AFRL-RH-WP-TR-2020-0066

COG-PACK™ Technical Design Document

2Lt Matt Rommel
Mr. Allen W. Dukes
Airman Biosciences Division

Mr. Scott J. Duberstein
Mr. Courtney J. Downs
Mr. Ethan Blackford
Ball Aerospace & Technologies

March 2020
Interim Report

AIR FORCE RESEARCH LABORATORY
711TH HUMAN PERFORMANCE WING,
AIRMAN SYSTEMS DIRECTORATE,
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by 88th Air Base Wing Public Affairs Office and is available to the general public including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RH-WP-TR-2020-0066 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

SOTO.ARMANDO.1262578328 Digitally signed by SOTO.ARMANDO.1262578328 Date: 2020.08.20 12:09:47 -04'00'

ARMANDO SOTO
Work Unit Manager
Performance Optimization Branch

MCKINLEY.RICHARD.A.1262239710 Digitally signed by MCKINLEY.RICHARD.A.1262239710 Date: 2020.08.24 10:26:45 -04'00'

RICHARD A. MCKINLEY, DR-III
Core Research Area Lead,
Prformance Optimization Branch
Airman Biosciences Division

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
30-07-2020	Interim	Jan 2019 to March 2020

4. TITLE AND SUBTITLE
COG-PACK™ Technical Design Document

5a. CONTRACT NUMBER
FA8650-16-C-6610

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62202F

6. AUTHOR(S)
2Lt Matt Rommel*
Mr. Allen W. Dukes*
Mr. Scott J. Duberstein+
Mr. Courtney J. Downs+
Mr. Ethan B. Blackford+

5d. PROJECT NUMBER
5329

5e. TASK NUMBER
08

5f. WORK UNIT NUMBER
H0KG (53290819)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ball Aerospace & Technologies+
2675 Presidential Drive, Fairborn, OH 45324

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Materiel Command*
Air Force Research Laboratory
711 Human Performance Wing
Airman Systems Directorate
Airman Biosciences Division
Performance Optimization Branch
Cognitive Neuroscience Section
Wright-Patterson AFB, OH 45433

Air Force Materiel Command**
Air Force Research Laboratory
711 Human Performance Wing
USAF School of Aerospace Medicine
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
711HPW/RHBC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RH-WP-TR-2020-0066

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION A. Approved for public release.

13. SUPPLEMENTARY NOTES
88ABW-2020-2622; cleared 19 August 2020

14. ABSTRACT Many of the most popular hardware and software products for measuring and improving human performance are developed and marketed for standalone use. For research and development or operational use, more effective insights into human performance may be obtained by leveraging diverse measurements from multiple sensors. Often the devices lack the ability to interoperate and the extensibility to do varies across devices. As a result, developers supporting human performance research and interested in utilizing a promising new device must generate a new set of requirements and sensor-specific code to integrate the device before researchers can begin the collection of physiological and performance-based data. In the current human performance assessment space there are few coherent software architecture or data formats that can be shared effectively among research groups, leaving groups to sit on an island unable to communicate effectively or integrate established and emerging technologies.

Cognitive Operations Gear (COG) Pack™ serves as a domain-flexible software architecture designed to collect and analyze human physiological and behavioral data with the end goal of providing operator state assessment. COG Pack currently supports a wide selection of hardware devices. Smart Eye Pro for eye tracking and wearables like Zephyr BioHarness for cardiorespiratory and activity measurement, to name a few. The underlying software architecture offers unique insights into (I) developing (Dukes, et al., 2019), (II) integrating, and (III) maintaining new hardware and software.

15. SUBJECT TERMS
COG Pack™, Technical Design Document, Software Architecture, System Design, Backend Services

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Armando Soto
Unclassified	Unclassified	Unclassified	SAR	27	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

COG PACK™ TECHNICAL DESIGN DOCUMENT

Descriptions and Definitions of Techniques and Solutions
Implemented in COG Pack

Scott J. Duberstein, Allen W. Dukes, Ethan B. Blackford, Courtney J. Downs, Matt T. Rommel

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

Many of the most popular hardware and software products for measuring and improving human performance are developed and marketed for standalone use. For research and development or operational use, more effective insights into human performance may be obtained by leveraging diverse measurements from multiple sensors. Often the devices lack the ability to interoperate and the extensibility to do varies across devices. As a result, developers supporting human performance research and interested in utilizing a promising new device must generate a new set of requirements and sensor-specific code to integrate the device before researchers can begin the collection of physiological and performance-based data. In the current human performance assessment space there are few coherent software architecture or data formats that can be shared effectively among research groups, leaving groups to sit on an island unable to communicate effectively or integrate established and emerging technologies.

Cognitive Operations Gear (COG) Pack™ serves as a domain-flexible software architecture designed to collect and analyze human physiological and behavioral data with the end goal of providing operator state assessment. COG Pack currently supports a wide selection of hardware devices. Smart Eye Pro for eye tracking and wearables like Zephyr BioHarness for cardiorespiratory and activity measurement, to name a few. The underlying software architecture offers unique insights into (I) developing (Dukes, et al., 2019), (II) integrating, and (III) maintaining new hardware and software.

1.0 INTRODUCTION

There are three targeted roles for those who interact with COG Pack: participant, researcher, and developer. A ***participant*** serves as the source of data in an experimental data collection; a ***researcher*** manages the collection of data during an experiment; and a ***developer*** tasked with integrating sensors or algorithms before the data collection process starts. Through this common vernacular of roles, COG Pack provides scope as to how to implement features for all three roles.

Additionally, through the implementation of Data-Flow Diagrams (DFD's), figures, and Sequence Diagrams, readers should understand the high and low-level structures that make up COG Pack as a software system.

Implemented primarily as an Object-Oriented C# based framework, COG Pack provides a reference architectural description of Sensors producing Signals. The creation and coordination of these elements are managed by the ***Kernel***. To start, this paper will formalize the abstract parts of C# code. (As a supplement, it is recommended that readers familiarize themselves with the white paper (Dukes, et al., 2019) and videos (Works Cited) to further understand how the system functions.)

2.0 ADDITIONAL BACKGROUND INFORMATION

Throughout the scenarios outlined in this paper, all API properties are described in object-oriented programming syntax. For example, "Signal**.**Name" (Signal-Dot-Name) is the colloquial phrase in software development to annotate, "A Signal's Name Property". These approaches derive from software engineering best practices with a focus on event-driven programming. (Rochlin, 2013). Furthermore, a Software Diagram Key outlines the contents of software figure diagrams, and the Selected API Attributes are intended to help readers familiarize themselves with the class properties described throughout the paper. The descriptions are designed to be thorough such that anyone can understand the scope of the processes at a high level.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Overall Architecture

You can control the camera by clicking and dragging on the screen, or by selecting options in the Camera Folder. (Figure 2.)

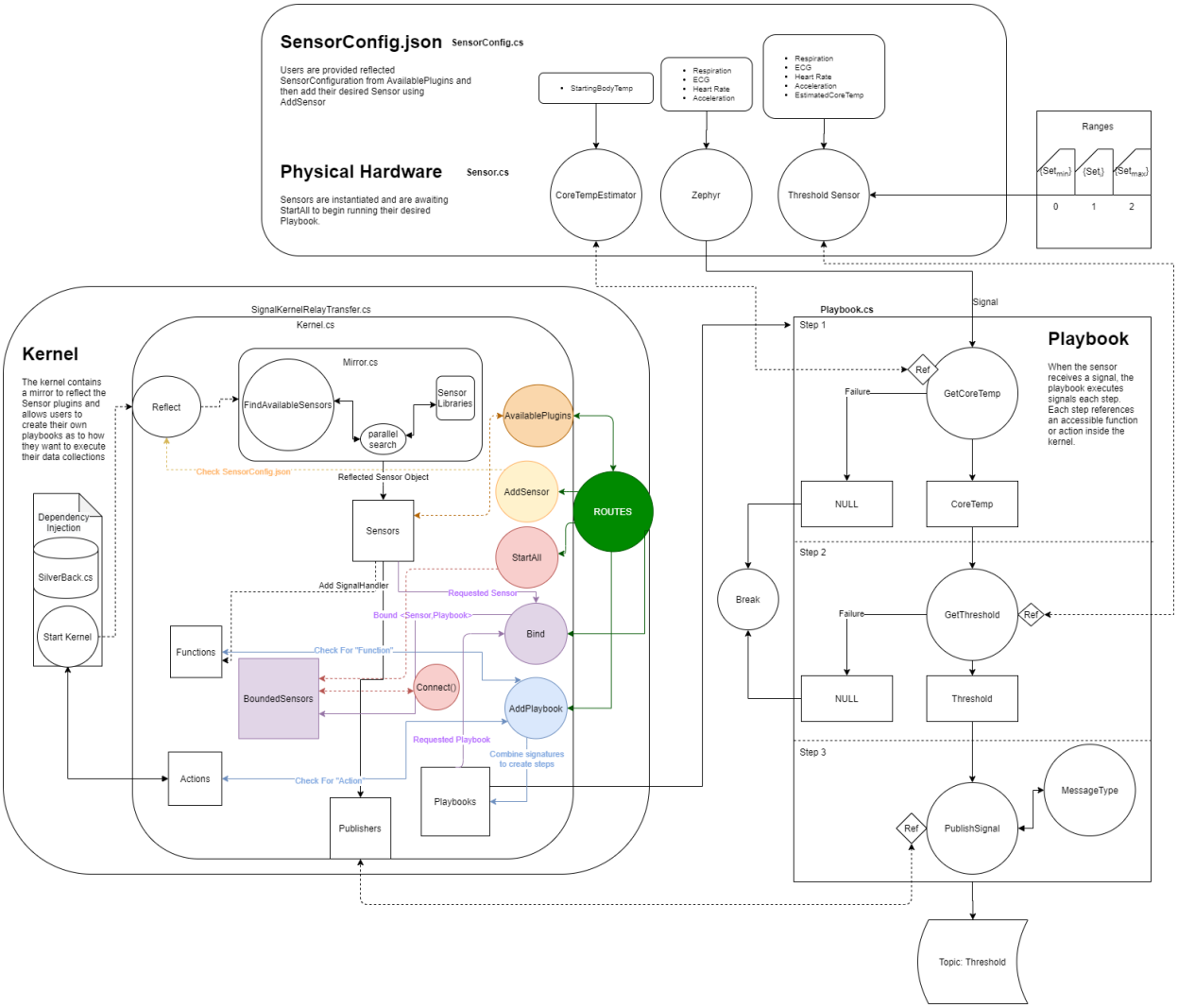


Figure 1. Overall Architecture DFD

The ***Kernel*** manages the multiple Sensor configurations and operations inside the framework. Figure 1 represents every ***Kernel*** interaction with each Sensor found inside the COG Pack architecture.

First, the assignment of configurations and operations inside the ***Kernel*** follow the left most box labeled ***Kernel*** which defines its classes in **Kernel.cs** and **Mirror.cs**. Internally, a web server in **SignalKernelRelayTransfer.cs** enables interaction with the ***Kernel*** and is colloquially

abbreviated as ***SKRT***. Lastly, the lower right DFD describes how the ***Kernel*** assigns requested operations through the use of ***Playbooks*** implemented in **Playbook.cs**.

Once ***SKRT*** starts the ***Kernel***, the ***Kernel*** uses its ***Mirror*** class (**Mirror.cs**) to search the contents of every Dynamic-Link Library (DLL) and Executable (EXE) for classes that inherit from the **COG_Pack.API.Sensor** class. The ***Kernel*** then uses this ***Mirror*** to 'reflect' available **SensorConfiguration** JSON examples for the *AvailalbePlugins* REST call (Fielding, 2000).

Users or other software systems can query the *AvailablePlugins* functionality to obtain a list of all **SensorConfiguration** JSON examples. Given these examples, the users or other software systems can alter the values, and provide that configuration by sending an *AddSensor* REST call. This first creates an instance of the Sensor object inside the **Kernel;** then adds the Sensor's ***SignalHandler*** dictionary to the **Kernel's** Function list to later be executed via a **Playbook**. Finally, **Actions** are added to the **Kernel** from **SKRT** to support additional dependency injection functionality accessible via **Playbooks**.

Given that **Functions** and **Actions** are now accessible and available in the **Kernel,** and all Sensor instances are instantiated, a **Playbook** can be created for each Sensor through the use of the *AddPlaybook* REST call by providing the appropriate JSON describing the requested **Playbook** functionality. Once a valid **Playbook** is created, the **Playbook** is bound to a Sensor or group of Sensors by making the *Bind* REST call. After the Sensors and **Playbooks** are bound, the dictionary association of bounded Sensors manages the functionality between each Sensor's ***OnSignalReceived*** EventHandler to the **Playbook**. Each individual Sensor object can only ever have one **Playbook** association.

To begin collecting Signals from the Sensors, initiate each Sensor's ***Connect*** routine by executing the *StartAll* REST call. If a Sensor is not bound with a **Playbook**, the Sensor's ***Connect*** routine will not execute.

When a Signal is detected, each Sensor will raise an ***OnSignalReceived*** Event and move the Signal as per the instructions designated in the **Playbook**. If the **Steps** in the **Playbook** fail to process the Signal, the **Playbook** will "break" as to prevent further code execution and not waste resources. Since the **Action** in Step 3 of this example **Playbook** is "PublishSignal", the ZMQ instance created in the **Kernel** calls its internal ZMQ dictionary to publish the Threshold signals. Optionally, Signals sent to **SilverBack** (**SilverBack.cs**) are logged using the MongoDB database driver for COG Pack.

(For further in-depth detail describing the contents of Sensor configuration reference either sample **SensorConfig.json** files provided, the API Documentation (Dukes, Duberstein, & Rommel, 2020) for the **Sensor.cs** and **SensorConfig.cs** classes, or the PostMan collection included in the git repo.)

3.2 Process Specification

3.2.1. Scenarios

Three unique scenarios describe the event-driven behavior of COG Pack. Each of these scenarios will provide readers the understanding necessary for 1), publishing Signal data, 2) raising threshold assessment signals, and 3) the execution of actions taken by COG Pack to provide users with data.

1) **Title:** *Publishing Signals*
   Actors: Developer, Sensor
   Description: Developer captures signal data inside an Event, ZMQ, and/or REST
2) **Title:** *Applying Thresholds to Signals*
   Actors: ThresholdCalulator, Sensor, **Kernel**
   Description: How a Threshold Sensor provides assessment
3) **Title**: *Executing a **Playbook***
   Actors: Playbook, **Kernel**
   Description: Executing instructions inside the **Playbook**

3.2.2. Scenario 1: Publishing Signals

Background

When a Sensor implements the ***OnSignalReceived*** EventHandler to raise a Signal, it is imperative for the system to both publish and save the result in real-time. When the Sensor collects data it first converts the data to a Signal object, then raises the Signal to be published. The act of publishing allows for other processes to perform additional operations on a Signal. After the Signal is created, it can be published in three ways. Figure 2 illustrates the velocity at which data is published from Event to ZMQ to REST.

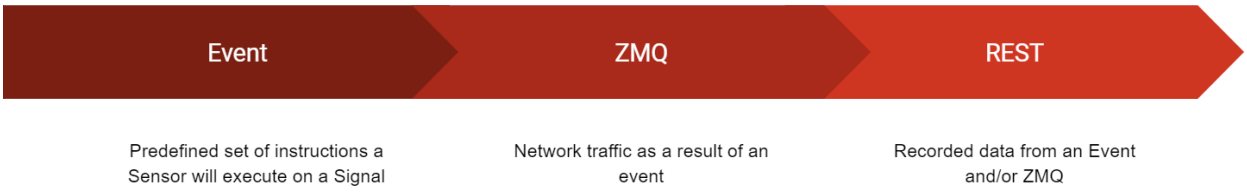


Figure 2. Publish End-Points

Implementation

Working in the event space provides the fastest execution, ideally for developers who have a custom project where they would like to directly leverage Sensors from COG Pack. Developers implement the appropriate execution paths by stitching together ***OnSignalReceived*** EventHandlers in a **Kernel** using **Playbooks** that describe the sequence of Signal processing.

ZeroMQ (ZMQ) is best leveraged when developers want to create a distributed system on a network of computers consisting of multiple Sensors, Signals, and Participants. ZMQ is a messaging system that enables mainstream network design patterns to be implemented (Hintjens, n.d.). A commonly implemented design pattern Publisher-Subscriber (Pub-Sub) is leveraged by the COG Pack **Kernel**. Publishers produce data to be received by subscribers in the form of "topics". Meaning a producer process can have multiple publishers with topics and a consumer process can receive a particular data stream via consuming the desired "token" topic. The **Kernel** describes the ZMQ publisher topics in two different ways. To help breakaway from nomenclature fatigue to describe data streams in code, COG Pack foregoes dots in objects to describe ZMQ topics and implements hyphen based delineation. These topics are set prior to data collection and execution:

1) SensorName-MessageType (SensorName Dash MessageType)
2) SignalName

1) "SensorName-MessageType" enables developers to access all Signals from a given specific Sensor. For example, "Zephyr-json" describes the set of all signals related to Zephyr and communicated as JSON.

Because Sensor Configuration is the first step in initialization, MessageType is declared inside the Sensor's configuration. Currently, COG Pack supports "json" and "msgpack" formats for data serialization. It remains critical for developers to handle serialization on their respective platforms, or reference and implement the MicroServiceReceiver class from COG Pack to deserialize the Sensor traffic into the Signal format.

2) "SignalName" By overriding the Topic to be a specific "Signal.Name" description, the publisher stream can only ever be given that specific Signal type.

In this case, a ZMQ publisher will only publish Signal data tagged with the "Name" property given by Topic. This is by design. When Signals are relayed from any Sensor in the Signal processing pipeline, the Signal can be routed to the next appropriate Sensor, based on the "Signal.Name" property. All Signals that are tagged via the "Signal.Name" property arrive in this format, regardless of when they are sampled, and published to this topic. In the case where one Sensor samples more frequently than another, both Sensors are still publishing to that stream and Signals of that name will arrive on the topic stream.

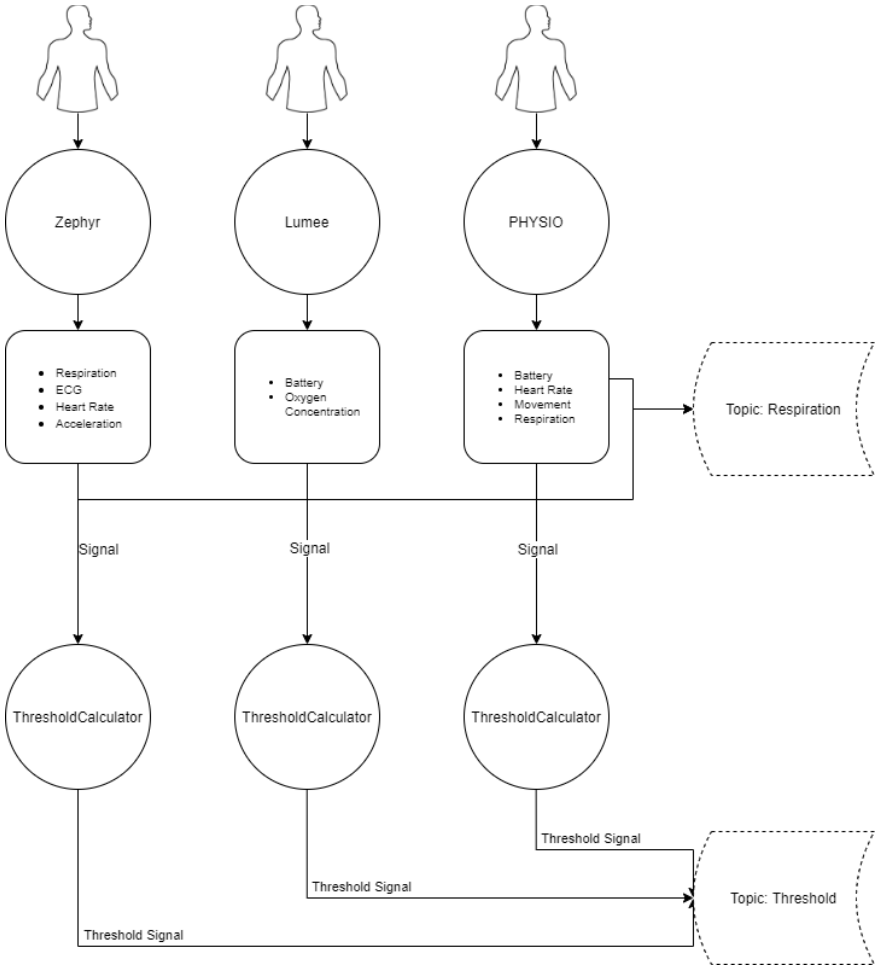


Figure 3. Sensors with Two Different Topics

Figure 3 illustrates how Zephyr, PHYSIO, and Lumee Sensors relay their Signal data to independent software Sensors. Overriding the topic adds the advantage of comparing two hardware Sensors that provide the same Signals. This figure also illustrates the process for comparing and contrasting "Respiration" and "Battery" Signals from each Sensor. Rather than having two separate subscribers aggregate each of the separate topic streams, developers are able to listen for traffic given the specific Signal name ("Respiration" or "Battery") from both sensors simultaneously and perform the same or different calculations.

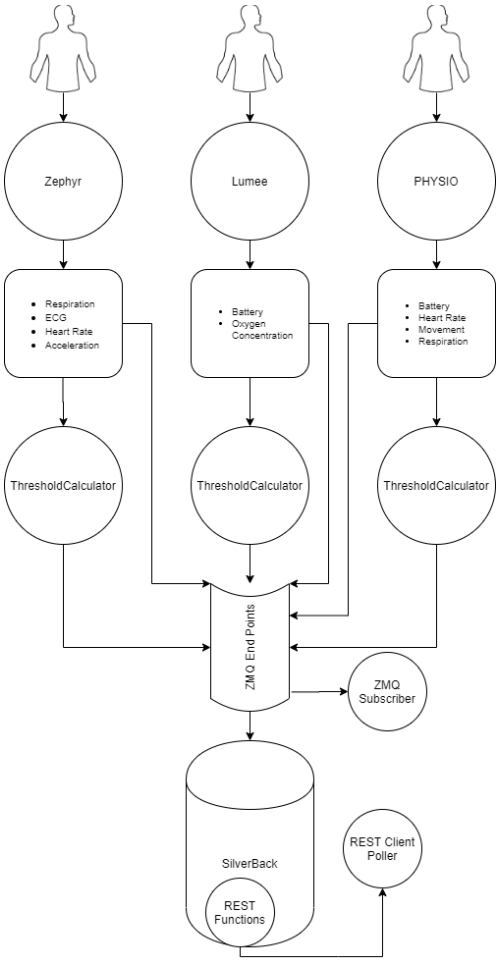


Figure 4. Three Sensors Storing Data from ZMQ

REST connections can leverage the MongoDB database instance implemented in COG Pack. While developers could write continuous timed polling to examine data in real-time and leverage REST as illustrated by Figure 4, REST remains the slowest way to evaluate Signal data. The REST sub-scenario is best for developers who need to examine data post-hoc. The data stored by COG Pack in a MongoDB database leverages the database's best parts. The data is sorted by timestamp order so users can query for specific timespans, all of a Signal's properties, or receive and export the entire database to a preferred format such as CSV, JSON, MongoDB's BSON, or other file format with an interface adapter.

3.2.3. Scenario 2: Applying Thresholds to Signals

Background

Collecting and publishing data brings a unique set of challenges for assessment. In fitness applications, knowing when a participant has a "high" heart rate can indicate they are "feeling the burn" in their workout. However, what is considered a high heart rate? A common scenario in "monitoring systems" is alerting users when a specific data point occurs. Professional sports athletes have considerably "lower" resting heart rates, how would the framework account for the

change in a more physically inclined outlier as a participant? Transmitting physiological Signal data for measurement and providing assessment are key features implemented by COG Pack.

Implementation

To classify and tag a hardware Sensor's physiological Signals that fall in specific zones, COG Pack provides a Signal named "Threshold". In order to differentiate Thresholds for different source Signals, each is Signal is processed by a Threshold Calculator through the **Playbook** implementation**.** Developers reference the "Signal.Units" property to differentiate Threshold Signals. For example, a Threshold Signal will always have "Signal.Name" set to 'Threshold', and the "Signal.Units" property will specify the data type's origin, such as 'Heart Rate' or 'Oxygen Concentration'. Separate Signals provided by different hardware Sensors, can then be equally evaluated, but then uniquely identified by the combination of "Signal.Name" 'Threshold' and "Signal.Units" such as 'Heart Rate' or 'Oxygen Concentration' as previously described.

Figure 3 also shows a ThresholdCalculator which is a software Sensor that can be tailored to a participant's baseline data. This software Sensor identifies when a Signal's Channel Value lies within a specified range. These points can be used to alert and assist researchers when participants are experiencing physiological events in an experiment, or simply when a hardware Sensor's battery needs recharged. Aggregating and performing these calculations in real-time can ease the post-analysis process by performing the calculations during the course of the data collection. After the fact, the REST client can be queried for Threshold Signals that occurred during data collection, i.e., "How often did a Participant experience 'LOW' oxygen saturation?"

As a Sensor, ThresholdCalculator is constantly assessing Signal data provided by another Sensor. ThresholdCalculators are injective by design, meaning one Threshold Calculator can map directly to the Signals of each hardware Sensor.

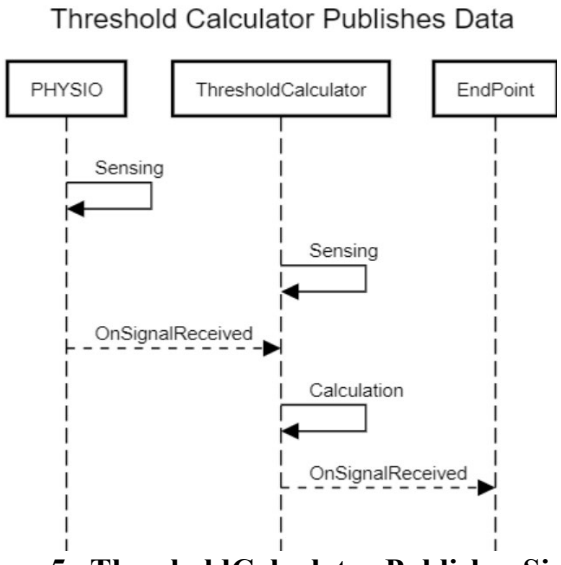


Figure 5. ThresholdCalculator Publishes Signal

As illustrated by Figure 5, both the PHYSIO and ThresholdCalculator are constantly processing (sensing) for data. PHYSIO provides Signal formatted data to a ThresholdCalculator which then calculates Threshold Signals based on a ruleset. The ruleset is defined using set theory as follows:

Rules:
Let "x" be the single channel value in $\mathbb{N}$(natural counting numbers) to measure from a given Signal,
Let **A** be the global minimum value where $x < min$,
Let **B** be the global maximum value where $x > max$,
Let **C** be the set of sets where $\mathbf{C_i}\{ x \mid x \geq (Cmin \,||A) \wedge x < (Cmax||B), C \subseteq A \vee C \subseteq B\}$

The ThresholdCalculator as an ordered set would look like **{ {A}, {C}, {B} }**. **Rule C** requires *that at least one value* is the first value exists from **A** OR $\mathbf{C_{i+1}}$ and the last value exists in **B** OR $\mathbf{C_{i-1}}$. Assuming there exists a range that is less than **B** AND not in **A**, Threshold Calculator requires that additional sets of ranges are created until $\{\{\mathbf{C_0}\}...\{\mathbf{C_i}\}\} < \mathbf{B}$.
Where $\mathbf{C_0}$ can be expressed as$\{\sum_{k=A}^{C[0]max} k+1\}$ and $\mathbf{C_i}\{\sum_{k=C[i]max+1}^{B} k+1\}$ where k is the iterator of the current set of **C** and i is the iterator of the **C**. In programming, this is equivalent to a two-dimensional array where "C[row][column]".
Thusly proving **{{A},{$\mathbf{C_0}$ ... $\mathbf{C_i}$}, {B}}** developers would be able to associate extrema labels to **A** and **B** and the following local labels as sub ranges in **C**.

In Table 1, a participant's set of ThresholdCalculator Heart Rate ranges can be defined as {Low, Medium, High, Very High}.

Table 1. Thresholding Heart Rate in Interval Notation

Range	Heart Rate Range	Zone
0	(80)	Low
1	[80,110)	Medium
2	[110,150)	High
3	[150)	Very High

This participant's Threshold Calculator is given the heart rate value of 80. If developers were to draw these ranges as interval notation it would look like Figure 6 Interval Notation of Table 1

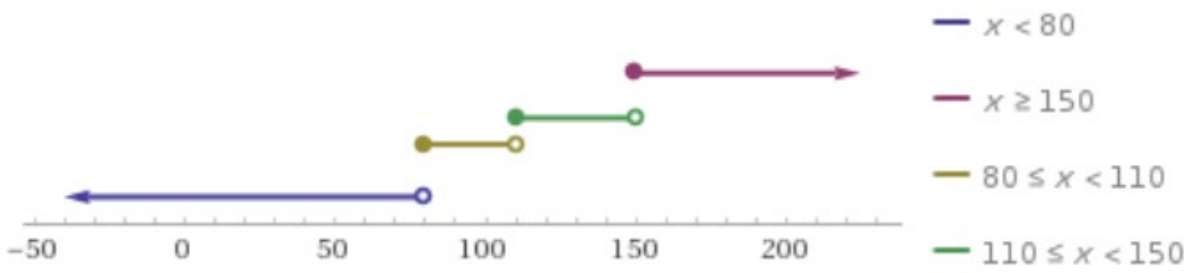


Figure 6. Interval Notation of Table

ThresholdCalculator generates a Threshold Signal with a minimum of two Channels. The first Channel is named "Range" and provides an index into the "Ranges" array found in the Signal's *ExtendedAttributes*. Any subsequent Channels contain the source values used to calculate the Threshold Signal. Threshold Signals based on "Heart Rate" and "Battery" will contain only 1 additional Channel with its name matching that of the Threshold Signal's "Units" property. For more advanced assessment Signals, such as Fixations from eye-tracking, these subsequent Channels may have names such as "X" and "Y" to denote the pixel that defined the Fixation Signal.

ThresholdCalculator naturally rejects inputs that are not declared in its configuration file "ThresholdCalculatorConfig.json". For instance, a PHYSIO "**P**" with the following Signals {Heart Rate, Respiration, Movement} and a Threshold Calculator "**T**" that generates thresholds {Heart Rate, Respiration} from **P**. $\mathbf{T} \supset \mathbf{P}$, **T** includes some properties from **P**. Therefore, if **P** were to pass **T** a Signal that was not defined in the configuration of **T**, **T** would return Ø or "null" as demonstrated in **Figure 7 ThresholdCalculator as n Injective Function**.

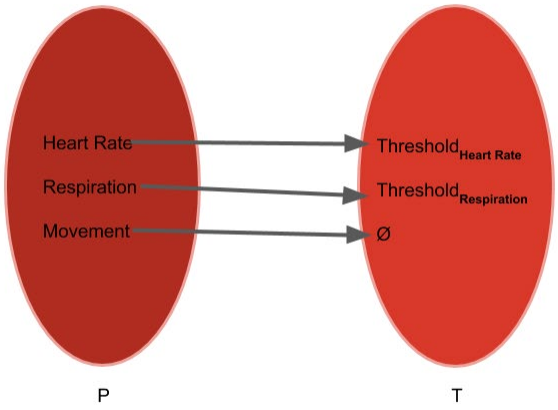


Figure 7. ThresholdCalculator as n Injective Function

COG Pack as a framework expects that software Sensors ***never*** assume that the data for evaluation is of a specific type. The functions that receive Signals and perform operations to generate a new Signal, must verify the receipt of the correct Signal type. For instance, if a machine learning Sensor specializes in evaluating electrocardiograms, then as a function, the machine learning Sensor returns Ø if the Signal it was given was not of type/name "ECG".

3.2.4. Scenario 3: Executing a Playbook

Background

Imagine a developer learned the basic rules and began to implement their own functions. This developer creates their own projects to test their functions as software Sensors in the event space as described in Scenario 1: Publishing Signals. The developer provides this function as publicly available code, having completed the testing phase of the development cycle. This functionality now exists as a newly minted Sensor and is ready to be deployed in the larger COG Pack framework.

A new requirement hits that developer's desk: the Sensor's data stream now needs to be written directly to the database, instead of publishing JSON strings. The developer could provide two valid solutions given the request.

1) Rewrite their project code to use the COG Pack MongoDB client, SilverBack.
2) Write a Playbook, leveraging the **Kernel** and Sensor's real-time interpretation of their Sensor, to write directly to **SilverBack**.

These are both "valid" in the sense that both solutions get the job done! However, COG Pack is meant to be flexible not only in the domain operational sense but as a fully fluid software architecture. This flexibility is made possible by **Playbooks**.

Implementation

What is a **Playbook**? A **Playbook** to a researcher is the JSON that manipulates the desired output of a Sensor's Signals, such as performing real time analysis and recording results to a database. To a developer, a **Playbook** is a class that manages the execution of ordered **Steps**, which are composed of **Signatures**. **Signatures** are the collection of function pointers that make up one single **Step** in a **Playbook**.

Signatures are either labeled as a **Function** or **Action**. An example of a **Function** would be "GetThreshold" whereas an **Action** would be "PublishSignal". A **Function** is implemented in the Sensor class, and **Functions** are only added to the **Kernel's** collection of *AvailableSignatures* when a Sensor provides it to the *SignalHandler* dictionary. Conversely, an **Action** is provided and generated by the **Kernel** at run-time (with the exception that ZMQ is only enabled when requested in a **SensorConfig** file as described in Scenario 1: Publishing Signals).

Identifying why **Functions** vs **Actions** exist plays a critical role in writing **Signatures** for a **Step**. A **Step** executes **Signatures** in stack order, meaning first in first out. **Steps** are also intended to be read like simple sentences, from bottom to top. Consider the following sentence:

Publish the Threshold SignalProcessingAlgorithm from a Heart Rate Sensor

In computer memory, the resulting sentence's **Step's Signature** looks like Figure 8

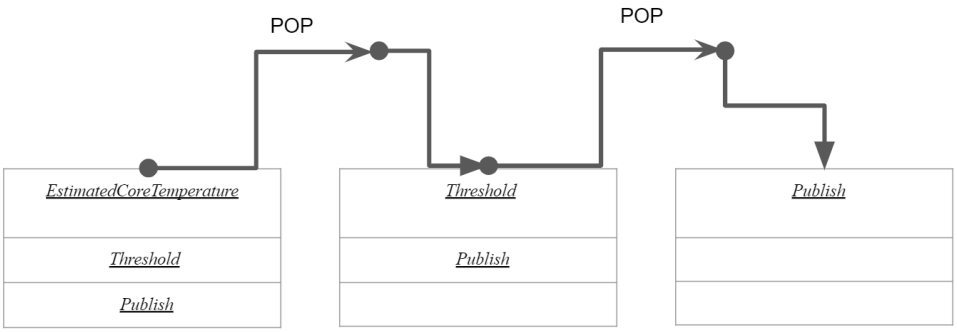


Figure 8. Executing a Step in Stack Order

Order matters, with each pop the function at the top of the stack is executed. Alternatively, expressing this **Step's Signatures** as a mathematical formula could be illustrated as Figure 9.

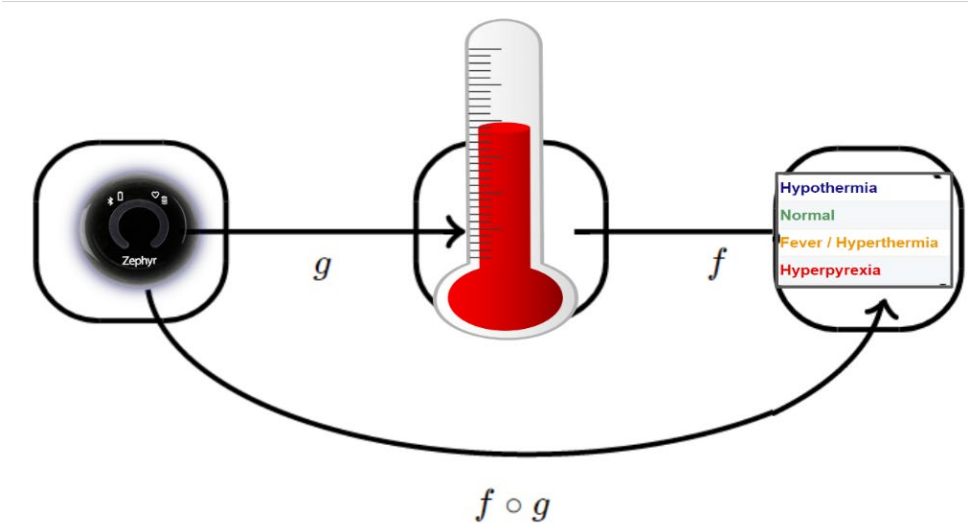


Figure 9. Playbook Function Composition Flow

Figure 9 and the function above highlight the implementation of a *single* **Step** in the C# run-time. A **Step's Signatures** are written as $f(g(x))$. Meaning that the innermost functions are executed first, then the additional functions execute in stack order. As mentioned in Scenario 2:

Applying Thresholds to Signals, if the function **Signature** provided identifies a Signal as null and fails evaluation, the other **Signatures** will NOT execute.

As an example best identified in Figure 9. Imagine the **Step** is provided with a "Respiration" Signal, but the **Step's Signature** functions only support a "Heart Rate" Signal. In this case, the innermost function would return null, resulting in not executing the next **Steps**. Developers are alerted about **Steps** failing through console messages. This reactive approach allows developers to chain **Signature** functions together and evaluate different aspects of their code for unit testing, hardware/software integration, and on-the-fly changes without having to change the overall **Playbook's** subsequent **Steps**.

Executing *multiple **Steps** with n count of **Signatures*** within compiled code demonstrates the modular nature of COG Pack. If one **Step** were to fail, other **Steps** in the **Playbook** can still execute, without failing, crashing, or becoming unstable. This prevents code paths from intersecting and interfering with one another. Notice the three **Steps** in Figure 10. The Zephyr by itself does not possess the functionality for CoreTempEstimation of Threshold nor can it write directly to the database. Stitching together these functional elements in a **Playbook**, allows access to **Functions** and **Actions** supplied by the **Kernel** or other Sensors. As the Zephyr provides Signals, the **Playbook's Step's Signatures** determines the route of each resulting Signal. To alter the route of Zephyr Signals and publish them, then a simple change of the **Action** signature to *PublishSignal* would implement that capability.

Figure 10. Zephyr Playbook Execution

Notice in Figure 11 that the *Playbook Structure* relies significantly less on other Sensors to execute immediately. In *Traditional Event Structure* the execution thread has to be managed by each Sensor object. When is it best to write a custom solution? In the event when a project requires incredibly granular solutions that COG Pack does not support by the Sensor's *OnSignalReceived* EventHandler. While **Playbooks** cover a vast majority of functionality for researchers and developers, an edge case not anticipated by the framework could be solved by leveraging the many different end-points of COG Pack.

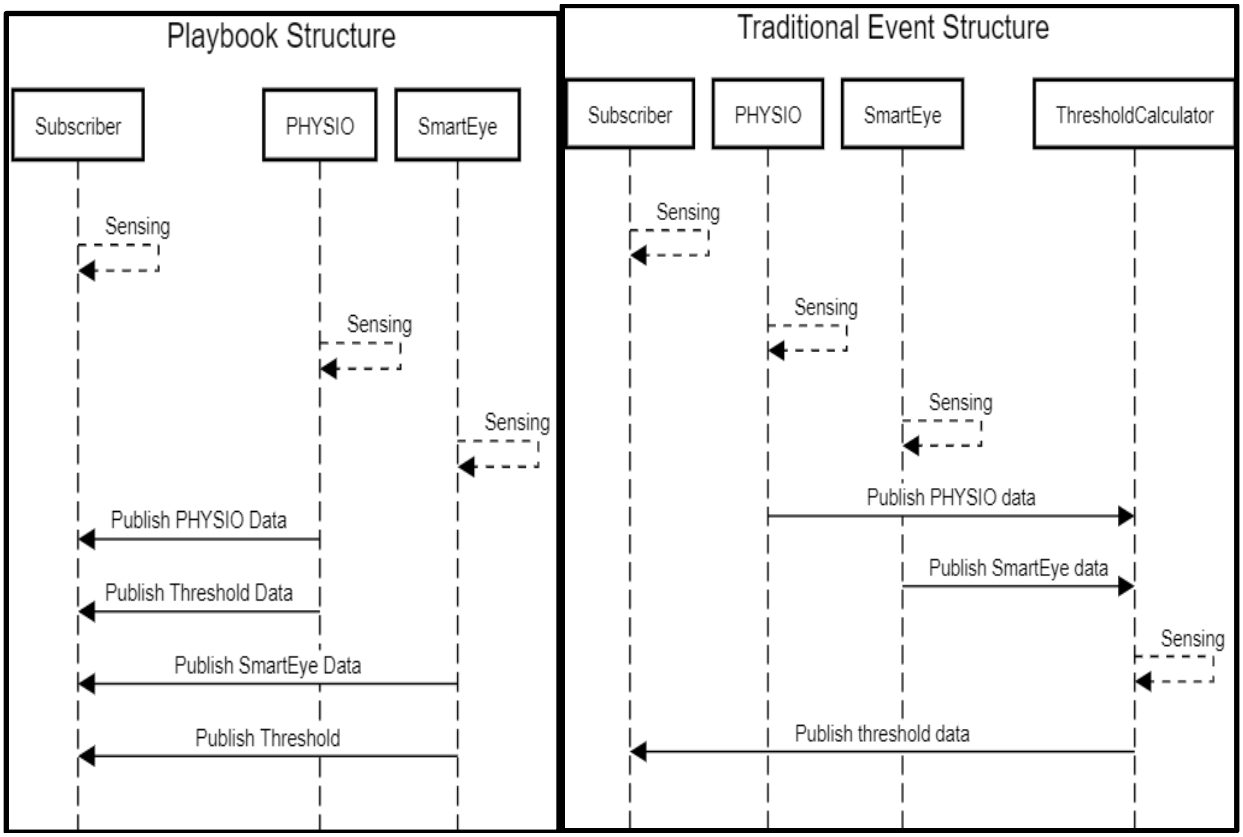


Figure 11. Playbook vs Event Structure

4.0 REFERENCES

Downs, C. J. (Writer), & Downs, C. J. (Director). (2020). *Sensor Setup* [Motion Picture]. 711 HPW/RHBCN.

Downs, C. J., & Dukes, A. W. (2020). *COG Pack™ Dashboard.* Wright Patterson: DTIC.

Downs, C. J., Dukes, A. W., & Duberstein, S. J. (2020). *Reflection Base UI - Dynamically Generated Configuration-based UI Elements with Example Implementations Found in COG Pack™.* Wright Patterson: DTIC.

Duberstein, S. J. (Writer), & Duberstein, S. J. (Director). (2020). *Implementing Playbooks With a Kernel* [Motion Picture]. 711 HPW/RHBCN.

Dukes, A. W. (Producer), Dukes, A. W. (Writer), & Dukes, A. W. (Director). (2020). *Developing and Integrating New Sensors As Plugins* [Motion Picture].

Dukes, A. W., Duberstein, S. J., & Rommel, M. T. (2020). *Cognitive Operations Gear (COG) Pack(TM) API Specification.* RHBCN, 711 HPW. Wright Patterson, OH: DTIC.

Dukes, A. W., Duberstein, S. J., Blackford, E. B., Klosterman, S. L., Rommel, M. T., & Downs, C. J. (2019). *Cognitive Operations Gear (COG) Pack™ Software API White Paper - An API Designed to Enable the Integration of Multi-modal Data-types for Real-time Analysis.* RHBCN, 711 HPW. Wright Patterson, OH: DTIC.

Fielding, R. T. (2000). *Representational State Transfer (REST)*. Retrieved 07 06, 2020 from Donald Bren School of Information and Computer Sciences: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Hintjens, P. (n.d.). *ZeroMQ Guide*. From Preface: http://zguide.zeromq.org/page:preface

Rochlin, M. (2013). *I don't understand Dot Notation.* From Code Academy: https://www.codecademy.com/forum_questions/5170307264a7402d9a0012f5#:~:text=Each%20Object%20is%20an%20instance,I%20am%20a%20person.&text=Dot%20notation%20allows%20us%20to,the%20methods%20inside%20that%20class.

Rommel, M. T. (Writer), & Rommel, M. T. (Director). (2020). *Eye-Gaze-Head Advanced Fixations Control* [Motion Picture]. 711 HPW/RHBCN.

Rommel, M. T. (Writer), & Rommel, M. T. (Director). (2020). *Eye-Gaze-Head Heatmap Control* [Motion Picture]. 711 HPW/RHBCN.

The MVVM Pattern. (2012, 10 04). From Microsoft Docs: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)?redirectedfrom=MSDN

APPENDICES

Backend Logic DFD

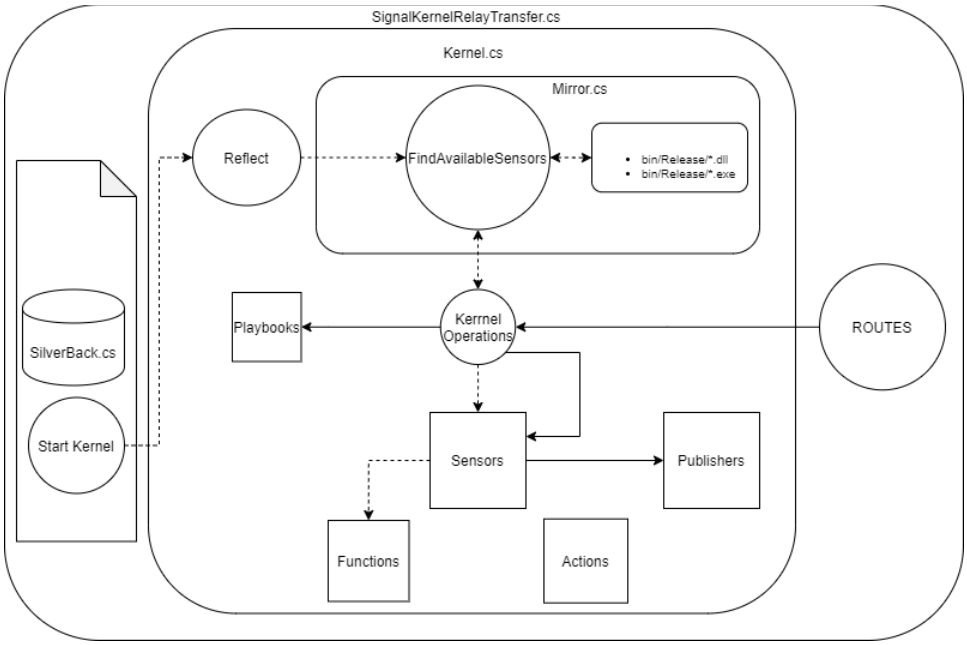


Figure 12. DFD 1 - Backend Logic

Figure 12 provides a high level "bird's eye view" interpretation of Figure 1. While Figure 1 contains every data transformation path, Figure 12 consolidates the low level interactions. In traditional software development, developers start with this first diagram then draw the subsequent interactions through iterative DFDs.

High Level Sequence Diagram

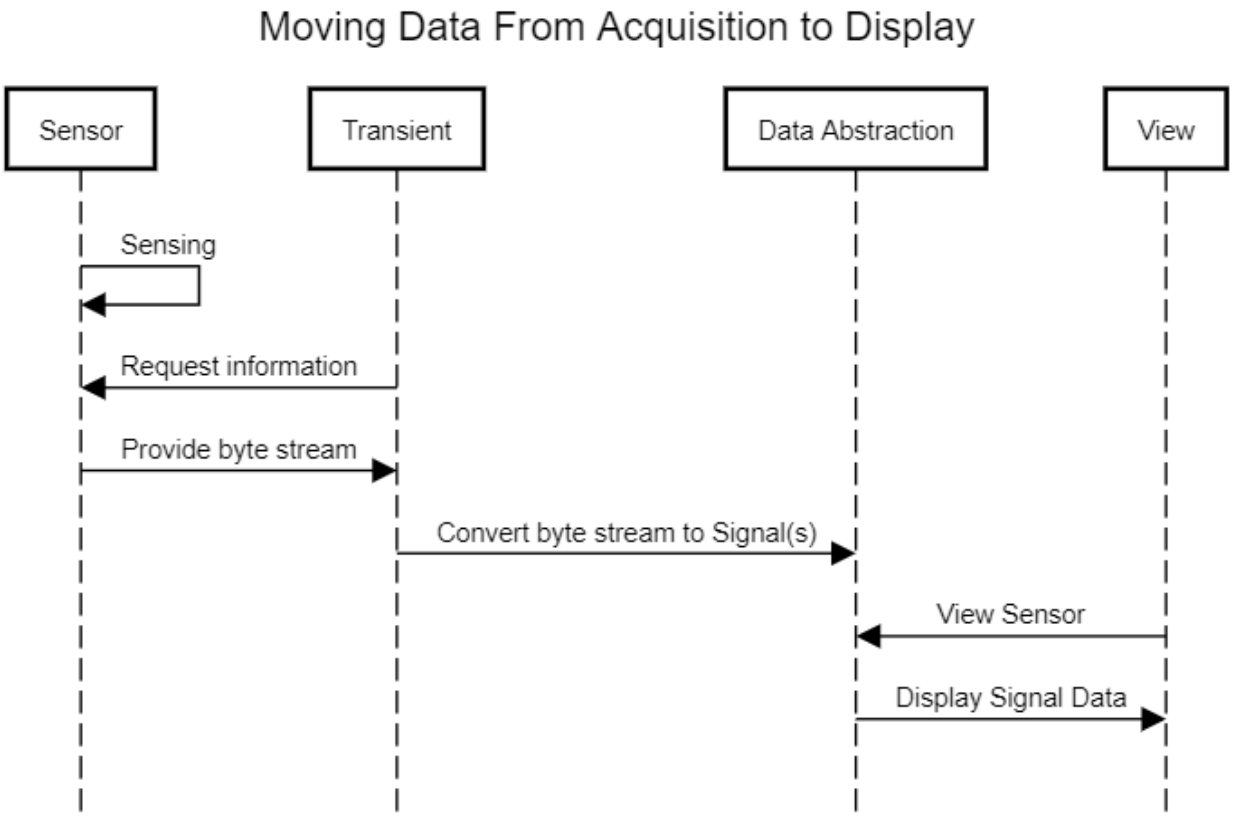


Figure 13. Moving Data from Acquisition to Display

Figure 13 conveys the process of moving data from acquisition to display. The physical Sensor constantly provides new data based on the Sensor's sampling rate. Next, the transient layer provides the byte stream to the abstraction layer and is converted to Signals. Once the Signals are converted, other software systems are then able to interpret and view the information in a View controller. This separation of data and view logic follows the design patterns of MVVM (The MVVM Pattern, 2012)

Package Diagram

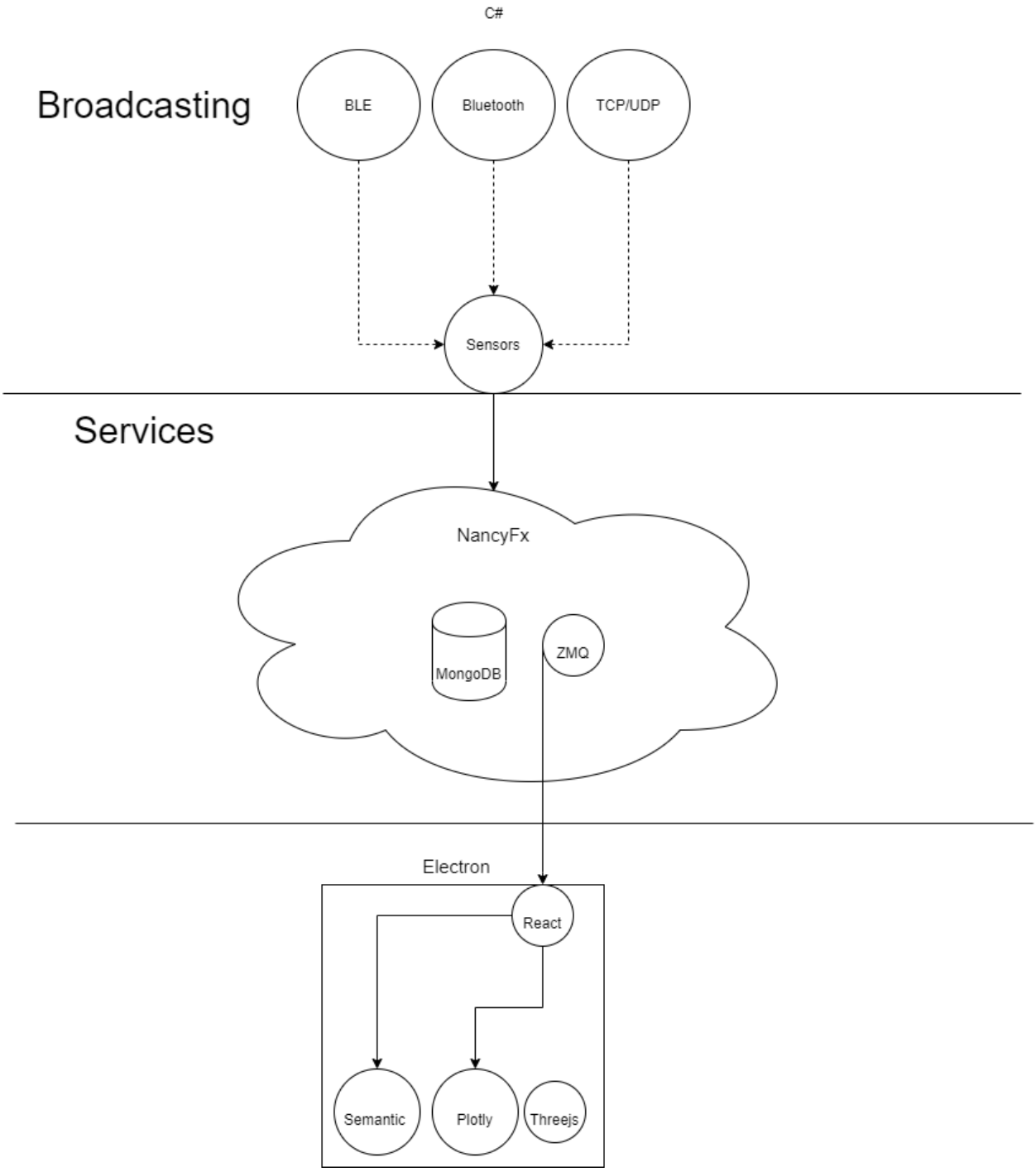
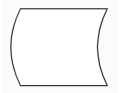


Figure 14. Package Diagram

The package diagram (Figure 14) demonstrates the "flow" of data between the software libraries utilized by COG Pack. The figure also displays the separation of the C# backend code from the

Electron user interface (Downs & Dukes, COG Pack™ Dashboard, 2020). C# libraries manage the communication with hardware Sensors at the top, ZMQ is used for streaming Signals to manage data flow, MongoDB for archiving to support post-analysis, and a NancyFx webserver to interface between the C# and Electron components. Finally, JSON formatted Signals are sent via ZMQ and received in Electron. Here visualization libraries are incorporated at this highest level for generating and managing HTML(React), framing UI elements(Semantic), graphing(Plotly), and 3D rendering(Threejs) (Downs & Dukes, COG Pack™ Dashboard, 2020).

Software Diagram Key

Name	Figure	Description
Database		Stores and accesses data
Data Flow		Carries data to a function, database, or output
Function		A function
Output		Resulting output
Sensing		Constantly processing
Topic		Data stream ZMQ access point

Subset of API Attributes

For the complete description of all API attributes for each object, reference (Dukes, Duberstein, & Rommel, Cognitive Operations Gear (COG) Pack(TM) API Specification, 2020)

Channel	"Channels":[
	{
	"Name":"","Values":[]
	}
	]

SensorConfig	"Emulation": true, "PlaybackFile": "", "Signals": [...], "SensorName": "", "SensorID": "", "ComputerName": null, "Location": "", "ZMQPort": 3017, "MessageType": "json"
Sensor	"State": "Disconnected", "SensorName": "Sensor", "SensorID": "", "Signals": [], "ExtendedAttributes": [], "ComputerName": "", "Location": "

Signal	"Units": "",
	"TimeStamp": {
	"SystemDateTime": ""
	},
	"Channels": [...],//See Channel
	"ExtendedAttributes": [],
	"Name": "",
	"Source": {...}//See sensor

Threshold Signal	"Units": "OriginSignal",
	"TimeStamp": {
	"SystemDateTime": ""
	},
	"Channels": [Name: “Range”,...],//See Channel
	"ExtendedAttributes": []
	"Name": "Threshold”,
	"Source": {...}//See sensor