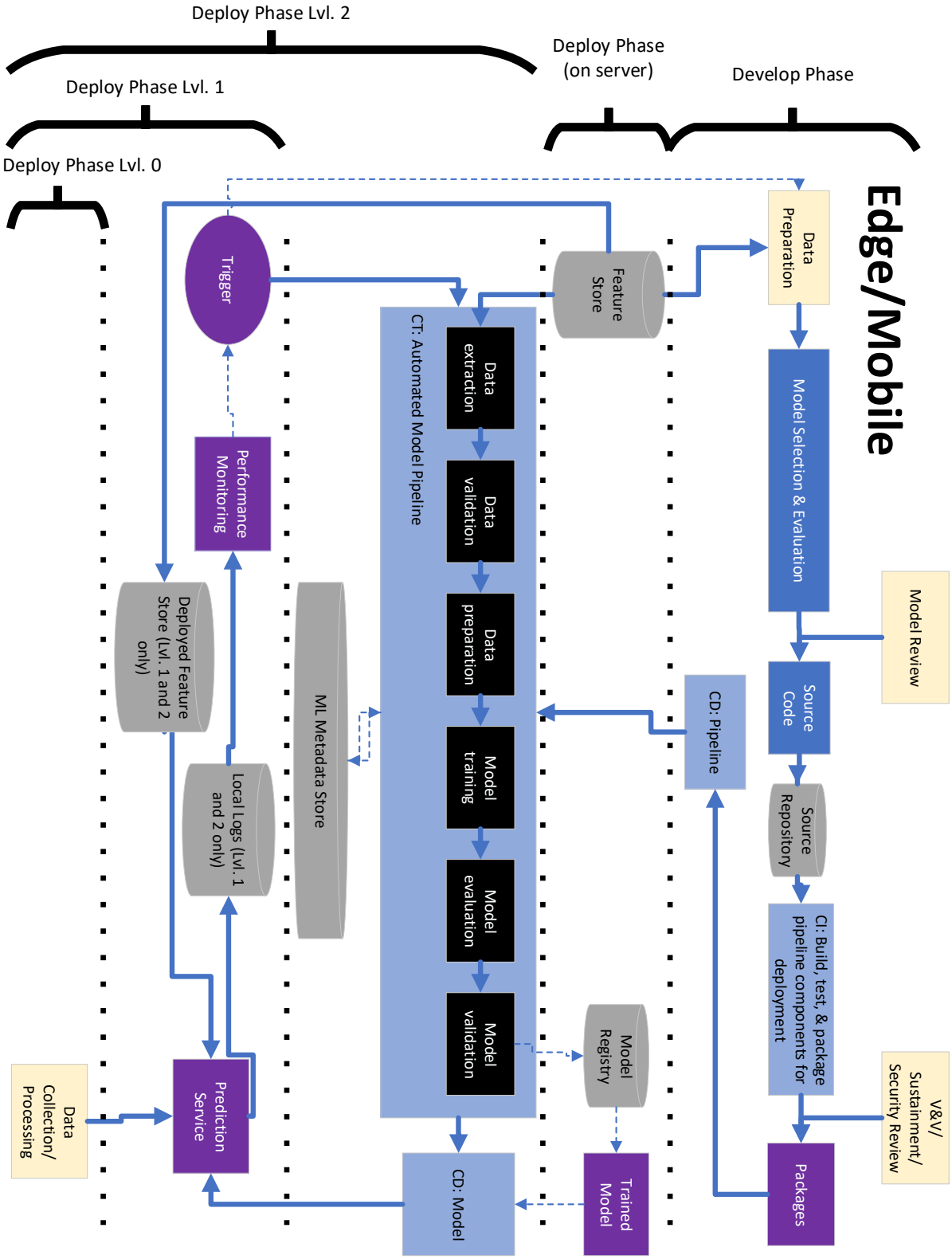

Fig. 5 Diagram of artifacts, pipelines, and processes needed to implement MLOps on a ground environment. Adapted from [34]

host the prediction service. This level of deployment requires the creation of two new optional artifacts during the Develop phase. The first is a system to locally log performance metrics of the model. The second is a smaller deployed version of the feature store. The purpose of these two artifacts is to reduce communication between the server and the edge device, increasing the speed with which an alert of performance degradation can be made and reducing prediction latency. The automated model pipeline and other artifacts necessary for Continuous Training remain on the host server.

The final level of deployment, Lvl. 2, does not add any additional artifacts but now the model is simple enough or the edge device powerful enough for the entire automated model pipeline to be hosted on the edge device. This final level of deployment minimizes the time between model degradation being noticed and the delivery of a new automatically produced model.

3. Space MLOps

Conducting MLOps in space requires addressing several additional complicating factors. In the space environment, not only is the available computing power extremely limited and shared across many applications, there are thermal/power duty cycles, limited telemetry, and radiation induced faults to contend with. Onboard AI/ML systems must also be made to work within the CONOPs of a satellite's mission.

Due to the limited compute available on a satellite it will generally not be possible to deploy a fully automated model pipeline capable of training and delivering a new model onboard. Even if the onboard processor was capable of handling the training of a model, the often long training times would interfere with the ability to perform other onboard processing, interfering with the mission. Additionally large amounts of data are necessary for training models, which would impact the limited data storage available onboard. Realistically, only the simplest of training schemes are feasible onboard with state of the art processors, such as online training of simple models. As a workaround to these limitations we propose using two automated model pipelines for AI/ML systems onboard satellites, as shown in Figure 6. The first of these pipelines would be the now familiar automated model pipeline deployed on a server, capable of handling the delivery of a new model from the data extraction stage all the way to model validation. The second pipeline (onboard model pipeline) would be a simpler version running onboard a satellite. The purpose of the onboard model pipeline would be to perform basic maintenance of a model, fault mitigation, and validation of the model's performance until a new model can be uplinked to the satellite.

In addition to the onboard model pipeline, methods will need to be put in place to handle uplink and downlink of model updates. For most missions there will only be a limited window of opportunity to handle the transmission of data and the sustainment plan of a model needs to be built around these windows. The developers will need to consider strategies such as making model updates small enough to fit within a single attempt to transmit or having the ability to send larger updates over multiple transmissions. There will also be additional considerations with what data to transmit back to the ground. The retraining a model may require the collection and transmission of additional data meant to be processed onboard.

In the space environment, limited computing power and varying duty cycles mean any AI/ML system will likely not be running at all times. All of the components of the AI/ML system must be designed to handle interruptions due to the normal scheduling of mission processing, loss of power, and overheating. Each step in the MLOps process needs to be designed to execute only when there are enough resources available to complete the step and methods need to be built in to handle incomplete execution of steps.

Conducting MLOps in the space environment adds significant complexity, but it is necessary. Some of this additional complexity comes from the need to build additional artifacts to maintain and schedule the AI/ML system. The greatest source of complexity is likely to be incorporating a plan operating, monitoring, and updating the system within the constraints of a mission's CONOPs. It is not a matter of if an AI/ML system's performance will degrade, but when. Different AI/ML system's will degrade at different rates but it is guaranteed to happen and procedures need to be in place to ensure the validity of automated mission critical decisions.

V. Hardware Landscape

To know what is possible with AI/ML in space it is necessary to have an understanding of the different types of processors or accelerators that are available for computation and their capabilities. Here we provide a brief overview of several types of processors and highlight their strengths and weaknesses for performing AI/ML acceleration in space. We then follow up with a discussion on specific space-grade computation hardware that is currently on the market and what we may see in the future. We note that in this section we only consider applications of using these processors for inference only and do not consider the applicability of different hardware for training.

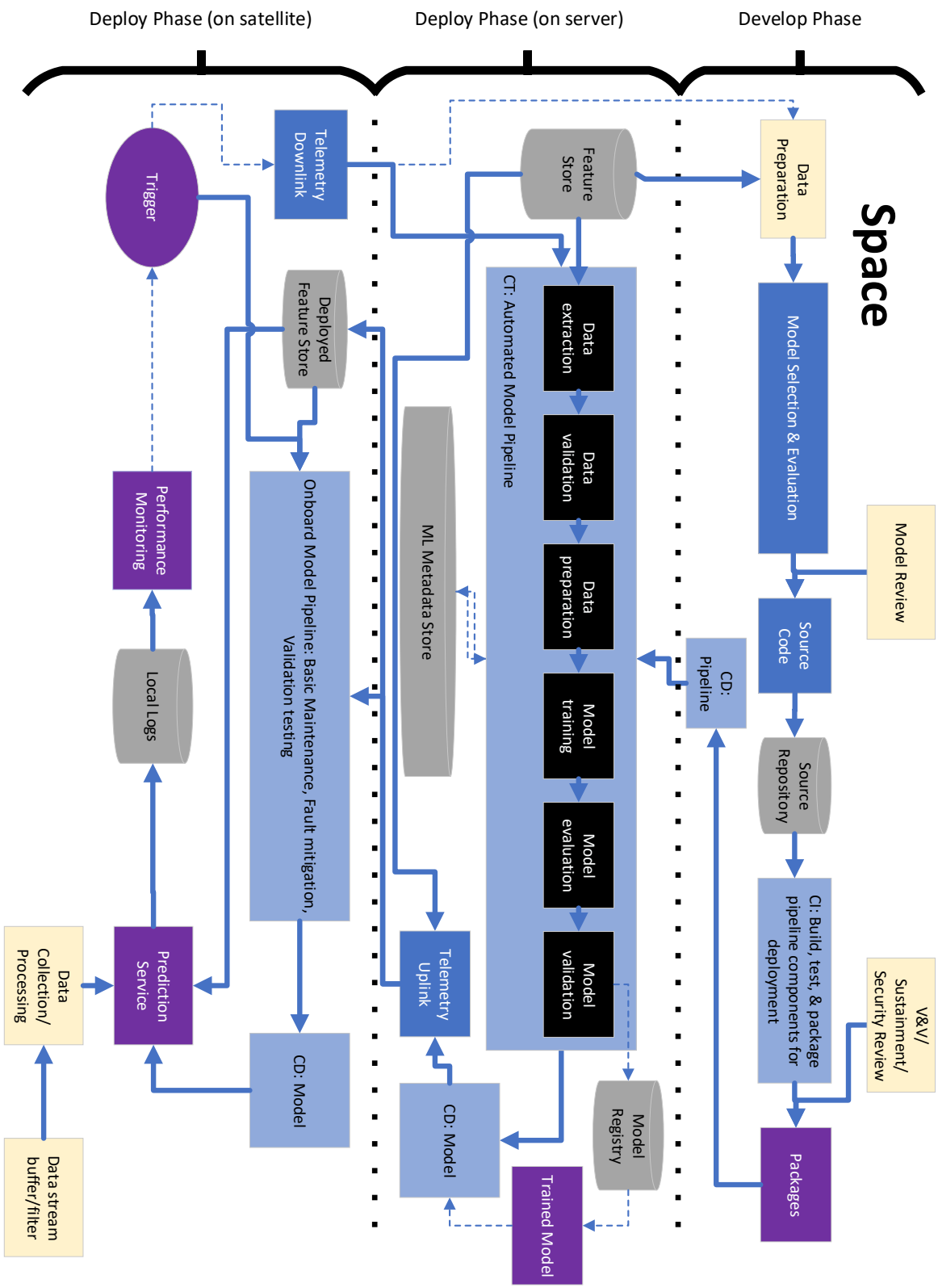

Fig. 6 Diagram of artifacts, pipelines, and processes needed to implement MLOps on a ground environment. Adapted from [34]

Throughout this section the following processing units will be discussed:

- CPU: Central Processing Unit
- FPGA: Field Programmable Gate Array
- GPU: Graphics Processing Unit

Generally speaking, a CPU is good at evaluating complicated decision trees and performing operations one task at a time. Conversely, a GPU is not very well suited for evaluating decision trees, but is very good at computing vectorized workloads such as matrix multiplication. This optimization for vectorized operations has lead to GPUs being the currently preferred computation unit for AI/ML as many AI/ML algorithms, deep learning in particular, are based on linear algebra. FPGAs are optimized to be a blank slate where any combination of digital logic can be implemented, though they generally lack in the high-bandwidth memory that GPUs tend to have. FPGAs have several advantages when it comes to AI/ML computations, which we discuss below, leading to increased interest in the units. In the following discussion we provide a comparison of the computation units with respect to AI/ML processing.

A. Processor Comparison

In order to better compare these very different computational units, the scope of comparison will be limited to the application of AI/ML. The subjects to be compared will include the following:

- Throughput: the quantity of data being processed through a system within a unit of time.
- Latency: the time taken from an input action to result in an output from the system.
- Sensor Integration: the process of retrieving data from a sensor, performing pre-processing steps and feeding the results into the model.
- Ease of Update: the difficulty in the process required to change the deployed model functionality.
- Radiation Tolerance: the tolerance of each piece of hardware to radiation exposure and commercial availability.
- Ease of development: the difficulty of deploying a model to each type of hardware.
- Power: the efficiency of computations with respect to power consumption for each type of hardware.

1. Throughput and Latency

The venerable CPU is capable of evaluating nearly any model due to the general purpose nature of the device[25]. However, it is not very performant in either throughput or latency due to the small number of parallel processes it can evaluate. Even with the comparably faster clock rates against both GPU's and FPGA's, both its throughput and latency can be lacking due to the smaller number of computation cores.

FPGAs on the other hand offer great performance with high throughput and low latency. FPGAs can inherently provide low latency as well as deterministic latency for real-time applications such as consuming a continuous data stream from a sensor and directly feeding the data to the model, which would bypass an ordinary CPU. Typically designers will build a neural network from the ground up and structure the FPGA to best suit the model. FPGAs can offer performance advantages over GPUs when the application demands low latency and low batch sizes – for example, with speech recognition and other "real-time" analysis workloads.

GPUs instead are able to provide a higher bandwidth than FPGAs, at the cost of higher latency. They are better for highly parallel computations, such as training deep neural networks, thanks to the very high memory bandwidth. Bandwidth is one of the main reasons why GPUs are faster for computing over CPUs. GPU cores are usually organized into blocks of 32 cores that all execute the same instruction on multiple data points at the same time making vectorized math much quicker to implement. Modern GPUs tend to have over two-thousand cores depending on the application environment, allowing for a huge amount of data to be processed in one clock cycle. Ignoring the transfer time from storage or RAM to a GPU's VRAM, GPUs have lower latency than CPUs for highly parallelized algorithms since they devote proportionally more transistors to arithmetic logic units and fewer to caches and flow control. GPUs found in edge devices will only have cores numbering in the hundreds, still allowing for a greater bandwidth than CPUs and FPGAs also found in edge devices. GPU's however require a CPU or FPGA to command the high-level logic to drive computations.

[25] Some models such as spiking neural networks and quantum neural networks are designed for different computing architectures but can be simulated on general purpose processors.

2. Sensor Integration

Sensor integration is a very broad topic, therefore the scope of this section will be limited to the flow of data. A CPU or GPU is likely to retrieve data through a peripheral interface. Peripheral interfaces will add additional latency to the pipeline and will potentially add a new bottleneck for data ingestion. However, the peripheral devices often can work off of direct-memory access which minimizes CPU overhead for acquiring the data. If preprocessing is required, both throughput and latency will be negatively impacted.

FPGAs can help overcome I/O bottlenecks. They are often used where data must traverse many different networks at low latency. They're useful for eliminating memory buffering and overcoming I/O bottlenecks—one of the most limiting factors in AI system performance. By accelerating data ingestion, FPGAs can speed the entire AI work-flow. FPGAs can also enable sensor fusion with minimal overhead. FPGAs excel when handling data input from multiple sensors, such as cameras, LIDAR, and other sensors.

3. Radiation Tolerance

CPUs and FPGAs have come in radiation hardened and tolerant models for decades thanks to the demand in the space industry. GPUs on the other hand have seen little demand for radiation tolerant models until recently and options available on the market remain limited. Most processing demanding highly parallelized computation has been performed on the ground. As there are no radiation hardened GPU designs, most efforts to make radiation tolerant GPUs involve shielding low-power GPUs designed for edge applications.

4. Power

CPUs are generally very power efficient for generalized workloads, but would require much more total energy to complete vectorized tasks than either FPGAs or GPUs. This is due to the serial nature of CPUs, they are optimized for control flow that allows for code branches to occur with little penalty.

GPUs tend to be more power efficient than CPUs when applied to a vectorized workload. They work well when operating on large independent datasets due to the exploitation of data and thread level parallelism. They also work very well on linear code that have very few changes in control flow. GPUs generally don't do well with branching code since it can starve functional units of work leaving them idle. They also don't generally have large caches, and the individual functional units are relatively slow which makes mispredicting a branch result in potentially serious performance and power penalties.

With FPGAs, designers can fine-tune the hardware to the application, helping meet power efficiency requirements. FPGAs can also accommodate multiple functions (e.g. sensor fusion), delivering more energy efficiency from the chip. It's possible to use a portion of an FPGA for a function, rather than the entire chip, allowing the FPGA to host multiple functions in parallel.

5. Ease of Development

Having the greatest possible computational performance does little good if it is exceptionally difficult to develop AI/ML models for the hardware. Most AI/ML model development libraries (e.g., Tensorflow and PyTorch) provide robust support for compiling models on a CPU. Compiling a model to a GPU requires additional effort to efficiently parallelize the computations across the many cores. In the case of NVIDIA GPUs this can be accomplished with CUDA and cuDNN libraries developed by NVIDIA. GPUs from other manufacturers require similar drivers and libraries. There is broad support for deep learning operations on GPUs as they are currently the primary accelerators of deep learning models. Initial installation of the drivers and libraries can be challenging but methods such as containerization have made this process easier as it only needs to be done once. The need to install these additional drivers and libraries makes deploying an AI/ML model on a GPU a little more difficult than a CPU, but the massive performance increase due to the parallelization makes the effort worthwhile. FPGAs can prove significantly more challenging to deploy AI/ML models to. Only recently have manufacturers started to provide tools to ease development of deep learning models for FPGAs, like Xilinx's Vitis AI. This presents another difficulty as each manufacturer provides a different toolset. Additionally the available tools for compiling deep learning models on FPGAs provide only limited support for deep learning operations. Deploying a model on a FPGA can require significant changes to a deep learning model to account for unsupported operations; this compounded with the steep learning curve associated with programming for FPGAs can lead to costly development.

6. Ease of Update

The process for updating an active model can vary between the various platforms. For CPUs and GPUs, the process usually embraces conventional and robust methods of swapping over runtime control to a new model. This usually results in minimal down time and allows for automatic fallback in the event of a failed update. FPGAs on the other hand require the logic fabric to be re-flashed which requires the hardware to go offline during the update process. The fabric is also unable to revert itself to the previous design in the event of a failure, the device overseeing the flash of the fabric will need to maintain that logic.

7. Summary

Figure 7 shows a notional comparison of the strengths of each hardware option. The further the representative area reaches from the origin of the chart, the better the performance in the labeled category. Generally speaking, the more surface area covered by each type of hardware the better the hardware is at performing AI/ML operations onboard space vehicles. As an example, for applications where latency and radiation tolerance are the key criteria (equally rated), then the FPGA would be the best choice since GPUs aren't radiation hardened at this time. Additionally, each type of hardware could theoretically be combined to achieve maximal performance in most categories. One such combination could be the combination of a GPU with an FPGA that has an on-board microprocessor.

B. Heterogeneous Compute

Heterogeneous computing refers to systems that use more than one type of processing core. These systems can benefit from higher processing efficiency by adding coprocessors that can perform specialized computing tasks, e.g. a FPU (floating-point unit), GPU, or FPGA. For example, GPU's need a CPU for the high-level logic but the CPU is also useful for preparing batches of data while the GPU is performing inference operations. An FPGA and GPU combination can also be beneficial by running small low-latency models on the FPGA and then utilizing the GPU when a more complicated model should be applied to the data. An example would be a two stage system for Automatic Target Recognition (ATR). Data streaming from a camera can produce more frames per second than an ATR model is able to evaluate. Instead it would be more efficient to deploy a low-latency model on the FPGA to detect important frames and then send the selected frames to a more complex model on the GPU, where targets would be identified. These systems do have their own challenges though that are not found in homogeneous systems. The presence of multiple processing cores have all the same issues involved with homogeneous parallel processing systems while adding potential non-uniformity in system development, programming practices, and overall system capability. As an example there needs to be an interface defined between various types of hardware, in the case of a CPU/GPU system this interface is a well-defined protocol on the PCIe bus. However, when interfacing a generic CPU with an FPGA the interface is not well defined and could involve a lot of overhead to design a solution. This is not always the case though, when using a Xilinx SOC with an embedded processor developers can make use of integrated solutions such as the proprietary AXI bus or DMA.

There are some options that may provide better overall performance in the future. Most of the examples are purpose-built ASICs that target AI/ML applications to accelerate training and inference of deep learning models or, in some cases, focus on inference-only applications. Some examples include ASICs such as the Tensor Processing Units (TPUs) and Edge TPUs coming from Google, and neuromorphic processors being developed by Intel and other manufacturers.

C. Space Hardware Landscape

In the previous sections we compared the three major types of computational units available in the space environment. In this section we look at the currently available processors utilized in space and compare them to what will and may be available in the near future. The primary metric we use for comparing computational performance is Giga-Floating-Point Operations Per Second (GFLOPS) which, for most devices, can be found either from datasheets or benchmark tests[35–55]. Additionally the number of GFLOPs for a single inference with a 32-bit neural network model can be calculated and the ratio of GFLOPs to GFLOPS can then provide a rough estimate of the time required for a single inference of a model on a particular type of hardware. We also use peak operating power for a device to estimate GFLOPS per Watt (GFLOPS/W) and use this as an indicator of computational efficiency. In Figure 8 we present a scatter plot of GFLOPS against peak operating power in Watts for various computational devices. Figure 8 is meant to provide a rough estimate of the performance of the various devices under consideration, to facilitate a discussion on current computational trends, and is not meant to be a definitive statement on the quality of each product. In the

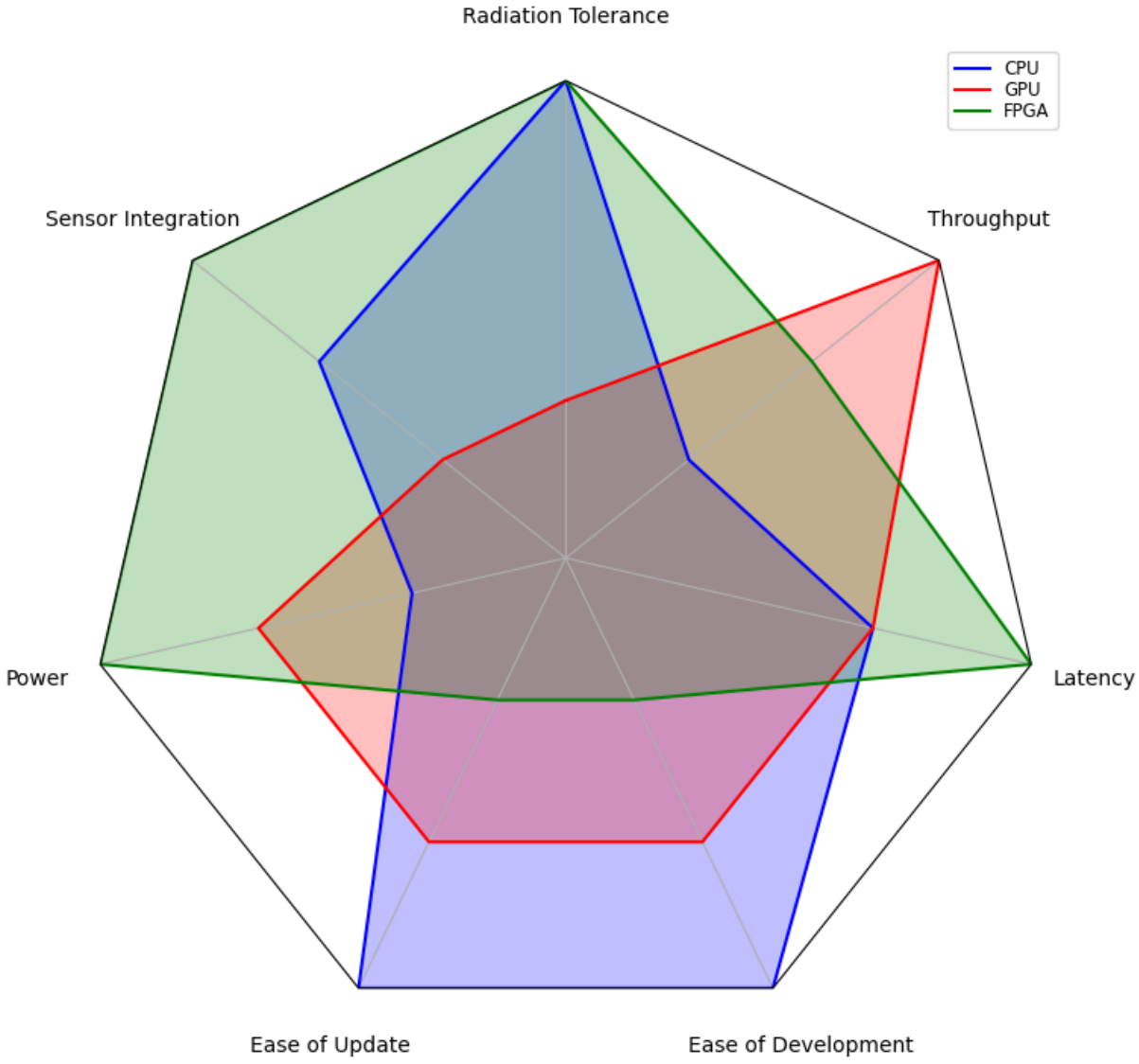

Fig. 7 Notional comparison of different AI/ML processing hardware.

following, we provide a brief discussion of the current state of computational power in space and infer what will be possible in the future based on developments in edge processing.

The current generation of space qualified processors (e.g., BAE RAD 750) lag significantly behind what is possible on the ground (green), in terms of GFLOPS, by several orders of magnitude. The BAE RAD750 is extremely deficient relative to the other devices and being able to run valuable AI/ML applications on the RAD750 is unlikely. The BAE RAD5545 and the Microsemi RTG4 are capable of roughly 10 times the GFLOPS as the RAD750, but, when compared to what is possible at the edge (orange), they are still vastly inferior. The RAD5545 and RTG4 would only be able to perform multiple inferences per second with small models like MobileNet (a neural network designed to perform image classification on mobile devices) and would only be able to perform near real-time inference on data streams with significantly smaller models. While the RAD750, RAD5545, and RTG4 are the least performative processors on the list they also provide the greatest tolerance to radiation, making the three devices more suitable for long term missions.

The Innoflight CFC-400 is a heterogeneous device with a CPU and an FPGA. We provide two data points for the CFC-400, labeled CFC-400 Max and CFC-400*. The former is the theoretical maximum GFLOPS possible assuming maximum utilization of the FPGA on the device, the latter is the GFLOPS achieved for an AI/ML application on a COTS (commercial off the shelf) equivalent processor [56]. We provided the two data points to highlight the difficulty in obtaining peak performance with an FPGA and that realistic peak performance is likely to be somewhere in the range of the two data points. The biggest drawback of the CFC-400 is using the FPGA for acceleration of AI/ML models, particularly for neural networks. Many of the popular tools for developing and training neural networks (e.g., Tensorflow, PyTorch) are designed around deployment on x86 CPUs and NVIDIA GPUs. Compiling and running a neural network on a FPGA, that was initially designed for another processor, can be a difficult process. Tools exist to ease this burden (see Section IV.D) but there are still various pain points in learning these tools along with a lack of full support for all operations found in neural networks. Achievable performance should improve as better tools for optimizing AI/ML models on FPGAs improve. The CFC-400 is capable of significantly greater inference rates than the RAD5545 and should be able to perform real-time inference with well optimized models or operate models at slower rates while additional processing is happening.

Newer space qualified devices, like the Innoflight CFC-500, Ibeos Edge 1U, and the Moog Deep Delphi iX5, will increase radiation tolerant computational power in space by several orders of magnitude relative to the RAD5545, while maintaining similar power needs. All three of these devices combine a CPU and GPU and will be able to achieve inference rates similar to what is possible with a highly optimized CFC-400. The main advantage presented by these devices is the utilization of a GPU instead of a FPGA. As mentioned previously, GPUs and CPUs are significantly easier to deploy AI/ML models to when compared to an FPGA. This ease of deployment is a significant advantage and potentially a worthwhile trade for an increase in power consumption and decrease in GFLOPS. The CFC-500 was released after the CFC-400 and the former has lower theoretical GFLOPS/W than the latter, but in application this may not materialize due to the relative ease with which models can be deployed on GPU. The CFC-500 was initially released with a NVIDIA Tegra K1 SoC GPU; however, that GPU is now obsolete and the future design of CFC-500 is unclear. With devices like the CFC-500 one could implement a two stage change detection algorithm; a smaller model like a MobileNet could be used to scan data streaming from a camera at very high inferences per second and if something of interest is detected the image could then be sent to a larger model for a more thorough analysis. The Ibeos Edge 1U is also of interest as powering two of these devices would only require slightly more energy than the CFC-500 but provide the ability to deploy multiple AI/ML systems simultaneously.

The edge environment is seeing massive gains in computing performance. The NVIDIA Jetson, AMD Ryzen Embedded, Xilinx Versal, and Intel Stratix lineups are able to achieve GFLOPS/W ratios that are better than the NVIDIA 1080Ti (a GPU that several years ago was considered top of the line for graphics processing) with a fraction of the volume. As technology continues to improve and the demand for AI/ML onboard increases, these devices (or similar ones) will find their way into space vehicles, allowing for complex AI/ML systems onboard. The Trident Versal Digital RF Transceiver (VDRT) will utilize a Xilinx Versal AI VC1902 (capable of $\sim$ 11.2 TFLOPS) as its primary processor and is slated to be released during the second quarter of 2022. This will likely make the VDRT one of the most powerful space-grade processors and bring computing to space that is on par with the edge.

We also highlight the Coral System-on-Module (SoM); while not the most performative device for AI/ML processing, it requires the least amount of energy and is able to achieve maximum performance with only 2 watts. The Coral SoM consists of an ARM CPU with an integrated GPU and a Google Edge TPU. Technically the Edge TPU is an ASIC that is only capable of 8-bit deep learning operations using Tensorflow Lite (the device only supports a limited number of common deep learning operations) and the great energy efficiency comes at the cost of this lack of versatility. Since the Coral SoM only performs 8-bit operations, we cannot directly compare the performance of the device to the others in

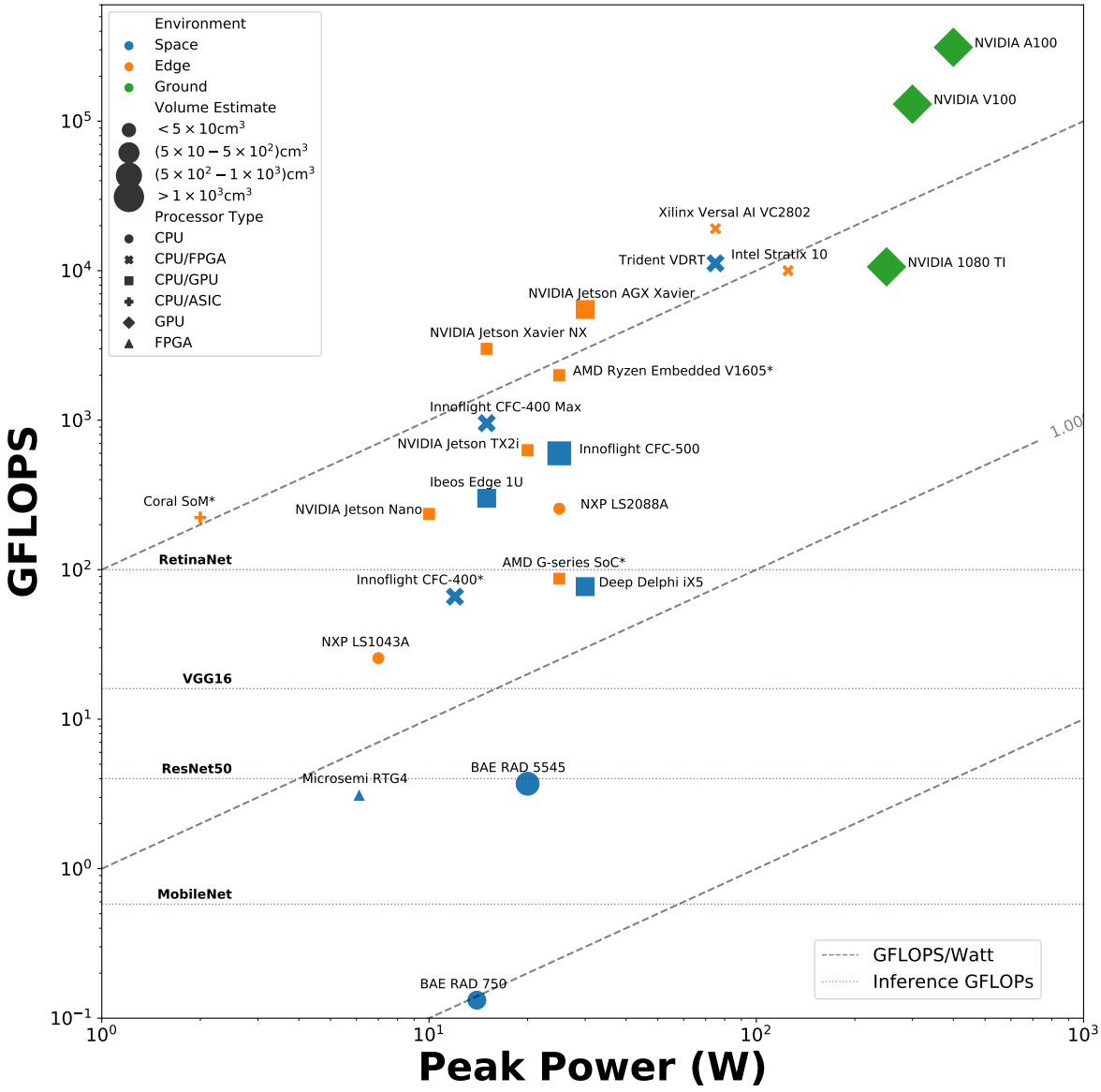

Fig. 8 Plot of GFLOPS against peak power consumption in Watts for various computational devices. Blue, orange, and green represent devices intended for the space, edge, and ground environments respectively. The size of the data points represents the estimated volume of the device. The shape indicates the types of processors on the device. The dashed diagonal gray lines represent contours of constant GFLOPS per Watt and the purple horizontal dotted lines represent the GFLOPs required for a single inference of a standard selection of deep learning models. Asterisks represent values obtained through reported benchmarks/tests, all others were obtained through datasheets. CFC-400 is shown for both a use case and the theoretical limit including FPGA (Max). The Coral SoM ASIC can only perform inference with 8-bit models, for comparison we show the GFLOPS needed to achieve the same latency with a 32-bit MobileNet model. The Nvidia V100 and A100 GFLOPS values are for the TensorFloat-32 format as these are the only devices capable of using this format and it is a more optimal precision for AI/ML. All other values are reported with Float32 precision.

terms of GFLOPS. Instead, we use latency benchmarks for model inference with the Coral SoM [55] and extrapolate the minimum GFLOPS needed to achieve the same latency with a 32-bit MobileNet model. Therefore, using an ASIC architecture and quantizing models to an 8-bit format allows one to achieve performance similar to the Jetson Nano with one fifth of the power. A system composed of several of these devices could conduct inference with many AI/ML models simultaneously with relatively little energy demand. Currently the radiation tolerance of the Coral SoM is unknown, but rad-tolerant ASICs have been produced and ASICs built specifically for accelerating deep learning models could be applied in space in the future. The biggest drawback to ASICs is the lack of reprogramability. CPUs, GPUs, and FPGAs can all be reprogrammed to perform completely different tasks, but ASICs can only do what they were initially designed for—an Edge TPU could never be used for any computation other than deep learning operations. Various models can be compiled on an Edge TPU, but the operations required for the execution of a model on the device must adhere to a short list of common matrix operations. Therefore, an ASIC could never be the only compute available in a payload. Instead, ASICs could be utilized to accelerate deep learning models while a second type of processer is used for other computational needs.

VI. Conclusion

Future space systems will rely increasingly more on AI/ML for autonomy and processing data faster than closed-loop ground processing timelines. AI/ML processing onboard satellites will add new requirements to meet operational, safety, and customer needs. The space environment poses new challenges for AI/ML processing not seen in the ground and edge environments including a more diverse set of heterogeneous, low-SWaP, rad-tolerant compute devices. Developing AI/ML applications for space requires considering the target environment throughout the development workflow and new approaches to implementing MLOps in low-SWaP, distributed, optionally-connected environments. While new advances in edge processing devices deliver significant acceleration capability for deep learning models, their applicability in a space environment is to be seen. In this white paper, we have derived the driving requirements for AI/ML solutions in space; identified the differences between the ground, edge, and space environments; provided a reference architecture for end-to-end development, deployment, and sustainment of AI/ML systems in the space environment; and evaluated the current and next-generation processors for accelerating AI/ML models in space.

Funding Sources

The work presented in this white paper was fully funded by Lockheed Martin Space as part of an internal R&D effort.

Acknowledgments

We would like to thank all the subject matter experts across Lockheed Martin Space that provided valuable insights and feedback that improved the content and presentation of this white paper.

References

[1] US Department of Defense, "SUMMARY OF THE 2018 DEPARTMENT OF DEFENSE ARTIFICIAL INTELLIGENCE STRATEGY," , 2018. URL https://media.defense.gov/2019/Feb/12/2002088963/-1/-1/1/SUMMARY-OF-DOD-AI-STRATEGY.PDF.

[2] Heller, C. H., "The Future Navy - Near-Term Applications of Artificial Intelligence," *Naval War College Review: Vol. 72: No. 4, Article 7*, 2019, pp. 72–99.

[3] CHIPS Magazine, "Transforming the DoD through AI – DoD East Coast AI Symposium & Expo," , 2020. URL https://www.doncio.navy.mil/CHIPS/ArticleDetails.aspx?ID=13343.

[4] NSCAI, "NSCAI Final Report," , 2021. URL https://reports.nscai.gov/final-report/.

[5] Matt Turek, "Explainable Artificial Intelligence (XAI)," , 2018. URL https://www.darpa.mil/program/explainable-artificial-intelligence.

[6] Sheldon Fernandez, "Dark AI and the Promise of Explainability," , 2020. URL https://medium.com/darwinai/dark-ai-and-the-promise-of-explainability-part-i-a6b35009a88c.

[7] Ribeiro, M., Singh, S., and Guestrin, C., ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 2016, pp. 97–101. https://doi.org/10.18653/v1/N16-3020.

[8] Lundberg, S. M., and Lee, S.-I., "A Unified Approach to Interpreting Model Predictions," *Proceedings of the 31st International Conference on Neural Information Processing Systems*, Curran Associates Inc., Red Hook, NY, USA, 2017, p. 4768–4777.

[9] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A., "Learning Deep Features for Discriminative Localization," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2921–2929. https://doi.org/10.1109/CVPR.2016.319.

[10] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D., "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 618–626. https://doi.org/10.1109/ICCV.2017.74.

[11] Qiu Lin, Z., Javad Shafiee, M., Bochkarev, S., St. Jules, M., Wang, X. Y., and Wong, A., "Do Explanations Reflect Decisions? A Machine-centric Strategy to Quantify the Performance of Explainability Algorithms," *arXiv e-prints*, 2019, arXiv:1910.07387.

[12] Langerman, D., Johnson, A., Buettner, K., and George, A. D., "Beyond Floating-Point Ops: CNN Performance Prediction with Critical Datapath Length," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–9. https://doi.org/10.1109/HPEC43674.2020.9286182.

[13] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," *OSDI*, 2018.

[14] O' Neill, J., "An Overview of Neural Network Compression," *arXiv e-prints*, 2020, arXiv:2006.03669.

[15] Han, S., Pool, J., Tran, J., and Dally, W. J., "Learning Both Weights and Connections for Efficient Neural Networks," *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, MIT Press, Cambridge, MA, USA, 2015, p. 1135–1143.

[16] Wong, A., Javad Shafiee, M., Chwyl, B., and Li, F., "FermiNets: Learning generative machines to generate efficient neural networks via generative synthesis," *arXiv e-prints*, 2018, arXiv:1809.05989.

[17] Javad Shafiee, M., Hryniowski, A., Li, F., Qiu Lin, Z., and Wong, A., "State of Compact Architecture Search For Deep Neural Networks," *arXiv e-prints*, 2019, arXiv:1910.06466.

[18] Guo, Y., "A Survey on Methods and Theories of Quantized Neural Networks," *arXiv e-prints*, 2018, arXiv:1808.04752.

[19] Courbariaux, M., Bengio, Y., and David, J.-P., "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations," *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, MIT Press, Cambridge, MA, USA, 2015, p. 3123–3131.

[20] Dettmers, T., "8-Bit Approximations for Parallelism in Deep Learning," *CoRR*, Vol. abs/1511.04561, 2016.

[21] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[22] Nayak, P., Zhang, D., and Chai, S., "Bit Efficient Quantization for Deep Neural Networks," *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, 2019, pp. 52–56. https://doi.org/10.1109/EMC2-NIPS53020.2019.00020.

[23] Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., and Qian, D., "The Deep Learning Compiler: A Comprehensive Survey," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 3, 2021, pp. 708–727. https://doi.org/10.1109/TPDS.2020.3030548.

[24] Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., Kimball, R., Knight, J., Korovaiko, N., Kumar, V., Lao, Y., Lishka, C. R., Menon, J., Myers, J., Aswath Narayana, S., Procter, A., and Webb, T. J., "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning," *arXiv e-prints*, 2018, arXiv:1801.08058.

[25] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A., "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *arXiv e-prints*, 2018, arXiv:1802.04730.

[26] Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., Montgomery, J., Maher, B., Nadathur, S., Olesen, J., Park, J., Rakhov, A., Smelyanskiy, M., and Wang, M., "Glow: Graph Lowering Compiler Techniques for Neural Networks," *arXiv e-prints*, 2018, arXiv:1805.00907.

[27] Goodfellow, I. J., Shlens, J., and Szegedy, C., "Explaining and Harnessing Adversarial Examples," *arXiv e-prints*, 2014, arXiv:1412.6572.

[28] Roffe, S., and George, A. D., "Evaluation of Algorithm-Based Fault Tolerance for Machine Learning and Computer Vision under Neutron Radiation," *2020 IEEE Aerospace Conference*, 2020, pp. 1–9. https://doi.org/10.1109/AERO47225.2020.9172799.

[29] Sadin, S. R., Povinelli, F. P., and Rosen, R., "The NASA technology push towards future space mission systems," *Acta Astronautica*, Vol. 20, 1989, pp. 73–77. https://doi.org/https://doi.org/10.1016/0094-5765(89)90054-4, URL https://www.sciencedirect.com/science/article/pii/0094576589900544.

[30] Goodfellow, I. J., Shlens, J., and Szegedy, C., "Explaining and Harnessing Adversarial Examples," , 2015.

[31] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., and Vladu, A., "Towards Deep Learning Models Resistant to Adversarial Attacks," , 2019.

[32] Luo, T., Cai, T., Zhang, M., Chen, S., and Wang, L., "RANDOM MASK: Towards Robust Convolutional Neural Networks," , 2020.

[33] Cao, Y., Xiao, C., Yang, D., Fang, J., Yang, R., Liu, M., and Li, B., "Adversarial Objects Against LiDAR-Based Autonomous Driving Systems," , 2019.

[34] "MLOps: Continuous delivery and automation pipelines in machine learning," , Nov 2020. URL https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[35] *RAD5545 multi-core system-on-chip power architecture processor*, BAE Systems, 3 2017.

[36] *Compact Flight Computer: CFC-500*, Innoflight, 3 2021.

[37] Belloch, J. A., León, G., Badía, J., Lindoso, A., and Millan, E. S., "Evaluating the computational performance of the Xilinx Ultrascale+ EG Heterogeneous MPSoC," *The Journal of Supercomputing*, Vol. 77, 2020, pp. 2124–2137.

[38] Fuller, Sam, "Industrial Solutions," , 2017. URL https://www.nxp.com/docs/en/supporting-information/DN-Robotics%20and%20LS104x.pdf.

[39] NXP, "Layerscape 2084A and 2044A Multicore Processors," , 2021. URL https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-2084a-and-2044a-multicore-processors:LS2084A.

[40] NVIDIA, "Autonomous Machines," , 2021. URL https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[41] Ryan Smith, "NVIDIA Gives Jetson AGX Xavier a Trim, Announces Nano-Sized Jetson Xavier NX," , 2019. URL https://www.anandtech.com/show/15070/nvidia-gives-jetson-xavier-a-trim-announces-nanosized-jetson-xavier-nx.

[42] Dustin Franklin, "NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics," , 2018. URL https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/.

[43] *DEEP DELPHI iX5 GPU COMPUTE SOLUTION WITH INTELLIGENT DATA PROCESSING AND MACHINE LEARNING CAPABILITY*, Moog, 3 2019.

[44] Coral, "Dev Board Datasheet," , 2020. URL https://coral.ai/static/files/Coral-Dev-Board-datasheet.pdf.

[45] NVIDIA, "NVIDIA V100 TENSOR CORE GPU," , 2020. URL https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[46] NVIDIA, "NVIDIA A100 TENSOR CORE GPU," , 2020. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

[47] Ibeos, "Ibeos EDGE Payload Processor," , 2019. URL https://219275cd-734f-47bd-895a-f7de56d5091e.filesusr.com/ugd/8e84d1_b033a34a2d814e8a98111bfbcee4180a.pdf.

[48] Intel, "Intel Stratix 10 GX/SX Device Overview," , 2020. URL https://219275cd-734f-47bd-895a-f7de56d5091e.filesusr.com/ugd/8e84d1_b033a34a2d814e8a98111bfbcee4180a.pdf.

[49] Bruhn, F., Tsog, N., Kunkel, F., Flordal, O., and Troxel, I., "Enabling radiation tolerant heterogeneous GPU-based onboard data processing in space," *CEAS Space Journal*, 2020, pp. 1–14.

[50] Xilinx, "Versal ACAP AI Core Series Product Selection Guide," , 2021. URL https://www.xilinx.com/support/documentation/selection-guides/versal-ai-core-product-selection-guide.pdf.

[51] Kalomoiris, I., Pitsis, G., Tsagkatakis, G., Ioannou, A., Kozanitis, C., Dollas, A., Tsakalides, P., and Katevenis, G., "An Experimental Analysis of the Opportunities to Use Field Programmable Gate Array Multiprocessors for On-board Satellite Deep Learning Classification of Spectroscopic Observations from Future ESA Space Missions," 2019.

[52] Lovelly, T. M., and George, A., "Comparative Analysis of Present and Future Space-Grade Processors with Device Metrics," *J. Aerosp. Inf. Syst.*, Vol. 14, 2017, pp. 184–197.

[53] Microsemi, "VRTG4 FPGA Technical Brief," , 2020. URL https://www.microsemi.com/document-portal/doc_download/134430-rtg4-fpgas-technical-brief.

[54] Trident, "VERSAL DIGITAL RF TRANSCEIVER," , 2020. URL https://www.tridsys.com/wp-content/uploads/2021/06/Trident-VDRT-Cut-Sheet.pdf.

[55] Coral, "Edge TPU performance benchmarks," , 2020. URL https://coral.ai/docs/edgetpu/benchmarks/.

[56] Kalomoiris, I., Pitsis, G., Tsagkatakis, G., Ioannou, A., Kozanitis, C., Dollas, A., Tsakalides, P., and Katevenis, G., "An Experimental Analysis of the Opportunities to Use Field Programmable Gate Array Multiprocessors for On-board Satellite Deep Learning Classification of Spectroscopic Observations from Future ESA Space Missions," 2019.