Prognostics As-A-Service: A Scalable Cloud Architecture for Prognostics

Jason Watkins[1], Christopher Teubert[2], and John Ossenfort[3]

[1,3] *SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035*
jason.watkins@nasa.gov
john.ossenfort@nasa.gov

[2] *NASA Ames Research Center, Moffett Field, CA 94035*
christopher.a.teubert@nasa.gov

ABSTRACT

Comprehensive aircraft system health-state awareness is critical for maintaining safe, efficient growth in global operations, enabling higher levels of autonomy, and facilitating new forms of aviation. Maintainers, vehicle operators, air traffic controllers, dispatchers, pilots, autonomous systems, and other decision-makers must have reliable real-time knowledge of the vehicle health, the health of its critical composite systems, predictions of how health changes with time, and forecasts of how its capabilities change with health degradation to preserve safety and efficiency. Providing this information in a reliable manner in computationally constrained environments and across a wide range of vehicles and systems continues to be a challenge. This challenge can be partially resolved through cloud computing, where the execution of prognostic and diagnostic algorithms is performed on a network of remote servers hosted on the internet. NASA is developing a cloud computing service, Prognostics As-A-Service (PaaS), that explores the feasibility and challenges of cloud-enhanced prognostics. Though such a system has broad applicability, this research effort is focused on aviation applications.

1. INTRODUCTION

Comprehensive aircraft system health-state awareness is critical for maintaining safe, efficient growth in global operations as well as enabling higher levels of autonomy and new forms of aviation. Maintainers, operators, controllers, dispatchers, pilots, autonomous systems, and other decision makers must have reliable real-time knowledge of the vehicle health, health of its critical composite systems, predictions of how health changes with time, and predictions of how its capabilities change with health degradation in order to preserve safety and efficiency. Providing this information reliably in computationally constrained environments and across the wide range of vehicles and systems continues to be a challenge.

This challenge can be partially resolved through leveraging of cloud computing. Leveraging external resources could enable aircraft with computationally constrained systems to gain improved efficiency and reduced lifecycle costs through resource sharing, and enables the use of new algorithms utilizing the large quantity of data aggregated from many users to provide better services to all. NASA is developing Prognostics As-A-Service, a cloud computing service for diagnostics and prognostics. While cloud computing architectures have many advantages, some challenges will need to be addressed, as described below:

1. Generality: PaaS must be capable of providing services across a wide spectrum of vehicle types and configurations. For prognostics, this requires flexible, configurable, generalized systems models, so as to describe the system of interest. Such generalized models often require extensive system characterization to derive model parameters. The challenges of generalization and parameter identification for generalized models are major technological barriers.
2. Communications: Many aviation systems rely on complex and often limited communications systems. Users of the PaaS system will need to rely on prognostic predictions from PaaS, even in the presence of communication constraints (latency, bandwidth) or dropout from communication failures.
3. Utility: A PaaS architecture must be capable of predicting with the precision, timeliness, and accuracy required for decision makers to take action to protect the safety and efficiency of the aircraft and others. These prediction attributes make up the Quality of Service (QoS) require-

ments for a PaaS. Decision makers could be operating in real-time, such as UAS operators, pilots, autonomous pilots, air traffic controllers, etc., or they could be operating in a strategic manner, such as maintainers.

4. Security: A PaaS architecture requires end-users to send information about the operation of that system over a network. Protecting the Confidentiality, Integrity, and Availability of that information and the prognostic estimates to the degree appropriate is a real challenge.
5. Environmental Complexity: Future load prediction and system degradation prediction can both be a function of the environment it operates in. Inaccurate or incomplete understanding of the environment can lead to imprecise or inaccurate predictions.
6. Trust: Predictions must be trusted in order to be used. This means that they must be both trustworthy and that the end user must be convinced of its trustworthiness.

The Prognostics As-A-Service effort at NASA explores the feasibility and challenges of cloud-enhanced prognostics. Though such a system has wide applicability, this research effort was focused on aviation applications. This effort is exploring and demonstrating the ability to address the six major challenges of a PaaS architecture, described above. This paper details the PaaS architecture and describes its use in NASA projects.

2. SIMILAR ARCHITECTURES

Cloud computing is a topic at this year's IEEE International Conference on Prognostics and Health Management, demonstrating the elevated interest in cloud prognostics architectures. A number of cloud-based prognostics or health management architectures are proposed in the literature (Lee, 2013; Deb, 2013; Ning, Huang, Shen, & Di, 2013). Peng Wang et al in A Computational Framework for Cloud-Based Machine Prognosis (Wang, Gao, Wu, & Terpenny, 2016) describes the value case and a basic formulation of a theoretical cloud-architecture for manufacturing prognostics. Jay Lee et. al. in Methodology and Framework of a Cloud-Based Prognostics and Health Management System for Manufacturing Industry (Lee, 2013) present a detailed architecture for cloud PHM built on the IMS Watchdog PHM Toolbox. Their architecture is designed to be configurable, leveraging configuration files to customize the PHM workflow. These architectures highlight the challenge of Generality for PaaS architectures and illustrate potential solutions.

Researchers at NASAs Diagnostics and Prognostics group created the Prognostics Virtual Lab (Kulkarni et al., 2017), a live-virtual-constructive (LVC) testbed for prognostics research. This work resulted in an extendable, modular framework for performing prognostics remotely using the LVC-DE software and message set (NASA, n.d.; Murphy, Jovic, & Otto, 2015; Murphy & Hoang, 2015). The work identified many of the challenges facing remote architectures for performing prognostics. Researchers struggled with communications, utility, and environmental complexity challenges.

Researchers with NASA Diagnostics and Prognostics group also created the Generic Software Architecture for Prognostics (GSAP) (Teubert, Daigle, Sankararaman, Goebel, & Watkins, 2017). GSAP is an open-source modular, extendable framework for performing prognostics. It was designed to reduce the investment required to create and deploy a prognostics application. The GSAP architecture has been implemented in C++ and open-sourced. Because of its extendibility and generality, GSAP was selected as the foundation of the NASA PaaS Architecture.

There are many Internet of Things (IOT) cloud based services available to the consumer with open source licenses and from well-known publicly traded companies, such as Google, Microsoft, General Electric, Amazon Web Services, Autodesk, Salesforce, and IBM. (*IoT Cloud Platform Landscape*, n.d.) These solutions provide secure, open (any device, operating system, data source, application software, or service), and scalable solutions that can be used to process data from sensors that are on-site at the edge (computation node near the sensor or sensors), or in the cloud. These platforms enable the user to integrate their applications for data processing and analytics.

Of particular relevance to the prognostic service developed at NASA, is the General Electric (GE) Predix Platform™, a distributed application and services platform for the Industrial Internet of Things (IIoT). GE's Predix Platform provides a scalable, public cloud infrastructure for industrial organizations to "rapidly build, securely deploy, and effectively run IIoT applications from edge to cloud." (*Predix Platform*, n.d.). Predix enables edge computing and cloud computing to optimize workload execution. GE's platform offers core platform-as-a-service components, a robust and secure data fabric layer with certificate management, authentication, and governance services, and a defined set of runtime, storage and development services. Services are provided by GE and third-party partners. Predix works in heterogeneous environments that could include GE equipment or equipment from other vendors. Customers can implement their own analytic service, such as prognostic digital twins and prognostic prediction algorithms.

Vitria is a company that sells an Internet of Things (IoT) Analytics Platform-as-a-Service that enables customers to utilize the Vitria analytics engine to monitor, detect, diagnose, analyze, and predict based on IoT data and configurations. The platform allows the customer to prioritize and classify incidents and to suggest automated actions to remediate and resolve anomalies and incidents. Customers can deploy their own algorithms and system models in this service platform.

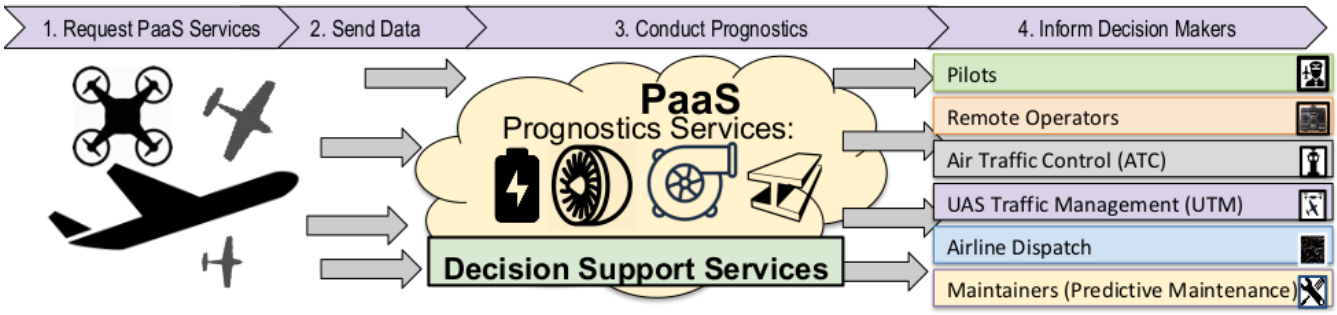

Figure 1. Overview of PaaS operation

3. GENERAL USE CASE

PaaS is designed to fulfil a need for in-flight prognostics beyond what can be achieved onboard. The PaaS service augments any onboard prognostics by applying more powerful hardware to enable more sophisticated prognostic algorithms. This includes both faster multi-core server CPUs as well as GPUs and potentially FPGAs. Abstracting these various hardware components behind a service also simplifies development by minimizing the number of changes to the vehicle configuration.

The service covered by this paper includes facilities to register and manage platforms, systems and components (defined in detail below), configure those entities, send data to the service, and retrieve events from the service. The registration and configuration steps only need to be done once for a given configuration. Once a complete platform is configured, the user can request that a session be started. Starting a session triggers the creation of the configured prognosers on the service backend. Once a session is active, the user can send data and periodically check for results.

The service does not include any kind of client-side user interface. It is our expectation that many end-users will prefer integration of the data provided by the service into their existing user interfaces over the addition of another separate interface to already crowded UAS ground station environments. We are also exploring possible graphical user interface designs as part of the System Wide Safety project.

As part of the service's operation, it naturally stores most data received and calculated in a database. In addition to the basic performance of prognostics, this also enables research by giving easy access to a large quantity of uniform data that can be used to tune and enhance prognostic algorithms.

4. SOFTWARE ARCHITECTURE DESCRIPTION

The Prognostics as a Service architecture builds on the Generic Software Architecture for Prognostics (GSAP) (Teubert et al., 2017) library. PaaS provides the infrastructure to store and manage data and configuration information associated with systems being prognosed, and to efficiently pass data to prognosers and results back to the user.

The PaaS prototype was developed over the last two years as part of the Convergent Aeronautic Solutions (CAS) Project's concept incubation process. The prototype system consists of a Prognostics Application Driver written in C++ that wraps GSAP in a thin communication framework that talks to the front-end process. The front-end process consists of a web server written in Java that exposes a RESTful API to end-users. The application also uses a PostgreSQL database to store prognostics data, configuration information, and application state.

4.1. Entities

PaaS models discrete prognostic problems as a hierarchy of entities that describe a complete set of things to be prognosed.

4.1.1. Platform

Each prognostic session is tied to a single platform. The platform represents a collection of prognostic targets to be analyzed together. The most common manifestation of a platform in our work is an unmanned aerial system (UAS). Each platform has one or more systems associated with it.

4.1.2. System

Each system represents a discrete object that can be analyzed by a single GSAP prognoser. The system is usually one half of the total representation of this object. It stores configuration information about the object that depends on the platform. The other half of the object is represented by the Component entity.

4.1.3. Component

Components represent a pluggable object that is used in a system. As an example, a UAS battery has a system that represents the requirement that the UAS have a battery attached during operation, while the specific battery used in any given flight is represented by a component. The component stores

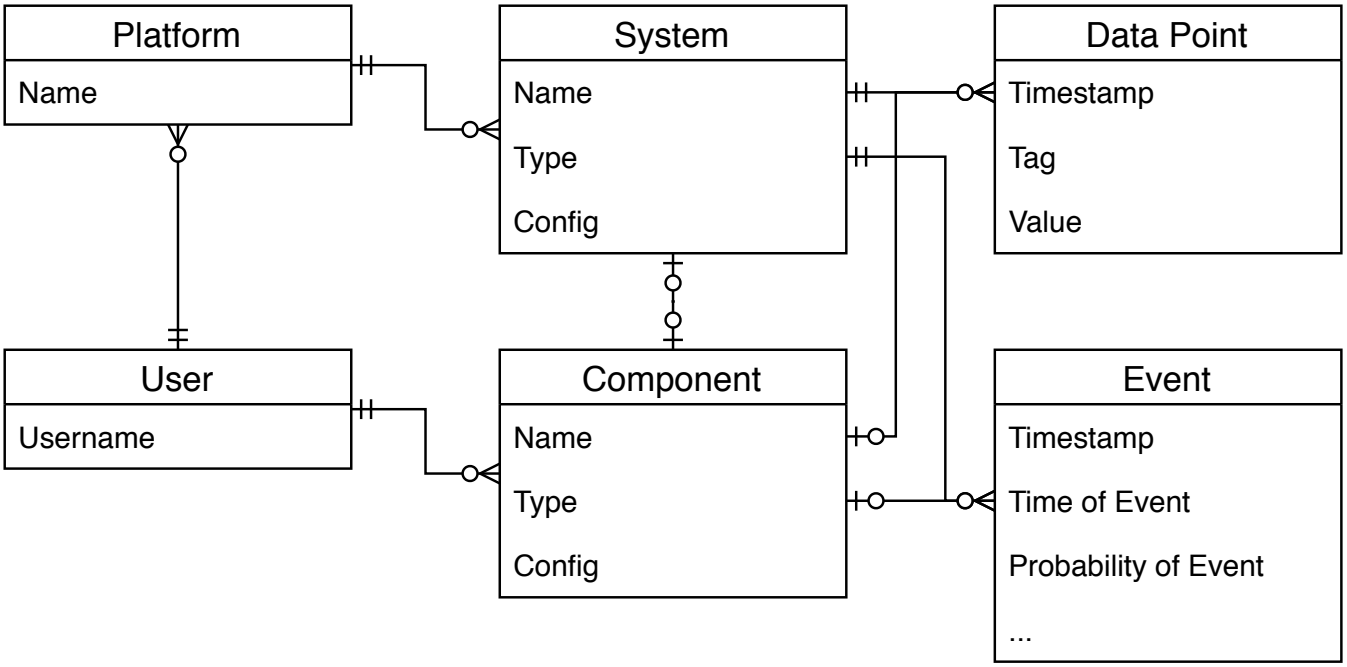

Figure 2. A simplified representation of PaaS entities. The relationship between a user and a component/platform and between a platform and system represent ownership (e.g. a user *owns* a platform). The relation ship between systems and components represents assignment (a component is *assigned to* a system). The relationship between data points/events and systems/components represents an association (e.g. a data point is *associated with* a system) These entities represent the user's conceptual view of the service. They map closely to the database entities used to store data used by the service. The Platform entity maps to a GSAP process, a System/Component pair map to a prognoser and Data Points and Events map to individual messages passed within the service.

configuration information that is specific to a particular physical device.

4.1.4. Data Point

Each data point represents a single input value, such as a battery voltage or vehicle latitude. Data points are published individually to allow prognostic components (primarily observers and predictors) to aggregate only the data they require.

4.1.5. Event

Events represent a piece of information that may be of interest to the user. The primary events of interest are prognostic events that contain the results of a prediction. Additional events may be generated that do not relate to a specific prognostic result. These include things like status events that notify the user of the status of particular parts of the system.

4.2. Service Structure

PaaS is organized into three distinct layers that each interact with the adjacent layer or layers. The lowest layer is the Prognostics Application Driver, which creates and manages the lifetime of GSAP prognosers. The PAD is also responsible for one half of the inter-process communication link that connects the C++ process running GSAP to the Java process running the REST server. The service layer is the core of the Java server. It handles the other half of the inter-process communication and handles all communication with the database. The service layer receives sensor data from the REST API layer and passes that data to the PAD, and also receives prognostic results from the PAD that it stores in the database for retrieval via the REST API. Finally, the REST API layer is a thin wrapper over the service layer that exposes the service architecture to the world as a set of HTTP endpoints to which requests can be made.

Data is passed between parts of the application using a lightweight publish/subscribe system. This system allows data to be published in granular pieces allows prognostic components to aggregate exactly the data they need to perform their calculations. This model also enables a very simple model for asynchronous execution.

4.2.1. Prognostic Application Driver (PAD)

Within the main Java application, the execution layer consists of two sets of components, first a set of repositories that encapsulate database operations. These are primarily standard Spring JPA repositories. Second, the execution layer con-

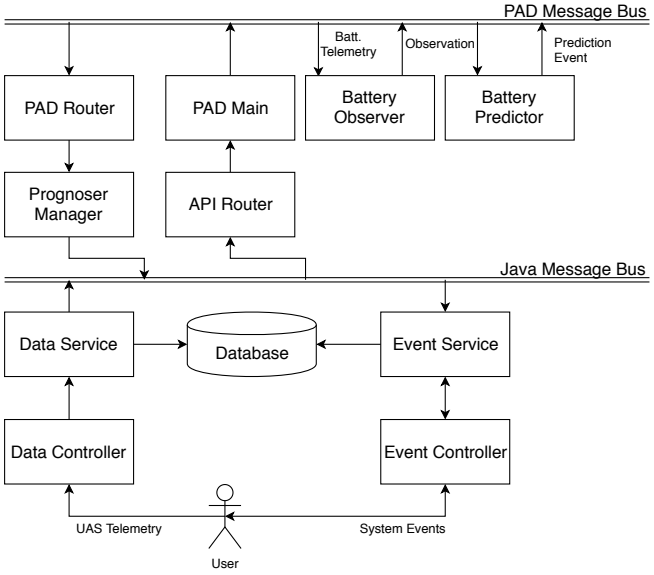

Figure 3. A simplified representation of the PaaS communication bus. A new PAD is created for each prognostic session. After initialization, the PAD's main function enters an infinite loop that reads messages from stdin until an "exit" message is received. During initialization, the PAD creates a Router object that intelligently subscribes to messages that need to be transferred to the front-end.

sists of a simple IPC messaging protocol and infrastructure for maintaining prognostics processes associated with each active session.

The current version of the GSAP framework implements a lightweight message bus architecture to enable the efficient routing of data between various parts of the application. This message bus architecture is also replicated in the service layer described below, and messages of interest are determined and serialized between the two busses as necessary.

On the other end of the IPC pipe, a C++ application provides a thin wrapper around the GSAP library that processes incoming messages, configures prognosers based on those messages and places incoming data on a message bus. It also monitors the message bus for prognostic events and passes those events back to the Java server application.

4.2.2. Service Layer

The service layer contains components that execute the main "business logic" of the system. This includes input validation, storage and retrieval of data from the database. The service layer also routes session data needed by the prognostics component to the prognostics component in the execution layer.

4.2.3. REST API Layer

Representational State Transfer (REST) is a web architecture that allows for both querying and updating of resources us-

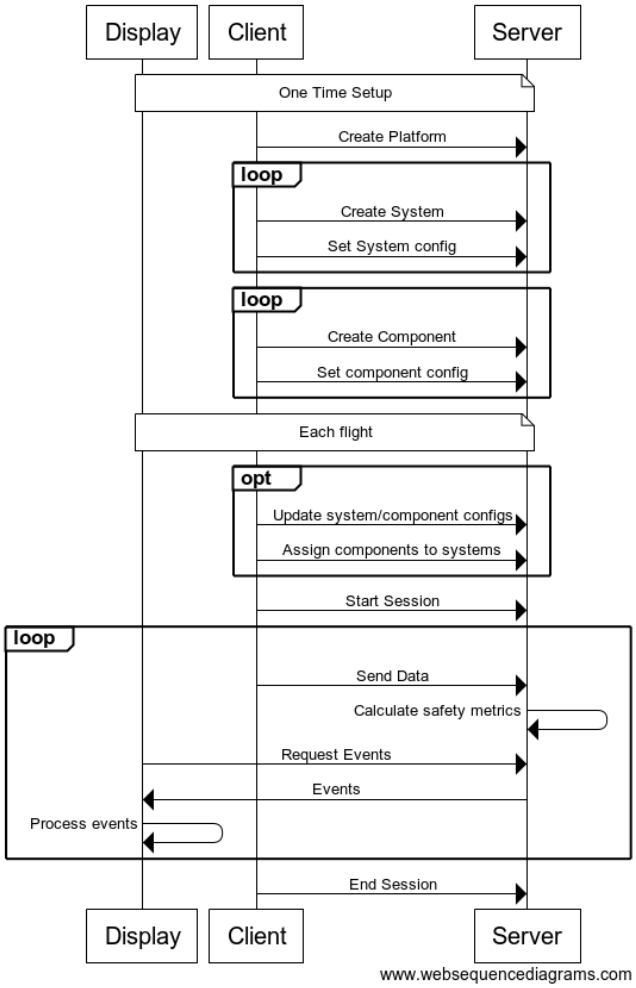

Figure 4. A high level overview of the sequence of requests used to set up and perform prognostics.

ing structured HTTP requests. The REST style provides a uniform stateless architecture that fits well into today's web-centric world. The API is not without limitation however. In particular, the REST format (and HTTP in general) do not provide any mechanism for push-based notifications. Due to this limitation, the current API requires that clients poll periodically to receive new events.

4.2.4. Data Flow

A typical user's workflow breaks down into three categories of operations. The user must perform certain one-time operations to create a workable set of components, must perform certain operations "pre-flight" to ensure that the pieces are configured correctly, and finally the user performs "in-flight" operations of sending data and receiving results.

When a user first registers, they must set up platforms, systems and components to tell the service about the things they wish to perform prognostics on. At a minimum, the user must

create a single platform (if they only wish to do vehicle-level prognostics), but most users will also create subsystems (e.g. batteries) for each platform. Finally the user creates components, which represent the specific serial-numbered part that can be assigned to a system. These operations can be performed at any time, but only need to be performed once. Once a platform exists in the system, it is stored in the database forever.

Once initial configuration is complete, the user can start a session at any time. To start a session, the user must make sure that all systems have a component assigned and then make a request to start a session. Once the session is started, the user can send sensor data to the service, and receive events back either by polling an endpoint in the RESTful interface or via direct pushes in the MAVLINK interface. Once the session is concluded, the user sends an end session request to the service to indicate that the service can release resources associated with the session.

5. CASE STUDIES

5.1. CASAS

The CASAS project was a NASA effort to demonstrate the integration of multiple intelligent systems technologies developed at NASA. As part of the project, PaaS was used as an operational battery health-monitoring tool. A Python client was developed to interface with the CASAS ground station software to collect battery voltage, current and temperature data in real time and publish that data to PaaS via an HTTP POST every second. The client simultaneously made an HTTP GET request each second to retrieve prognostic results from the server.

PaaS was integrated as one source of decision making in the project. To accomplish this, one of the UAS platforms flown by the project was set up in PaaS by creating a Platform representing the UAS and a System representing the battery powering the UAS's motors. A Component was also created for each of the flight batteries used by the project. During each test flight, the battery for the flight was assigned to the battery system and the vehicle ground station started sending battery sensor data to PaaS as soon as the vehicle was powered on.

The project successfully demonstrated the application of cloud-based battery prognostics, but execution was not without issue. We encountered significant safety challenges in deploying the service. The primary issue we encountered was a security concern by NASA's flight safety review board with connecting the UAS ground station to the network. To work around this, we had to slave a second computer to the active ground station and connect that machine to the network so that it could stream data to PaaS.

5.2. System Wide Safety (SWS)

The goal of the System Wide Safety (SWS) project is to provide system-wide, model-based predictive capabilities in the UAS domain to ensure overall system safety. This would potentially include areas such as weather, navigation or communication performance, population density models, vehicle system health, and more. Much of the work performed in the initial year of the project was a reimplementation of existing technologies that had been developed for the Real-Time System Monitoring (RTSM) project in order to provide similar assurances for commercial aircraft in the National airspace. Because the initial RTSM implementation was developed as a demonstration of the technology, it did not provide a scalable or accessible solution for multiple aircraft and did not provide the functionality needed to collect aircraft telemetry from multiple data sources. The PaaS architecture was adopted in order to solve both of these shortcomings, and also provide the general framework needed to port over battery prognostics that were planned as one of the initial safety metrics for SWS.

An important ongoing deliverable of the SWS project is in supporting flight tests to investigate and demonstrate advancement of its goals under the technical challenge for In-Time Safety Nets for Emerging Operations. Specifically, these tests would demonstrate automated in-time risk identification and mitigation for small UAVs over the duration of the flight, and to that end the SWS project has been working towards integration of its technologies into the UAS Traffic Management (UTM) ecosystem as a Supplemental Data Service Provider (SDSP). The intent of the SDSP is to respond to queries from UAS operators in order to increase system awareness and overall safety in the airspace. In its current implementation, the SWS SDSP has focused on battery health monitoring and obstacle proximity information as the initial safety metrics being calculated. Using the RESTful web service provided by PaaS, the SWS project participated in UTM "Sprint 3" and "Sprint 4" integration activities using simulated flight data. Telemetry items such as latitude, longitude, altitude, speed, battery voltage, current and temperature were sent to the SDSP via the PaaS RESTful interface, and obstacle awareness (nearby building and trees) and battery health monitoring data were returned.

A second demonstration effort was also performed in concert with standalone flights at Langley Research Center in Virginia. This earlier, initial implementation specified MAVlink protocol messages to pack and unpack the required data being sent between the UAS ground station and SWS SDSP. Rather than utilizing the RESTful interface, this data was sent via a TCP port and relied on intermediate front-end Java code to translate between the MAVlink data messages and the internal PaaS API for storing data and performing stateful session management.

Integrating the battery monitoring and obstacle awareness metrics into PaaS each had different challenges. The battery health monitoring effort had already begun, and served as the primary metric to prove out the PaaS/GSAP model. This made it a natural fit for the SWS SDSP, and in general there were few issues with the integration. Although this integration went smoothly overall, we did find that the API documentation did not sufficiently specify the expected units for various telemetry items. As a result, we encountered several instances where the client would initially send the right data but in the wrong units (e.g. 100mV rather than 0.1V). Another shortcoming in the use of PaaS is a relatively low ceiling for high-rate data. For all testing so far, the SWS project has specified a rate of 1Hz for all incoming telemetry. Higher data rates are desirable for some prognostic applications, but the nature of HTTP, especially over a secure connection may make this impractical.

The integration of obstacle awareness using PaaS proved to be slightly more difficult, or at least unwieldy in its implementation. Part of the reason for this is the PaaS treatment of "components" as the underlying basis for its system or platform health prediction. In viewing the system as a collection of components, it does not easily support those data inputs that describe the behavior of the system as a whole, or the relationship of other nearby systems or objects. While these deficiencies did not prevent PaaS from being a useful tool it sometimes added additional overhead. Having a more comprehensive schema or development of optional domain-specific libraries would be useful in solving this. Presently the GSAP component of PaaS is not being used for obstacle awareness.

Despite any shortcomings, leveraging the health management framework of PaaS proved to be extremely useful in providing the generalized prognostic tools as well as the front-end API and back-end database. Adoption of PaaS greatly simplified the process of getting the SWS server running and collecting user data from the customer in a short timeframe. It is hoped that the SWS project will be able to attract more customers and allow us to test the PaaS framework over hundreds of simultaneous users, potentially tracking thousands of objects. Scaling up to this size in a robust and reliable way would be the gold standard for proving PaaS in the field.

6. CONCLUSIONS/FUTURE WORK

Prognostics can inform decisions to protect the safety of a vehicle and reduce lifetime costs through predictive maintenance. The PaaS architecture has the potential to bring precision prognostics capabilities to vehicles that otherwise would not have access to such information or supplement any onboard capabilities.

The work on PaaS so far has partially addressed the 6 challenges outlined in the introduction:

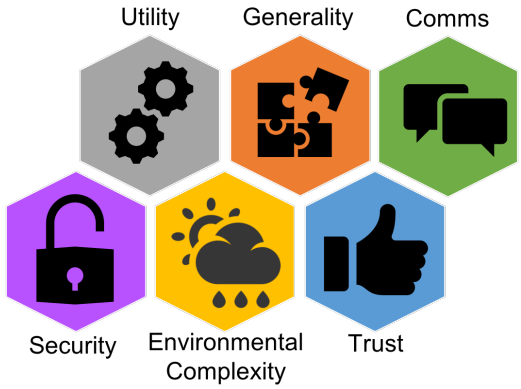

Figure 5. The 6 key challenges addressed by PaaS

1. Generality: PaaS is built on the GSAP library, and therefore inherits its generic prognostics architecture, including configurable and generic prognostics models. Additionally, the interface has been carefully designed to accept a wide range of data, and to produce its output in a general and extensible format.
2. Communications: The PaaS architecture is built on RESTful principles, which are in turn built on the HTTP protocol. The stateless nature of HTTP requests gives the service an inherent resilience in the face of temporary communication loss. This is balanced against a certain amount of overhead involved in using JSON (a text-based format) for all data. In the future, we may need to develop binary data formats to maximize data throughput.
3. Utility: The PaaS architecture has been tested in two use cases described above. The architecture has and continues to evolve in response to those tests.
4. Security: Development of a security infrastructure is largely unmentioned in this paper, however much of the security will come from adherence to software industry standards. REST API's of the kind described here are increasingly common in the software world and there is a wide set of best practices to draw from. The service is available only over TLS, which provides encryption of data in flight, and we are developing authentication and authorization controls that adhere to both industry and NASA standards.
5. Environmental Complexity: The SWS project is actively working on quantifying and refining our environmental models, including advanced trajectory generation (Corbetta, Banerjee, Okolo, Gorospe, & Luchinsky, 2019) and battery power estimation.
6. Trust: The PaaS architecture is not yet fully mature. Initial tests in the case studies above show promise, but the

architecture is not yet ready for deployment in safety-critical applications.

Feedback from the CASAS and SWS projects show a potential for improvement in usability when addressing domains outside of component health management. Using more generalized system models has its advantages in being applicable to many domains, but also increases the burden of translating inputs and outputs to something useful. Perhaps allowing an end user to map a domain-specific ontology onto the generic PaaS interface would allow for a simplification of integration and promote further use, possibly in novel ways.

Further feedback from the CASAS project suggests that connecting vehicles directly to internet services is likely to receive pushback from those concerned with aircraft safety. Further investigation into how to demonstrate safe operation of a highly-connected UAS is necessary for PaaS to fully succeed.

The SWS project will continue to mature their safety metrics and safety assessment algorithms in the PaaS platform. The PaaS architecture will be matured to better support these advancements. Additionally, the SWS project will test the PaaS architecture with their algorithms in flight tests, providing an additional opportunity to reveal and fix weaknesses in the PaaS architecture. Additionally, the authors are planning a study to better understand the stakeholder QoS requirements and to understand the performance of the prototype PaaS system. These will be used to identify the performance gaps, areas where performance does not meet requirements, for future work.

ACKNOWLEDGMENT

The authors thank the NASA Airspace Operations and Safety Program (AOSP) System Wide Safety (SWS) project and the NASA Transformative Aeronautics Concepts Program (TACP) Convergent Aeronautic Solutions (CAS) project for supporting this research.

REFERENCES

Corbetta, M., Banerjee, P., Okolo, W., Gorospe, G., & Luchinsky, D. G. (2019). Real-time uav trajectory prediction for safety monitoring in low-altitude airspace. In *Aiaa aviation 2019 forum* (p. 3514).

Deb, B. e. a. (2013). Towards systems level prognostics in the cloud. *IEEE Conference on Prognostics and Health Management (PHM)*, *1*(6).

Iot cloud platform landscape. (n.d.). https://www.postscapes.com/internet-of-things-platforms/. (Accessed: 2019-04-30)

Kulkarni, C., Gorospe, G., Teubert, C., Quach, C. C., Hogge, E., & Darafsheh, K. (2017). A virtual laboratory for aviation and airspace prognostics research. *AIAA*.

Lee, J. e. a. (2013). Methodology and framework of a cloud-based prognostics and health management system for manufacturing industry.

Murphy, J. R., & Hoang, T. (2015). *Uas integration in the nas project: Integrated test and lvc infrastructure* (Tech. Rep.). NASA.

Murphy, J. R., Jovic, S., & Otto, N. M. (2015). Message latency characterization of a distributed live, virtual, constructive simulation environment. *AIAA Infortech Aerospace Conference*.

NASA. (n.d.). *Live virtual constructive distributed environment (lvc) lvc gateway, gateway toolbox* (Tech. Rep.). Author.

Ning, D., Huang, J., Shen, J., & Di, D. (2013). A cloud based framework of prognostics and health management for manufacturing industry. *IEEE International Conference on Prognostics and Health Management*.

Predix platform. (n.d.). https://www.ge.com/digital/iiot-platform. (Accessed: 2019-04-29)

Teubert, C., Daigle, M., Sankararaman, S., Goebel, K., & Watkins, J. (2017). A generic software architecture for prognostics. *International Journal of Prognostics and Health Management*, *8*(13).

Wang, P., Gao, R. X., Wu, D., & Terpenny, J. (2016). A computational framework for cloud-based machine prognosis. *Procedia CIRP*, *57*, 309-314.

BIOGRAPHIES

Jason Watkins is a software engineer with Stinger Ghafarrian Technologies (SGT) working for the Diagnostics and Prognostics group at NASA Ames Research center. He currently works on the System-Wide Safety project at NASA Ames. Jason received his B.S. in Computer Science from University of California, Irvine in 2018. Prior to graduating and joining SGT full time, Jason completed several undergraduate internships with NASA and SGT. As part of those internships, Jason has worked on several NASA software projects, including the open source projects X-Plane Connect and Generic Software Architecture for Prognostics (GSAP).

Christopher Teubert is a subproject technical lead for Autonomous Systems and group lead of the Diagnostics and Prognostics group at NASA Ames Research Center. He is also the principal investigator for the Generic Software Architecture for Prognostics (GSAP). Christopher received his B.S. in Aerospace Engineering from Iowa State University in 2012 and is currently working on his M.S. in Computer Science and Engineering at Santa Clara University. Chris worked as a research engineer with Stinger Ghafarrian Technologies (SGT) at NASA Ames Research Center from 2012-2016. Since 2016, Chris has been a computer engineer and lead with the Diagnostics and Prognostics group.

In 2018 Chris took on the role of subproject technical lead for autonomous systems.

John Ossenfort is a Computer Scientist and employee of SGT/KBRWyle at NASA Ames Research Center. He is currently writing software as part of the System-Wide Safety (SWS) project, under the Airspace Operations and Safety Program (AOSP). He also works in the Discovery and Systems Health research area, integrating fault management technologies with advanced testing and demonstration for the Orion Multi-purpose Crew Vehicle. In the past he has worked on several exploration projects, including the Ares I-X flight test, Exploration Flight Test 1 (EFT-1) and Deep Space Habitat (DSH). John has a dual BA degree in Anthropology and East Asian Studies from Washington University in St. Louis.